rocSOLVER Documentation

Release 3.17.0

Advanced Micro Devices

Apr 08, 2022

CONTENTS

1 rocSOLVER User Guide **3**

1.1 Introduction . . . 3

1.1.1 Library overview . . . 3

1.1.2 Currently implemented functionality . . . 3

LAPACK auxiliary functions . . . 3

LAPACK main functions . . . 5

LAPACK-like functions . . . 7

1.2 Building and Installation . . . 7

1.2.1 Prerequisites . . . 7

1.2.2 Installing from pre-built packages . . . 8

1.2.3 Building & installing from source . . . 8

Using the install.sh script . . . 8

Manual building and installation . . . 9

1.3 Using rocSOLVER . . . 11

1.3.1 QR factorization of a single matrix . . . 11

1.3.2 QR factorization of a batch of matrices . . . 13

Strided_batched version . . . 13

Batched version . . . 16

1.4 Memory Model . . . 18

1.4.1 Automatic workspace . . . 19

1.4.2 User-managed workspace . . . 19

Minimum required size . . . 19

Using an environment variable . . . 20

Using helper functions . . . 20

1.4.3 User-owned workspace . . . 20

1.5 Multi-level Logging . . . 20

1.5.1 Logging modes . . . 21

Trace logging . . . 21

Bench logging . . . 21

Profile logging . . . 21

1.5.2 Initialization and set-up . . . 21

1.5.3 Example code . . . 22

1.5.4 Kernel logging . . . 23

1.5.5 Multiple host threads . . . 23

1.6 Clients . . . 23

1.6.1 Testing rocSOLVER . . . 24

1.6.2 Benchmarking rocSOLVER . . . 24

1.6.3 rocSOLVER sample code . . . 25

2 rocSOLVER Library Design Guide **27**

2.1 Introduction . . . 27
2.2 Batched rocSOLVER . . . 27
2.3 Tuning rocSOLVER Performance . . . 27
2.3.1 geqr2/geqrf and geql2/geqlf functions . . . 30
GEQxF_BLOCKSIZE . . . 30
GEQxF_GEQx2_SWITCHSIZE . . . 30
2.3.2 gerq2/gerqf and gelq2/gelqf functions . . . 30
GExQF_BLOCKSIZE . . . 30
GExQF_GExQ2_SWITCHSIZE . . . 31
2.3.3 org2r/orgqr, org2l/orgql, ung2r/ungqr and ung2l/ungql functions . . . 31
xxGQx_BLOCKSIZE . . . 31
xxGQx_xxGQx2_SWITCHSIZE . . . 31
2.3.4 orgr2/orgrq, orgl2/orglq, ungr2/ungrq and ungl2/unglq functions . . . 31
xxGxQ_BLOCKSIZE . . . 32
xxGxQ_xxGxQ2_SWITCHSIZE . . . 32
2.3.5 orm2r/ormqr, orm2l/ormql, unm2r/unmqr and unm2l/unmql functions . . . 32
xxMQx_BLOCKSIZE . . . 32
2.3.6 ormr2/ormrq, orml2/ormlq, unmr2/unmrq and unml2/unmlq functions . . . 32
xxMxQ_BLOCKSIZE . . . 33
2.3.7 gebd2/gebrd and labrd functions . . . 33
GEBRD_BLOCKSIZE . . . 33
GEBRD_GEBD2_SWITCHSIZE . . . 33
2.3.8 gesvd function . . . 33
THIN_SVD_SWITCH . . . 34
2.3.9 sytd2/sytrd, hetd2/hetrd and latrd functions . . . 34
xxTRD_BLOCKSIZE . . . 34
xxTRD_xxTD2_SWITCHSIZE . . . 34
2.3.10 sygs2/sygst and hegs2/hegst functions . . . 34
xxGST_BLOCKSIZE . . . 35
2.3.11 syevd, heevd and stedc functions . . . 35
STEDC_MIN_DC_SIZE . . . 35
2.3.12 potf2/potrf functions . . . 35
POTRF_BLOCKSIZE . . . 35
POTRF_POTF2_SWITCHSIZE . . . 36
2.3.13 sytf2/sytrf and lasyf functions . . . 36
SYTRF_BLOCKSIZE . . . 36
SYTRF_SYTF2_SWITCHSIZE . . . 36
2.3.14 getf2/getrf functions . . . 36
GETF2_MAX_COLS . . . 36
GETF2_MAX_THDS . . . 36
GETF2_OPTIM_NGRP . . . 36
GETRF_NUM_INTERVALS . . . 36
GETRF_INTERVALS . . . 36
GETRF_BLKSIZES . . . 36
GETRF_BATCH_NUM_INTERVALS . . . 37
GETRF_BATCH_INTERVALS . . . 37
GETRF_BATCH_BLKSIZES . . . 37
GETRF_NPVT_NUM_INTERVALS . . . 37
GETRF_NPVT_INTERVALS . . . 37
GETRF_NPVT_BLKSIZES . . . 37
GETRF_NPVT_BATCH_NUM_INTERVALS . . . 37
GETRF_NPVT_BATCH_INTERVALS . . . 37
GETRF_NPVT_BATCH_BLKSIZES . . . 37
2.3.15 getri function . . . 37

GETRI_MAX_COLS . . . 37
GETRI_TINY_SIZE . . . 37
GETRI_NUM_INTERVALS . . . 37
GETRI_INTERVALS . . . 37
GETRI_BLKSIZES . . . 37
GETRI_BATCH_TINY_SIZE . . . 37
GETRI_BATCH_NUM_INTERVALS . . . 37
GETRI_BATCH_INTERVALS . . . 37
GETRI_BATCH_BLKSIZES . . . 37
2.3.16 trtri function . . . 37
TRTRI_MAX_COLS . . . 37
TRTRI_NUM_INTERVALS . . . 37
TRTRI_INTERVALS . . . 37
TRTRI_BLKSIZES . . . 37
TRTRI_BATCH_NUM_INTERVALS . . . 37
TRTRI_BATCH_INTERVALS . . . 37
TRTRI_BATCH_BLKSIZES . . . 38
2.4 Contributing Guidelines . . . 38

3 rocSOLVER API **39**
3.1 Types . . . 39
3.1.1 Additional types . . . 39
rocblas_direct . . . 39
rocblas_storev . . . 40
rocblas_svect . . . 40
rocblas_evect . . . 40
rocblas_workmode . . . 41
rocblas_eform . . . 41
3.2 LAPACK Auxiliary Functions . . . 41
3.2.1 Vector and Matrix manipulations . . . 42
rocsolver_<type>lacgv() . . . 42
rocsolver_<type>laswp() . . . 42
3.2.2 Householder reflections . . . 43
rocsolver_<type>larfg() . . . 43
rocsolver_<type>larft() . . . 44
rocsolver_<type>larf() . . . 45
rocsolver_<type>larfb() . . . 46
3.2.3 Bidiagonal forms . . . 48
rocsolver_<type>labrd() . . . 48
rocsolver_<type>bdsqr() . . . 50
3.2.4 Tridiagonal forms . . . 51
rocsolver_<type>latrd() . . . 51
rocsolver_<type>sterf() . . . 53
rocsolver_<type>steqr() . . . 53
rocsolver_<type>stedc() . . . 54
3.2.5 Symmetric matrices . . . 55
rocsolver_<type>lasyf() . . . 56
3.2.6 Orthonormal matrices . . . 57
rocsolver_<type>org2r() . . . 57
rocsolver_<type>orgqr() . . . 58
rocsolver_<type>orgl2() . . . 59
rocsolver_<type>orglq() . . . 59
rocsolver_<type>org2l() . . . 60
rocsolver_<type>orgql() . . . 61

rocsolver_<type>orgbr() 61
rocsolver_<type>orgtr() 62
rocsolver_<type>orm2r() 63
rocsolver_<type>ormqr() 64
rocsolver_<type>orml2() 65
rocsolver_<type>ormlq() 66
rocsolver_<type>orm2l() 67
rocsolver_<type>ormql() 68
rocsolver_<type>ormbr() 69
rocsolver_<type>ormtr() 70
3.2.7 Unitary matrices 72
rocsolver_<type>ung2r() 72
rocsolver_<type>ungqr() 73
rocsolver_<type>ungl2() 74
rocsolver_<type>unglq() 74
rocsolver_<type>ung2l() 75
rocsolver_<type>ungql() 76
rocsolver_<type>ungbr() 76
rocsolver_<type>ungtr() 77
rocsolver_<type>unm2r() 78
rocsolver_<type>unmqr() 79
rocsolver_<type>unml2() 80
rocsolver_<type>unmlq() 82
rocsolver_<type>unm2l() 83
rocsolver_<type>unmql() 84
rocsolver_<type>unmbr() 85
rocsolver_<type>unmtr() 86
3.3 LAPACK Functions 87
3.3.1 Triangular factorizations 88
rocsolver_<type>potf2() 88
rocsolver_<type>potf2_batched() 89
rocsolver_<type>potf2_strided_batched() 90
rocsolver_<type>potrf() 91
rocsolver_<type>potrf_batched() 92
rocsolver_<type>potrf_strided_batched() 93
rocsolver_<type>getf2() 94
rocsolver_<type>getf2_batched() 95
rocsolver_<type>getf2_strided_batched() 96
rocsolver_<type>getrf() 97
rocsolver_<type>getrf_batched() 98
rocsolver_<type>getrf_strided_batched() 99
rocsolver_<type>sytf2() 100
rocsolver_<type>sytf2_batched() 102
rocsolver_<type>sytf2_strided_batched() 104
rocsolver_<type>sytrf() 105
rocsolver_<type>sytrf_batched() 107
rocsolver_<type>sytrf_strided_batched() 109
3.3.2 Orthogonal factorizations 110
rocsolver_<type>geqr2() 111
rocsolver_<type>geqr2_batched() 112
rocsolver_<type>geqr2_strided_batched() 113
rocsolver_<type>geqrf() 115
rocsolver_<type>geqrf_batched() 116
rocsolver_<type>geqrf_strided_batched() 117

rocsolver_<type>gerq2() . . . 118
rocsolver_<type>gerq2_batched() . . . 119
rocsolver_<type>gerq2_strided_batched() . . . 120
rocsolver_<type>gerqf() . . . 122
rocsolver_<type>gerqf_batched() . . . 123
rocsolver_<type>gerqf_strided_batched() . . . 124
rocsolver_<type>geql2() . . . 125
rocsolver_<type>geql2_batched() . . . 126
rocsolver_<type>geql2_strided_batched() . . . 127
rocsolver_<type>geqlf() . . . 129
rocsolver_<type>geqlf_batched() . . . 130
rocsolver_<type>geqlf_strided_batched() . . . 131
rocsolver_<type>gelq2() . . . 132
rocsolver_<type>gelq2_batched() . . . 133
rocsolver_<type>gelq2_strided_batched() . . . 134
rocsolver_<type>gelqf() . . . 135
rocsolver_<type>gelqf_batched() . . . 136
rocsolver_<type>gelqf_strided_batched() . . . 138
3.3.3 Problem and matrix reductions . . . 139
rocsolver_<type>gebd2() . . . 140
rocsolver_<type>gebd2_batched() . . . 141
rocsolver_<type>gebd2_strided_batched() . . . 143
rocsolver_<type>gebrd() . . . 145
rocsolver_<type>gebrd_batched() . . . 146
rocsolver_<type>gebrd_strided_batched() . . . 148
rocsolver_<type>sytd2() . . . 150
rocsolver_<type>sytd2_batched() . . . 151
rocsolver_<type>sytd2_strided_batched() . . . 153
rocsolver_<type>hetd2() . . . 154
rocsolver_<type>hetd2_batched() . . . 155
rocsolver_<type>hetd2_strided_batched() . . . 156
rocsolver_<type>sytrd() . . . 158
rocsolver_<type>sytrd_batched() . . . 159
rocsolver_<type>sytrd_strided_batched() . . . 160
rocsolver_<type>hetrd() . . . 162
rocsolver_<type>hetrd_batched() . . . 163
rocsolver_<type>hetrd_strided_batched() . . . 164
rocsolver_<type>sygs2() . . . 165
rocsolver_<type>sygs2_batched() . . . 167
rocsolver_<type>sygs2_strided_batched() . . . 168
rocsolver_<type>hegs2() . . . 169
rocsolver_<type>hegs2_batched() . . . 170
rocsolver_<type>hegs2_strided_batched() . . . 171
rocsolver_<type>sygst() . . . 173
rocsolver_<type>sygst_batched() . . . 174
rocsolver_<type>sygst_strided_batched() . . . 175
rocsolver_<type>hegst() . . . 176
rocsolver_<type>hegst_batched() . . . 177
rocsolver_<type>hegst_strided_batched() . . . 178
3.3.4 Linear-systems solvers . . . 180
rocsolver_<type>trtri() . . . 180
rocsolver_<type>trtri_batched() . . . 181
rocsolver_<type>trtri_strided_batched() . . . 182
rocsolver_<type>getri() . . . 183

rocsolver_<type>getri_batched() . . . 184
rocsolver_<type>getri_strided_batched() . . . 185
rocsolver_<type>getrs() . . . 186
rocsolver_<type>getrs_batched() . . . 187
rocsolver_<type>getrs_strided_batched() . . . 188
rocsolver_<type>gesv() . . . 189
rocsolver_<type>gesv_batched() . . . 190
rocsolver_<type>gesv_strided_batched() . . . 192
rocsolver_<type>potri() . . . 193
rocsolver_<type>potri_batched() . . . 194
rocsolver_<type>potri_strided_batched() . . . 195
rocsolver_<type>potrs() . . . 196
rocsolver_<type>potrs_batched() . . . 197
rocsolver_<type>potrs_strided_batched() . . . 198
rocsolver_<type>posv() . . . 199
rocsolver_<type>posv_batched() . . . 200
rocsolver_<type>posv_strided_batched() . . . 201
3.3.5 Least-squares solvers . . . 203
rocsolver_<type>gels() . . . 203
rocsolver_<type>gels_batched() . . . 204
rocsolver_<type>gels_strided_batched() . . . 205
3.3.6 Symmetric eigensolvers . . . 207
rocsolver_<type>syev() . . . 208
rocsolver_<type>syev_batched() . . . 209
rocsolver_<type>syev_strided_batched() . . . 210
rocsolver_<type>heev() . . . 211
rocsolver_<type>heev_batched() . . . 212
rocsolver_<type>heev_strided_batched() . . . 213
rocsolver_<type>syevd() . . . 214
rocsolver_<type>syevd_batched() . . . 215
rocsolver_<type>syevd_strided_batched() . . . 216
rocsolver_<type>heevd() . . . 217
rocsolver_<type>heevd_batched() . . . 218
rocsolver_<type>heevd_strided_batched() . . . 219
rocsolver_<type>sygv() . . . 220
rocsolver_<type>sygv_batched() . . . 221
rocsolver_<type>sygv_strided_batched() . . . 223
rocsolver_<type>hegv() . . . 224
rocsolver_<type>hegv_batched() . . . 225
rocsolver_<type>hegv_strided_batched() . . . 227
rocsolver_<type>sygvd() . . . 229
rocsolver_<type>sygvd_batched() . . . 230
rocsolver_<type>sygvd_strided_batched() . . . 231
rocsolver_<type>hegvd() . . . 233
rocsolver_<type>hegvd_batched() . . . 234
rocsolver_<type>hegvd_strided_batched() . . . 236
3.3.7 Singular value decomposition . . . 238
rocsolver_<type>gesvd() . . . 238
rocsolver_<type>gesvd_batched() . . . 240
rocsolver_<type>gesvd_strided_batched() . . . 242
3.4 Lapack-like Functions . . . 245
3.4.1 Triangular factorizations . . . 245
rocsolver_<type>getf2_npvt() . . . 246
rocsolver_<type>getf2_npvt_batched() . . . 247

rocsolver_<type>getf2_npvt_strided_batched() . . . 248
rocsolver_<type>getrf_npvt() . . . 249
rocsolver_<type>getrf_npvt_batched() . . . 250
rocsolver_<type>getrf_npvt_strided_batched() . . . 251
3.4.2 Linear-systems solvers . . . 252
rocsolver_<type>getri_npvt() . . . 252
rocsolver_<type>getri_npvt_batched() . . . 253
rocsolver_<type>getri_npvt_strided_batched() . . . 254
rocsolver_<type>getri_outofplace() . . . 255
rocsolver_<type>getri_outofplace_batched() . . . 256
rocsolver_<type>getri_outofplace_strided_batched() . . . 257
rocsolver_<type>getri_npvt_outofplace() . . . 259
rocsolver_<type>getri_npvt_outofplace_batched() . . . 259
rocsolver_<type>getri_npvt_outofplace_strided_batched() . . . 261
3.5 Logging Functions and Library Information . . . 263
3.5.1 Logging functions . . . 263
rocsolver_log_begin() . . . 263
rocsolver_log_end() . . . 263
rocsolver_log_set_layer_mode() . . . 263
rocsolver_log_set_max_levels() . . . 264
rocsolver_log_restore_defaults() . . . 264
rocsolver_log_write_profile() . . . 264
rocsolver_log_flush_profile() . . . 264
3.5.2 Library information . . . 264
rocsolver_get_version_string() . . . 264
rocsolver_get_version_string_size() . . . 265
3.6 Deprecated . . . 265
3.6.1 Types . . . 265
rocsolver_int . . . 265
rocsolver_handle . . . 266
rocsolver_direction . . . 266
rocsolver_storev . . . 266
rocsolver_operation . . . 266
rocsolver_fill . . . 266
rocsolver_diagonal . . . 266
rocsolver_side . . . 267
rocsolver_status . . . 267
3.6.2 Auxiliary functions . . . 267
rocsolver_create_handle() . . . 267
rocsolver_destroy_handle() . . . 268
rocsolver_set_stream() . . . 268
rocsolver_get_stream() . . . 268
rocsolver_set_vector() . . . 268
rocsolver_get_vector() . . . 268
rocsolver_set_matrix() . . . 269
rocsolver_get_matrix() . . . 269

4 License & Attributions **271**

Index **273**

Legal Disclaimer

Contents

rocSOLVER's documentation consists of 3 main Chapters. The User Guide is the starting point for new users of the library, and a basic reference for current users and/or users of LAPACK. Advanced users and developers who want to further understand or extend the rocSOLVER library may wish to refer to the Library Design Guide. For a list of currently implemented routines, and a description of each's functionality and input and output parameters, see the rocSOLVER API.

CHAPTER

ONE

ROCSOLVER USER GUIDE

1.1 Introduction

Table of contents

- *Library overview*
- *Currently implemented functionality*
  - *LAPACK auxiliary functions*
  - *LAPACK main functions*
  - *LAPACK-like functions*

1.1.1 Library overview

rocSOLVER is an implementation of LAPACK routines on top of the AMD's open source ROCm platform. rocSOLVER is implemented in the HIP programming language and optimized for AMD's latest discrete GPUs.

1.1.2 Currently implemented functionality

The rocSOLVER library is in the early stages of active development. New features are being continuously added, with new functionality documented at each release of the ROCm platform.

The following tables summarize the LAPACK functionality implemented for the different supported precisions in rocSOLVER's latest release. All LAPACK and LAPACK-like main functions include *_batched* and *_strided_batched* versions. For a complete description of the listed routines, please see the *rocSOLVER API* document.

LAPACK auxiliary functions

Table 1: Vector and matrix manipulations

Function	single	double	single complex	double complex
rocsolver_lacgv	x	x	x	x
rocsolver_laswp	x	x	x	x

Table 2: Householder reflections

Function	single	double	single complex	double complex
rocsolver_larfg	x	x	x	x
rocsolver_larf	x	x	x	x
rocsolver_larft	x	x	x	x
rocsolver_larfb	x	x	x	x

Table 3: Bidiagonal forms

Function	single	double	single complex	double complex
rocsolver_labrd	x	x	x	x
rocsolver_bdsqr	x	x	x	x

Table 4: Tridiagonal forms

Function	single	double	single complex	double complex
rocsolver_sterf	x	x		
rocsolver_latrd	x	x	x	x
rocsolver_steqr	x	x	x	x
rocsolver_stedc	x	x	x	x

Table 5: Symmetric matrices

Function	single	double	single complex	double complex
rocsolver_lasyf	x	x	x	x

Table 6: Orthonormal matrices

Function	single	double	single complex	double complex
rocsolver_org2r	x	x		
rocsolver_orgqr	x	x		
rocsolver_orgl2	x	x		
rocsolver_orglq	x	x		
rocsolver_org2l	x	x		
rocsolver_orgql	x	x		
rocsolver_orgbr	x	x		
rocsolver_orgtr	x	x		
rocsolver_orm2r	x	x		
rocsolver_ormqr	x	x		
rocsolver_orml2	x	x		
rocsolver_ormlq	x	x		
rocsolver_orm2l	x	x		
rocsolver_ormql	x	x		
rocsolver_ormbr	x	x		
rocsolver_ormtr	x	x		

Table 7: Unitary matrices

Function	single	double	single complex	double complex
rocsolver_ung2r			x	x
rocsolver_ungqr			x	x
rocsolver_ungl2			x	x
rocsolver_unglq			x	x
rocsolver_ung2l			x	x
rocsolver_ungql			x	x
rocsolver_ungbr			x	x
rocsolver_ungtr			x	x
rocsolver_unm2r			x	x
rocsolver_unmqr			x	x
rocsolver_unml2			x	x
rocsolver_unmlq			x	x
rocsolver_unm2l			x	x
rocsolver_unmql			x	x
rocsolver_unmbr			x	x
rocsolver_unmtr			x	x

LAPACK main functions

Table 8: Triangular factorizations

Function	single	double	single complex	double complex
rocsolver_potf2	x	x	x	x
rocsolver_potrf	x	x	x	x
rocsolver_getf2	x	x	x	x
rocsolver_getrf	x	x	x	x
rocsolver_sytf2	x	x	x	x
rocsolver_sytrf	x	x	x	x

Table 9: Orthogonal factorizations

Function	single	double	single complex	double complex
rocsolver_geqr2	x	x	x	x
rocsolver_geqrf	x	x	x	x
rocsolver_gerq2	x	x	x	x
rocsolver_gerqf	x	x	x	x
rocsolver_gelq2	x	x	x	x
rocsolver_gelqf	x	x	x	x
rocsolver_geql2	x	x	x	x
rocsolver_geqlf	x	x	x	x

Table 10: Problem and matrix reductions

Function	single	double	single complex	double complex
rocsolver_sytd2	x	x		
rocsolver_sytrd	x	x		
rocsolver_sygs2	x	x		
rocsolver_sygst	x	x		
rocsolver_hetd2			x	x
rocsolver_hetrd			x	x
rocsolver_hegs2			x	x
rocsolver_hegst			x	x
rocsolver_gebd2	x	x	x	x
rocsolver_gebrd	x	x	x	x

Table 11: Linear-systems solvers

Function	single	double	single complex	double complex
rocsolver_trtri	x	x	x	x
rocsolver_getri	x	x	x	x
rocsolver_getrs	x	x	x	x
rocsolver_gesv	x	x	x	x
rocsolver_potri	x	x	x	x
rocsolver_potrs	x	x	x	x
rocsolver_posv	x	x	x	x

Table 12: Least-square solvers

Function	single	double	single complex	double complex
rocsolver_gels	x	x	x	x

Table 13: Symmetric eigensolvers

Function	single	double	single complex	double complex
rocsolver_syev	x	x		
rocsolver_syevd	x	x		
rocsolver_sygv	x	x		
rocsolver_sygvd	x	x		
rocsolver_heev			x	x
rocsolver_heevd			x	x
rocsolver_hegv			x	x
rocsolver_hegvd			x	x

Table 14: Singular value decomposition

Function	single	double	single complex	double complex
rocsolver_gesvd	x	x	x	x

LAPACK-like functions

Table 15: Triangular factorizations

Function	single	double	single complex	double complex
rocsolver_getf2_npvt	x	x	x	x
rocsolver_getrf_npvt	x	x	x	x

Table 16: Linear-systems solvers

Function	single	double	single complex	double complex
rocsolver_getri_npvt	x	x	x	x
rocsolver_getri_outofplace	x	x	x	x
rocsolver_getri_npvt_outofplace	x	x	x	x

1.2 Building and Installation

Table of contents

- *Prerequisites*
- *Installing from pre-built packages*
- *Building & installing from source*
  - *Using the install.sh script*
  - *Manual building and installation*

1.2.1 Prerequisites

rocSOLVER requires a ROCm-enabled platform. For more information, see the ROCm install guide.

rocSOLVER also requires a compatible version of rocBLAS installed on the system. For more information, see the rocBLAS install guide.

rocBLAS and rocSOLVER are both still under active development, and it is hard to define minimal compatibility versions. For now, a good rule of thumb is to always use rocSOLVER together with the matching rocBLAS version. For example, if you want to install rocSOLVER from the ROCm 3.3 release, then be sure that the ROCm 3.3 version of rocBLAS is also installed; if you are building the rocSOLVER branch tip, then you will need to build and install the rocBLAS branch tip as well.

1.2.2 Installing from pre-built packages

If you have added the ROCm repositories to your Linux distribution, the latest release version of rocSOLVER can be installed using a package manager. On Ubuntu, for example, use the commands:

```
sudo apt-get update
sudo apt-get install rocsolver
```

1.2.3 Building & installing from source

The rocSOLVER source code is hosted on GitHub. Download the code and checkout the desired branch using:

```
git clone -b <desired_branch_name> https://github.com/ROCmSoftwarePlatform/rocSOLVER.
↪git
cd rocSOLVER
```

To build from source, some external dependencies such as CMake and Python are required. Additionally, if the library clients are to be built (by default they are not), then LAPACK and GoogleTest will be also required. (The library clients, rocsolver-test and rocsolver-bench, provide the infrastructure for testing and benchmarking rocSOLVER. For more details see the *clients section* of this user's guide).

Using the install.sh script

It is recommended that the provided install.sh script be used to build and install rocSOLVER. The command

```
./install.sh --help
```

gives detailed information on how to use this installation script.

Next, some common use cases are listed:

```
./install.sh
```

This command builds rocSOLVER and puts the generated library files, such as headers and `librocsolver.so`, in the output directory: `rocSOLVER/build/release/rocsolver-install`. Other output files from the configuration and building process can also be found in the `rocSOLVER/build` and `rocSOLVER/build/release` directories. It is assumed that all external library dependencies have been installed. It also assumes that the rocBLAS library is located at `/opt/rocm/rocblas`.

```
./install.sh -g
```

Use the `-g` flag to build in debug mode. In this case the generated library files will be located at `rocSOLVER/build/debug/rocsolver-install`. Other output files from the configuration and building process can also be found in the `rocSOLVER/build` and `rocSOLVER/build/debug` directories.

```
./install.sh --lib_dir /home/user/rocsolverlib --build_dir buildoutput
```

Use `--lib_dir` and `--build_dir` to change output directories. In this case, for example, the installer will put the headers and library files in `/home/user/rocsolverlib`, while the outputs of the configuration and building processes will be in `rocSOLVER/buildoutput` and `rocSOLVER/buildoutput/release`. The selected output directories must be local, otherwise the user may require sudo privileges. To install rocSOLVER system-wide, we recommend the use of the `-i` flag as shown below.

```
./install.sh --rocblas_dir /alternative/rocblas/location
```

Use `--rocblas_dir` to change where the build system will search for the rocBLAS library. In this case, for example, the installer will look for the rocBLAS library at `/alternative/rocblas/location`.

```
./install.sh -s
```

With the `-s` flag, the installer will generate a static library (`librocsolver.a`) instead.

```
./install.sh -d
```

With the `-d` flag, the installer will first install all the external dependencies required by the rocSOLVER library in `/usr/local`. This flag only needs to be used once. For subsequent invocations of install.sh it is not necessary to rebuild the dependencies.

```
./install.sh -c
```

With the `-c` flag, the installer will additionally build the library clients `rocsolver-bench` and `rocsolver-test`. The binaries will be located at `rocSOLVER/build/release/clients/staging`. It is assumed that all external dependencies for the client have been installed.

```
./install.sh -dc
```

By combining the `-c` and `-d` flags, the installer will also install all the external dependencies required by rocSOLVER clients. Again, the `-d` flag only needs to be used once.

```
./install.sh -i
```

With the `-i` flag, the installer will additionally generate a pre-built rocSOLVER package and install it, using a suitable package manager, at the standard location: `/opt/rocm/rocsolver`. This is the preferred approach to install rocSOLVER on a system, as it will allow the library to be safely removed using the package manager.

```
./install.sh -p
```

With the `-p` flag, the installer will also generate the rocSOLVER package, but it will not be installed.

```
./install.sh -i --install_dir /package/install/path
```

When generating a package, use `--install_dir` to change the directory where it will be installed. In this case, for example, the rocSOLVER package will be installed at `/package/install/path`.

Manual building and installation

Manual installation of all the external dependencies is not an easy task. Get more information on how to install each dependency at the corresponding documentation sites:

- CMake (version 3.16 is recommended).
- LAPACK (which internally depends on a Fortran compiler), and
- GoogleTest
- fmt

Once all dependencies are installed (including ROCm and rocBLAS), rocSOLVER can be manually built using a combination of CMake and Make commands. Using CMake options can provide more flexibility in tailoring the building and installation process. Here we provide a list of examples of common use cases (see the CMake documentation for more information on CMake options).

```
mkdir -p build/release && cd build/release
CXX=/opt/rocm/bin/hipcc cmake -DCMAKE_INSTALL_PREFIX=rocsolver-install ../..
make install
```

This is equivalent to `./install.sh`.

```
mkdir -p buildoutput/release && cd buildoutput/release
CXX=/opt/rocm/bin/hipcc cmake -DCMAKE_INSTALL_PREFIX=/home/user/rocsolverlib ../..
make install
```

This is equivalent to `./install.sh --lib_dir /home/user/rocsolverlib --build_dir buildoutput`.

```
mkdir -p build/release && cd build/release
CXX=/opt/rocm/bin/hipcc cmake -DCMAKE_INSTALL_PREFIX=rocsolver-install -Drocblas_DIR=/
↪alternative/rocblas/location ../..
make install
```

This is equivalent to `./install.sh --rocblas_dir /alternative/rocblas/location`.

```
mkdir -p build/debug && cd build/debug
CXX=/opt/rocm/bin/hipcc cmake -DCMAKE_INSTALL_PREFIX=rocsolver-install -DCMAKE_BUILD_
↪TYPE=Debug ../..
make install
```

This is equivalent to `./install.sh -g`.

```
mkdir -p build/release && cd build/release
CXX=/opt/rocm/bin/hipcc cmake -DCMAKE_INSTALL_PREFIX=rocsolver-install -DBUILD_SHARED_
↪LIBS=OFF ../..
make install
```

This is equivalent to `./install.sh -s`.

```
mkdir -p build/release && cd build/release
CXX=/opt/rocm/bin/hipcc cmake -DCMAKE_INSTALL_PREFIX=rocsolver-install -DBUILD_
↪CLIENTS_TESTS=ON -DBUILD_CLIENTS_BENCHMARKS=ON ../..
make install
```

This is equivalent to `./install.sh -c`.

```
mkdir -p build/release && cd build/release
CXX=/opt/rocm/bin/hipcc cmake -DCMAKE_INSTALL_PREFIX=rocsolver-install -DCPACK_SET_
↪DESTDIR=OFF -DCPACK_PACKAGING_INSTALL_PREFIX=/opt/rocm ../..
make install
make package
```

This is equivalent to `./install.sh -p`.

```
mkdir -p build/release && cd build/release
CXX=/opt/rocm/bin/hipcc cmake -DCMAKE_INSTALL_PREFIX=rocsolver-install -DCPACK_SET_
↪DESTDIR=OFF -DCPACK_PACKAGING_INSTALL_PREFIX=/package/install/path ../..
make install
make package
sudo dpkg -i rocsolver[-\_]*.deb
```

On an Ubuntu system, for example, this would be equivalent to `./install.sh -i --install_dir /package/install/path`.

1.3 Using rocSOLVER

Once installed, rocSOLVER can be used just like any other library with a C API. The header file will need to be included in the user code, and both the rocBLAS and rocSOLVER shared libraries will become link-time and run-time dependencies for the user application.

Next, some examples are used to illustrate the basic use of rocSOLVER API and rocSOLVER batched API.

Table of contents

- *QR factorization of a single matrix*
- *QR factorization of a batch of matrices*
  - *Strided_batched version*
  - *Batched version*

1.3.1 QR factorization of a single matrix

The following code snippet uses rocSOLVER to compute the QR factorization of a general m-by-n real matrix in double precision. For a full description of the used rocSOLVER routine, see the API documentation here: *rocsolver_dgeqrf()*.

```
#include <hip/hip_runtime_api.h> // for hip functions
#include <rocsolver.h> // for all the rocsolver C interfaces and type declarations
#include <stdio.h>   // for printf
#include <stdlib.h>  // for malloc

// Example: Compute the QR Factorization of a matrix on the GPU

double *create_example_matrix(rocblas_int *M_out,
                              rocblas_int *N_out,
                              rocblas_int *lda_out) {
  // a *very* small example input; not a very efficient use of the API
  const double A[3][3] = { {  12, -51,   4},
                           {   6, 167, -68},
                           {  -4,  24, -41} };
  const rocblas_int M = 3;
  const rocblas_int N = 3;
  const rocblas_int lda = 3;
  *M_out = M;
  *N_out = N;
  *lda_out = lda;
  // note: rocsolver matrices must be stored in column major format,
  //       i.e. entry (i,j) should be accessed by hA[i + j*lda]
  double *hA = (double*)malloc(sizeof(double)*lda*N);
  for (size_t i = 0; i < M; ++i) {
    for (size_t j = 0; j < N; ++j) {
      // copy A (2D array) into hA (1D array, column-major)
      hA[i + j*lda] = A[i][j];
    }
  }
  return hA;
}
```

(continues on next page)

(continued from previous page)

```
// We use rocsolver_dgeqrf to factor a real M-by-N matrix, A.
// See https://rocsolver.readthedocs.io/en/latest/api_lapackfunc.html#c.rocsolver_
↪dgeqrf
// and https://www.netlib.org/lapack/explore-html/df/dc5/group__variants_g_
↪ecomputational_ga3766ea903391b5cf9008132f7440ec7b.html
int main() {
  rocblas_int M;          // rows
  rocblas_int N;          // cols
  rocblas_int lda;        // leading dimension
  double *hA = create_example_matrix(&M, &N, &lda); // input matrix on CPU

  // let's print the input matrix, just to see it
  printf("A = [\n");
  for (size_t i = 0; i < M; ++i) {
    printf("  ");
    for (size_t j = 0; j < N; ++j) {
      printf("% .3f ", hA[i + j*lda]);
    }
    printf(";\n");
  }
  printf("]\n");

  // initialization
  rocblas_handle handle;
  rocblas_create_handle(&handle);

  // Some rocsolver functions may trigger rocblas to load its GEMM kernels.
  // You can preload the kernels by explicitly invoking rocblas_initialize
  // (e.g., to exclude one-time initialization overhead from benchmarking).

  // preload rocBLAS GEMM kernels (optional)
  // rocblas_initialize();

  // calculate the sizes of our arrays
  size_t size_A = lda * (size_t)N;   // count of elements in matrix A
  size_t size_piv = (M < N) ? M : N; // count of Householder scalars

  // allocate memory on GPU
  double *dA, *dIpiv;
  hipMalloc((void**)&dA, sizeof(double)*size_A);
  hipMalloc((void**)&dIpiv, sizeof(double)*size_piv);

  // copy data to GPU
  hipMemcpy(dA, hA, sizeof(double)*size_A, hipMemcpyHostToDevice);

  // compute the QR factorization on the GPU
  rocsolver_dgeqrf(handle, M, N, dA, lda, dIpiv);

  // copy the results back to CPU
  double *hIpiv = (double*)malloc(sizeof(double)*size_piv); // householder scalars on␣
↪CPU
  hipMemcpy(hA, dA, sizeof(double)*size_A, hipMemcpyDeviceToHost);
  hipMemcpy(hIpiv, dIpiv, sizeof(double)*size_piv, hipMemcpyDeviceToHost);

  // the results are now in hA and hIpiv
  // we can print some of the results if we want to see them
```

(continues on next page)

(continued from previous page)

```
  printf("R = [\n");
  for (size_t i = 0; i < M; ++i) {
    printf("  ");
    for (size_t j = 0; j < N; ++j) {
      printf("% .3f ", (i <= j) ? hA[i + j*lda] : 0);
    }
    printf(";\n");
  }
  printf("]\n");

  // clean up
  free(hIpiv);
  hipFree(dA);
  hipFree(dIpiv);
  free(hA);
  rocblas_destroy_handle(handle);
}
```

The exact command used to compile the example above may vary depending on the system environment, but here is a typical example:

```
/opt/rocm/bin/hipcc -I/opt/rocm/include -c example.c
/opt/rocm/bin/hipcc -o example -L/opt/rocm/lib -lrocsolver -lrocblas example.o
```

1.3.2 QR factorization of a batch of matrices

One of the advantages of using GPUs is the ability to execute in parallel many operations of the same type but on different data sets. Based on this idea, rocSOLVER and rocBLAS provide a *batch* version of most of their routines. These batch versions allow the user to execute the same operation on a set of different matrices and/or vectors with a single library call. For more details on the approach to batch functionality followed in rocSOLVER, see *Batched rocSOLVER*.

Strided_batched version

The following code snippet uses rocSOLVER to compute the QR factorization of a series of general m-by-n real matrices in double precision. The matrices must be stored in contiguous memory locations on the GPU, and are accessed by a pointer to the first matrix and a stride value that gives the separation between one matrix and the next. For a full description of the used rocSOLVER routine, see the API documentation here: *rocsolver_dgeqrf_strided_batched()*.

```
#include <hip/hip_runtime_api.h> // for hip functions
#include <rocsolver.h> // for all the rocsolver C interfaces and type declarations
#include <stdio.h>   // for printf
#include <stdlib.h> // for malloc

// Example: Compute the QR Factorizations of an array of matrices on the GPU

double *create_example_matrices(rocblas_int *M_out,
                                rocblas_int *N_out,
                                rocblas_int *lda_out,
                                rocblas_stride *strideA_out,
                                rocblas_int *batch_count_out) {
  const double A[2][3][3] = {
    // First input matrix
```

(continues on next page)

(continued from previous page)

```
    { { 12, -51,   4},
      {  6, 167, -68},
      { -4,  24, -41} },

    // Second input matrix
    { {  3, -12,  11},
      {  4, -46,  -2},
      {  0,   5,  15} } };

  const rocblas_int M = 3;
  const rocblas_int N = 3;
  const rocblas_int lda = 3;
  const rocblas_stride strideA = lda * N;
  const rocblas_int batch_count = 2;
  *M_out = M;
  *N_out = N;
  *lda_out = lda;
  *strideA_out = strideA;
  *batch_count_out = batch_count;

  // allocate space for input matrix data on CPU
  double *hA = (double*)malloc(sizeof(double)*strideA*batch_count);

  // copy A (3D array) into hA (1D array, column-major)
  for (size_t b = 0; b < batch_count; ++b)
    for (size_t i = 0; i < M; ++i)
      for (size_t j = 0; j < N; ++j)
        hA[i + j*lda + b*strideA] = A[b][i][j];

  return hA;
}

// Use rocsolver_dgeqrf_strided_batched to factor an array of real M-by-N matrices.
int main() {
  rocblas_int M;          // rows
  rocblas_int N;          // cols
  rocblas_int lda;        // leading dimension
  rocblas_stride strideA; // stride from start of one matrix to the next
  rocblas_int batch_count; // number of matricies
  double *hA = create_example_matrices(&M, &N, &lda, &strideA, &batch_count);

  // print the input matrices
  for (size_t b = 0; b < batch_count; ++b) {
    printf("A[%zu] = [\n", b);
    for (size_t i = 0; i < M; ++i) {
      printf("  ");
      for (size_t j = 0; j < N; ++j) {
        printf("% 4.f ", hA[i + j*lda + strideA*b]);
      }
      printf(";\n");
    }
    printf("]\n");
  }

  // initialization
  rocblas_handle handle;
  rocblas_create_handle(&handle);
```

(continues on next page)

(continued from previous page)

```
  // preload rocBLAS GEMM kernels (optional)
  // rocblas_initialize();

  // calculate the sizes of our arrays
  size_t size_A = strideA * (size_t)batch_count;   // elements in array for matrices
  rocblas_stride strideP = (M < N) ? M : N;         // stride of Householder scalar␣
↪sets
  size_t size_piv = strideP * (size_t)batch_count; // elements in array for␣
↪Householder scalars

  // allocate memory on GPU
  double *dA, *dIpiv;
  hipMalloc((void**)&dA, sizeof(double)*size_A);
  hipMalloc((void**)&dIpiv, sizeof(double)*size_piv);

  // copy data to GPU
  hipMemcpy(dA, hA, sizeof(double)*size_A, hipMemcpyHostToDevice);

  // compute the QR factorizations on the GPU
  rocsolver_dgeqrf_strided_batched(handle, M, N, dA, lda, strideA, dIpiv, strideP,␣
↪batch_count);

  // copy the results back to CPU
  double *hIpiv = (double*)malloc(sizeof(double)*size_piv); // householder scalars on␣
↪CPU
  hipMemcpy(hA, dA, sizeof(double)*size_A, hipMemcpyDeviceToHost);
  hipMemcpy(hIpiv, dIpiv, sizeof(double)*size_piv, hipMemcpyDeviceToHost);

  // the results are now in hA and hIpiv
  // print some of the results
  for (size_t b = 0; b < batch_count; ++b) {
    printf("R[%zu] = [\n", b);
    for (size_t i = 0; i < M; ++i) {
      printf("  ");
      for (size_t j = 0; j < N; ++j) {
        printf("% 4.f ", (i <= j) ? hA[i + j*lda + strideA*b] : 0);
      }
      printf(";\n");
    }
    printf("]\n");
  }

  // clean up
  free(hIpiv);
  hipFree(dA);
  hipFree(dIpiv);
  free(hA);
  rocblas_destroy_handle(handle);
}
```

Batched version

The following code snippet uses rocSOLVER to compute the QR factorization of a series of general m-by-n real matrices in double precision. The matrices do not need to be in contiguous memory locations on the GPU, and will be accessed by an array of pointers. For a full description of the used rocSOLVER routine, see the API documentation here: *rocsolver_dgeqrf_batched*.

```
#include <hip/hip_runtime_api.h> // for hip functions
#include <rocsolver.h> // for all the rocsolver C interfaces and type declarations
#include <stdio.h> // for printf
#include <stdlib.h> // for malloc

// Example: Compute the QR Factorizations of a batch of matrices on the GPU

double **create_example_matrices(rocblas_int *M_out,
                                 rocblas_int *N_out,
                                 rocblas_int *lda_out,
                                 rocblas_int *batch_count_out) {
  // a small example input
  const double A[2][3][3] = {
    // First input matrix
    { { 12, -51,   4},
      {  6, 167, -68},
      { -4,  24, -41} },

    // Second input matrix
    { {  3, -12,  11},
      {  4, -46,  -2},
      {  0,   5,  15} } };

  const rocblas_int M = 3;
  const rocblas_int N = 3;
  const rocblas_int lda = 3;
  const rocblas_int batch_count = 2;
  *M_out = M;
  *N_out = N;
  *lda_out = lda;
  *batch_count_out = batch_count;

  // allocate space for input matrix data on CPU
  double **hA = (double**)malloc(sizeof(double*)*batch_count);
  hA[0] = (double*)malloc(sizeof(double)*lda*N);
  hA[1] = (double*)malloc(sizeof(double)*lda*N);

  for (size_t b = 0; b < batch_count; ++b)
    for (size_t i = 0; i < M; ++i)
      for (size_t j = 0; j < N; ++j)
        hA[b][i + j*lda] = A[b][i][j];

  return hA;
}

// Use rocsolver_dgeqrf_batched to factor a batch of real M-by-N matrices.
int main() {
  rocblas_int M;           // rows
  rocblas_int N;           // cols
  rocblas_int lda;         // leading dimension
  rocblas_int batch_count; // number of matricies
```

(continues on next page)

(continued from previous page)

```
double **hA = create_example_matrices(&M, &N, &lda, &batch_count);

// print the input matrices
for (size_t b = 0; b < batch_count; ++b) {
  printf("A[%zu] = [\n", b);
  for (size_t i = 0; i < M; ++i) {
    printf("  ");
    for (size_t j = 0; j < N; ++j) {
      printf("% 4.f ", hA[b][i + j*lda]);
    }
    printf(";\n");
  }
  printf("]\n");
}

// initialization
rocblas_handle handle;
rocblas_create_handle(&handle);

// preload rocBLAS GEMM kernels (optional)
// rocblas_initialize();

// calculate the sizes of the arrays
size_t size_A = lda * (size_t)N;          // count of elements in each matrix A
rocblas_stride strideP = (M < N) ? M : N; // stride of Householder scalar sets
size_t size_piv = strideP * (size_t)batch_count; // elements in array for␣
↪Householder scalars

// allocate memory on the CPU for an array of pointers,
// then allocate memory for each matrix on the GPU.
double **A = (double**)malloc(sizeof(double*)*batch_count);
for (rocblas_int b = 0; b < batch_count; ++b)
  hipMalloc((void**)&A[b], sizeof(double)*size_A);

// allocate memory on GPU for the array of pointers and Householder scalars
double **dA, *dIpiv;
hipMalloc((void**)&dA, sizeof(double*)*batch_count);
hipMalloc((void**)&dIpiv, sizeof(double)*size_piv);

// copy each matrix to the GPU
for (rocblas_int b = 0; b < batch_count; ++b)
  hipMemcpy(A[b], hA[b], sizeof(double)*size_A, hipMemcpyHostToDevice);

// copy the array of pointers to the GPU
hipMemcpy(dA, A, sizeof(double*)*batch_count, hipMemcpyHostToDevice);

// compute the QR factorizations on the GPU
rocsolver_dgeqrf_batched(handle, M, N, dA, lda, dIpiv, strideP, batch_count);

// copy the results back to CPU
double *hIpiv = (double*)malloc(sizeof(double)*size_piv); // householder scalars on␣
↪CPU
hipMemcpy(hIpiv, dIpiv, sizeof(double)*size_piv, hipMemcpyDeviceToHost);
for (rocblas_int b = 0; b < batch_count; ++b)
  hipMemcpy(hA[b], A[b], sizeof(double)*size_A, hipMemcpyDeviceToHost);

// the results are now in hA and hIpiv
```

(continues on next page)

(continued from previous page)

```
  // print some of the results
  for (size_t b = 0; b < batch_count; ++b) {
    printf("R[%zu] = [\n", b);
    for (size_t i = 0; i < M; ++i) {
      printf("  ");
      for (size_t j = 0; j < N; ++j) {
        printf("% 4.f ", (i <= j) ? hA[b][i + j*lda] : 0);
      }
      printf(";\n");
    }
    printf("]\n");
  }

  // clean up
  free(hIpiv);
  for (rocblas_int b = 0; b < batch_count; ++b)
    free(hA[b]);
  free(hA);
  for (rocblas_int b = 0; b < batch_count; ++b)
    hipFree(A[b]);
  free(A);
  hipFree(dA);
  hipFree(dIpiv);
  rocblas_destroy_handle(handle);
}
```

1.4 Memory Model

Almost all LAPACK and rocSOLVER routines require workspace memory in order to compute their results. In contrast to LAPACK, however, pointers to the workspace are not explicitly passed to rocSOLVER functions as arguments; instead, they are managed behind-the-scenes using a configurable device memory model.

rocSOLVER makes use of and is integrated with rocBLAS's memory model. Workspace memory, and the scheme used to manage it, is tracked on a per-`rocblas_handle` basis, and the same functionality that is used to manipulate rocBLAS's workspace memory can and will also affect rocSOLVER's workspace memory.

There are 4 schemes for device memory management:

- Automatic (managed by rocSOLVER/rocBLAS): The default scheme. Device memory persists between function calls and will be automatically reallocated if more memory is required by a function.
- User-managed (preallocated): The desired workspace size is specified by the user as an environment variable before handle creation, and cannot be altered after the handle is created.
- User-managed (manual): The desired workspace size can be manipulated using rocBLAS helper functions.
- User-owned: The user manually allocates device memory and calls a rocBLAS helper function to use it as the workspace.

Table of contents

- *Automatic workspace*
- *User-managed workspace*

  - *Minimum required size*
  - *Using an environment variable*
  - *Using helper functions*
- *User-owned workspace*

1.4.1 Automatic workspace

By default, rocSOLVER will automatically allocate device memory to be used as internal workspace using the rocBLAS memory model, and will increase the amount of allocated memory as needed by rocSOLVER functions. If this scheme is in use, the function `rocblas_is_managing_device_memory` will return `true`. In order to re-enable this scheme if it is not in use, a `nullptr` or zero size can be passed to the helper functions `rocblas_set_device_memory_size` or `rocblas_set_workspace`. For more details on these rocBLAS APIs, see the rocBLAS documentation.

This scheme has the disadvantage that automatic reallocation is synchronizing, and the user cannot control when this synchronization happens.

1.4.2 User-managed workspace

Alternatively, the user can manually specify an amount of memory to be allocated by rocSOLVER/rocBLAS. This allows the user to control when and if memory is reallocated and synchronization occurs. However, function calls will fail if there is not enough allocated memory.

Minimum required size

In order to choose an appropriate amount of memory to allocate, rocSOLVER can be queried to determine the minimum amount of memory required for functions to complete. The query can be started by calling `rocblas_start_device_memory_size_query`, followed by calls to the desired functions with appropriate problem sizes (a null pointer can be passed to the device pointer arguments). A final call to `rocblas_stop_device_memory_size_query` will return the minimum required size.

For example, the following code snippet will return the memory size required to solve a 1024*1024 linear system with 1 right-hand side (involving calls to `getrf` and `getrs`):

```
size_t memory_size;
rocblas_start_device_memory_size_query(handle);
rocsolver_dgetrf(handle, 1024, 1024, nullptr, lda, nullptr, nullptr);
rocsolver_dgetrs(handle, rocblas_operation_none, 1024, 1, nullptr, lda, nullptr,␣
↪nullptr, ldb);
rocblas_stop_device_memory_size_query(handle, &memory_size);
```

For more details on the rocBLAS APIs, see the rocBLAS documentation.

Using an environment variable

The desired workspace size can be provided before creation of the `rocblas_handle` by setting the value of environment variable `ROCBLAS_DEVICE_MEMORY_SIZE`. If this variable is unset or the value is == 0, then it will be ignored. Note that a workspace size set in this way cannot be changed once the handle has been created.

Using helper functions

Another way to set the desired workspace size is by using the helper function `rocblas_set_device_memory_size`. This function is called after handle creation and can be called multiple times; however, it is recommended to first synchronize the handle stream if a rocSOLVER or rocBLAS routine has already been called. For example:

```
hipStream_t stream;
rocblas_get_stream(handle, &stream);
hipStreamSynchronize(stream);

rocblas_set_device_memory_size(handle, memory_size);
```

For more details on the rocBLAS APIs, see the rocBLAS documentation.

1.4.3 User-owned workspace

Finally, the user may opt to manage the workspace memory manually using HIP. By calling the function `rocblas_set_workspace`, the user may pass a pointer to device memory to rocBLAS that will be used as the workspace for rocSOLVER. For example:

```
void* device_memory;
hipMalloc(&device_memory, memory_size);
rocblas_set_workspace(handle, device_memory, memory_size);

// perform computations here

rocblas_set_workspace(handle, nullptr, 0);
hipFree(device_memory);
```

For more details on the rocBLAS APIs, see the rocBLAS documentation.

1.5 Multi-level Logging

Similar to rocBLAS logging, rocSOLVER provides logging facilities that can be used to output information on rocSOLVER function calls. Three modes of logging are supported: trace logging, bench logging, and profile logging.

Note that performance will degrade when logging is enabled.

Table of contents

- *Logging modes*
  - *Trace logging*
  - *Bench logging*

  - *Profile logging*
- *Initialization and set-up*
- *Example code*
- *Kernel logging*
- *Multiple host threads*

1.5.1 Logging modes

Trace logging

Trace logging outputs a line each time an internal rocSOLVER or rocBLAS routine is called, outputting the function name and the values of its arguments (excluding stride arguments). The maximum depth of nested function calls that can appear in the log is specified by the user.

Bench logging

Bench logging outputs a line each time a public rocSOLVER routine is called (excluding auxiliary library functions), outputting a line that can be used with the executable `rocsolver-bench` to call the function with the same size arguments.

Profile logging

Profile logging, upon calling `rocsolver_log_write_profile` or `rocsolver_log_flush_profile`, or terminating the logging session using `rocsolver_log_end`, will output statistics on each called internal rocSOLVER and rocBLAS routine. These include the number of times each function was called, the total program runtime occupied by the function, and the total program runtime occupied by its nested function calls. As with trace logging, the maximum depth of nested output is specified by the user. Note that, when profile logging is enabled, the stream will be synchronized after every internal function call.

1.5.2 Initialization and set-up

In order to use rocSOLVER's logging facilities, the user must first call `rocsolver_log_begin` in order to allocate the internal data structures used for logging and begin the logging session. The user may then specify a layer mode and max level depth, either programmatically using `rocsolver_log_set_layer_mode`, `rocsolver_log_set_max_levels`, or by setting the corresponding environment variables.

The layer mode specifies which logging type(s) are activated, and can be `rocblas_layer_mode_none`, `rocblas_layer_mode_log_trace`, `rocblas_layer_mode_log_bench`, `rocblas_layer_mode_log_profile`, or a bitwise combination of these. The max level depth specifies the default maximum depth of nested function calls that may appear in the trace and profile logging.

Both the default layer mode and max level depth can be specified using environment variables.

- `ROCSOLVER_LAYER`
- `ROCSOLVER_LEVELS`

If these variables are not set, the layer mode will default to `rocblas_layer_mode_none` and the max level depth will default to 1. These defaults can be restored by calling the function `rocsolver_log_restore_defaults`.

`ROCSOLVER_LAYER` is a bitwise OR of zero or more bit masks as follows:

- If `ROCSOLVER_LAYER` is not set, then there is no logging
- If `(ROCSOLVER_LAYER & 1) != 0`, then there is trace logging
- If `(ROCSOLVER_LAYER & 2) != 0`, then there is bench logging
- If `(ROCSOLVER_LAYER & 4) != 0`, then there is profile logging

Three environment variables can set the full path name for a log file:

- `ROCSOLVER_LOG_TRACE_PATH` sets the full path name for trace logging
- `ROCSOLVER_LOG_BENCH_PATH` sets the full path name for bench logging
- `ROCSOLVER_LOG_PROFILE_PATH` sets the full path name for profile logging

If one of these environment variables is not set, then `ROCSOLVER_LOG_PATH` sets the full path for the corresponding logging, if it is set. If neither the above nor `ROCSOLVER_LOG_PATH` are set, then the corresponding logging output is streamed to standard error.

The results of profile logging, if enabled, can be printed using `rocsolver_log_write_profile` or `rocsolver_log_flush_profile`. Once logging facilities are no longer required (e.g. at program termination), the user must call `rocsolver_log_end` to free the data structures used for logging. If the profile log has not been flushed beforehand, then `rocsolver_log_end` will also output the results of profile logging.

For more details on the mentioned logging functions, see the *Logging functions section* on the rocSOLVER API document.

1.5.3 Example code

Code examples that illustrate the use of rocSOLVER's multi-level logging facilities can be found in this section or in the `example_logging.cpp` file in the `clients/samples` directory.

The following example shows some basic use: enabling trace and profile logging, and setting the max depth for their output.

```
// initialization
rocblas_handle handle;
rocblas_create_handle(&handle);
rocsolver_log_begin();

// begin trace logging and profile logging (max depth = 5)
rocsolver_log_set_layer_mode(rocblas_layer_mode_log_trace | rocblas_layer_mode_log_
↪profile);
rocsolver_log_set_max_levels(5);

// call rocSOLVER functions...

// terminate logging and print profile results
rocsolver_log_flush_profile();
rocsolver_log_end();
rocblas_destroy_handle(handle);
```

Alternatively, users may control which logging modes are enabled by using environment variables. The benefit of this approach is that the program does not need to be recompiled if a different logging environment is desired. This requires that `rocsolver_log_set_layer_mode` and `rocsolver_log_set_max_levels` are not called in the code, e.g.

```
// initialization
rocblas_handle handle;
rocblas_create_handle(&handle);
rocsolver_log_begin();

// call rocSOLVER functions...

// termination
rocsolver_log_end();
rocblas_destroy_handle(handle);
```

The user may then set the desired logging modes and max depth on the command line as follows:

```
export ROCSOLVER_LAYER=5
export ROCSOLVER_LEVELS=5
```

1.5.4 Kernel logging

Kernel launches from within rocSOLVER can be added to the trace and profile logs using an additional layer mode flag. The flag `rocblas_layer_mode_ex_log_kernel` can be combined with `rocblas_layer_mode` flags and passed to `rocsolver_log_set_layer_mode` in order to enable kernel logging. Alternatively, the environment variable `ROCSOLVER_LAYER` can be set such that `(ROCSOLVER_LAYER & 16) != 0`:

- If `(ROCSOLVER_LAYER & 17) != 0`, then kernel calls will be added to the trace log
- If `(ROCSOLVER_LAYER & 20) != 0`, then kernel calls will be added to the profile log

1.5.5 Multiple host threads

The logging facilities for rocSOLVER assume that each `rocblas_handle` is associated with at most one host thread. When using rocSOLVER's multi-level logging setup, it is recommended to create a separate `rocblas_handle` for each host thread.

The rocsolver_log_* functions are not thread-safe. Calling a log function while any rocSOLVER routine is executing on another host thread will result in undefined behaviour. Once enabled, logging data collection is thread-safe. However, note that trace logging will likely result in garbled trace trees if rocSOLVER routines are called from multiple host threads.

1.6 Clients

rocSOLVER has an infrastructure for testing and benchmarking similar to that of rocBLAS, as well as sample code illustrating basic use of the library.

Client binaries are not built by default; they require specific flags to be passed to the install script or CMake system. If the `-c` flag is passed to `install.sh`, the client binaries will be located in the directory `<rocsolverDIR>/build/release/clients/staging`. If both the `-c` and `-g` flags are passed to `install.sh`, the client binaries will be located in `<rocsolverDIR>/build/debug/clients/staging`. If the `-DBUILD_CLIENTS_TESTS=ON` flag, the `-DBUILD_CLIENTS_BENCHMARKS=ON` flag, and/or the `-DBUILD_CLIENTS_SAMPLES=ON` flag are passed to the CMake system, the relevant client binaries will normally be located in the directory `<rocsolverDIR>/build/clients/staging`. See the *Building and installation section* of the User Guide for more information on building the library and its clients.

Table of contents

- *Testing rocSOLVER*
- *Benchmarking rocSOLVER*
- *rocSOLVER sample code*

1.6.1 Testing rocSOLVER

The `rocsolver-test` client executes a suite of Google tests (*gtest*) that verifies the correct functioning of the library. The results computed by rocSOLVER, given random input data, are normally compared with the results computed by NETLib LAPACK on the CPU, or tested implicitly in the context of the solved problem. It will be built if the `-c` flag is passed to `install.sh` or if the `-DBUILD_CLIENTS_TESTS=ON` flag is passed to the CMake system.

Calling the rocSOLVER gtest client with the `--help` flag

```
./rocsolver-test --help
```

returns information on different flags that control the behavior of the gtests.

One of the most useful flags is the `--gtest_filter` flag, which allows the user to choose which tests to run from the suite. For example, the following command will run the tests for only geqrf:

```
./rocsolver-test --gtest_filter=*GEQRF*
```

Note that rocSOLVER's tests are divided into two separate groupings: `checkin_lapack` and `daily_lapack`. Tests in the `checkin_lapack` group are small and quick to execute, and verify basic correctness and error handling. Tests in the `daily_lapack` group are large and slower to execute, and verify correctness of large problem sizes. Users may run one test group or the other using `--gtest_filter`, e.g.

```
./rocsolver-test --gtest_filter=*checkin_lapack*
./rocsolver-test --gtest_filter=*daily_lapack*
```

1.6.2 Benchmarking rocSOLVER

The `rocsolver-bench` client runs any rocSOLVER function with random data of the specified dimensions. It compares basic performance information (i.e. execution times) between NETLib LAPACK on the CPU and rocSOLVER on the GPU. It will be built if the `-c` flag is passed to `install.sh` or if the `-DBUILD_CLIENTS_BENCHMARKS=ON` flag is passed to the CMake system.

Calling the rocSOLVER bench client with the `--help` flag

```
./rocsolver-bench --help
```

returns information on the different parameters and flags that control the behavior of the benchmark client.

Two of the most important flags for `rocsolver-bench` are the `-f` and `-r` flags. The `-f` (or `--function`) flag allows the user to select which function to benchmark. The `-r` (or `--precision`) flag allows the user to select the data precision for the function, and can be one of s (single precision), d (double precision), c (single precision complex), or z (double precision complex).

The non-pointer arguments for a function can be passed to `rocsolver-bench` by using the argument name as a flag (see the *rocSOLVER API* document for information on the function arguments and their names). For example, the function `rocsolver_dgeqrf_strided_batched` has the following method signature:

```
rocblas_status
rocsolver_dgeqrf_strided_batched(rocblas_handle handle,
                                 const rocblas_int m,
                                 const rocblas_int n,
                                 double* A,
                                 const rocblas_int lda,
                                 const rocblas_stride strideA,
                                 double* ipiv,
                                 const rocblas_stride strideP,
                                 const rocblas_int batch_count);
```

A call to `rocsolver-bench` that runs this function on a batch of one hundred 30x30 matrices could look like this:

```
./rocsolver-bench -f geqrf_strided_batched -r d -m 30 -n 30 --lda 30 --strideA 900 --
↪strideP 30 --batch_count 100
```

Generally, `rocsolver-bench` will attempt to provide or calculate a suitable default value for these arguments, though at least one size argument must always be specified by the user. Functions that take m and n as arguments typically require m to be provided, and a square matrix will be assumed. For example, the previous command is equivalent to:

```
./rocsolver-bench -f geqrf_strided_batched -r d -m 30 --batch_count 100
```

Other useful benchmarking options include the `--perf` flag, which will disable the LAPACK computation and only time and print the rocSOLVER performance result; the `-i` (or `--iters`) flag, which indicates the number of times to run the GPU timing loop (the performance result would be the average of all the runs); and the `--profile` flag, which enables *profile logging* indicating the maximum depth of the nested output.

```
./rocsolver-bench -f geqrf_strided_batched -r d -m 30 --batch_count 100 --perf 1
./rocsolver-bench -f geqrf_strided_batched -r d -m 30 --batch_count 100 --iters 20
./rocsolver-bench -f geqrf_strided_batched -r d -m 30 --batch_count 100 --profile 5
```

In addition to the benchmarking functionality, the rocSOLVER bench client can also provide the norm of the error in the computations when the `-v` (or `--verify`) flag is used; and return the amount of device memory required as workspace for the given function, if the `--mem_query` flag is passed.

```
./rocsolver-bench -f geqrf_strided_batched -r d -m 30 --batch_count 100 --verify 1
./rocsolver-bench -f geqrf_strided_batched -r d -m 30 --batch_count 100 --mem_query 1
```

1.6.3 rocSOLVER sample code

rocSOLVER's sample programs provide illustrative examples of how to work with the rocSOLVER library. They will be built if the `-c` flag is passed to `install.sh` or if the `-DBUILD_CLIENTS_SAMPLES=ON` flag is passed to the CMake system.

Currently, sample code exists to demonstrate the following:

- Basic use of rocSOLVER in C, C++, and Fortran, using the example of *rocsolver_geqrf*;
- Use of batched and strided_batched functions, using *rocsolver_geqrf_batched* and *rocsolver_geqrf_strided_batched* as examples;
- Use of rocSOLVER with the Heterogeneous Memory Management (HMM) model; and
- Use of rocSOLVER's *multi-level logging* functionality.

CHAPTER

TWO

ROCSOLVER LIBRARY DESIGN GUIDE

2.1 Introduction

More to come later. . .

2.2 Batched rocSOLVER

More to come later. . .

2.3 Tuning rocSOLVER Performance

Some compile-time parameters in rocSOLVER can be modified to tune the performance of the library functions in a given context (e.g., for a particular matrix size or shape). A description of these tunable constants is presented in this section.

To facilitate the description, the constants are grouped by the family of functions they affect. Some aspects of the involved algorithms are also depicted here for the sake of clarity; however, this section is not intended to be a review of the well-known methods for different matrix computations. These constants are specific to the rocSOLVER implementation and are only described within that context.

All described constants can be found in `library/src/include/ideal_sizes.hpp`. These are not run-time arguments for the associated API functions. The library must be *rebuilt from source* for any change to take effect.

> **Warning:** The effect of changing a tunable constant on the performance of the library is difficult to predict, and such analysis is beyond the scope of this document. Advanced users and developers tuning these values should proceed with caution. New values may (or may not) improve or worsen the performance of the associated functions.

Table of contents

- *geqr2/geqrf and geql2/geqlf functions*
  - *GEQxF_BLOCKSIZE*
  - *GEQxF_GEQx2_SWITCHSIZE*
- *gerq2/gerqf and gelq2/gelqf functions*

  - *GExQF_BLOCKSIZE*
  - *GExQF_GExQ2_SWITCHSIZE*
- *org2r/orgqr, org2l/orgql, ung2r/ungqr and ung2l/ungql functions*
  - *xxGQx_BLOCKSIZE*
  - *xxGQx_xxGQx2_SWITCHSIZE*
- *orgr2/orgrq, orgl2/orglq, ungr2/ungrq and ungl2/unglq functions*
  - *xxGxQ_BLOCKSIZE*
  - *xxGxQ_xxGxQ2_SWITCHSIZE*
- *orm2r/ormqr, orm2l/ormql, unm2r/unmqr and unm2l/unmql functions*
  - *xxMQx_BLOCKSIZE*
- *ormr2/ormrq, orml2/ormlq, unmr2/unmrq and unml2/unmlq functions*
  - *xxMxQ_BLOCKSIZE*
- *gebd2/gebrd and labrd functions*
  - *GEBRD_BLOCKSIZE*
  - *GEBRD_GEBD2_SWITCHSIZE*
- *gesvd function*
  - *THIN_SVD_SWITCH*
- *sytd2/sytrd, hetd2/hetrd and latrd functions*
  - *xxTRD_BLOCKSIZE*
  - *xxTRD_xxTD2_SWITCHSIZE*
- *sygs2/sygst and hegs2/hegst functions*
  - *xxGST_BLOCKSIZE*
- *syevd, heevd and stedc functions*
  - *STEDC_MIN_DC_SIZE*
- *potf2/potrf functions*
  - *POTRF_BLOCKSIZE*
  - *POTRF_POTF2_SWITCHSIZE*
- *sytf2/sytrf and lasyf functions*
  - *SYTRF_BLOCKSIZE*
  - *SYTRF_SYTF2_SWITCHSIZE*
- *getf2/getrf functions*
  - *GETF2_MAX_COLS*
  - *GETF2_MAX_THDS*
  - *GETF2_OPTIM_NGRP*
  - *GETRF_NUM_INTERVALS*

    - *GETRF_INTERVALS*
    - *GETRF_BLKSIZES*
    - *GETRF_BATCH_NUM_INTERVALS*
    - *GETRF_BATCH_INTERVALS*
    - *GETRF_BATCH_BLKSIZES*
    - *GETRF_NPVT_NUM_INTERVALS*
    - *GETRF_NPVT_INTERVALS*
    - *GETRF_NPVT_BLKSIZES*
    - *GETRF_NPVT_BATCH_NUM_INTERVALS*
    - *GETRF_NPVT_BATCH_INTERVALS*
    - *GETRF_NPVT_BATCH_BLKSIZES*
- *getri function*
    - *GETRI_MAX_COLS*
    - *GETRI_TINY_SIZE*
    - *GETRI_NUM_INTERVALS*
    - *GETRI_INTERVALS*
    - *GETRI_BLKSIZES*
    - *GETRI_BATCH_TINY_SIZE*
    - *GETRI_BATCH_NUM_INTERVALS*
    - *GETRI_BATCH_INTERVALS*
    - *GETRI_BATCH_BLKSIZES*
- *trtri function*
    - *TRTRI_MAX_COLS*
    - *TRTRI_NUM_INTERVALS*
    - *TRTRI_INTERVALS*
    - *TRTRI_BLKSIZES*
    - *TRTRI_BATCH_NUM_INTERVALS*
    - *TRTRI_BATCH_INTERVALS*
    - *TRTRI_BATCH_BLKSIZES*

2.3.1 geqr2/geqrf and geql2/geqlf functions

The orthogonal factorizations from the left (QR or QL factorizations) are separated into two versions: blocked and unblocked. The unblocked routines GEQR2 and GEQL2 are based on BLAS Level 2 operations and work by applying Householder reflectors one column at a time. The blocked routines GEQRF and GEQLF factorize a block of columns at each step using the unblocked functions (provided the matrix is large enough) and apply the resulting block reflectors to update the rest of the matrix. The application of the block reflectors is based on matrix-matrix operations (BLAS Level 3), which, in general, can give better performance on the GPU.

GEQxF_BLOCKSIZE

GEQxF_BLOCKSIZE

Determines the size of the block column factorized at each step in the blocked QR or QL algorithm (GEQRF or GEQLF). It also applies to the corresponding batched and strided-batched routines.

GEQxF_GEQx2_SWITCHSIZE

GEQxF_GEQx2_SWITCHSIZE

Determines the size at which rocSOLVER switches from the unblocked to the blocked algorithm when executing GEQRF or GEQLF. It also applies to the corresponding batched and strided-batched routines.

GEQRF or GEQLF will factorize blocks of GEQxF_BLOCKSIZE columns at a time until the rest of the matrix has no more than GEQxF_GEQx2_SWITCHSIZE rows or columns; at this point the last block, if any, will be factorized with the unblocked algorithm (GEQR2 or GEQL2).

(As of the current rocSOLVER release, these constants have not been tuned for any specific cases.)

2.3.2 gerq2/gerqf and gelq2/gelqf functions

The orthogonal factorizations from the right (RQ or LQ factorizations) are separated into two versions: blocked and unblocked. The unblocked routines GERQ2 and GELQ2 are based on BLAS Level 2 operations and work by applying Householder reflectors one row at a time. The blocked routines GERQF and GELQF factorize a block of rows at each step using the unblocked functions (provided the matrix is large enough) and apply the resulting block reflectors to update the rest of the matrix. The application of the block reflectors is based on matrix-matrix operations (BLAS Level 3), which, in general, can give better performance on the GPU.

GExQF_BLOCKSIZE

GExQF_BLOCKSIZE

Determines the size of the block row factorized at each step in the blocked RQ or LQ algorithm (GERQF or GELQF). It also applies to the corresponding batched and strided-batched routines.

GExQF_GExQ2_SWITCHSIZE

GExQF_GExQ2_SWITCHSIZE

Determines the size at which rocSOLVER switches from the unblocked to the blocked algorithm when executing GERQF or GELQF. It also applies to the corresponding batched and strided-batched routines.

GERQF or GELQF will factorize blocks of GExQF_BLOCKSIZE rows at a time until the rest of the matrix has no more than GExQF_GExQ2_SWITCHSIZE rows or columns; at this point the last block, if any, will be factorized with the unblocked algorithm (GERQ2 or GELQ2).

(As of the current rocSOLVER release, these constants have not been tuned for any specific cases.)

2.3.3 org2r/orgqr, org2l/orgql, ung2r/ungqr and ung2l/ungql functions

The generators of a matrix Q with orthonormal columns (as products of Householder reflectors derived from the QR or QL factorizations) are also separated into blocked and unblocked versions. The unblocked routines ORG2R/UNG2R and ORG2L/UNG2L, based on BLAS Level 2 operations, work by accumulating one Householder reflector at a time. The blocked routines ORGQR/UNGQR and ORGQL/UNGQL multiply a set of reflectors at each step using the unblocked functions (provided there are enough reflectors to accumulate) and apply the resulting block reflector to update Q. The application of the block reflectors is based on matrix-matrix operations (BLAS Level 3), which, in general, can give better performance on the GPU.

xxGQx_BLOCKSIZE

xxGQx_BLOCKSIZE

Determines the size of the block reflector that is applied at each step when generating a matrix Q with orthonormal columns with the blocked algorithm (ORGQR/UNGQR or ORGQL/UNGQL).

xxGQx_xxGQx2_SWITCHSIZE

xxGQx_xxGQx2_SWITCHSIZE

Determines the size at which rocSOLVER switches from the unblocked to the blocked algorithm when executing ORGQR/UNGQR or ORGQL/UNGQL.

ORGQR/UNGQR or ORGQL/UNGQL will accumulate xxGQx_BLOCKSIZE reflectors at a time until there are no more than xxGQx_xxGQx2_SWITCHSIZE reflectors left; the remaining reflectors, if any, are applied one by one using the unblocked algorithm (ORG2R/UNG2R or ORG2L/UNG2L).

(As of the current rocSOLVER release, these constants have not been tuned for any specific cases.)

2.3.4 orgr2/orgrq, orgl2/orglq, ungr2/ungrq and ungl2/unglq functions

The generators of a matrix Q with orthonormal rows (as products of Householder reflectors derived from the RQ or LQ factorizations) are also separated into blocked and unblocked versions. The unblocked routines ORGR2/UNGR2 and ORGL2/UNGL2, based on BLAS Level 2 operations, work by accumulating one Householder reflector at a time. The blocked routines ORGRQ/UNGRQ and ORGLQ/UNGLQ multiply a set of reflectors at each step using the unblocked functions (provided there are enough reflectors to accumulate) and apply the resulting block reflector to update Q. The application of the block reflectors is based on matrix-matrix operations (BLAS Level 3), which, in general, can give better performance on the GPU.

xxGxQ_BLOCKSIZE

xxGxQ_BLOCKSIZE
Determines the size of the block reflector that is applied at each step when generating a matrix Q with orthonormal rows with the blocked algorithm (ORGRQ/UNGRQ or ORGLQ/UNGLQ).

xxGxQ_xxGxQ2_SWITCHSIZE

xxGxQ_xxGxQ2_SWITCHSIZE
Determines the size at which rocSOLVER switches from the unblocked to the blocked algorithm when executing ORGRQ/UNGRQ or ORGLQ/UNGLQ.

ORGRQ/UNGRQ or ORGLQ/UNGLQ will accumulate xxGxQ_BLOCKSIZE reflectors at a time until there are no more than xxGxQ_xxGxQ2_SWITCHSIZE reflectors left; the remaining reflectors, if any, are applied one by one using the unblocked algorithm (ORGR2/UNGR2 or ORGL2/UNGL2).

(As of the current rocSOLVER release, these constants have not been tuned for any specific cases.)

2.3.5 orm2r/ormqr, orm2l/ormql, unm2r/unmqr and unm2l/unmql functions

As with the generators of orthonormal/unitary matrices, the routines to multiply a general matrix C by a matrix Q with orthonormal columns are separated into blocked and unblocked versions. The unblocked routines ORM2R/UNM2R and ORM2L/UNM2L, based on BLAS Level 2 operations, work by multiplying one Householder reflector at a time, while the blocked routines ORMQR/UNMQR and ORMQL/UNMQL apply a set of reflectors at each step (provided there are enough reflectors to start with). The application of the block reflectors is based on matrix-matrix operations (BLAS Level 3), which, in general, can give better performance on the GPU.

xxMQx_BLOCKSIZE

xxMQx_BLOCKSIZE
Determines the size of the block reflector that multiplies the matrix C at each step with the blocked algorithm (ORMQR/UNMQR or ORMQL/UNMQL).

xxMQx_BLOCKSIZE also acts as a switch size; if the total number of reflectors is not greater than xxMQx_BLOCKSIZE (k <= xxMQx_BLOCKSIZE), ORMQR/UNMQR or ORMQL/UNMQL will directly call the unblocked routines (ORM2R/UNM2R or ORM2L/UNM2L). However, when k is not a multiple of xxMQx_BLOCKSIZE, the last block that updates C in the blocked process is allowed to be smaller than xxMQx_BLOCKSIZE.

(As of the current rocSOLVER release, this constant has not been tuned for any specific cases.)

2.3.6 ormr2/ormrq, orml2/ormlq, unmr2/unmrq and unml2/unmlq functions

As with the generators of orthonormal/unitary matrices, the routines to multiply a general matrix C by a matrix Q with orthonormal rows are separated into blocked and unblocked versions. The unblocked routines ORMR2/UNMR2 and ORML2/UNML2, based on BLAS Level 2 operations, work by multiplying one Householder reflector at a time, while the blocked routines ORMRQ/UNMRQ and ORMLQ/UNMLQ apply a set of reflectors at each step (provided there are enough reflectors to start with). The application of the block reflectors is based on matrix-matrix operations (BLAS Level 3), which, in general, can give better performance on the GPU.

xxMxQ_BLOCKSIZE

xxMxQ_BLOCKSIZE

Determines the size of the block reflector that multiplies the matrix C at each step with the blocked algorithm (ORMRQ/UNMRQ or ORMLQ/UNMLQ).

xxMxQ_BLOCKSIZE also acts as a switch size; if the total number of reflectors is not greater than xxMxQ_BLOCKSIZE (k <= xxMxQ_BLOCKSIZE), ORMRQ/UNMRQ or ORMLQ/UNMLQ will directly call the unblocked routines (ORMR2/UNMR2 or ORML2/UNML2). However, when k is not a multiple of xxMxQ_BLOCKSIZE, the last block that updates C in the blocked process is allowed to be smaller than xxMxQ_BLOCKSIZE.

(As of the current rocSOLVER release, this constant has not been tuned for any specific cases.)

2.3.7 gebd2/gebrd and labrd functions

The computation of the bidiagonal form of a matrix is separated into blocked and unblocked versions. The unblocked routine GEBD2 (and the auxiliary LABRD), based on BLAS Level 2 operations, apply Householder reflections to one column and row at a time. The blocked routine GEBRD reduces a leading block of rows and columns at each step using the unblocked function LABRD (provided the matrix is large enough), and applies the resulting block reflectors to update the trailing submatrix. The application of the block reflectors is based on matrix-matrix operations (BLAS Level 3), which, in general, can give better performance on the GPU.

GEBRD_BLOCKSIZE

GEBRD_BLOCKSIZE

Determines the size of the leading block that is reduced to bidiagonal form at each step when using the blocked algorithm (GEBRD). It also applies to the corresponding batched and strided-batched routines.

GEBRD_GEBD2_SWITCHSIZE

GEBRD_GEBD2_SWITCHSIZE

Determines the size at which rocSOLVER switches from the unblocked to the blocked algorithm when executing GEBRD. It also applies to the corresponding batched and strided-batched routines.

GEBRD will use LABRD to reduce blocks of GEBRD_BLOCKSIZE rows and columns at a time until the trailing submatrix has no more than GEBRD_GEBD2_SWITCHSIZE rows or columns; at this point the last block, if any, will be reduced with the unblocked algorithm (GEBD2).

(As of the current rocSOLVER release, these constants have not been tuned for any specific cases.)

2.3.8 gesvd function

The Singular Value Decomposition of a matrix A could be sped up for matrices with sufficiently many more rows than columns (or columns than rows) by starting with a QR factorization (or LQ factorization) of A and working with the triangular factor afterwards.

THIN_SVD_SWITCH

THIN_SVD_SWITCH

Determines the factor by which one dimension of a matrix should exceed the other dimension for the thin SVD to be computed when executing GESVD. It also applies to the corresponding batched and strided-batched routines.

When a m-by-n matrix A is passed to GESVD, if m >= THIN_SVD_SWITCH*n or n >= THIN_SVD_SWITCH*m, then the thin SVD is computed.

(As of the current rocSOLVER release, this constant has not been tuned for any specific cases.)

2.3.9 sytd2/sytrd, hetd2/hetrd and latrd functions

The computation of the tridiagonal form of a symmetric/Hermitian matrix is separated into blocked and unblocked versions. The unblocked routines SYTD2/HETD2 (and the auxiliary LATRD), based on BLAS Level 2 operations, apply Householder reflections to one column/row at a time. The blocked routine SYTRD reduces a block of rows and columns at each step using the unblocked function LATRD (provided the matrix is large enough) and applies the resulting block reflector to update the rest of the matrix. The application of the block reflectors is based on matrix-matrix operations (BLAS Level 3), which, in general, can give better performance on the GPU.

xxTRD_BLOCKSIZE

xxTRD_BLOCKSIZE

Determines the size of the leading block that is reduced to tridiagonal form at each step when using the blocked algorithm (SYTRD/HETRD). It also applies to the corresponding batched and strided-batched routines.

xxTRD_xxTD2_SWITCHSIZE

xxTRD_xxTD2_SWITCHSIZE

Determines the size at which rocSOLVER switches from the unblocked to the blocked algorithm when executing SYTRD/HETRD. It also applies to the corresponding batched and strided-batched routines.

SYTRD/HETRD will use LATRD to reduce blocks of xxTRD_BLOCKSIZE rows and columns at a time until the rest of the matrix has no more than xxTRD_xxTD2_SWITCHSIZE rows or columns; at this point the last block, if any, will be reduced with the unblocked algorithm (SYTD2/HETD2).

(As of the current rocSOLVER release, these constants have not been tuned for any specific cases.)

2.3.10 sygs2/sygst and hegs2/hegst functions

The reduction of a symmetric/Hermitian-definite generalized eigenproblem to standard form is separated into blocked and unblocked versions. The unblocked routines SYGS2/HEGS2 reduce the matrix A one column/row at a time with vector operations and rank-2 updates (BLAS Level 2). The blocked routines SYGST/HEGST reduce a leading block of A at each step using the unblocked methods (provided A is large enough) and update the trailing matrix with BLAS Level 3 operations (matrix products and rank-2k updates), which, in general, can give better performance on the GPU.

xxGST_BLOCKSIZE

xxGST_BLOCKSIZE

Determines the size of the leading block that is reduced to standard form at each step when using the blocked algorithm (SYGST/HEGST). It also applies to the corresponding batched and strided-batched routines.

xxGST_BLOCKSIZE also acts as a switch size; if the original size of the problem is not larger than xxGST_BLOCKSIZE (n <= xxGST_BLOCKSIZE), SYGST/HEGST will directly call the unblocked routines (SYGS2/HEGS2). However, when n is not a multiple of xxGST_BLOCKSIZE, the last block reduced in the blocked process is allowed to be smaller than xxGST_BLOCKSIZE.

(As of the current rocSOLVER release, this constant has not been tuned for any specific cases.)

2.3.11 syevd, heevd and stedc functions

When running SYEVD/HEEVD (or the corresponding batched and strided-batched routines), the computation of the eigenvectors of the associated tridiagonal matrix can be sped up using a divide-and-conquer approach (implemented in STEDC), provided the size of the independent block is large enough.

STEDC_MIN_DC_SIZE

STEDC_MIN_DC_SIZE

Determines the minimum size required for the eigenvectors of an independent block of a tridiagonal matrix to be computed using the divide-and-conquer algorithm (STEDC).

If the size of the block is not greater than STEDC_MIN_DC_SIZE (bs <= STEDC_MIN_DC_SIZE), the eigenvectors are computed with the normal QR algorithm.

(As of the current rocSOLVER release, this constant has not been tuned for any specific cases.)

2.3.12 potf2/potrf functions

The Cholesky factorization is separated into blocked (right-looking) and unblocked versions. The unblocked routine POTF2, based on BLAS Level 2 operations, computes one diagonal element at a time and scales the corresponding row/column. The blocked routine POTRF factorizes a leading block of rows/columns at each step using the unblocked algorithm (provided the matrix is large enough) and updates the trailing matrix with BLAS Level 3 operations (symmetric rank-k updates), which, in general, can give better performance on the GPU.

POTRF_BLOCKSIZE

POTRF_BLOCKSIZE

Determines the size of the leading block that is factorized at each step when using the blocked algorithm (POTRF). It also applies to the corresponding batched and strided-batched routines.

POTRF_POTF2_SWITCHSIZE

POTRF_POTF2_SWITCHSIZE

Determines the size at which rocSOLVER switches from the unblocked to the blocked algorithm when executing POTRF. It also applies to the corresponding batched and strided-batched routines.

POTRF will factorize blocks of POTRF_BLOCKSIZE columns at a time until the rest of the matrix has no more than POTRF_POTF2_SWITCHSIZE columns; at this point the last block, if any, will be factorized with the unblocked algorithm (POTF2).

(As of the current rocSOLVER release, these constants have not been tuned for any specific cases.)

2.3.13 sytf2/sytrf and lasyf functions

The Bunch-Kaufman factorization is separated into blocked and unblocked versions. The unblocked routine SYTF2 generates one 1-by-1 or 2-by-2 diagonal block at a time and applies a rank-1 update. The blocked routine SYTRF executes a partial factorization of a given maximum number of diagonal elements (LASYF) at each step (provided the matrix is large enough), and updates the rest of the matrix with matrix-matrix operations (BLAS Level 3), which, in general, can give better performance on the GPU.

SYTRF_BLOCKSIZE

SYTRF_BLOCKSIZE

Determines the maximum size of the partial factorization executed at each step when using the blocked algorithm (SYTRF). It also applies to the corresponding batched and strided-batched routines.

SYTRF_SYTF2_SWITCHSIZE

SYTRF_SYTF2_SWITCHSIZE

Determines the size at which rocSOLVER switches from the unblocked to the blocked algorithm when executing SYTRF. It also applies to the corresponding batched and strided-batched routines.

SYTRF will use LASYF to factorize a submatrix of at most SYTRF_BLOCKSIZE columns at a time until the rest of the matrix has no more than SYTRF_SYTF2_SWITCHSIZE columns; at this point the last block, if any, will be factorized with the unblocked algorithm (SYTF2).

(As of the current rocSOLVER release, these constants have not been tuned for any specific cases.)

2.3.14 getf2/getrf functions

GETF2_MAX_COLS

GETF2_MAX_THDS

GETF2_OPTIM_NGRP

GETRF_NUM_INTERVALS

GETRF_INTERVALS

GETRF_BLKSIZES

GETRF_BATCH_NUM_INTERVALS

GETRF_BATCH_INTERVALS

GETRF_BATCH_BLKSIZES

GETRF_NPVT_NUM_INTERVALS

GETRF_NPVT_INTERVALS

GETRF_NPVT_BLKSIZES

GETRF_NPVT_BATCH_NUM_INTERVALS

GETRF_NPVT_BATCH_INTERVALS

GETRF_NPVT_BATCH_BLKSIZES

2.3.15 getri function

GETRI_MAX_COLS

GETRI_TINY_SIZE

GETRI_NUM_INTERVALS

GETRI_INTERVALS

GETRI_BLKSIZES

GETRI_BATCH_TINY_SIZE

GETRI_BATCH_NUM_INTERVALS

GETRI_BATCH_INTERVALS

GETRI_BATCH_BLKSIZES

2.3.16 trtri function

TRTRI_MAX_COLS

TRTRI_NUM_INTERVALS

TRTRI_INTERVALS

TRTRI_BLKSIZES

TRTRI_BATCH_NUM_INTERVALS

TRTRI_BATCH_INTERVALS

TRTRI_BATCH_BLKSIZES

2.4 Contributing Guidelines

More to come later. . .

CHAPTER

THREE

ROCSOLVER API

3.1 Types

rocSOLVER uses types and enumerations defined by the rocBLAS API. For more information, see the rocBLAS types documentation. Next we present additional types, only used in rocSOLVER, that extend the rocBLAS API.

3.1.1 Additional types

List of additional types

- *rocblas_direct*
- *rocblas_storev*
- *rocblas_svect*
- *rocblas_evect*
- *rocblas_workmode*
- *rocblas_eform*

rocblas_direct

enum rocblas_direct

Used to specify the order in which multiple Householder matrices are applied together.

Values:

enumerator rocblas_forward_direction

Householder matrices applied from the right.

enumerator rocblas_backward_direction

Householder matrices applied from the left.

rocblas_storev

enum rocblas_storev

Used to specify how householder vectors are stored in a matrix of vectors.

Values:

enumerator rocblas_column_wise

Householder vectors are stored in the columns of a matrix.

enumerator rocblas_row_wise

Householder vectors are stored in the rows of a matrix.

rocblas_svect

enum rocblas_svect

Used to specify how the singular vectors are to be computed and stored.

Values:

enumerator rocblas_svect_all

The entire associated orthogonal/unitary matrix is computed.

enumerator rocblas_svect_singular

Only the singular vectors are computed and stored in output array.

enumerator rocblas_svect_overwrite

Only the singular vectors are computed and overwrite the input matrix.

enumerator rocblas_svect_none

No singular vectors are computed.

rocblas_evect

enum rocblas_evect

Used to specify how the eigenvectors are to be computed.

Values:

enumerator rocblas_evect_original

Compute eigenvectors for the original symmetric/Hermitian matrix.

enumerator rocblas_evect_tridiagonal

Compute eigenvectors for the symmetric tridiagonal matrix.

enumerator rocblas_evect_none

No eigenvectors are computed.

rocblas_workmode

enum rocblas_workmode

Used to enable the use of fast algorithms (with out-of-place computations) in some of the routines.

Values:

enumerator rocblas_outofplace

Out-of-place computations are allowed; this requires extra device memory for workspace.

enumerator rocblas_inplace

If not enough memory is available, this forces in-place computations.

rocblas_eform

enum rocblas_eform

Used to specify the form of the generalized eigenproblem.

Values:

enumerator rocblas_eform_ax

The problem is $Ax = \lambda Bx$.

enumerator rocblas_eform_abx

The problem is $ABx = \lambda x$.

enumerator rocblas_eform_bax

The problem is $BAx = \lambda x$.

3.2 LAPACK Auxiliary Functions

These are functions that support more *advanced LAPACK routines*. The auxiliary functions are divided into the following categories:

- *Vector and Matrix manipulations*. Some basic operations with vectors and matrices that are not part of the BLAS standard.
- *Householder reflections*. Generation and application of Householder matrices.
- *Bidiagonal forms*. Computations specialized in bidiagonal matrices.
- *Tridiagonal forms*. Computations specialized in tridiagonal matrices.
- *Symmetric matrices*. Computations specialized in symmetric matrices.
- *Orthonormal matrices*. Generation and application of orthonormal matrices.
- *Unitary matrices*. Generation and application of unitary matrices.

Note: Throughout the APIs' descriptions, we use the following notations:

- x[i] stands for the i-th element of vector x, while A[i,j] represents the element in the i-th row and j-th column of matrix A. Indices are 1-based, i.e. x[1] is the first element of x.
- If X is a real vector or matrix, X^T indicates its transpose; if X is complex, then X^H represents its conjugate transpose. When X could be real or complex, we use X' to indicate X transposed or X conjugate transposed, accordingly.
- x_i = x_i; we sometimes use both notations, x_i when displaying mathematical equations, and x_i in the text describing the function parameters.

3.2.1 Vector and Matrix manipulations

List of vector and matrix manipulations

- *rocsolver_<type>lacgv()*
- *rocsolver_<type>laswp()*

rocsolver_<type>lacgv()

rocblas_status **rocsolver_zlacgv** (rocblas_handle *handle*, **const** rocblas_int *n*, rocblas_double_complex **x*, **const** rocblas_int *incx*)

rocblas_status **rocsolver_clacgv** (rocblas_handle *handle*, **const** rocblas_int *n*, rocblas_float_complex **x*, **const** rocblas_int *incx*)

LACGV conjugates the complex vector x.

It conjugates the n entries of a complex vector x with increment incx.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] n`: rocblas_int. n >= 0. The dimension of vector x.
- `[inout] x`: pointer to type. Array on the GPU of size at least n (size depends on the value of incx). On entry, the vector x. On exit, each entry is overwritten with its conjugate value.
- `[in] incx`: rocblas_int. incx != 0. The distance between two consecutive elements of x. If incx is negative, the elements of x are indexed in reverse order.

rocsolver_<type>laswp()

rocblas_status **rocsolver_zlaswp** (rocblas_handle *handle*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, **const** rocblas_int *k1*, **const** rocblas_int *k2*, **const** rocblas_int **ipiv*, **const** rocblas_int *incx*)

rocblas_status **rocsolver_claswp** (rocblas_handle *handle*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, **const** rocblas_int *k1*, **const** rocblas_int *k2*, **const** rocblas_int **ipiv*, **const** rocblas_int *incx*)

rocblas_status **rocsolver_dlaswp** (rocblas_handle *handle*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, **const** rocblas_int *k1*, **const** rocblas_int *k2*, **const** rocblas_int **ipiv*, **const** rocblas_int *incx*)

rocblas_status **rocsolver_slaswp** (rocblas_handle *handle*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, **const** rocblas_int *k1*, **const** rocblas_int *k2*, **const** rocblas_int **ipiv*, **const** rocblas_int *incx*)

LASWP performs a series of row interchanges on the matrix A.

Row interchanges are done one by one. If ipiv$[k_1 + (j - k_1) \cdot \text{abs}(\text{incx})] = r$, then the j-th row of A will be interchanged with the r-th row of A, for $j = k_1, k_1 + 1, \ldots, k_2$. Indices k_1 and k_2 are 1-based indices.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] n`: rocblas_int. n >= 0. The number of columns of the matrix A.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the matrix to which the row interchanges will be applied. On exit, the resulting permuted matrix.
- `[in] lda`: rocblas_int. lda > 0. The leading dimension of the array A.
- `[in] k1`: rocblas_int. k1 > 0. The k_1 index. It is the first element of ipiv for which a row interchange will be done. This is a 1-based index.
- `[in] k2`: rocblas_int. k2 > k1 > 0. The k_2 index. k_2 - k_1 + 1 is the number of elements of ipiv for which a row interchange will be done. This is a 1-based index.
- `[in] ipiv`: pointer to rocblas_int. Array on the GPU of dimension at least k_1 + (k_2 - k_1)*abs(incx). The vector of pivot indices. Only the elements in positions k_1 through k_1 + (k_2 - k_1)*abs(incx) of this vector are accessed. Elements of ipiv are considered 1-based.
- `[in] incx`: rocblas_int. incx != 0. The distance between successive values of ipiv. If incx is negative, the pivots are applied in reverse order.

3.2.2 Householder reflections

List of Householder functions

- *rocsolver_<type>larfg()*
- *rocsolver_<type>larft()*
- *rocsolver_<type>larf()*
- *rocsolver_<type>larfb()*

rocsolver_<type>larfg()

rocblas_status **rocsolver_zlarfg** (rocblas_handle *handle*, **const** rocblas_int *n*, rocblas_double_complex **alpha*, rocblas_double_complex **x*, **const** rocblas_int *incx*, rocblas_double_complex **tau*)

rocblas_status **rocsolver_clarfg** (rocblas_handle *handle*, **const** rocblas_int *n*, rocblas_float_complex **alpha*, rocblas_float_complex **x*, **const** rocblas_int *incx*, rocblas_float_complex **tau*)

rocblas_status **rocsolver_dlarfg** (rocblas_handle *handle*, **const** rocblas_int *n*, double **alpha*, double **x*, **const** rocblas_int *incx*, double **tau*)

rocblas_status **rocsolver_slarfg** (rocblas_handle *handle*, **const** rocblas_int *n*, float **alpha*, float **x*, **const** rocblas_int *incx*, float **tau*)

LARFG generates a Householder reflector H of order n.

The reflector H is such that

$$H' \begin{bmatrix} \text{alpha} \\ x \end{bmatrix} = \begin{bmatrix} \text{beta} \\ 0 \end{bmatrix}$$

where x is an n-1 vector, and alpha and beta are scalars. Matrix H can be generated as

$$H = I - \text{tau} \begin{bmatrix} 1 \\ v \end{bmatrix} \begin{bmatrix} 1 & v' \end{bmatrix}$$

where v is an n-1 vector, and tau is a scalar known as the Householder scalar. The vector

$$\bar{v} = \begin{bmatrix} 1 \\ v \end{bmatrix}$$

is the Householder vector associated with the reflection.

Note The matrix H is orthogonal/unitary (i.e. $H'H = HH' = I$). It is symmetric when real (i.e. $H^T = H$), but not Hermitian when complex (i.e. $H^H \neq H$ in general).

Parameters

- `[in] handle`: rocblas_handle.
- `[in] n`: rocblas_int. n >= 0. The order (size) of reflector H.
- `[inout] alpha`: pointer to type. A scalar on the GPU. On entry, the scalar alpha. On exit, it is overwritten with beta.
- `[inout] x`: pointer to type. Array on the GPU of size at least n-1 (size depends on the value of incx). On entry, the vector x, On exit, it is overwritten with vector v.
- `[in] incx`: rocblas_int. incx > 0. The distance between two consecutive elements of x.
- `[out] tau`: pointer to type. A scalar on the GPU. The Householder scalar tau.

rocsolver_<type>larft()

rocblas_status **rocsolver_zlarft**(rocblas_handle *handle*, **const** *rocblas_direct* *direct*, **const** *rocblas_storev* *storev*, **const** rocblas_int *n*, **const** rocblas_int *k*, rocblas_double_complex **V*, **const** rocblas_int *ldv*, rocblas_double_complex **tau*, rocblas_double_complex **T*, **const** rocblas_int *ldt*)

rocblas_status **rocsolver_clarft**(rocblas_handle *handle*, **const** *rocblas_direct* *direct*, **const** *rocblas_storev* *storev*, **const** rocblas_int *n*, **const** rocblas_int *k*, rocblas_float_complex **V*, **const** rocblas_int *ldv*, rocblas_float_complex **tau*, rocblas_float_complex **T*, **const** rocblas_int *ldt*)

rocblas_status **rocsolver_dlarft**(rocblas_handle *handle*, **const** *rocblas_direct* *direct*, **const** *rocblas_storev* *storev*, **const** rocblas_int *n*, **const** rocblas_int *k*, double **V*, **const** rocblas_int *ldv*, double **tau*, double **T*, **const** rocblas_int *ldt*)

rocblas_status **rocsolver_slarft**(rocblas_handle *handle*, **const** *rocblas_direct* *direct*, **const** *rocblas_storev* *storev*, **const** rocblas_int *n*, **const** rocblas_int *k*, float **V*, **const** rocblas_int *ldv*, float **tau*, float **T*, **const** rocblas_int *ldt*)

LARFT generates the triangular factor T of a block reflector H of order n.

The block reflector H is defined as the product of k Householder matrices

$$H = H_1 H_2 \cdots H_k \quad \text{if direct indicates forward direction, or}$$
$$H = H_k \cdots H_2 H_1 \quad \text{if direct indicates backward direction}$$

The triangular factor T is upper triangular in the forward direction and lower triangular in the backward direction. If storev is column-wise, then

$$H = I - VTV'$$

where the i-th column of matrix V contains the Householder vector associated with H_i. If storev is row-wise, then

$$H = I - V'TV$$

where the i-th row of matrix V contains the Householder vector associated with H_i.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] direct`: *rocblas_direct*. Specifies the direction in which the Householder matrices are applied.
- `[in] storev`: *rocblas_storev*. Specifies how the Householder vectors are stored in matrix V.
- `[in] n`: rocblas_int. n >= 0. The order (size) of the block reflector.
- `[in] k`: rocblas_int. k >= 1. The number of Householder matrices forming H.
- `[in] V`: pointer to type. Array on the GPU of size ldv*k if column-wise, or ldv*n if row-wise. The matrix of Householder vectors.
- `[in] ldv`: rocblas_int. ldv >= n if column-wise, or ldv >= k if row-wise. Leading dimension of V.
- `[in] tau`: pointer to type. Array of k scalars on the GPU. The vector of all the Householder scalars.
- `[out] T`: pointer to type. Array on the GPU of dimension ldt*k. The triangular factor. T is upper triangular if direct indicates forward direction, otherwise it is lower triangular. The rest of the array is not used.
- `[in] ldt`: rocblas_int. ldt >= k. The leading dimension of T.

rocsolver_<type>larf()

rocblas_status **rocsolver_zlarf** (rocblas_handle *handle*, **const** rocblas_side *side*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_double_complex **x*, **const** rocblas_int *incx*, **const** rocblas_double_complex **alpha*, rocblas_double_complex **A*, **const** rocblas_int *lda*)

rocblas_status **rocsolver_clarf** (rocblas_handle *handle*, **const** rocblas_side *side*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_float_complex **x*, **const** rocblas_int *incx*, **const** rocblas_float_complex **alpha*, rocblas_float_complex **A*, **const** rocblas_int *lda*) rocblas_status **rocsolver_dlarf** (rocblas_handle *handle*, **const** rocblas_side *side*, **const** rocblas_int *m*, **const** rocblas_int *n*, double **x*, **const** rocblas_int *incx*, **const** double **alpha*, double **A*, **const** rocblas_int *lda*)

rocblas_status **rocsolver_slarf** (rocblas_handle *handle*, **const** rocblas_side *side*, **const** rocblas_int *m*, **const** rocblas_int *n*, float **x*, **const** rocblas_int *incx*, **const** float **alpha*, float **A*, **const** rocblas_int *lda*)

LARF applies a Householder reflector H to a general matrix A.

The Householder reflector H, of order m or n, is to be applied to an m-by-n matrix A from the left or the right, depending on the value of side. H is given by

$$H = I - \text{alpha} \cdot xx'$$

where alpha is the Householder scalar and x is a Householder vector. H is never actually computed.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] side`: rocblas_side. Determines whether H is applied from the left or the right.
- `[in] m`: rocblas_int. m >= 0. Number of rows of A.
- `[in] n`: rocblas_int. n >= 0. Number of columns of A.
- `[in] x`: pointer to type. Array on the GPU of size at least 1 + (m-1)*abs(incx) if left side, or at least 1 + (n-1)*abs(incx) if right side. The Householder vector x.
- `[in] incx`: rocblas_int. incx != 0. Distance between two consecutive elements of x. If incx < 0, the elements of x are indexed in reverse order.
- `[in] alpha`: pointer to type. A scalar on the GPU. The Householder scalar. If alpha = 0, then H = I (A will remain the same; x is never used)
- `[inout] A`: pointer to type. Array on the GPU of size lda*n. On entry, the matrix A. On exit, it is overwritten with H*A (or A*H).
- `[in] lda`: rocblas_int. lda >= m. Leading dimension of A.

rocsolver_<type>larfb()

rocblas_status **rocsolver_zlarfb** (rocblas_handle *handle*, **const** rocblas_side *side*, **const** rocblas_operation *trans*, **const** *rocblas_direct* *direct*, **const** *rocblas_storev* *storev*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, rocblas_double_complex **V*, **const** rocblas_int *ldv*, rocblas_double_complex **T*, **const** rocblas_int *ldt*, rocblas_double_complex **A*, **const** rocblas_int *lda*)

rocblas_status **rocsolver_clarfb** (rocblas_handle *handle*, **const** rocblas_side *side*, **const** rocblas_operation *trans*, **const** *rocblas_direct* *direct*, **const** *rocblas_storev* *storev*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, rocblas_float_complex **V*, **const** rocblas_int *ldv*, rocblas_float_complex **T*, **const** rocblas_int *ldt*, rocblas_float_complex **A*, **const** rocblas_int *lda*) rocblas_status **rocsolver_dlarfb**(rocblas_handle *handle*, **const** rocblas_side *side*, **const** rocblas_operation *trans*, **const** *rocblas_direct* *direct*, **const** *rocblas_storev* *storev*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, double **V*, **const** rocblas_int *ldv*, double **T*, **const** rocblas_int *ldt*, double **A*, **const** rocblas_int *lda*)

rocblas_status **rocsolver_slarfb**(rocblas_handle *handle*, **const** rocblas_side *side*, **const** rocblas_operation *trans*, **const** *rocblas_direct* *direct*, **const** *rocblas_storev* *storev*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, float **V*, **const** rocblas_int *ldv*, float **T*, **const** rocblas_int *ldt*, float **A*, **const** rocblas_int *lda*)

LARFB applies a block reflector H to a general m-by-n matrix A.

The block reflector H is applied in one of the following forms, depending on the values of side and trans:

$$\begin{array}{cl} HA & \text{(No transpose from the left),} \\ H'A & \text{(Transpose or conjugate transpose from the left),} \\ AH & \text{(No transpose from the right), or} \\ AH' & \text{(Transpose or conjugate transpose from the right).} \end{array}$$

The block reflector H is defined as the product of k Householder matrices as

$$\begin{array}{ll} H = H_1 H_2 \cdots H_k & \text{if direct indicates forward direction, or} \\ H = H_k \cdots H_2 H_1 & \text{if direct indicates backward direction} \end{array}$$

H is never stored. It is calculated as

$$H = I - VTV'$$

where the i-th column of matrix V contains the Householder vector associated with H_i, if storev is column-wise; or

$$H = I - V'TV$$

where the i-th row of matrix V contains the Householder vector associated with H_i, if storev is row-wise. T is the associated triangular factor as computed by *LARFT*.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] side`: rocblas_side. Specifies from which side to apply H.
- `[in] trans`: rocblas_operation. Specifies whether the block reflector or its transpose/conjugate transpose is to be applied.
- `[in] direct`: *rocblas_direct*. Specifies the direction in which the Householder matrices are to be applied to generate H.
- `[in] storev`: *rocblas_storev*. Specifies how the Householder vectors are stored in matrix V.

- `[in] m`: rocblas_int. m >= 0. Number of rows of matrix A.
- `[in] n`: rocblas_int. n >= 0. Number of columns of matrix A.
- `[in] k`: rocblas_int. k >= 1. The number of Householder matrices.
- `[in] V`: pointer to type. Array on the GPU of size ldv*k if column-wise, ldv*n if row-wise and applying from the right, or ldv*m if row-wise and applying from the left. The matrix of Householder vectors.
- `[in] ldv`: rocblas_int. ldv >= k if row-wise, ldv >= m if column-wise and applying from the left, or ldv >= n if column-wise and applying from the right. Leading dimension of V.
- `[in] T`: pointer to type. Array on the GPU of dimension ldt*k. The triangular factor of the block reflector.
- `[in] ldt`: rocblas_int. ldt >= k. The leading dimension of T.
- `[inout] A`: pointer to type. Array on the GPU of size lda*n. On entry, the matrix A. On exit, it is overwritten with H*A, A*H, H'*A, or A*H'.
- `[in] lda`: rocblas_int. lda >= m. Leading dimension of A.

3.2.3 Bidiagonal forms

List of functions for bidiagonal forms

- *rocsolver_<type>labrd()*
- *rocsolver_<type>bdsqr()*

rocsolver_<type>labrd()

rocblas_status **rocsolver_zlabrd**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, rocblas_double_complex **A*, **const** rocblas_int *lda*, double **D*, double **E*, rocblas_double_complex **tauq*, rocblas_double_complex **taup*, rocblas_double_complex **X*, **const** rocblas_int *ldx*, rocblas_double_complex **Y*, **const** rocblas_int *ldy*)

rocblas_status **rocsolver_clabrd**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, rocblas_float_complex **A*, **const** rocblas_int *lda*, float **D*, float **E*, rocblas_float_complex **tauq*, rocblas_float_complex **taup*, rocblas_float_complex **X*, **const** rocblas_int *ldx*, rocblas_float_complex **Y*, **const** rocblas_int *ldy*)

rocblas_status **rocsolver_dlabrd**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, double **A*, **const** rocblas_int *lda*, double **D*, double **E*, double **tauq*, double **taup*, double **X*, **const** rocblas_int *ldx*, double **Y*, **const** rocblas_int *ldy*)

rocblas_status **rocsolver_slabrd**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, float **A*, **const** rocblas_int *lda*, float **D*, float **E*, float **tauq*, float **taup*, float **X*, **const** rocblas_int *ldx*, float **Y*, **const** rocblas_int *ldy*)

LABRD computes the bidiagonal form of the first k rows and columns of a general m-by-n matrix A, as well as the matrices X and Y needed to reduce the remaining part of A.

The reduced form is given by:

$$B = Q'AP$$

where the leading k-by-k block of B is upper bidiagonal if m >= n, or lower bidiagonal if m < n. Q and P are orthogonal/unitary matrices represented as the product of Householder matrices

$$\begin{array}{cl} Q = H_1H_2\cdots H_k, & \text{and} \\ P = G_1G_2\cdots G_k. & \end{array}$$

Each Householder matrix H_i and G_i is given by

$$\begin{array}{cl} H_i = I - \text{tauq}[i]\cdot v_iv_i', & \text{and} \\ G_i = I - \text{taup}[i]\cdot u_iu_i'. & \end{array}$$

If m >= n, the first i-1 elements of the Householder vector v_i are zero, and $v_i[i] = 1$; while the first i elements of the Householder vector u_i are zero, and $u_i[i+1] = 1$. If m < n, the first i elements of the Householder vector v_i are zero, and $v_i[i+1] = 1$; while the first i-1 elements of the Householder vector u_i are zero, and $u_i[i] = 1$.

The unreduced part of the matrix A can be updated using the block update

$$A = A - VY' - XU'$$

where V and U are the m-by-k and n-by-k matrices formed with the vectors v_i and u_i, respectively.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. m >= 0. The number of rows of the matrix A.
- `[in] n`: rocblas_int. n >= 0. The number of columns of the matrix A.
- `[in] k`: rocblas_int. min(m,n) >= k >= 0. The number of leading rows and columns of matrix A that will be reduced.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the m-by-n matrix to be reduced. On exit, the first k elements on the diagonal and superdiagonal (if m >= n), or subdiagonal (if m < n), contain the bidiagonal form B. If m >= n, the elements below the diagonal of the first k columns are the possibly non-zero elements of the Householder vectors associated with Q, while the elements above the superdiagonal of the first k rows are the n - i - 1 possibly non-zero elements of the Householder vectors related to P. If m < n, the elements below the subdiagonal of the first k columns are the m - i - 1 possibly non-zero elements of the Householder vectors related to Q, while the elements above the diagonal of the first k rows are the n - i possibly non-zero elements of the vectors associated with P.
- `[in] lda`: rocblas_int. lda >= m. specifies the leading dimension of A.
- `[out] D`: pointer to real type. Array on the GPU of dimension k. The diagonal elements of B.
- `[out] E`: pointer to real type. Array on the GPU of dimension k. The off-diagonal elements of B.
- `[out] tauq`: pointer to type. Array on the GPU of dimension k. The Householder scalars associated with matrix Q.

- [out] taup: pointer to type. Array on the GPU of dimension k. The Householder scalars associated with matrix P.
- [out] X: pointer to type. Array on the GPU of dimension ldx*k. The m-by-k matrix needed to update the unreduced part of A.
- [in] ldx: rocblas_int. ldx >= m. The leading dimension of X.
- [out] Y: pointer to type. Array on the GPU of dimension ldy*k. The n-by-k matrix needed to update the unreduced part of A.
- [in] ldy: rocblas_int. ldy >= n. The leading dimension of Y.

rocsolver_<type>bdsqr()

rocblas_status **rocsolver_zbdsqr**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, **const** rocblas_int *nv*, **const** rocblas_int *nu*, **const** rocblas_int *nc*, double **D*, double **E*, rocblas_double_complex **V*, **const** rocblas_int *ldv*, rocblas_double_complex **U*, **const** rocblas_int *ldu*, rocblas_double_complex **C*, **const** rocblas_int *ldc*, rocblas_int **info*)

rocblas_status **rocsolver_cbdsqr**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, **const** rocblas_int *nv*, **const** rocblas_int *nu*, **const** rocblas_int *nc*, float **D*, float **E*, rocblas_float_complex **V*, **const** rocblas_int *ldv*, rocblas_float_complex **U*, **const** rocblas_int *ldu*, rocblas_float_complex **C*, **const** rocblas_int *ldc*, rocblas_int **info*)

rocblas_status **rocsolver_dbdsqr**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, **const** rocblas_int *nv*, **const** rocblas_int *nu*, **const** rocblas_int *nc*, double **D*, double **E*, double **V*, **const** rocblas_int *ldv*, double **U*, **const** rocblas_int *ldu*, double **C*, **const** rocblas_int *ldc*, rocblas_int **info*)

rocblas_status **rocsolver_sbdsqr**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, **const** rocblas_int *nv*, **const** rocblas_int *nu*, **const** rocblas_int *nc*, float **D*, float **E*, float **V*, **const** rocblas_int *ldv*, float **U*, **const** rocblas_int *ldu*, float **C*, **const** rocblas_int *ldc*, rocblas_int **info*)

BDSQR computes the singular value decomposition (SVD) of an n-by-n bidiagonal matrix B, using the implicit QR algorithm.

The SVD of B has the form:

$$B = QSP'$$

where S is the n-by-n diagonal matrix of singular values of B, the columns of Q are the left singular vectors of B, and the columns of P are its right singular vectors.

The computation of the singular vectors is optional; this function accepts input matrices U (of size nu-by-n) and V (of size n-by-nv) that are overwritten with UQ and $P'V$. If nu = 0 no left vectors are computed; if nv = 0 no right vectors are computed.

Optionally, this function can also compute $Q'C$ for a given n-by-nc input matrix C.

Parameters

- [in] handle: rocblas_handle.

- `[in] uplo`: rocblas_fill. Specifies whether B is upper or lower bidiagonal.
- `[in] n`: rocblas_int. n >= 0. The number of rows and columns of matrix B.
- `[in] nv`: rocblas_int. nv >= 0. The number of columns of matrix V.
- `[in] nu`: rocblas_int. nu >= 0. The number of rows of matrix U.
- `[in] nc`: rocblas_int. nu >= 0. The number of columns of matrix C.
- `[inout] D`: pointer to real type. Array on the GPU of dimension n. On entry, the diagonal elements of B. On exit, if info = 0, the singular values of B in decreasing order; if info > 0, the diagonal elements of a bidiagonal matrix orthogonally equivalent to B.
- `[inout] E`: pointer to real type. Array on the GPU of dimension n-1. On entry, the off-diagonal elements of B. On exit, if info > 0, the off-diagonal elements of a bidiagonal matrix orthogonally equivalent to B (if info = 0 this matrix converges to zero).
- `[inout] V`: pointer to type. Array on the GPU of dimension ldv*nv. On entry, the matrix V. On exit, it is overwritten with P'*V. (Not referenced if nv = 0).
- `[in] ldv`: rocblas_int. ldv >= n if nv > 0, or ldv >=1 if nv = 0. The leading dimension of V.
- `[inout] U`: pointer to type. Array on the GPU of dimension ldu*n. On entry, the matrix U. On exit, it is overwritten with U*Q. (Not referenced if nu = 0).
- `[in] ldu`: rocblas_int. ldu >= nu. The leading dimension of U.
- `[inout] C`: pointer to type. Array on the GPU of dimension ldc*nc. On entry, the matrix C. On exit, it is overwritten with Q'*C. (Not referenced if nc = 0).
- `[in] ldc`: rocblas_int. ldc >= n if nc > 0, or ldc >=1 if nc = 0. The leading dimension of C.
- `[out] info`: pointer to a rocblas_int on the GPU. If info = 0, successful exit. If info = i > 0, i elements of E have not converged to zero.

3.2.4 Tridiagonal forms

List of functions for tridiagonal forms

- *rocsolver_<type>latrd()*
- *rocsolver_<type>sterf()*
- *rocsolver_<type>steqr()*
- *rocsolver_<type>stedc()*

rocsolver_<type>latrd()

rocblas_status **rocsolver_zlatrd**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, **const** rocblas_int *k*, rocblas_double_complex **A*, **const** rocblas_int *lda*, double **E*, rocblas_double_complex **tau*, rocblas_double_complex **W*, **const** rocblas_int *ldw*)

rocblas_status **rocsolver_clatrd**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, **const** rocblas_int *k*, rocblas_float_complex **A*, **const** rocblas_int *lda*, float **E*, rocblas_float_complex **tau*, rocblas_float_complex **W*, **const** rocblas_int *ldw*) rocblas_status **rocsolver_dlatrd**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, **const** rocblas_int *k*, double **A*, **const** rocblas_int *lda*, double **E*, double **tau*, double **W*, **const** rocblas_int *ldw*)

rocblas_status **rocsolver_slatrd**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, **const** rocblas_int *k*, float **A*, **const** rocblas_int *lda*, float **E*, float **tau*, float **W*, **const** rocblas_int *ldw*)

LATRD computes the tridiagonal form of k rows and columns of a symmetric/hermitian matrix A, as well as the matrix W needed to update the remaining part of A.

The reduced form is given by:

$$T = Q'AQ$$

If uplo is lower, the first k rows and columns of T form the tridiagonal block. If uplo is upper, then the last k rows and columns of T form the tridiagonal block. Q is an orthogonal/unitary matrix represented as the product of Householder matrices

$$\begin{array}{cl} Q = H_1H_2\cdots H_k & \text{if uplo indicates lower, or} \\ Q = H_nH_{n-1}\cdots H_{n-k+1} & \text{if uplo is upper.} \end{array}$$

Each Householder matrix H_i is given by

$$H_i = I - \text{tau}[i] \cdot v_i v_i'$$

where tau[i] is the corresponding Householder scalar. When uplo indicates lower, the first i elements of the Householder vector v_i are zero, and $v_i[i+1] = 1$. If uplo is upper, the last n-i elements of the Householder vector v_i are zero, and $v_i[i] = 1$.

The unreduced part of the matrix A can be updated using a rank update of the form:

$$A = A - VW' - WV'$$

where V is the n-by-k matrix formed by the vectors v_i.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower part of the matrix A is stored. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.
- `[in] n`: rocblas_int. n >= 0. The number of rows and columns of the matrix A.
- `[in] k`: rocblas_int. 0 <= k <= n. The number of rows and columns of the matrix A to be reduced.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the n-by-n matrix to be reduced. On exit, if uplo is lower, the first k columns have been reduced to tridiagonal form (given in the diagonal elements of A and the array E), the elements below the diagonal contain the possibly non-zero entries of the Householder vectors associated with Q, stored as columns. If uplo is upper, the last k columns have been reduced to tridiagonal form (given in the diagonal elements of A and the array E), the elements above the diagonal contain the possibly non-zero entries of the Householder vectors associated with Q, stored as columns.

- `[in] lda`: rocblas_int. lda >= n. The leading dimension of A.
- `[out] E`: pointer to real type. Array on the GPU of dimension n-1. If upper (lower), the last (first) k elements of E are the off-diagonal elements of the computed tridiagonal block.
- `[out] tau`: pointer to type. Array on the GPU of dimension n-1. If upper (lower), the last (first) k elements of tau are the Householder scalars related to Q.
- `[out] W`: pointer to type. Array on the GPU of dimension ldw*k. The n-by-k matrix needed to update the unreduced part of A.
- `[in] ldw`: rocblas_int. ldw >= n. The leading dimension of W.

rocsolver_<type>sterf()

rocblas_status **rocsolver_dsterf**(rocblas_handle *handle*, **const** rocblas_int *n*, double **D*, double **E*, rocblas_int **info*)

rocblas_status **rocsolver_ssterf**(rocblas_handle *handle*, **const** rocblas_int *n*, float **D*, float **E*, rocblas_int **info*)

STERF computes the eigenvalues of a symmetric tridiagonal matrix.

The eigenvalues of the symmetric tridiagonal matrix are computed by the Pal-Walker-Kahan variant of the QL/QR algorithm, and returned in increasing order.

The matrix is not represented explicitly, but rather as the array of diagonal elements D and the array of symmetric off-diagonal elements E.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] n`: rocblas_int. n >= 0. The number of rows and columns of the tridiagonal matrix.
- `[inout] D`: pointer to real type. Array on the GPU of dimension n. On entry, the diagonal elements of the tridiagonal matrix. On exit, if info = 0, the eigenvalues in increasing order. If info > 0, the diagonal elements of a tridiagonal matrix that is similar to the original matrix (i.e. has the same eigenvalues).
- `[inout] E`: pointer to real type. Array on the GPU of dimension n-1. On entry, the off-diagonal elements of the tridiagonal matrix. On exit, if info = 0, this array converges to zero. If info > 0, the off-diagonal elements of a tridiagonal matrix that is similar to the original matrix (i.e. has the same eigenvalues).
- `[out] info`: pointer to a rocblas_int on the GPU. If info = 0, successful exit. If info = i > 0, STERF did not converge. i elements of E did not converge to zero.

rocsolver_<type>steqr()

rocblas_status **rocsolver_zsteqr**(rocblas_handle *handle*, **const** *rocblas_evect* *evect*, **const** rocblas_int *n*, double **D*, double **E*, rocblas_double_complex **C*, **const** rocblas_int *ldc*, rocblas_int **info*)

rocblas_status **rocsolver_csteqr**(rocblas_handle *handle*, **const** *rocblas_evect* *evect*, **const** rocblas_int *n*, float **D*, float **E*, rocblas_float_complex **C*, **const** rocblas_int *ldc*, rocblas_int **info*)

rocblas_status **rocsolver_dsteqr**(rocblas_handle *handle*, **const** *rocblas_evect* *evect*, **const** rocblas_int *n*, double **D*, double **E*, double **C*, **const** rocblas_int *ldc*, rocblas_int **info*) rocblas_status **rocsolver_ssteqr** (rocblas_handle *handle*, **const** *rocblas_evect* *evect*, **const** rocblas_int *n*, float **D*, float **E*, float **C*, **const** rocblas_int *ldc*, rocblas_int **info*)

STEQR computes the eigenvalues and (optionally) eigenvectors of a symmetric tridiagonal matrix.

The eigenvalues of the symmetric tridiagonal matrix are computed by the implicit QL/QR algorithm, and returned in increasing order.

The matrix is not represented explicitly, but rather as the array of diagonal elements D and the array of symmetric off-diagonal elements E. When D and E correspond to the tridiagonal form of a full symmetric/Hermitian matrix, as returned by, e.g., *SYTRD* or *HETRD*, the eigenvectors of the original matrix can also be computed, depending on the value of evect.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] evect`: *rocblas_evect*. Specifies how the eigenvectors are computed.
- `[in] n`: rocblas_int. n >= 0. The number of rows and columns of the tridiagonal matrix.
- `[inout] D`: pointer to real type. Array on the GPU of dimension n. On entry, the diagonal elements of the tridiagonal matrix. On exit, if info = 0, the eigenvalues in increasing order. If info > 0, the diagonal elements of a tridiagonal matrix that is similar to the original matrix (i.e. has the same eigenvalues).
- `[inout] E`: pointer to real type. Array on the GPU of dimension n-1. On entry, the off-diagonal elements of the tridiagonal matrix. On exit, if info = 0, this array converges to zero. If info > 0, the off-diagonal elements of a tridiagonal matrix that is similar to the original matrix (i.e. has the same eigenvalues).
- `[inout] C`: pointer to type. Array on the GPU of dimension ldc*n. On entry, if evect is original, the orthogonal/unitary matrix used for the reduction to tridiagonal form as returned by, e.g., *ORGTR* or *UNGTR*. On exit, it is overwritten with the eigenvectors of the original symmetric/Hermitian matrix (if evect is original), or the eigenvectors of the tridiagonal matrix (if evect is tridiagonal). (Not referenced if evect is none).
- `[in] ldc`: rocblas_int. ldc >= n if evect is original or tridiagonal. Specifies the leading dimension of C. (Not referenced if evect is none).
- `[out] info`: pointer to a rocblas_int on the GPU. If info = 0, successful exit. If info = i > 0, STEQR did not converge. i elements of E did not converge to zero.

rocsolver_<type>stedc()

rocblas_status **rocsolver_zstedc** (rocblas_handle *handle*, **const** *rocblas_evect* *evect*, **const** rocblas_int *n*, double **D*, double **E*, rocblas_double_complex **C*, **const** rocblas_int *ldc*, rocblas_int **info*)

rocblas_status **rocsolver_cstedc** (rocblas_handle *handle*, **const** *rocblas_evect* *evect*, **const** rocblas_int *n*, float **D*, float **E*, rocblas_float_complex **C*, **const** rocblas_int *ldc*, rocblas_int **info*)

rocblas_status **rocsolver_dstedc** (rocblas_handle *handle*, **const** *rocblas_evect* *evect*, **const** rocblas_int *n*, double **D*, double **E*, double **C*, **const** rocblas_int *ldc*, rocblas_int **info*) rocblas_status **rocsolver_sstedc**(rocblas_handle *handle*, **const** *rocblas_evect* *evect*, **const** rocblas_int *n*, float **D*, float **E*, float **C*, **const** rocblas_int *ldc*, rocblas_int **info*)

STEDC computes the eigenvalues and (optionally) eigenvectors of a symmetric tridiagonal matrix.

This function uses the divide and conquer method to compute the eigenvectors. The eigenvalues are returned in increasing order.

The matrix is not represented explicitly, but rather as the array of diagonal elements D and the array of symmetric off-diagonal elements E. When D and E correspond to the tridiagonal form of a full symmetric/Hermitian matrix, as returned by, e.g., *SYTRD* or *HETRD*, the eigenvectors of the original matrix can also be computed, depending on the value of evect.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] evect`: *rocblas_evect*. Specifies how the eigenvectors are computed.
- `[in] n`: rocblas_int. n >= 0. The number of rows and columns of the tridiagonal matrix.
- `[inout] D`: pointer to real type. Array on the GPU of dimension n. On entry, the diagonal elements of the tridiagonal matrix. On exit, if info = 0, the eigenvalues in increasing order.
- `[inout] E`: pointer to real type. Array on the GPU of dimension n-1. On entry, the off-diagonal elements of the tridiagonal matrix. On exit, if info = 0, the values of this array are destroyed.
- `[inout] C`: pointer to type. Array on the GPU of dimension ldc*n. On entry, if evect is original, the orthogonal/unitary matrix used for the reduction to tridiagonal form as returned by, e.g., *ORGTR* or *UNGTR*. On exit, if info = 0, it is overwritten with the eigenvectors of the original symmetric/Hermitian matrix (if evect is original), or the eigenvectors of the tridiagonal matrix (if evect is tridiagonal). (Not referenced if evect is none).
- `[in] ldc`: rocblas_int. ldc >= n if evect is original or tridiagonal. Specifies the leading dimension of C. (Not referenced if evect is none).
- `[out] info`: pointer to a rocblas_int on the GPU. If info = 0, successful exit. If info = i > 0, STEDC failed to compute an eigenvalue on the sub-matrix formed by the rows and columns info/(n+1) through mod(info,n+1).

3.2.5 Symmetric matrices

List of functions for symmetric matrices

- *rocsolver_<type>lasyf()*

rocsolver_<type>lasyf()

rocblas_status **rocsolver_zlasyf** (rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, **const** rocblas_int *nb*, rocblas_int **kb*, rocblas_double_complex **A*, **const** rocblas_int *lda*, rocblas_int **ipiv*, rocblas_int **info*)

rocblas_status **rocsolver_clasyf** (rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, **const** rocblas_int *nb*, rocblas_int **kb*, rocblas_float_complex **A*, **const** rocblas_int *lda*, rocblas_int **ipiv*, rocblas_int **info*)

rocblas_status **rocsolver_dlasyf** (rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, **const** rocblas_int *nb*, rocblas_int **kb*, double **A*, **const** rocblas_int *lda*, rocblas_int **ipiv*, rocblas_int **info*)

rocblas_status **rocsolver_slasyf** (rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, **const** rocblas_int *nb*, rocblas_int **kb*, float **A*, **const** rocblas_int *lda*, rocblas_int **ipiv*, rocblas_int **info*)

LASYF computes a partial factorization of a symmetric matrix A using Bunch-Kaufman diagonal pivoting.

The partial factorization has the form

$$A = \begin{bmatrix} I & U_{12} \\ 0 & U_{22} \end{bmatrix} \begin{bmatrix} A_{11} & 0 \\ 0 & D \end{bmatrix} \begin{bmatrix} I & 0 \\ U_{12}^T & U_{22}^T \end{bmatrix}$$

or

$$A = \begin{bmatrix} L_{11} & 0 \\ L_{21} & I \end{bmatrix} \begin{bmatrix} D & 0 \\ 0 & A_{22} \end{bmatrix} \begin{bmatrix} L_{11}^T & L_{21}^T \\ 0 & I \end{bmatrix}$$

depending on the value of uplo. The order of the block diagonal matrix D is either nb or $nb-1$, and is returned in the argument kb.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower part of the matrix A is stored. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.
- `[in] n`: rocblas_int. n >= 0. The number of rows and columns of the matrix A.
- `[in] nb`: rocblas_int. 2 <= nb <= n. The number of columns of A to be factored.
- `[out] kb`: pointer to a rocblas_int on the GPU. The number of columns of A that were actually factored (either nb or nb-1).
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the symmetric matrix A to be factored. On exit, the partially factored matrix.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of A.
- `[out] ipiv`: pointer to rocblas_int. Array on the GPU of dimension n. The vector of pivot indices. Elements of ipiv are 1-based indices. If uplo is upper, then only the last kb elements of ipiv will be set. For n - kb < k <= n, if ipiv[k] > 0 then rows and columns k and ipiv[k] were interchanged and D[k,k] is a 1-by-1 diagonal block. If, instead, ipiv[k] = ipiv[k-1] < 0, then rows and columns k-1 and -ipiv[k] were interchanged and D[k-1,k-1] to D[k,k] is a 2-by-2 diagonal block. If uplo is lower, then only the first kb elements of ipiv will be set. For 1 <= k <= kb, if ipiv[k] > 0 then rows and columns k

and ipiv[k] were interchanged and D[k,k] is a 1-by-1 diagonal block. If, instead, ipiv[k] = ipiv[k+1] < 0, then rows and columns k+1 and -ipiv[k] were interchanged and D[k,k] to D[k+1,k+1] is a 2-by-2 diagonal block.

- `[out] info`: pointer to a rocblas_int on the GPU. If info = 0, successful exit. If info[i] = j > 0, D is singular. D[j,j] is the first diagonal zero.

3.2.6 Orthonormal matrices

List of functions for orthonormal matrices

- *rocsolver_<type>org2r()*
- *rocsolver_<type>orgqr()*
- *rocsolver_<type>orgl2()*
- *rocsolver_<type>orglq()*
- *rocsolver_<type>org2l()*
- *rocsolver_<type>orgql()*
- *rocsolver_<type>orgbr()*
- *rocsolver_<type>orgtr()*
- *rocsolver_<type>orm2r()*
- *rocsolver_<type>ormqr()*
- *rocsolver_<type>orml2()*
- *rocsolver_<type>ormlq()*
- *rocsolver_<type>orm2l()*
- *rocsolver_<type>ormql()*
- *rocsolver_<type>ormbr()*
- *rocsolver_<type>ormtr()*

rocsolver_<type>org2r()

rocblas_status **rocsolver_dorg2r**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, double **A*, **const** rocblas_int *lda*, double **ipiv*)

rocblas_status **rocsolver_sorg2r**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, float **A*, **const** rocblas_int *lda*, float **ipiv*)

ORG2R generates an m-by-n Matrix Q with orthonormal columns.

(This is the unblocked version of the algorithm).

The matrix Q is defined as the first n columns of the product of k Householder reflectors of order m

$$Q = H_1 H_2 \cdots H_k.$$

The Householder matrices H_i are never stored, they are computed from its corresponding Householder vectors v_i and scalars ipiv$[i]$, as returned by *GEQRF*.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. m >= 0. The number of rows of the matrix Q.
- `[in] n`: rocblas_int. 0 <= n <= m. The number of columns of the matrix Q.
- `[in] k`: rocblas_int. 0 <= k <= n. The number of Householder reflectors.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the matrix A as returned by *GEQRF*, with the Householder vectors in the first k columns. On exit, the computed matrix Q.
- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of A.
- `[in] ipiv`: pointer to type. Array on the GPU of dimension at least k. The Householder scalars as returned by *GEQRF*.

rocsolver_<type>orgqr()

rocblas_status **rocsolver_dorgqr**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, double **A*, **const** rocblas_int *lda*, double **ipiv*)

rocblas_status **rocsolver_sorgqr**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, float **A*, **const** rocblas_int *lda*, float **ipiv*)

ORGQR generates an m-by-n Matrix Q with orthonormal columns.

(This is the blocked version of the algorithm).

The matrix Q is defined as the first n columns of the product of k Householder reflectors of order m

$$Q = H_1 H_2 \cdots H_k$$

The Householder matrices H_i are never stored, they are computed from its corresponding Householder vectors v_i and scalars ipiv$[i]$, as returned by *GEQRF*.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. m >= 0. The number of rows of the matrix Q.
- `[in] n`: rocblas_int. 0 <= n <= m. The number of columns of the matrix Q.
- `[in] k`: rocblas_int. 0 <= k <= n. The number of Householder reflectors.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the matrix A as returned by *GEQRF*, with the Householder vectors in the first k columns. On exit, the computed matrix Q.
- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of A.
- `[in] ipiv`: pointer to type. Array on the GPU of dimension at least k. The Householder scalars as returned by *GEQRF*.

rocsolver_<type>orgl2()

rocblas_status **rocsolver_dorgl2** (rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, double **A*, **const** rocblas_int *lda*, double **ipiv*)

rocblas_status **rocsolver_sorgl2** (rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, float **A*, **const** rocblas_int *lda*, float **ipiv*)

ORGL2 generates an m-by-n Matrix Q with orthonormal rows.

(This is the unblocked version of the algorithm).

The matrix Q is defined as the first m rows of the product of k Householder reflectors of order n

$$Q = H_k H_{k-1} \cdots H_1$$

The Householder matrices H_i are never stored, they are computed from its corresponding Householder vectors v_i and scalars ipiv$[i]$, as returned by *GELQF*.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. 0 <= m <= n. The number of rows of the matrix Q.
- `[in] n`: rocblas_int. n >= 0. The number of columns of the matrix Q.
- `[in] k`: rocblas_int. 0 <= k <= m. The number of Householder reflectors.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the matrix A as returned by *GELQF*, with the Householder vectors in the first k rows. On exit, the computed matrix Q.
- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of A.
- `[in] ipiv`: pointer to type. Array on the GPU of dimension at least k. The Householder scalars as returned by *GELQF*.

rocsolver_<type>orglq()

rocblas_status **rocsolver_dorglq** (rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, double **A*, **const** rocblas_int *lda*, double **ipiv*)

rocblas_status **rocsolver_sorglq** (rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, float **A*, **const** rocblas_int *lda*, float **ipiv*)

ORGLQ generates an m-by-n Matrix Q with orthonormal rows.

(This is the blocked version of the algorithm).

The matrix Q is defined as the first m rows of the product of k Householder reflectors of order n

$$Q = H_k H_{k-1} \cdots H_1$$

The Householder matrices H_i are never stored, they are computed from its corresponding Householder vectors v_i and scalars ipiv$[i]$, as returned by *GELQF*.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. 0 <= m <= n. The number of rows of the matrix Q.
- `[in] n`: rocblas_int. n >= 0. The number of columns of the matrix Q.
- `[in] k`: rocblas_int. 0 <= k <= m. The number of Householder reflectors.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the matrix A as returned by *GELQF*, with the Householder vectors in the first k rows. On exit, the computed matrix Q.
- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of A.
- `[in] ipiv`: pointer to type. Array on the GPU of dimension at least k. The Householder scalars as returned by *GELQF*.

rocsolver_<type>org2l()

rocblas_status **rocsolver_dorg2l** (rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, double **A*, **const** rocblas_int *lda*, double **ipiv*)

rocblas_status **rocsolver_sorg2l** (rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, float **A*, **const** rocblas_int *lda*, float **ipiv*)

ORG2L generates an m-by-n Matrix Q with orthonormal columns.

(This is the unblocked version of the algorithm).

The matrix Q is defined as the last n columns of the product of k Householder reflectors of order m

$$Q = H_k H_{k-1} \cdots H_1$$

The Householder matrices H_i are never stored, they are computed from its corresponding Householder vectors v_i and scalars ipiv$[i]$, as returned by *GEQLF*.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. m >= 0. The number of rows of the matrix Q.
- `[in] n`: rocblas_int. 0 <= n <= m. The number of columns of the matrix Q.
- `[in] k`: rocblas_int. 0 <= k <= n. The number of Householder reflectors.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the matrix A as returned by *GEQLF*, with the Householder vectors in the last k columns. On exit, the computed matrix Q.
- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of A.
- `[in] ipiv`: pointer to type. Array on the GPU of dimension at least k. The Householder scalars as returned by *GEQLF*.

rocsolver_<type>orgql()

rocblas_status **rocsolver_dorgql** (rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, double **A*, **const** rocblas_int *lda*, double **ipiv*)

rocblas_status **rocsolver_sorgql** (rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, float **A*, **const** rocblas_int *lda*, float **ipiv*)

ORGQL generates an m-by-n Matrix Q with orthonormal columns.

(This is the blocked version of the algorithm).

The matrix Q is defined as the last n column of the product of k Householder reflectors of order m

$$Q = H_k H_{k-1} \cdots H_1$$

The Householder matrices H_i are never stored, they are computed from its corresponding Householder vectors v_i and scalars ipiv$[i]$, as returned by *GEQLF*.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. m >= 0. The number of rows of the matrix Q.
- `[in] n`: rocblas_int. 0 <= n <= m. The number of columns of the matrix Q.
- `[in] k`: rocblas_int. 0 <= k <= n. The number of Householder reflectors.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the matrix A as returned by *GEQLF*, with the Householder vectors in the last k columns. On exit, the computed matrix Q.
- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of A.
- `[in] ipiv`: pointer to type. Array on the GPU of dimension at least k. The Householder scalars as returned by *GEQLF*.

rocsolver_<type>orgbr()

rocblas_status **rocsolver_dorgbr** (rocblas_handle *handle*, **const** *rocblas_storev* *storev*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, double **A*, **const** rocblas_int *lda*, double **ipiv*)

rocblas_status **rocsolver_sorgbr** (rocblas_handle *handle*, **const** *rocblas_storev* *storev*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, float **A*, **const** rocblas_int *lda*, float **ipiv*)

ORGBR generates an m-by-n Matrix Q with orthonormal rows or columns.

If storev is column-wise, then the matrix Q has orthonormal columns. If m >= k, Q is defined as the first n columns of the product of k Householder reflectors of order m

$$Q = H_1 H_2 \cdots H_k$$

If m < k, Q is defined as the product of Householder reflectors of order m

$$Q = H_1 H_2 \cdots H_{m-1}$$

On the other hand, if storev is row-wise, then the matrix Q has orthonormal rows. If n > k, Q is defined as the first m rows of the product of k Householder reflectors of order n

$$Q = H_k H_{k-1} \cdots H_1$$

If n <= k, Q is defined as the product of Householder reflectors of order n

$$Q = H_{n-1} H_{n-2} \cdots H_1$$

The Householder matrices H_i are never stored, they are computed from its corresponding Householder vectors v_i and scalars ipiv$[i]$, as returned by *GEBRD* in its arguments A and tauq or taup.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] storev`: *rocblas_storev*. Specifies whether to work column-wise or row-wise.
- `[in] m`: rocblas_int. m >= 0. The number of rows of the matrix Q. If row-wise, then min(n,k) <= m <= n.
- `[in] n`: rocblas_int. n >= 0. The number of columns of the matrix Q. If column-wise, then min(m,k) <= n <= m.
- `[in] k`: rocblas_int. k >= 0. The number of columns (if storev is column-wise) or rows (if row-wise) of the original matrix reduced by *GEBRD*.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the Householder vectors as returned by *GEBRD*. On exit, the computed matrix Q.
- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of A.
- `[in] ipiv`: pointer to type. Array on the GPU of dimension min(m,k) if column-wise, or min(n,k) if row-wise. The Householder scalars as returned by *GEBRD*.

rocsolver_<type>orgtr()

rocblas_status **rocsolver_dorgtr**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, double **ipiv*)

rocblas_status **rocsolver_sorgtr**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, float **ipiv*)

ORGTR generates an n-by-n orthogonal Matrix Q.

Q is defined as the product of n-1 Householder reflectors of order n. If uplo indicates upper, then Q has the form

$$Q = H_{n-1} H_{n-2} \cdots H_1$$

On the other hand, if uplo indicates lower, then Q has the form

$$Q = H_1 H_2 \cdots H_{n-1}$$

The Householder matrices H_i are never stored, they are computed from its corresponding Householder vectors v_i and scalars ipiv$[i]$, as returned by *SYTRD* in its arguments A and tau.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] uplo`: rocblas_fill. Specifies whether the *SYTRD* factorization was upper or lower triangular. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.
- `[in] n`: rocblas_int. n >= 0. The number of rows and columns of the matrix Q.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the Householder vectors as returned by *SYTRD*. On exit, the computed matrix Q.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of A.
- `[in] ipiv`: pointer to type. Array on the GPU of dimension n-1. The Householder scalars as returned by *SYTRD*.

rocsolver_<type>orm2r()

rocblas_status **rocsolver_dormr2r**(rocblas_handle *handle*, **const** rocblas_side *side*, **const** rocblas_operation *trans*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, double **A*, **const** rocblas_int *lda*, double **ipiv*, double **C*, **const** rocblas_int *ldc*)

rocblas_status **rocsolver_sorm2r**(rocblas_handle *handle*, **const** rocblas_side *side*, **const** rocblas_operation *trans*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, float **A*, **const** rocblas_int *lda*, float **ipiv*, float **C*, **const** rocblas_int *ldc*)

ORM2R multiplies a matrix Q with orthonormal columns by a general m-by-n matrix C.

(This is the unblocked version of the algorithm).

The matrix Q is applied in one of the following forms, depending on the values of side and trans:

$$\begin{array}{cl} QC & \text{No transpose from the left,} \\ Q^TC & \text{Transpose from the left,} \\ CQ & \text{No transpose from the right, and} \\ CQ^T & \text{Transpose from the right.} \end{array}$$

Q is defined as the product of k Householder reflectors

$$Q = H_1 H_2 \cdots H_k$$

of order m if applying from the left, or n if applying from the right. Q is never stored, it is calculated from the Householder vectors and scalars returned by the QR factorization *GEQRF*.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] side`: rocblas_side. Specifies from which side to apply Q.
- `[in] trans`: rocblas_operation. Specifies whether the matrix Q or its transpose is to be applied.
- `[in] m`: rocblas_int. m >= 0. Number of rows of matrix C.
- `[in] n`: rocblas_int. n >= 0. Number of columns of matrix C.
- `[in] k`: rocblas_int. k >= 0; k <= m if side is left, k <= n if side is right. The number of Householder reflectors that form Q.
- `[in] A`: pointer to type. Array on the GPU of size lda*k. The Householder vectors as returned by *GEQRF* in the first k columns of its argument A.
- `[in] lda`: rocblas_int. lda >= m if side is left, or lda >= n if side is right. Leading dimension of A.
- `[in] ipiv`: pointer to type. Array on the GPU of dimension at least k. The Householder scalars as returned by *GEQRF*.
- `[inout] C`: pointer to type. Array on the GPU of size ldc*n. On entry, the matrix C. On exit, it is overwritten with Q*C, C*Q, Q'*C, or C*Q'.
- `[in] ldc`: rocblas_int. ldc >= m. Leading dimension of C.

rocsolver_<type>ormqr()

rocblas_status **rocsolver_dormqr**(rocblas_handle *handle*, **const** rocblas_side *side*, **const** rocblas_operation *trans*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, double **A*, **const** rocblas_int *lda*, double **ipiv*, double **C*, **const** rocblas_int *ldc*)

rocblas_status **rocsolver_sormqr**(rocblas_handle *handle*, **const** rocblas_side *side*, **const** rocblas_operation *trans*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, float **A*, **const** rocblas_int *lda*, float **ipiv*, float **C*, **const** rocblas_int *ldc*)

ORMQR multiplies a matrix Q with orthonormal columns by a general m-by-n matrix C.

(This is the blocked version of the algorithm).

The matrix Q is applied in one of the following forms, depending on the values of side and trans:

$$
\begin{array}{ll}
QC & \text{No transpose from the left,} \\
Q^T C & \text{Transpose from the left,} \\
CQ & \text{No transpose from the right, and} \\
CQ^T & \text{Transpose from the right.}
\end{array}
$$

Q is defined as the product of k Householder reflectors

$$Q = H_1 H_2 \cdots H_k$$

of order m if applying from the left, or n if applying from the right. Q is never stored, it is calculated from the Householder vectors and scalars returned by the QR factorization *GEQRF*.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] side`: rocblas_side. Specifies from which side to apply Q.
- `[in] trans`: rocblas_operation. Specifies whether the matrix Q or its transpose is to be applied.
- `[in] m`: rocblas_int. m >= 0. Number of rows of matrix C.
- `[in] n`: rocblas_int. n >= 0. Number of columns of matrix C.
- `[in] k`: rocblas_int. k >= 0; k <= m if side is left, k <= n if side is right. The number of Householder reflectors that form Q.
- `[in] A`: pointer to type. Array on the GPU of size lda*k. The Householder vectors as returned by *GEQRF* in the first k columns of its argument A.
- `[in] lda`: rocblas_int. lda >= m if side is left, or lda >= n if side is right. Leading dimension of A.
- `[in] ipiv`: pointer to type. Array on the GPU of dimension at least k. The Householder scalars as returned by *GEQRF*.
- `[inout] C`: pointer to type. Array on the GPU of size ldc*n. On entry, the matrix C. On exit, it is overwritten with Q*C, C*Q, Q'*C, or C*Q'.
- `[in] ldc`: rocblas_int. ldc >= m. Leading dimension of C.

rocsolver_<type>orml2()

rocblas_status **rocsolver_dorml2**(rocblas_handle *handle*, **const** rocblas_side *side*, **const** rocblas_operation *trans*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, double **A*, **const** rocblas_int *lda*, double **ipiv*, double **C*, **const** rocblas_int *ldc*)

rocblas_status **rocsolver_sorml2**(rocblas_handle *handle*, **const** rocblas_side *side*, **const** rocblas_operation *trans*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, float **A*, **const** rocblas_int *lda*, float **ipiv*, float **C*, **const** rocblas_int *ldc*)

ORML2 multiplies a matrix Q with orthonormal rows by a general m-by-n matrix C.

(This is the unblocked version of the algorithm).

The matrix Q is applied in one of the following forms, depending on the values of side and trans:

$$\begin{array}{ll} QC & \text{No transpose from the left,} \\ Q^TC & \text{Transpose from the left,} \\ CQ & \text{No transpose from the right, and} \\ CQ^T & \text{Transpose from the right.} \end{array}$$

Q is defined as the product of k Householder reflectors

$$Q = H_k H_{k-1} \cdots H_1$$

of order m if applying from the left, or n if applying from the right. Q is never stored, it is calculated from the Householder vectors and scalars returned by the LQ factorization *GELQF*.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] side`: rocblas_side. Specifies from which side to apply Q.
- `[in] trans`: rocblas_operation. Specifies whether the matrix Q or its transpose is to be applied.
- `[in] m`: rocblas_int. m >= 0. Number of rows of matrix C.
- `[in] n`: rocblas_int. n >= 0. Number of columns of matrix C.
- `[in] k`: rocblas_int. k >= 0; k <= m if side is left, k <= n if side is right. The number of Householder reflectors that form Q.
- `[in] A`: pointer to type. Array on the GPU of size lda*m if side is left, or lda*n if side is right. The Householder vectors as returned by *GELQF* in the first k rows of its argument A.
- `[in] lda`: rocblas_int. lda >= k. Leading dimension of A.
- `[in] ipiv`: pointer to type. Array on the GPU of dimension at least k. The Householder scalars as returned by *GELQF*.
- `[inout] C`: pointer to type. Array on the GPU of size ldc*n. On entry, the matrix C. On exit, it is overwritten with Q*C, C*Q, Q'*C, or C*Q'.
- `[in] ldc`: rocblas_int. ldc >= m. Leading dimension of C.

rocsolver_<type>ormlq()

rocblas_status **rocsolver_dormlq**(rocblas_handle *handle*, **const** rocblas_side *side*, **const** rocblas_operation *trans*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, double **A*, **const** rocblas_int *lda*, double **ipiv*, double **C*, **const** rocblas_int *ldc*)

rocblas_status **rocsolver_sormlq**(rocblas_handle *handle*, **const** rocblas_side *side*, **const** rocblas_operation *trans*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, float **A*, **const** rocblas_int *lda*, float **ipiv*, float **C*, **const** rocblas_int *ldc*)

ORMLQ multiplies a matrix Q with orthonormal rows by a general m-by-n matrix C.

(This is the blocked version of the algorithm).

The matrix Q is applied in one of the following forms, depending on the values of side and trans:

$$\begin{array}{cl} QC & \text{No transpose from the left,} \\ Q^TC & \text{Transpose from the left,} \\ CQ & \text{No transpose from the right, and} \\ CQ^T & \text{Transpose from the right.} \end{array}$$

Q is defined as the product of k Householder reflectors

$$Q = H_k H_{k-1} \cdots H_1$$

of order m if applying from the left, or n if applying from the right. Q is never stored, it is calculated from the Householder vectors and scalars returned by the LQ factorization *GELQF*.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] side`: rocblas_side. Specifies from which side to apply Q.
- `[in] trans`: rocblas_operation. Specifies whether the matrix Q or its transpose is to be applied.
- `[in] m`: rocblas_int. m >= 0. Number of rows of matrix C.
- `[in] n`: rocblas_int. n >= 0. Number of columns of matrix C.
- `[in] k`: rocblas_int. k >= 0; k <= m if side is left, k <= n if side is right. The number of Householder reflectors that form Q.
- `[in] A`: pointer to type. Array on the GPU of size lda*m if side is left, or lda*n if side is right. The Householder vectors as returned by *GELQF* in the first k rows of its argument A.
- `[in] lda`: rocblas_int. lda >= k. Leading dimension of A.
- `[in] ipiv`: pointer to type. Array on the GPU of dimension at least k. The Householder scalars as returned by *GELQF*.
- `[inout] C`: pointer to type. Array on the GPU of size ldc*n. On entry, the matrix C. On exit, it is overwritten with Q*C, C*Q, Q'*C, or C*Q'.
- `[in] ldc`: rocblas_int. ldc >= m. Leading dimension of C.

rocsolver_<type>orm2l()

rocblas_status **rocsolver_dorm2l** (rocblas_handle *handle*, **const** rocblas_side *side*, **const** rocblas_operation *trans*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, double **A*, **const** rocblas_int *lda*, double **ipiv*, double **C*, **const** rocblas_int *ldc*)

rocblas_status **rocsolver_sorm2l** (rocblas_handle *handle*, **const** rocblas_side *side*, **const** rocblas_operation *trans*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, float **A*, **const** rocblas_int *lda*, float **ipiv*, float **C*, **const** rocblas_int *ldc*)

ORM2L multiplies a matrix Q with orthonormal columns by a general m-by-n matrix C.

(This is the unblocked version of the algorithm).

The matrix Q is applied in one of the following forms, depending on the values of side and trans:

$$\begin{array}{cl} QC & \text{No transpose from the left,} \\ Q^TC & \text{Transpose from the left,} \\ CQ & \text{No transpose from the right, and} \\ CQ^T & \text{Transpose from the right.} \end{array}$$

Q is defined as the product of k Householder reflectors

$$Q = H_k H_{k-1} \cdots H_1$$

of order m if applying from the left, or n if applying from the right. Q is never stored, it is calculated from the Householder vectors and scalars returned by the QL factorization *GEQLF*.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] side`: rocblas_side. Specifies from which side to apply Q.
- `[in] trans`: rocblas_operation. Specifies whether the matrix Q or its transpose is to be applied.
- `[in] m`: rocblas_int. m >= 0. Number of rows of matrix C.
- `[in] n`: rocblas_int. n >= 0. Number of columns of matrix C.
- `[in] k`: rocblas_int. k >= 0; k <= m if side is left, k <= n if side is right. The number of Householder reflectors that form Q.
- `[in] A`: pointer to type. Array on the GPU of size lda*k. The Householder vectors as returned by *GEQLF* in the last k columns of its argument A.
- `[in] lda`: rocblas_int. lda >= m if side is left, lda >= n if side is right. Leading dimension of A.
- `[in] ipiv`: pointer to type. Array on the GPU of dimension at least k. The Householder scalars as returned by *GEQLF*.
- `[inout] C`: pointer to type. Array on the GPU of size ldc*n. On entry, the matrix C. On exit, it is overwritten with Q*C, C*Q, Q'*C, or C*Q'.
- `[in] ldc`: rocblas_int. ldc >= m. Leading dimension of C.

rocsolver_<type>ormql()

rocblas_status **rocsolver_dormql**(rocblas_handle *handle*, **const** rocblas_side *side*, **const** rocblas_operation *trans*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, double **A*, **const** rocblas_int *lda*, double **ipiv*, double **C*, **const** rocblas_int *ldc*)

rocblas_status **rocsolver_sormql**(rocblas_handle *handle*, **const** rocblas_side *side*, **const** rocblas_operation *trans*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, float **A*, **const** rocblas_int *lda*, float **ipiv*, float **C*, **const** rocblas_int *ldc*)

ORMQL multiplies a matrix Q with orthonormal columns by a general m-by-n matrix C.

(This is the blocked version of the algorithm).

The matrix Q is applied in one of the following forms, depending on the values of side and trans:

$$\begin{array}{ll} QC & \text{No transpose from the left,} \\ Q^T C & \text{Transpose from the left,} \\ CQ & \text{No transpose from the right, and} \\ CQ^T & \text{Transpose from the right.} \end{array}$$

Q is defined as the product of k Householder reflectors

$$Q = H_k H_{k-1} \cdots H_1$$

of order m if applying from the left, or n if applying from the right. Q is never stored, it is calculated from the Householder vectors and scalars returned by the QL factorization *GEQLF*.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] side`: rocblas_side. Specifies from which side to apply Q.
- `[in] trans`: rocblas_operation. Specifies whether the matrix Q or its transpose is to be applied.
- `[in] m`: rocblas_int. m >= 0. Number of rows of matrix C.
- `[in] n`: rocblas_int. n >= 0. Number of columns of matrix C.
- `[in] k`: rocblas_int. k >= 0; k <= m if side is left, k <= n if side is right. The number of Householder reflectors that form Q.
- `[in] A`: pointer to type. Array on the GPU of size lda*k. The Householder vectors as returned by *GEQLF* in the last k columns of its argument A.
- `[in] lda`: rocblas_int. lda >= m if side is left, lda >= n if side is right. Leading dimension of A.
- `[in] ipiv`: pointer to type. Array on the GPU of dimension at least k. The Householder scalars as returned by *GEQLF*.
- `[inout] C`: pointer to type. Array on the GPU of size ldc*n. On entry, the matrix C. On exit, it is overwritten with Q*C, C*Q, Q'*C, or C*Q'.
- `[in] ldc`: rocblas_int. ldc >= m. Leading dimension of C.

rocsolver_<type>ormbr()

rocblas_status **rocsolver_dormbr**(rocblas_handle *handle*, **const** *rocblas_storev* *storev*, **const** rocblas_side *side*, **const** rocblas_operation *trans*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, double **A*, **const** rocblas_int *lda*, double **ipiv*, double **C*, **const** rocblas_int *ldc*)

rocblas_status **rocsolver_sormbr**(rocblas_handle *handle*, **const** *rocblas_storev* *storev*, **const** rocblas_side *side*, **const** rocblas_operation *trans*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, float **A*, **const** rocblas_int *lda*, float **ipiv*, float **C*, **const** rocblas_int *ldc*)

ORMBR multiplies a matrix Q with orthonormal rows or columns by a general m-by-n matrix C.

If storev is column-wise, then the matrix Q has orthonormal columns. If storev is row-wise, then the matrix Q has orthonormal rows. The matrix Q is applied in one of the following forms, depending on the values of side and trans:

$$\begin{array}{cl} QC & \text{No transpose from the left,} \\ Q^TC & \text{Transpose from the left,} \\ CQ & \text{No transpose from the right, and} \\ CQ^T & \text{Transpose from the right.} \end{array}$$

The order q of the orthogonal matrix Q is q = m if applying from the left, or q = n if applying from the right.

When storev is column-wise, if q >= k, then Q is defined as the product of k Householder reflectors

$$Q = H_1H_2\cdots H_k,$$

and if q < k, then Q is defined as the product

$$Q = H_1 H_2 \cdots H_{q-1}.$$

When storev is row-wise, if q > k, then Q is defined as the product of k Householder reflectors

$$Q = H_1 H_2 \cdots H_k,$$

and if q <= k, Q is defined as the product

$$Q = H_1 H_2 \cdots H_{q-1}.$$

The Householder matrices H_i are never stored, they are computed from its corresponding Householder vectors and scalars as returned by *GEBRD* in its arguments A and tauq or taup.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] storev`: *rocblas_storev*. Specifies whether to work column-wise or row-wise.
- `[in] side`: rocblas_side. Specifies from which side to apply Q.
- `[in] trans`: rocblas_operation. Specifies whether the matrix Q or its transpose is to be applied.
- `[in] m`: rocblas_int. m >= 0. Number of rows of matrix C.
- `[in] n`: rocblas_int. n >= 0. Number of columns of matrix C.
- `[in] k`: rocblas_int. k >= 0. The number of columns (if storev is column-wise) or rows (if row-wise) of the original matrix reduced by *GEBRD*.
- `[in] A`: pointer to type. Array on the GPU of size lda*min(q,k) if column-wise, or lda*q if row-wise. The Householder vectors as returned by *GEBRD*.
- `[in] lda`: rocblas_int. lda >= q if column-wise, or lda >= min(q,k) if row-wise. Leading dimension of A.
- `[in] ipiv`: pointer to type. Array on the GPU of dimension at least min(q,k). The Householder scalars as returned by *GEBRD*.
- `[inout] C`: pointer to type. Array on the GPU of size ldc*n. On entry, the matrix C. On exit, it is overwritten with Q*C, C*Q, Q'*C, or C*Q'.
- `[in] ldc`: rocblas_int. ldc >= m. Leading dimension of C.

rocsolver_<type>ormtr()

rocblas_status **rocsolver_dormtr**(rocblas_handle *handle*, **const** rocblas_side *side*, **const** rocblas_fill *uplo*, **const** rocblas_operation *trans*, **const** rocblas_int *m*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, double **ipiv*, double **C*, **const** rocblas_int *ldc*) rocblas_status **rocsolver_sormtr**(rocblas_handle *handle*, **const** rocblas_side *side*, **const** rocblas_fill *uplo*, **const** rocblas_operation *trans*, **const** rocblas_int *m*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, float **ipiv*, float **C*, **const** rocblas_int *ldc*)

ORMTR multiplies an orthogonal matrix Q by a general m-by-n matrix C.

The matrix Q is applied in one of the following forms, depending on the values of side and trans:

QC	No transpose from the left,
Q^TC	Transpose from the left,
CQ	No transpose from the right, and
CQ^T	Transpose from the right.

The order q of the orthogonal matrix Q is q = m if applying from the left, or q = n if applying from the right.

Q is defined as a product of q-1 Householder reflectors. If uplo indicates upper, then Q has the form

$$Q = H_{q-1}H_{q-2}\cdots H_1.$$

On the other hand, if uplo indicates lower, then Q has the form

$$Q = H_1H_2\cdots H_{q-1}$$

The Householder matrices H_i are never stored, they are computed from its corresponding Householder vectors and scalars as returned by *SYTRD* in its arguments A and tau.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] side`: rocblas_side. Specifies from which side to apply Q.
- `[in] uplo`: rocblas_fill. Specifies whether the *SYTRD* factorization was upper or lower triangular. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.
- `[in] trans`: rocblas_operation. Specifies whether the matrix Q or its transpose is to be applied.
- `[in] m`: rocblas_int. m >= 0. Number of rows of matrix C.
- `[in] n`: rocblas_int. n >= 0. Number of columns of matrix C.
- `[in] A`: pointer to type. Array on the GPU of size lda*q. On entry, the Householder vectors as returned by *SYTRD*.
- `[in] lda`: rocblas_int. lda >= q. Leading dimension of A.
- `[in] ipiv`: pointer to type. Array on the GPU of dimension at least q-1. The Householder scalars as returned by *SYTRD*.
- `[inout] C`: pointer to type. Array on the GPU of size ldc*n. On entry, the matrix C. On exit, it is overwritten with Q*C, C*Q, Q'*C, or C*Q'.
- `[in] ldc`: rocblas_int. ldc >= m. Leading dimension of C.

3.2.7 Unitary matrices

List of functions for unitary matrices

- *rocsolver_<type>ung2r()*
- *rocsolver_<type>ungqr()*
- *rocsolver_<type>ungl2()*
- *rocsolver_<type>unglq()*
- *rocsolver_<type>ung2l()*
- *rocsolver_<type>ungql()*
- *rocsolver_<type>ungbr()*
- *rocsolver_<type>ungtr()*
- *rocsolver_<type>unm2r()*
- *rocsolver_<type>unmqr()*
- *rocsolver_<type>unml2()*
- *rocsolver_<type>unmlq()*
- *rocsolver_<type>unm2l()*
- *rocsolver_<type>unmql()*
- *rocsolver_<type>unmbr()*
- *rocsolver_<type>unmtr()*

rocsolver_<type>ung2r()

rocblas_status **rocsolver_zung2r** (rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, rocblas_double_complex **A*, **const** rocblas_int *lda*, rocblas_double_complex **ipiv*)

rocblas_status **rocsolver_cung2r** (rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, rocblas_float_complex **A*, **const** rocblas_int *lda*, rocblas_float_complex **ipiv*)

UNG2R generates an m-by-n complex Matrix Q with orthonormal columns.

(This is the unblocked version of the algorithm).

The matrix Q is defined as the first n columns of the product of k Householder reflectors of order m

$$Q = H_1 H_2 \cdots H_k$$

The Householder matrices H_i are never stored, they are computed from its corresponding Householder vectors v_i and scalars ipiv$[i]$, as returned by *GEQRF*.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. m >= 0. The number of rows of the matrix Q.
- `[in] n`: rocblas_int. 0 <= n <= m. The number of columns of the matrix Q.
- `[in] k`: rocblas_int. 0 <= k <= n. The number of Householder reflectors.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the matrix A as returned by *GEQRF*, with the Householder vectors in the first k columns. On exit, the computed matrix Q.
- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of A.
- `[in] ipiv`: pointer to type. Array on the GPU of dimension at least k. The Householder scalars as returned by *GEQRF*.

rocsolver_<type>ungqr()

rocblas_status **rocsolver_zungqr** (rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, rocblas_double_complex **A*, **const** rocblas_int *lda*, rocblas_double_complex **ipiv*)

rocblas_status **rocsolver_cungqr** (rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, rocblas_float_complex **A*, **const** rocblas_int *lda*, rocblas_float_complex **ipiv*)

UNGQR generates an m-by-n complex Matrix Q with orthonormal columns.

(This is the blocked version of the algorithm).

The matrix Q is defined as the first n columns of the product of k Householder reflectors of order m

$$Q = H_1 H_2 \cdots H_k$$

Householder matrices H_i are never stored, they are computed from its corresponding Householder vectors v_i and scalars ipiv$[i]$, as returned by *GEQRF*.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. m >= 0. The number of rows of the matrix Q.
- `[in] n`: rocblas_int. 0 <= n <= m. The number of columns of the matrix Q.
- `[in] k`: rocblas_int. 0 <= k <= n. The number of Householder reflectors.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the matrix A as returned by *GEQRF*, with the Householder vectors in the first k columns. On exit, the computed matrix Q.
- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of A.
- `[in] ipiv`: pointer to type. Array on the GPU of dimension at least k. The Householder scalars as returned by *GEQRF*.

rocsolver_<type>ungl2()

rocblas_status **rocsolver_zungl2** (rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, rocblas_double_complex **A*, **const** rocblas_int *lda*, rocblas_double_complex **ipiv*)

rocblas_status **rocsolver_cungl2** (rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, rocblas_float_complex **A*, **const** rocblas_int *lda*, rocblas_float_complex **ipiv*)

UNGL2 generates an m-by-n complex Matrix Q with orthonormal rows.

(This is the unblocked version of the algorithm).

The matrix Q is defined as the first m rows of the product of k Householder reflectors of order n

$$Q = H_k^H H_{k-1}^H \cdots H_1^H$$

The Householder matrices H_i are never stored, they are computed from its corresponding Householder vectors v_i and scalars ipiv$[i]$, as returned by *GELQF*.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. 0 <= m <= n. The number of rows of the matrix Q.
- `[in] n`: rocblas_int. n >= 0. The number of columns of the matrix Q.
- `[in] k`: rocblas_int. 0 <= k <= m. The number of Householder reflectors.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the matrix A as returned by *GELQF*, with the Householder vectors in the first k rows. On exit, the computed matrix Q.
- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of A.
- `[in] ipiv`: pointer to type. Array on the GPU of dimension at least k. The Householder scalars as returned by *GELQF*.

rocsolver_<type>unglq()

rocblas_status **rocsolver_zunglq** (rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, rocblas_double_complex **A*, **const** rocblas_int *lda*, rocblas_double_complex **ipiv*)

rocblas_status **rocsolver_cunglq** (rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, rocblas_float_complex **A*, **const** rocblas_int *lda*, rocblas_float_complex **ipiv*)

UNGLQ generates an m-by-n complex Matrix Q with orthonormal rows.

(This is the blocked version of the algorithm).

The matrix Q is defined as the first m rows of the product of k Householder reflectors of order n

$$Q = H_k^H H_{k-1}^H \cdots H_1^H$$

The Householder matrices H_i are never stored, they are computed from its corresponding Householder vectors v_i and scalars ipiv$[i]$, as returned by *GELQF*.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. 0 <= m <= n. The number of rows of the matrix Q.
- `[in] n`: rocblas_int. n >= 0. The number of columns of the matrix Q.
- `[in] k`: rocblas_int. 0 <= k <= m. The number of Householder reflectors.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the matrix A as returned by *GELQF*, with the Householder vectors in the first k rows. On exit, the computed matrix Q.
- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of A.
- `[in] ipiv`: pointer to type. Array on the GPU of dimension at least k. The Householder scalars as returned by *GELQF*.

rocsolver_<type>ung2l()

rocblas_status **rocsolver_zung2l** (rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, rocblas_double_complex **A*, **const** rocblas_int *lda*, rocblas_double_complex **ipiv*)

rocblas_status **rocsolver_cung2l** (rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, rocblas_float_complex **A*, **const** rocblas_int *lda*, rocblas_float_complex **ipiv*)

UNG2L generates an m-by-n complex Matrix Q with orthonormal columns.

(This is the unblocked version of the algorithm).

The matrix Q is defined as the last n columns of the product of k Householder reflectors of order m

$$Q = H_k H_{k-1} \cdots H_1$$

The Householder matrices H_i are never stored, they are computed from its corresponding Householder vectors v_i and scalars ipiv$[i]$, as returned by *GEQLF*.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. m >= 0. The number of rows of the matrix Q.
- `[in] n`: rocblas_int. 0 <= n <= m. The number of columns of the matrix Q.
- `[in] k`: rocblas_int. 0 <= k <= n. The number of Householder reflectors.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the matrix A as returned by *GEQLF*, with the Householder vectors in the last k columns. On exit, the computed matrix Q.
- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of A.

- `[in] ipiv`: pointer to type. Array on the GPU of dimension at least k. The Householder scalars as returned by *GEQLF*.

rocsolver_<type>ungql()

rocblas_status **rocsolver_zungql** (rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, rocblas_double_complex **A*, **const** rocblas_int *lda*, rocblas_double_complex **ipiv*)

rocblas_status **rocsolver_cungql** (rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, rocblas_float_complex **A*, **const** rocblas_int *lda*, rocblas_float_complex **ipiv*)

UNGQL generates an m-by-n complex Matrix Q with orthonormal columns.

(This is the blocked version of the algorithm).

The matrix Q is defined as the last n columns of the product of k Householder reflectors of order m

$$Q = H_k H_{k-1} \cdots H_1$$

The Householder matrices H_i are never stored, they are computed from its corresponding Householder vectors v_i and scalars ipiv$[i]$, as returned by *GEQLF*.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. m >= 0. The number of rows of the matrix Q.
- `[in] n`: rocblas_int. 0 <= n <= m. The number of columns of the matrix Q.
- `[in] k`: rocblas_int. 0 <= k <= n. The number of Householder reflectors.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the matrix A as returned by *GEQLF*, with the Householder vectors in the last k columns. On exit, the computed matrix Q.
- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of A.
- `[in] ipiv`: pointer to type. Array on the GPU of dimension at least k. The Householder scalars as returned by *GEQLF*.

rocsolver_<type>ungbr()

rocblas_status **rocsolver_zungbr** (rocblas_handle *handle*, **const** *rocblas_storev* *storev*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, rocblas_double_complex **A*, **const** rocblas_int *lda*, rocblas_double_complex **ipiv*)

rocblas_status **rocsolver_cungbr** (rocblas_handle *handle*, **const** *rocblas_storev* *storev*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, rocblas_float_complex **A*, **const** rocblas_int *lda*, rocblas_float_complex **ipiv*)

UNGBR generates an m-by-n complex Matrix Q with orthonormal rows or columns.

If storev is column-wise, then the matrix Q has orthonormal columns. If m >= k, Q is defined as the first n columns of the product of k Householder reflectors of order m

$$Q = H_1 H_2 \cdots H_k$$

If m < k, Q is defined as the product of Householder reflectors of order m

$$Q = H_1 H_2 \cdots H_{m-1}$$

On the other hand, if storev is row-wise, then the matrix Q has orthonormal rows. If n > k, Q is defined as the first m rows of the product of k Householder reflectors of order n

$$Q = H_k H_{k-1} \cdots H_1$$

If n <= k, Q is defined as the product of Householder reflectors of order n

$$Q = H_{n-1} H_{n-2} \cdots H_1$$

The Householder matrices H_i are never stored, they are computed from its corresponding Householder vectors v_i and scalars ipiv$[i]$, as returned by *GEBRD* in its arguments A and tauq or taup.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] storev`: *rocblas_storev*. Specifies whether to work column-wise or row-wise.
- `[in] m`: rocblas_int. m >= 0. The number of rows of the matrix Q. If row-wise, then min(n,k) <= m <= n.
- `[in] n`: rocblas_int. n >= 0. The number of columns of the matrix Q. If column-wise, then min(m,k) <= n <= m.
- `[in] k`: rocblas_int. k >= 0. The number of columns (if storev is column-wise) or rows (if row-wise) of the original matrix reduced by *GEBRD*.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the Householder vectors as returned by *GEBRD*. On exit, the computed matrix Q.
- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of A.
- `[in] ipiv`: pointer to type. Array on the GPU of dimension min(m,k) if column-wise, or min(n,k) if row-wise. The Householder scalars as returned by *GEBRD*.

rocsolver_<type>ungtr()

rocblas_status **`rocsolver_zungtr`** (rocblas_handle *handle*, **`const`** rocblas_fill *uplo*, **`const`** rocblas_int *n*, rocblas_double_complex **A*, **`const`** rocblas_int *lda*, rocblas_double_complex **ipiv*) rocblas_status **rocsolver_cungtr**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, rocblas_float_complex **ipiv*)

UNGTR generates an n-by-n unitary Matrix Q.

Q is defined as the product of n-1 Householder reflectors of order n. If uplo indicates upper, then Q has the form

$$Q = H_{n-1}H_{n-2}\cdots H_1$$

On the other hand, if uplo indicates lower, then Q has the form

$$Q = H_1H_2\cdots H_{n-1}$$

The Householder matrices H_i are never stored, they are computed from its corresponding Householder vectors v_i and scalars ipiv$[i]$, as returned by *HETRD* in its arguments A and tau.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] uplo`: rocblas_fill. Specifies whether the *HETRD* factorization was upper or lower triangular. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.
- `[in] n`: rocblas_int. n >= 0. The number of rows and columns of the matrix Q.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the Householder vectors as returned by *HETRD*. On exit, the computed matrix Q.
- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of A.
- `[in] ipiv`: pointer to type. Array on the GPU of dimension n-1. The Householder scalars as returned by *HETRD*.

rocsolver_<type>unm2r()

rocblas_status **rocsolver_zunm2r**(rocblas_handle *handle*, **const** rocblas_side *side*, **const** rocblas_operation *trans*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, rocblas_double_complex **A*, **const** rocblas_int *lda*, rocblas_double_complex **ipiv*, rocblas_double_complex **C*, **const** rocblas_int *ldc*)

rocblas_status **rocsolver_cunm2r**(rocblas_handle *handle*, **const** rocblas_side *side*, **const** rocblas_operation *trans*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, rocblas_float_complex **A*, **const** rocblas_int *lda*, rocblas_float_complex **ipiv*, rocblas_float_complex **C*, **const** rocblas_int *ldc*)

UNM2R multiplies a complex matrix Q with orthonormal columns by a general m-by-n matrix C.

(This is the unblocked version of the algorithm).

The matrix Q is applied in one of the following forms, depending on the values of side and trans:

QC	No transpose from the left,
$Q^H C$	Conjugate transpose from the left,
CQ	No transpose from the right, and
CQ^H	Conjugate transpose from the right.

Q is defined as the product of k Householder reflectors

$$Q = H_1 H_2 \cdots H_k$$

of order m if applying from the left, or n if applying from the right. Q is never stored, it is calculated from the Householder vectors and scalars returned by the QR factorization *GEQRF*.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] side`: rocblas_side. Specifies from which side to apply Q.
- `[in] trans`: rocblas_operation. Specifies whether the matrix Q or its conjugate transpose is to be applied.
- `[in] m`: rocblas_int. m >= 0. Number of rows of matrix C.
- `[in] n`: rocblas_int. n >= 0. Number of columns of matrix C.
- `[in] k`: rocblas_int. k >= 0; k <= m if side is left, k <= n if side is right. The number of Householder reflectors that form Q.
- `[in] A`: pointer to type. Array on the GPU of size lda*k. The Householder vectors as returned by *GEQRF* in the first k columns of its argument A.
- `[in] lda`: rocblas_int. lda >= m if side is left, or lda >= n if side is right. Leading dimension of A.
- `[in] ipiv`: pointer to type. Array on the GPU of dimension at least k. The Householder scalars as returned by *GEQRF*.
- `[inout] C`: pointer to type. Array on the GPU of size ldc*n. On entry, the matrix C. On exit, it is overwritten with Q*C, C*Q, Q'*C, or C*Q'.
- `[in] ldc`: rocblas_int. ldc >= m. Leading dimension of C.

rocsolver_<type>unmqr()

rocblas_status **rocsolver_zunmqr**(rocblas_handle *handle*, **const** rocblas_side *side*, **const** rocblas_operation *trans*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, rocblas_double_complex **A*, **const** rocblas_int *lda*, rocblas_double_complex **ipiv*, rocblas_double_complex **C*, **const** rocblas_int *ldc*)

rocblas_status **rocsolver_cunmqr**(rocblas_handle *handle*, **const** rocblas_side *side*, **const** rocblas_operation *trans*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, rocblas_float_complex **A*, **const** rocblas_int *lda*, rocblas_float_complex **ipiv*, rocblas_float_complex **C*, **const** rocblas_int *ldc*)

UNMQR multiplies a complex matrix Q with orthonormal columns by a general m-by-n matrix C.

(This is the blocked version of the algorithm).

The matrix Q is applied in one of the following forms, depending on the values of side and trans:

$$\begin{array}{cl} QC & \text{No transpose from the left,} \\ Q^H C & \text{Conjugate transpose from the left,} \\ CQ & \text{No transpose from the right, and} \\ CQ^H & \text{Conjugate transpose from the right.} \end{array}$$

Q is defined as the product of k Householder reflectors

$$Q = H_1 H_2 \cdots H_k$$

of order m if applying from the left, or n if applying from the right. Q is never stored, it is calculated from the Householder vectors and scalars returned by the QR factorization *GEQRF*.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] side`: rocblas_side. Specifies from which side to apply Q.
- `[in] trans`: rocblas_operation. Specifies whether the matrix Q or its conjugate transpose is to be applied.
- `[in] m`: rocblas_int. m >= 0. Number of rows of matrix C.
- `[in] n`: rocblas_int. n >= 0. Number of columns of matrix C.
- `[in] k`: rocblas_int. k >= 0; k <= m if side is left, k <= n if side is right. The number of Householder reflectors that form Q.
- `[in] A`: pointer to type. Array on the GPU of size lda*k. The Householder vectors as returned by *GEQRF* in the first k columns of its argument A.
- `[in] lda`: rocblas_int. lda >= m if side is left, or lda >= n if side is right. Leading dimension of A.
- `[in] ipiv`: pointer to type. Array on the GPU of dimension at least k. The Householder scalars as returned by *GEQRF*.
- `[inout] C`: pointer to type. Array on the GPU of size ldc*n. On entry, the matrix C. On exit, it is overwritten with Q*C, C*Q, Q'*C, or C*Q'.
- `[in] ldc`: rocblas_int. ldc >= m. Leading dimension of C.

rocsolver_<type>unml2()

rocblas_status **rocsolver_zunml2**(rocblas_handle *handle*, **const** rocblas_side *side*, **const** rocblas_operation *trans*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, rocblas_double_complex **A*, **const** rocblas_int *lda*, rocblas_double_complex **ipiv*, rocblas_double_complex **C*, **const** rocblas_int *ldc*) rocblas_status **rocsolver_cunml2**(rocblas_handle *handle*, **const** rocblas_side *side*, **const** rocblas_operation *trans*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, rocblas_float_complex **A*, **const** rocblas_int *lda*, rocblas_float_complex **ipiv*, rocblas_float_complex **C*, **const** rocblas_int *ldc*)

UNML2 multiplies a complex matrix Q with orthonormal rows by a general m-by-n matrix C.

(This is the unblocked version of the algorithm).

The matrix Q is applied in one of the following forms, depending on the values of side and trans:

$$\begin{array}{cl} QC & \text{No transpose from the left,} \\ Q^H C & \text{Conjugate transpose from the left,} \\ CQ & \text{No transpose from the right, and} \\ CQ^H & \text{Conjugate transpose from the right.} \end{array}$$

Q is defined as the product of k Householder reflectors

$$Q = H_k^H H_{k-1}^H \cdots H_1^H$$

of order m if applying from the left, or n if applying from the right. Q is never stored, it is calculated from the Householder vectors and scalars returned by the LQ factorization *GELQF*.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] side`: rocblas_side. Specifies from which side to apply Q.
- `[in] trans`: rocblas_operation. Specifies whether the matrix Q or its conjugate transpose is to be applied.
- `[in] m`: rocblas_int. m >= 0. Number of rows of matrix C.
- `[in] n`: rocblas_int. n >= 0. Number of columns of matrix C.
- `[in] k`: rocblas_int. k >= 0; k <= m if side is left, k <= n if side is right. The number of Householder reflectors that form Q.
- `[in] A`: pointer to type. Array on the GPU of size lda*m if side is left, or lda*n if side is right. The Householder vectors as returned by *GELQF* in the first k rows of its argument A.
- `[in] lda`: rocblas_int. lda >= k. Leading dimension of A.
- `[in] ipiv`: pointer to type. Array on the GPU of dimension at least k. The Householder scalars as returned by *GELQF*.
- `[inout] C`: pointer to type. Array on the GPU of size ldc*n. On entry, the matrix C. On exit, it is overwritten with Q*C, C*Q, Q'*C, or C*Q'.
- `[in] ldc`: rocblas_int. ldc >= m. Leading dimension of C.

rocsolver_<type>unmlq()

rocblas_status **rocsolver_zunmlq**(rocblas_handle *handle*, **const** rocblas_side *side*, **const** rocblas_operation *trans*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, rocblas_double_complex **A*, **const** rocblas_int *lda*, rocblas_double_complex **ipiv*, rocblas_double_complex **C*, **const** rocblas_int *ldc*)

rocblas_status **rocsolver_cunmlq**(rocblas_handle *handle*, **const** rocblas_side *side*, **const** rocblas_operation *trans*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, rocblas_float_complex **A*, **const** rocblas_int *lda*, rocblas_float_complex **ipiv*, rocblas_float_complex **C*, **const** rocblas_int *ldc*)

UNMLQ multiplies a complex matrix Q with orthonormal rows by a general m-by-n matrix C.

(This is the blocked version of the algorithm).

The matrix Q is applied in one of the following forms, depending on the values of side and trans:

$$\begin{array}{cl} QC & \text{No transpose from the left,} \\ Q^HC & \text{Conjugate transpose from the left,} \\ CQ & \text{No transpose from the right, and} \\ CQ^H & \text{Conjugate transpose from the right.} \end{array}$$

Q is defined as the product of k Householder reflectors

$$Q = H_k^H H_{k-1}^H \cdots H_1^H$$

of order m if applying from the left, or n if applying from the right. Q is never stored, it is calculated from the Householder vectors and scalars returned by the LQ factorization *GELQF*.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] side`: rocblas_side. Specifies from which side to apply Q.
- `[in] trans`: rocblas_operation. Specifies whether the matrix Q or its conjugate transpose is to be applied.
- `[in] m`: rocblas_int. m >= 0. Number of rows of matrix C.
- `[in] n`: rocblas_int. n >= 0. Number of columns of matrix C.
- `[in] k`: rocblas_int. k >= 0; k <= m if side is left, k <= n if side is right. The number of Householder reflectors that form Q.
- `[in] A`: pointer to type. Array on the GPU of size lda*m if side is left, or lda*n if side is right. The Householder vectors as returned by *GELQF* in the first k rows of its argument A.
- `[in] lda`: rocblas_int. lda >= k. Leading dimension of A.
- `[in] ipiv`: pointer to type. Array on the GPU of dimension at least k. The Householder scalars as returned by *GELQF*.
- `[inout] C`: pointer to type. Array on the GPU of size ldc*n. On entry, the matrix C. On exit, it is overwritten with Q*C, C*Q, Q'*C, or C*Q'.
- `[in] ldc`: rocblas_int. ldc >= m. Leading dimension of C.

rocsolver_<type>unm2l()

rocblas_status **rocsolver_zunm2l** (rocblas_handle *handle*, **const** rocblas_side *side*, **const** rocblas_operation *trans*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, rocblas_double_complex **A*, **const** rocblas_int *lda*, rocblas_double_complex **ipiv*, rocblas_double_complex **C*, **const** rocblas_int *ldc*)

rocblas_status **rocsolver_cunm2l** (rocblas_handle *handle*, **const** rocblas_side *side*, **const** rocblas_operation *trans*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, rocblas_float_complex **A*, **const** rocblas_int *lda*, rocblas_float_complex **ipiv*, rocblas_float_complex **C*, **const** rocblas_int *ldc*)

UNM2L multiplies a complex matrix Q with orthonormal columns by a general m-by-n matrix C.

(This is the unblocked version of the algorithm).

The matrix Q is applied in one of the following forms, depending on the values of side and trans:

$$\begin{array}{cl} QC & \text{No transpose from the left,} \\ Q^HC & \text{Conjugate transpose from the left,} \\ CQ & \text{No transpose from the right, and} \\ CQ^H & \text{Conjugate transpose from the right.} \end{array}$$

Q is defined as the product of k Householder reflectors

$$Q = H_k H_{k-1} \cdots H_1$$

of order m if applying from the left, or n if applying from the right. Q is never stored, it is calculated from the Householder vectors and scalars returned by the QL factorization *GEQLF*.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] side`: rocblas_side. Specifies from which side to apply Q.
- `[in] trans`: rocblas_operation. Specifies whether the matrix Q or its conjugate transpose is to be applied.
- `[in] m`: rocblas_int. m >= 0. Number of rows of matrix C.
- `[in] n`: rocblas_int. n >= 0. Number of columns of matrix C.
- `[in] k`: rocblas_int. k >= 0; k <= m if side is left, k <= n if side is right. The number of Householder reflectors that form Q.
- `[in] A`: pointer to type. Array on the GPU of size lda*k. The Householder vectors as returned by *GEQLF* in the last k columns of its argument A.
- `[in] lda`: rocblas_int. lda >= m if side is left, lda >= n if side is right. Leading dimension of A.
- `[in] ipiv`: pointer to type. Array on the GPU of dimension at least k. The Householder scalars as returned by *GEQLF*.
- `[inout] C`: pointer to type. Array on the GPU of size ldc*n. On entry, the matrix C. On exit, it is overwritten with Q*C, C*Q, Q'*C, or C*Q'.
- `[in] ldc`: rocblas_int. ldc >= m. Leading dimension of C.

rocsolver_<type>unmql()

rocblas_status **rocsolver_zunmql** (rocblas_handle *handle*, **const** rocblas_side *side*, **const** rocblas_operation *trans*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, rocblas_double_complex **A*, **const** rocblas_int *lda*, rocblas_double_complex **ipiv*, rocblas_double_complex **C*, **const** rocblas_int *ldc*)

rocblas_status **rocsolver_cunmql** (rocblas_handle *handle*, **const** rocblas_side *side*, **const** rocblas_operation *trans*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, rocblas_float_complex **A*, **const** rocblas_int *lda*, rocblas_float_complex **ipiv*, rocblas_float_complex **C*, **const** rocblas_int *ldc*)

UNMQL multiplies a complex matrix Q with orthonormal columns by a general m-by-n matrix C.

(This is the blocked version of the algorithm).

The matrix Q is applied in one of the following forms, depending on the values of side and trans:

$$
\begin{array}{ll}
QC & \text{No transpose from the left,} \\
Q^H C & \text{Conjugate transpose from the left,} \\
CQ & \text{No transpose from the right, and} \\
CQ^H & \text{Conjugate transpose from the right.}
\end{array}
$$

Q is defined as the product of k Householder reflectors

$$Q = H_k H_{k-1} \cdots H_1$$

of order m if applying from the left, or n if applying from the right. Q is never stored, it is calculated from the Householder vectors and scalars returned by the QL factorization *GEQLF*.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] side`: rocblas_side. Specifies from which side to apply Q.
- `[in] trans`: rocblas_operation. Specifies whether the matrix Q or its conjugate transpose is to be applied.
- `[in] m`: rocblas_int. m >= 0. Number of rows of matrix C.
- `[in] n`: rocblas_int. n >= 0. Number of columns of matrix C.
- `[in] k`: rocblas_int. k >= 0; k <= m if side is left, k <= n if side is right. The number of Householder reflectors that form Q.
- `[in] A`: pointer to type. Array on the GPU of size lda*k. The Householder vectors as returned by *GEQLF* in the last k columns of its argument A.
- `[in] lda`: rocblas_int. lda >= m if side is left, lda >= n if side is right. Leading dimension of A.
- `[in] ipiv`: pointer to type. Array on the GPU of dimension at least k. The Householder scalars as returned by *GEQLF*.
- `[inout] C`: pointer to type. Array on the GPU of size ldc*n. On entry, the matrix C. On exit, it is overwritten with Q*C, C*Q, Q'*C, or C*Q'.
- `[in] ldc`: rocblas_int. ldc >= m. Leading dimension of C.

rocsolver_<type>unmbr()

rocblas_status **rocsolver_zunmbr**(rocblas_handle *handle*, **const** *rocblas_storev* *storev*, **const** rocblas_side *side*, **const** rocblas_operation *trans*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, rocblas_double_complex **A*, **const** rocblas_int *lda*, rocblas_double_complex **ipiv*, rocblas_double_complex **C*, **const** rocblas_int *ldc*)

rocblas_status **rocsolver_cunmbr**(rocblas_handle *handle*, **const** *rocblas_storev* *storev*, **const** rocblas_side *side*, **const** rocblas_operation *trans*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *k*, rocblas_float_complex **A*, **const** rocblas_int *lda*, rocblas_float_complex **ipiv*, rocblas_float_complex **C*, **const** rocblas_int *ldc*)

UNMBR multiplies a complex matrix Q with orthonormal rows or columns by a general m-by-n matrix C.

If storev is column-wise, then the matrix Q has orthonormal columns. If storev is row-wise, then the matrix Q has orthonormal rows. The matrix Q is applied in one of the following forms, depending on the values of side and trans:

QC	No transpose from the left,
$Q^H C$	Conjugate transpose from the left,
CQ	No transpose from the right, and
CQ^H	Conjugate transpose from the right.

The order q of the unitary matrix Q is q = m if applying from the left, or q = n if applying from the right.

When storev is column-wise, if q >= k, then Q is defined as the product of k Householder reflectors

$$Q = H_1 H_2 \cdots H_k,$$

and if q < k, then Q is defined as the product

$$Q = H_1 H_2 \cdots H_{q-1}.$$

When storev is row-wise, if q > k, then Q is defined as the product of k Householder reflectors

$$Q = H_1 H_2 \cdots H_k,$$

and if q <= k, Q is defined as the product

$$Q = H_1 H_2 \cdots H_{q-1}.$$

The Householder matrices H_i are never stored, they are computed from its corresponding Householder vectors and scalars as returned by *GEBRD* in its arguments A and tauq or taup.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] storev`: *rocblas_storev*. Specifies whether to work column-wise or row-wise.
- `[in] side`: rocblas_side. Specifies from which side to apply Q.
- `[in] trans`: rocblas_operation. Specifies whether the matrix Q or its conjugate transpose is to be applied.
- `[in] m`: rocblas_int. m >= 0. Number of rows of matrix C.
- `[in] n`: rocblas_int. n >= 0. Number of columns of matrix C.
- `[in] k`: rocblas_int. k >= 0. The number of columns (if storev is column-wise) or rows (if row-wise) of the original matrix reduced by *GEBRD*.
- `[in] A`: pointer to type. Array on the GPU of size lda*min(q,k) if column-wise, or lda*q if row-wise. The Householder vectors as returned by *GEBRD*.
- `[in] lda`: rocblas_int. lda >= q if column-wise, or lda >= min(q,k) if row-wise. Leading dimension of A.
- `[in] ipiv`: pointer to type. Array on the GPU of dimension at least min(q,k). The Householder scalars as returned by *GEBRD*.
- `[inout] C`: pointer to type. Array on the GPU of size ldc*n. On entry, the matrix C. On exit, it is overwritten with Q*C, C*Q, Q'*C, or C*Q'.
- `[in] ldc`: rocblas_int. ldc >= m. Leading dimension of C.

rocsolver_<type>unmtr()

rocblas_status **rocsolver_zunmtr**(rocblas_handle *handle*, **const** rocblas_side *side*, **const** rocblas_fill *uplo*, **const** rocblas_operation *trans*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, rocblas_double_complex **ipiv*, rocblas_double_complex **C*, **const** rocblas_int *ldc*)

rocblas_status **rocsolver_cunmtr**(rocblas_handle *handle*, **const** rocblas_side *side*, **const** rocblas_fill *uplo*, **const** rocblas_operation *trans*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, rocblas_float_complex **ipiv*, rocblas_float_complex **C*, **const** rocblas_int *ldc*)

UNMTR multiplies a unitary matrix Q by a general m-by-n matrix C.

The matrix Q is applied in one of the following forms, depending on the values of side and trans:

QC	No transpose from the left,
Q^HC	Conjugate transpose from the left,
CQ	No transpose from the right, and
CQ^H	Conjugate transpose from the right.

The order q of the unitary matrix Q is q = m if applying from the left, or q = n if applying from the right.

Q is defined as a product of q-1 Householder reflectors. If uplo indicates upper, then Q has the form

$$Q = H_{q-1}H_{q-2}\cdots H_1.$$

On the other hand, if uplo indicates lower, then Q has the form

$$Q = H_1 H_2 \cdots H_{q-1}$$

The Householder matrices H_i are never stored, they are computed from its corresponding Householder vectors and scalars as returned by *HETRD* in its arguments A and tau.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] side`: rocblas_side. Specifies from which side to apply Q.
- `[in] uplo`: rocblas_fill. Specifies whether the *HETRD* factorization was upper or lower triangular. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.
- `[in] trans`: rocblas_operation. Specifies whether the matrix Q or its conjugate transpose is to be applied.
- `[in] m`: rocblas_int. m >= 0. Number of rows of matrix C.
- `[in] n`: rocblas_int. n >= 0. Number of columns of matrix C.
- `[in] A`: pointer to type. Array on the GPU of size lda*q. On entry, the Householder vectors as returned by *HETRD*.
- `[in] lda`: rocblas_int. lda >= q. Leading dimension of A.
- `[in] ipiv`: pointer to type. Array on the GPU of dimension at least q-1. The Householder scalars as returned by *HETRD*.
- `[inout] C`: pointer to type. Array on the GPU of size ldc*n. On entry, the matrix C. On exit, it is overwritten with Q*C, C*Q, Q'*C, or C*Q'.
- `[in] ldc`: rocblas_int. ldc >= m. Leading dimension of C.

3.3 LAPACK Functions

LAPACK routines solve complex Numerical Linear Algebra problems. These functions are organized in the following categories:

- *Triangular factorizations*. Based on Gaussian elimination.
- *Orthogonal factorizations*. Based on Householder reflections.
- *Problem and matrix reductions*. Transformation of matrices and problems into equivalent forms.
- *Linear-systems solvers*. Based on triangular factorizations.
- *Least-squares solvers*. Based on orthogonal factorizations.
- *Symmetric eigensolvers*. Eigenproblems for symmetric matrices.
- *Singular value decomposition*. Singular values and related problems for general matrices.

Note: Throughout the APIs' descriptions, we use the following notations:

- x[i] stands for the i-th element of vector x, while A[i,j] represents the element in the i-th row and j-th column of matrix A. Indices are 1-based, i.e. x[1] is the first element of x.

- If X is a real vector or matrix, X^T indicates its transpose; if X is complex, then X^H represents its conjugate transpose. When X could be real or complex, we use X' to indicate X transposed or X conjugate transposed, accordingly.
- x_i = x_i; we sometimes use both notations, x_i when displaying mathematical equations, and x_i in the text describing the function parameters.

3.3.1 Triangular factorizations

List of triangular factorizations

- *rocsolver_<type>potf2()*
- *rocsolver_<type>potf2_batched()*
- *rocsolver_<type>potf2_strided_batched()*
- *rocsolver_<type>potrf()*
- *rocsolver_<type>potrf_batched()*
- *rocsolver_<type>potrf_strided_batched()*
- *rocsolver_<type>getf2()*
- *rocsolver_<type>getf2_batched()*
- *rocsolver_<type>getf2_strided_batched()*
- *rocsolver_<type>getrf()*
- *rocsolver_<type>getrf_batched()*
- *rocsolver_<type>getrf_strided_batched()*
- *rocsolver_<type>sytf2()*
- *rocsolver_<type>sytf2_batched()*
- *rocsolver_<type>sytf2_strided_batched()*
- *rocsolver_<type>sytrf()*
- *rocsolver_<type>sytrf_batched()*
- *rocsolver_<type>sytrf_strided_batched()*

rocsolver_<type>potf2()

rocblas_status **rocsolver_zpotf2** (rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, rocblas_int **info*)

rocblas_status **rocsolver_cpotf2** (rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, rocblas_int **info*)

rocblas_status **rocsolver_dpotf2** (rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, rocblas_int **info*) rocblas_status **rocsolver_spotf2** (rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, rocblas_int **info*)

POTF2 computes the Cholesky factorization of a real symmetric (complex Hermitian) positive definite matrix A.

(This is the unblocked version of the algorithm).

The factorization has the form:

$$\begin{array}{cl} A = U'U & \text{if uplo is upper, or} \\ A = LL' & \text{if uplo is lower.} \end{array}$$

U is an upper triangular matrix and L is lower triangular.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] uplo`: rocblas_fill. Specifies whether the factorization is upper or lower triangular. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.
- `[in] n`: rocblas_int. n >= 0. The number of rows and columns of matrix A.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the matrix A to be factored. On exit, the lower or upper triangular factor.
- `[in] lda`: rocblas_int. lda >= n. specifies the leading dimension of A.
- `[out] info`: pointer to a rocblas_int on the GPU. If info = 0, successful factorization of matrix A. If info = j > 0, the leading minor of order j of A is not positive definite. The factorization stopped at this point.

rocsolver_<type>potf2_batched()

rocblas_status **rocsolver_zpotf2_batched** (rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_double_complex ****const** *A*[], **const** rocblas_int *lda*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cpotf2_batched** (rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_float_complex ****const** *A*[], **const** rocblas_int *lda*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dpotf2_batched** (rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, double ****const** *A*[], **const** rocblas_int *lda*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_spotf2_batched** (rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, float ****const** *A*[], **const** rocblas_int *lda*, rocblas_int **info*, **const** rocblas_int *batch_count*)

POTF2_BATCHED computes the Cholesky factorization of a batch of real symmetric (complex Hermitian) positive definite matrices.

(This is the unblocked version of the algorithm).

The factorization of matrix A_i in the batch has the form:

$$A_i = U_i' U_i \quad \text{if uplo is upper, or}$$
$$A_i = L_i L_i' \quad \text{if uplo is lower.}$$

U_i is an upper triangular matrix and L_i is lower triangular.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] uplo`: rocblas_fill. Specifies whether the factorization is upper or lower triangular. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.
- `[in] n`: rocblas_int. n >= 0. The number of rows and columns of matrix A_i.
- `[inout] A`: array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n. On entry, the matrices A_i to be factored. On exit, the upper or lower triangular factors.
- `[in] lda`: rocblas_int. lda >= n. specifies the leading dimension of A_i.
- `[out] info`: pointer to rocblas_int. Array of batch_count integers on the GPU. If info[i] = 0, successful factorization of matrix A_i. If info[i] = j > 0, the leading minor of order j of A_i is not positive definite. The i-th factorization stopped at this point.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>potf2_strided_batched()

rocblas_status **rocsolver_zpotf2_strided_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cpotf2_strided_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dpotf2_strided_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_spotf2_strided_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **info*, **const** rocblas_int *batch_count*)

POTF2_STRIDED_BATCHED computes the Cholesky factorization of a batch of real symmetric (complex Hermitian) positive definite matrices.

(This is the unblocked version of the algorithm).

The factorization of matrix A_i in the batch has the form:

$$A_i = U_i' U_i \quad \text{if uplo is upper, or}$$
$$A_i = L_i L_i' \quad \text{if uplo is lower.}$$

U_i is an upper triangular matrix and L_i is lower triangular.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] uplo`: rocblas_fill. Specifies whether the factorization is upper or lower triangular. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.
- `[in] n`: rocblas_int. n >= 0. The number of rows and columns of matrix A_i.
- `[inout] A`: pointer to type. Array on the GPU (the size depends on the value of strideA). On entry, the matrices A_i to be factored. On exit, the upper or lower triangular factors.
- `[in] lda`: rocblas_int. lda >= n. specifies the leading dimension of A_i.
- `[in] strideA`: rocblas_stride. Stride from the start of one matrix A_i to the next one A_(i+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n.
- `[out] info`: pointer to rocblas_int. Array of batch_count integers on the GPU. If info[i] = 0, successful factorization of matrix A_i. If info[i] = j > 0, the leading minor of order j of A_i is not positive definite. The i-th factorization stopped at this point.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>potrf()

rocblas_status **rocsolver_zpotrf**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, rocblas_int **info*)

rocblas_status **rocsolver_cpotrf**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, rocblas_int **info*)

rocblas_status **rocsolver_dpotrf**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, rocblas_int **info*)

rocblas_status **rocsolver_spotrf**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, rocblas_int **info*)

POTRF computes the Cholesky factorization of a real symmetric (complex Hermitian) positive definite matrix A.

(This is the blocked version of the algorithm).

The factorization has the form:

$$A = U'U \quad \text{if uplo is upper, or}$$
$$A = LL' \quad \text{if uplo is lower.}$$

U is an upper triangular matrix and L is lower triangular.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] uplo`: rocblas_fill. Specifies whether the factorization is upper or lower triangular. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.
- `[in] n`: rocblas_int. n >= 0. The number of rows and columns of matrix A.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the matrix A to be factored. On exit, the lower or upper triangular factor.
- `[in] lda`: rocblas_int. lda >= n. specifies the leading dimension of A.
- `[out] info`: pointer to a rocblas_int on the GPU. If info = 0, successful factorization of matrix A. If info = j > 0, the leading minor of order j of A is not positive definite. The factorization stopped at this point.

rocsolver_<type>potrf_batched()

rocblas_status **rocsolver_zpotrf_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_double_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cpotrf_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_float_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dpotrf_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, double ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_spotrf_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, float ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **info*, **const** rocblas_int *batch_count*)

POTRF_BATCHED computes the Cholesky factorization of a batch of real symmetric (complex Hermitian) positive definite matrices.

(This is the blocked version of the algorithm).

The factorization of matrix A_i in the batch has the form:

$$
\begin{array}{cl}
A_i = U_i' U_i & \text{if uplo is upper, or} \\
A_i = L_i L_i' & \text{if uplo is lower.}
\end{array}
$$

U_i is an upper triangular matrix and L_i is lower triangular.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] uplo`: rocblas_fill. Specifies whether the factorization is upper or lower triangular. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.
- `[in] n`: rocblas_int. n >= 0. The number of rows and columns of matrix A_i.
- `[inout] A`: array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n. On entry, the matrices A_i to be factored. On exit, the upper or lower triangular factors.

- `[in] lda`: rocblas_int. lda >= n. specifies the leading dimension of A_i.
- `[out] info`: pointer to rocblas_int. Array of batch_count integers on the GPU. If info[i] = 0, successful factorization of matrix A_i. If info[i] = j > 0, the leading minor of order j of A_i is not positive definite. The i-th factorization stopped at this point.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>potrf_strided_batched()

rocblas_status **rocsolver_zpotrf_strided_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cpotrf_strided_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dpotrf_strided_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_spotrf_strided_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **info*, **const** rocblas_int *batch_count*)

POTRF_STRIDED_BATCHED computes the Cholesky factorization of a batch of real symmetric (complex Hermitian) positive definite matrices.

(This is the blocked version of the algorithm).

The factorization of matrix A_i in the batch has the form:

$$\begin{array}{cl} A_i = U_i' U_i & \text{if uplo is upper, or} \\ A_i = L_i L_i' & \text{if uplo is lower.} \end{array}$$

U_i is an upper triangular matrix and L_i is lower triangular.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] uplo`: rocblas_fill. Specifies whether the factorization is upper or lower triangular. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.
- `[in] n`: rocblas_int. n >= 0. The number of rows and columns of matrix A_i.
- `[inout] A`: pointer to type. Array on the GPU (the size depends on the value of strideA). On entry, the matrices A_i to be factored. On exit, the upper or lower triangular factors.

- `[in] lda`: rocblas_int. lda >= n. specifies the leading dimension of A_i.
- `[in] strideA`: rocblas_stride. Stride from the start of one matrix A_i to the next one A_(i+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n.
- `[out] info`: pointer to rocblas_int. Array of batch_count integers on the GPU. If info[i] = 0, successful factorization of matrix A_i. If info[i] = j > 0, the leading minor of order j of A_i is not positive definite. The i-th factorization stopped at this point.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>getf2()

rocblas_status **rocsolver_zgetf2**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, rocblas_int **ipiv*, rocblas_int **info*)

rocblas_status **rocsolver_cgetf2**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, rocblas_int **ipiv*, rocblas_int **info*)

rocblas_status **rocsolver_dgetf2**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, rocblas_int **ipiv*, rocblas_int **info*)

rocblas_status **rocsolver_sgetf2**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, rocblas_int **ipiv*, rocblas_int **info*)

GETF2 computes the LU factorization of a general m-by-n matrix A using partial pivoting with row interchanges.

(This is the unblocked Level-2-BLAS version of the algorithm. An optimized internal implementation without rocBLAS calls could be executed with small and mid-size matrices if optimizations are enabled (default option). For more details, see the "Tuning rocSOLVER performance" section of the Library Design Guide).

The factorization has the form

$$A = PLU$$

where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if m > n), and U is upper triangular (upper trapezoidal if m < n).

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. m >= 0. The number of rows of the matrix A.
- `[in] n`: rocblas_int. n >= 0. The number of columns of the matrix A.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the m-by-n matrix A to be factored. On exit, the factors L and U from the factorization. The unit diagonal elements of L are not stored.
- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of A.
- `[out] ipiv`: pointer to rocblas_int. Array on the GPU of dimension min(m,n). The vector of pivot indices. Elements of ipiv are 1-based indices. For 1 <= i <= min(m,n), the row i of the matrix was interchanged with row ipiv[i]. Matrix P of the factorization can be derived from ipiv.

- [out] info: pointer to a rocblas_int on the GPU. If info = 0, successful exit. If info = j > 0, U is singular. U[j,j] is the first zero pivot.

rocsolver_<type>getf2_batched()

rocblas_status **rocsolver_zgetf2_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_double_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cgetf2_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_float_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dgetf2_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, double ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_sgetf2_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, float ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_int **info*, **const** rocblas_int *batch_count*)

GETF2_BATCHED computes the LU factorization of a batch of general m-by-n matrices using partial pivoting with row interchanges.

(This is the unblocked Level-2-BLAS version of the algorithm. An optimized internal implementation without rocBLAS calls could be executed with small and mid-size matrices if optimizations are enabled (default option). For more details, see the "Tuning rocSOLVER performance" section of the Library Design Guide).

The factorization of matrix A_i in the batch has the form

$$A_i = P_i L_i U_i$$

where P_i is a permutation matrix, L_i is lower triangular with unit diagonal elements (lower trapezoidal if m > n), and U_i is upper triangular (upper trapezoidal if m < n).

Parameters

- [in] handle: rocblas_handle.
- [in] m: rocblas_int. m >= 0. The number of rows of all matrices A_i in the batch.
- [in] n: rocblas_int. n >= 0. The number of columns of all matrices A_i in the batch.
- [inout] A: array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n. On entry, the m-by-n matrices A_i to be factored. On exit, the factors L_i and U_i from the factorizations. The unit diagonal elements of L_i are not stored.
- [in] lda: rocblas_int. lda >= m. Specifies the leading dimension of matrices A_i.
- [out] ipiv: pointer to rocblas_int. Array on the GPU (the size depends on the value of strideP). Contains the vectors of pivot indices ipiv_i (corresponding to A_i). Dimension of ipiv_i is min(m,n). Elements of ipiv_i are 1-based indices. For each instance A_i in the batch and for 1 <= j <= min(m,n),

the row j of the matrix A_i was interchanged with row ipiv_i[j]. Matrix P_i of the factorization can be derived from ipiv_i.

- `[in] strideP`: rocblas_stride. Stride from the start of one vector ipiv_i to the next one ipiv_(i+1). There is no restriction for the value of strideP. Normal use case is strideP >= min(m,n).
- `[out] info`: pointer to rocblas_int. Array of batch_count integers on the GPU. If info[i] = 0, successful exit for factorization of A_i. If info[i] = j > 0, U_i is singular. U_i[j,j] is the first zero pivot.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>getf2_strided_batched()

rocblas_status **rocsolver_zgetf2_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cgetf2_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dgetf2_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_sgetf2_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_int **info*, **const** rocblas_int *batch_count*)

GETF2_STRIDED_BATCHED computes the LU factorization of a batch of general m-by-n matrices using partial pivoting with row interchanges.

(This is the unblocked Level-2-BLAS version of the algorithm. An optimized internal implementation without rocBLAS calls could be executed with small and mid-size matrices if optimizations are enabled (default option). For more details, see the “Tuning rocSOLVER performance” section of the Library Design Guide).

The factorization of matrix A_i in the batch has the form

$$A_i = P_i L_i U_i$$

where P_i is a permutation matrix, L_i is lower triangular with unit diagonal elements (lower trapezoidal if m > n), and U_i is upper triangular (upper trapezoidal if m < n).

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. m >= 0. The number of rows of all matrices A_i in the batch.
- `[in] n`: rocblas_int. n >= 0. The number of columns of all matrices A_i in the batch.
- `[inout] A`: pointer to type. Array on the GPU (the size depends on the value of strideA). On entry, the m-by-n matrices A_i to be factored. On exit, the factors L_i and U_i from the factorization. The unit diagonal elements of L_i are not stored.
- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of matrices A_i.
- `[in] strideA`: rocblas_stride. Stride from the start of one matrix A_i to the next one A_(i+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n
- `[out] ipiv`: pointer to rocblas_int. Array on the GPU (the size depends on the value of strideP). Contains the vectors of pivots indices ipiv_i (corresponding to A_i). Dimension of ipiv_i is min(m,n). Elements of ipiv_i are 1-based indices. For each instance A_i in the batch and for 1 <= j <= min(m,n), the row j of the matrix A_i was interchanged with row ipiv_i[j]. Matrix P_i of the factorization can be derived from ipiv_i.
- `[in] strideP`: rocblas_stride. Stride from the start of one vector ipiv_i to the next one ipiv_(i+1). There is no restriction for the value of strideP. Normal use case is strideP >= min(m,n).
- `[out] info`: pointer to rocblas_int. Array of batch_count integers on the GPU. If info[i] = 0, successful exit for factorization of A_i. If info[i] = j > 0, U_i is singular. U_i[j,j] is the first zero pivot.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>getrf()

rocblas_status **rocsolver_zgetrf**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, rocblas_int **ipiv*, rocblas_int **info*)

rocblas_status **rocsolver_cgetrf**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, rocblas_int **ipiv*, rocblas_int **info*)

rocblas_status **rocsolver_dgetrf**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, rocblas_int **ipiv*, rocblas_int **info*)

rocblas_status **rocsolver_sgetrf**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, rocblas_int **ipiv*, rocblas_int **info*)

GETRF computes the LU factorization of a general m-by-n matrix A using partial pivoting with row interchanges.

(This is the blocked Level-3-BLAS version of the algorithm. An optimized internal implementation without rocBLAS calls could be executed with mid-size matrices if optimizations are enabled (default option). For more details, see the "Tuning rocSOLVER performance" section of the Library Design Guide).

The factorization has the form

$$A = PLU$$

where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if m > n), and U is upper triangular (upper trapezoidal if m < n).

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. m >= 0. The number of rows of the matrix A.
- `[in] n`: rocblas_int. n >= 0. The number of columns of the matrix A.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the m-by-n matrix A to be factored. On exit, the factors L and U from the factorization. The unit diagonal elements of L are not stored.
- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of A.
- `[out] ipiv`: pointer to rocblas_int. Array on the GPU of dimension min(m,n). The vector of pivot indices. Elements of ipiv are 1-based indices. For 1 <= i <= min(m,n), the row i of the matrix was interchanged with row ipiv[i]. Matrix P of the factorization can be derived from ipiv.
- `[out] info`: pointer to a rocblas_int on the GPU. If info = 0, successful exit. If info = j > 0, U is singular. U[j,j] is the first zero pivot.

rocsolver_<type>getrf_batched()

rocblas_status **rocsolver_zgetrf_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_double_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cgetrf_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_float_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dgetrf_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, double ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_sgetrf_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, float ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_int **info*, **const** rocblas_int *batch_count*)

GETRF_BATCHED computes the LU factorization of a batch of general m-by-n matrices using partial pivoting with row interchanges.

(This is the blocked Level-3-BLAS version of the algorithm. An optimized internal implementation without rocBLAS calls could be executed with mid-size matrices if optimizations are enabled (default option). For more details, see the “Tuning rocSOLVER performance” section of the Library Design Guide).

The factorization of matrix A_i in the batch has the form

$$A_i = P_i L_i U_i$$

where P_i is a permutation matrix, L_i is lower triangular with unit diagonal elements (lower trapezoidal if m > n), and U_i is upper triangular (upper trapezoidal if m < n).

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. m >= 0. The number of rows of all matrices A_i in the batch.
- `[in] n`: rocblas_int. n >= 0. The number of columns of all matrices A_i in the batch.
- `[inout] A`: array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n. On entry, the m-by-n matrices A_i to be factored. On exit, the factors L_i and U_i from the factorizations. The unit diagonal elements of L_i are not stored.
- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of matrices A_i.
- `[out] ipiv`: pointer to rocblas_int. Array on the GPU (the size depends on the value of strideP). Contains the vectors of pivot indices ipiv_i (corresponding to A_i). Dimension of ipiv_i is min(m,n). Elements of ipiv_i are 1-based indices. For each instance A_i in the batch and for 1 <= j <= min(m,n), the row j of the matrix A_i was interchanged with row ipiv_i[j]. Matrix P_i of the factorization can be derived from ipiv_i.
- `[in] strideP`: rocblas_stride. Stride from the start of one vector ipiv_i to the next one ipiv_(i+1). There is no restriction for the value of strideP. Normal use case is strideP >= min(m,n).
- `[out] info`: pointer to rocblas_int. Array of batch_count integers on the GPU. If info[i] = 0, successful exit for factorization of A_i. If info[i] = j > 0, U_i is singular. U_i[j,j] is the first zero pivot.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>getrf_strided_batched()

rocblas_status **rocsolver_zgetrf_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cgetrf_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dgetrf_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_sgetrf_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_int **info*, **const** rocblas_int *batch_count*)

GETRF_STRIDED_BATCHED computes the LU factorization of a batch of general m-by-n matrices using partial pivoting with row interchanges.

(This is the blocked Level-3-BLAS version of the algorithm. An optimized internal implementation without rocBLAS calls could be executed with mid-size matrices if optimizations are enabled (default option). For more details, see the "Tuning rocSOLVER performance" section of the Library Design Guide).

The factorization of matrix A_i in the batch has the form

$$A_i = P_i L_i U_i$$

where P_i is a permutation matrix, L_i is lower triangular with unit diagonal elements (lower trapezoidal if m > n), and U_i is upper triangular (upper trapezoidal if m < n).

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. m >= 0. The number of rows of all matrices A_i in the batch.
- `[in] n`: rocblas_int. n >= 0. The number of columns of all matrices A_i in the batch.
- `[inout] A`: pointer to type. Array on the GPU (the size depends on the value of strideA). On entry, the m-by-n matrices A_i to be factored. On exit, the factors L_i and U_i from the factorization. The unit diagonal elements of L_i are not stored.
- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of matrices A_i.
- `[in] strideA`: rocblas_stride. Stride from the start of one matrix A_i to the next one A_(i+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n
- `[out] ipiv`: pointer to rocblas_int. Array on the GPU (the size depends on the value of strideP). Contains the vectors of pivots indices ipiv_i (corresponding to A_i). Dimension of ipiv_i is min(m,n). Elements of ipiv_i are 1-based indices. For each instance A_i in the batch and for 1 <= j <= min(m,n), the row j of the matrix A_i was interchanged with row ipiv_i[j]. Matrix P_i of the factorization can be derived from ipiv_i.
- `[in] strideP`: rocblas_stride. Stride from the start of one vector ipiv_i to the next one ipiv_(i+1). There is no restriction for the value of strideP. Normal use case is strideP >= min(m,n).
- `[out] info`: pointer to rocblas_int. Array of batch_count integers on the GPU. If info[i] = 0, successful exit for factorization of A_i. If info[i] = j > 0, U_i is singular. U_i[j,j] is the first zero pivot.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>sytf2()

rocblas_status **rocsolver_zsytf2** (rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, rocblas_int **ipiv*, rocblas_int **info*)

rocblas_status **rocsolver_csytf2** (rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, rocblas_int **ipiv*, rocblas_int **info*)

rocblas_status **rocsolver_dsytf2** (rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, rocblas_int **ipiv*, rocblas_int **info*) rocblas_status **rocsolver_ssytf2** (rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, rocblas_int **ipiv*, rocblas_int **info*)

SYTF2 computes the factorization of a symmetric indefinite matrix A using Bunch-Kaufman diagonal pivoting.

(This is the unblocked version of the algorithm).

The factorization has the form

$$\begin{array}{cc} A = UDU^T & \text{or} \\ A = LDL^T & \end{array}$$

where U or L is a product of permutation and unit upper/lower triangular matrices (depending on the value of uplo), and D is a symmetric block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks $D(k)$.

Specifically, U and L are computed as

$$\begin{array}{cc} U = P(n)U(n)\cdots P(k)U(k)\cdots & \text{and} \\ L = P(1)L(1)\cdots P(k)L(k)\cdots & \end{array}$$

where k decreases from n to 1 (increases from 1 to n) in steps of 1 or 2, depending on the order of block $D(k)$, and $P(k)$ is a permutation matrix defined by $ipiv[k]$. If we let s denote the order of block $D(k)$, then $U(k)$ and $L(k)$ are unit upper/lower triangular matrices defined as

$$U(k) = \left[\begin{array}{ccc} I_{k-s} & v & 0 \\ 0 & I_s & 0 \\ 0 & 0 & I_{n-k} \end{array}\right]$$

and

$$L(k) = \left[\begin{array}{ccc} I_{k-1} & 0 & 0 \\ 0 & I_s & 0 \\ 0 & v & I_{n-k-s+1} \end{array}\right].$$

If $s = 1$, then $D(k)$ is stored in $A[k, k]$ and v is stored in the upper/lower part of column k of A. If $s = 2$ and uplo is upper, then $D(k)$ is stored in $A[k-1, k-1]$, $A[k-1, k]$, and $A[k, k]$, and v is stored in the upper parts of columns $k-1$ and k of A. If $s = 2$ and uplo is lower, then $D(k)$ is stored in $A[k, k]$, $A[k+1, k]$, and $A[k+1, k+1]$, and v is stored in the lower parts of columns k and $k+1$ of A.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower part of the matrix A is stored. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.
- `[in] n`: rocblas_int. n >= 0. The number of rows and columns of the matrix A.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the symmetric matrix A to be factored. On exit, the block diagonal matrix D and the multipliers needed to compute U or L.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of A.

- [out] ipiv: pointer to rocblas_int. Array on the GPU of dimension n. The vector of pivot indices. Elements of ipiv are 1-based indices. For 1 <= k <= n, if ipiv[k] > 0 then rows and columns k and ipiv[k] were interchanged and D[k,k] is a 1-by-1 diagonal block. If, instead, ipiv[k] = ipiv[k-1] < 0 and uplo is upper (or ipiv[k] = ipiv[k+1] < 0 and uplo is lower), then rows and columns k-1 and -ipiv[k] (or rows and columns k+1 and -ipiv[k]) were interchanged and D[k-1,k-1] to D[k,k] (or D[k,k] to D[k+1,k+1]) is a 2-by-2 diagonal block.
- [out] info: pointer to a rocblas_int on the GPU. If info = 0, successful exit. If info[i] = j > 0, D is singular. D[j,j] is the first diagonal zero.

rocsolver_<type>sytf2_batched()

rocblas_status **rocsolver_zsytf2_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_double_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_csytf2_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_float_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dsytf2_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, double ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_ssytf2_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, float ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_int **info*, **const** rocblas_int *batch_count*)

SYTF2_BATCHED computes the factorization of a batch of symmetric indefinite matrices using Bunch-Kaufman diagonal pivoting.

(This is the unblocked version of the algorithm).

The factorization has the form

$$\begin{array}{cc} A_i = U_i D_i U_i^T & \text{or} \\ A_i = L_i D_i L_i^T & \end{array}$$

where U_i or L_i is a product of permutation and unit upper/lower triangular matrices (depending on the value of uplo), and D_i is a symmetric block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks $D_i(k)$.

Specifically, U_i and L_i are computed as

$$\begin{array}{cc} U_i = P_i(n)U_i(n)\cdots P_i(k)U_i(k)\cdots & \text{and} \\ L_i = P_i(1)L_i(1)\cdots P_i(k)L_i(k)\cdots & \end{array}$$

where k decreases from n to 1 (increases from 1 to n) in steps of 1 or 2, depending on the order of block $D_i(k)$, and $P_i(k)$ is a permutation matrix defined by $ipiv_i[k]$. If we let s denote the order of block $D_i(k)$, then $U_i(k)$ and $L_i(k)$ are unit upper/lower triangular matrices defined as

$$U_i(k) = \begin{bmatrix} I_{k-s} & v & 0 \\ 0 & I_s & 0 \\ 0 & 0 & I_{n-k} \end{bmatrix}$$

and

$$L_i(k) = \begin{bmatrix} I_{k-1} & 0 & 0 \\ 0 & I_s & 0 \\ 0 & v & I_{n-k-s+1} \end{bmatrix}.$$

If $s = 1$, then $D_i(k)$ is stored in $A_i[k, k]$ and v is stored in the upper/lower part of column k of A_i. If $s = 2$ and uplo is upper, then $D_i(k)$ is stored in $A_i[k-1, k-1]$, $A_i[k-1, k]$, and $A_i[k, k]$, and v is stored in the upper parts of columns $k-1$ and k of A_i. If $s = 2$ and uplo is lower, then $D_i(k)$ is stored in $A_i[k, k]$, $A_i[k+1, k]$, and $A_i[k+1, k+1]$, and v is stored in the lower parts of columns k and $k+1$ of A_i.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower part of the matrices A_i are stored. If uplo indicates lower (or upper), then the upper (or lower) part of A_i is not used.
- `[in] n`: rocblas_int. n >= 0. The number of rows and columns of all matrices A_i in the batch.
- `[inout] A`: array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n. On entry, the symmetric matrices A_i to be factored. On exit, the block diagonal matrices D_i and the multipliers needed to compute U_i or L_i.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of matrices A_i.
- `[out] ipiv`: pointer to rocblas_int. Array on the GPU of dimension n. The vector of pivot indices. Elements of ipiv are 1-based indices. For 1 <= k <= n, if ipiv_i[k] > 0 then rows and columns k and ipiv_i[k] were interchanged and D_i[k,k] is a 1-by-1 diagonal block. If, instead, ipiv_i[k] = ipiv_i[k-1] < 0 and uplo is upper (or ipiv_i[k] = ipiv_i[k+1] < 0 and uplo is lower), then rows and columns k-1 and -ipiv_i[k] (or rows and columns k+1 and -ipiv_i[k]) were interchanged and D_i[k-1,k-1] to D_i[k,k] (or D_i[k,k] to D_i[k+1,k+1]) is a 2-by-2 diagonal block.
- `[in] strideP`: rocblas_stride. Stride from the start of one vector ipiv_i to the next one ipiv_(i+1). There is no restriction for the value of strideP. Normal use case is strideP >= n.
- `[out] info`: pointer to rocblas_int. Array of batch_count integers on the GPU. If info[i] = 0, successful exit for factorization of A_i. If info[i] = j > 0, D_i is singular. D_i[j,j] is the first diagonal zero.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>sytf2_strided_batched()

rocblas_status **rocsolver_zsytf2_strided_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_csytf2_strided_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dsytf2_strided_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_ssytf2_strided_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_int **info*, **const** rocblas_int *batch_count*)

SYTF2_STRIDED_BATCHED computes the factorization of a batch of symmetric indefinite matrices using Bunch-Kaufman diagonal pivoting.

(This is the unblocked version of the algorithm).

The factorization has the form

$$\begin{array}{cl} A_i = U_i D_i U_i^T & \text{or} \\ A_i = L_i D_i L_i^T & \end{array}$$

where U_i or L_i is a product of permutation and unit upper/lower triangular matrices (depending on the value of uplo), and D_i is a symmetric block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks $D_i(k)$.

Specifically, U_i and L_i are computed as

$$\begin{array}{cl} U_i = P_i(n)U_i(n)\cdots P_i(k)U_i(k)\cdots & \text{and} \\ L_i = P_i(1)L_i(1)\cdots P_i(k)L_i(k)\cdots & \end{array}$$

where k decreases from n to 1 (increases from 1 to n) in steps of 1 or 2, depending on the order of block $D_i(k)$, and $P_i(k)$ is a permutation matrix defined by $ipiv_i[k]$. If we let s denote the order of block $D_i(k)$, then $U_i(k)$ and $L_i(k)$ are unit upper/lower triangular matrices defined as

$$U_i(k) = \begin{bmatrix} I_{k-s} & v & 0 \\ 0 & I_s & 0 \\ 0 & 0 & I_{n-k} \end{bmatrix}$$

and

$$L_i(k) = \begin{bmatrix} I_{k-1} & 0 & 0 \\ 0 & I_s & 0 \\ 0 & v & I_{n-k-s+1} \end{bmatrix}.$$

If $s = 1$, then $D_i(k)$ is stored in $A_i[k, k]$ and v is stored in the upper/lower part of column k of A_i. If $s = 2$ and uplo is upper, then $D_i(k)$ is stored in $A_i[k-1, k-1]$, $A_i[k-1, k]$, and $A_i[k, k]$, and v is stored in the upper parts of columns $k-1$ and k of A_i. If $s = 2$ and uplo is lower, then $D_i(k)$ is stored in $A_i[k, k]$, $A_i[k+1, k]$, and $A_i[k+1, k+1]$, and v is stored in the lower parts of columns k and $k+1$ of A_i.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower part of the matrices A_i are stored. If uplo indicates lower (or upper), then the upper (or lower) part of A_i is not used.
- `[in] n`: rocblas_int. n >= 0. The number of rows and columns of all matrices A_i in the batch.
- `[inout] A`: pointer to type. Array on the GPU (the size depends on the value of strideA). On entry, the symmetric matrices A_i to be factored. On exit, the block diagonal matrices D_i and the multipliers needed to compute U_i or L_i.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of matrices A_i.
- `[in] strideA`: rocblas_stride. Stride from the start of one matrix A_i to the next one A_(i+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n
- `[out] ipiv`: pointer to rocblas_int. Array on the GPU of dimension n. The vector of pivot indices. Elements of ipiv are 1-based indices. For 1 <= k <= n, if ipiv_i[k] > 0 then rows and columns k and ipiv_i[k] were interchanged and D_i[k,k] is a 1-by-1 diagonal block. If, instead, ipiv_i[k] = ipiv_i[k-1] < 0 and uplo is upper (or ipiv_i[k] = ipiv_i[k+1] < 0 and uplo is lower), then rows and columns k-1 and -ipiv_i[k] (or rows and columns k+1 and -ipiv_i[k]) were interchanged and D_i[k-1,k-1] to D_i[k,k] (or D_i[k,k] to D_i[k+1,k+1]) is a 2-by-2 diagonal block.
- `[in] strideP`: rocblas_stride. Stride from the start of one vector ipiv_i to the next one ipiv_(i+1). There is no restriction for the value of strideP. Normal use case is strideP >= n.
- `[out] info`: pointer to rocblas_int. Array of batch_count integers on the GPU. If info[i] = 0, successful exit for factorization of A_i. If info[i] = j > 0, D_i is singular. D_i[j,j] is the first diagonal zero.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>sytrf()

rocblas_status **rocsolver_zsytrf**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, rocblas_int **ipiv*, rocblas_int **info*)

rocblas_status **rocsolver_csytrf**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, rocblas_int **ipiv*, rocblas_int **info*)

rocblas_status **rocsolver_dsytrf**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, rocblas_int **ipiv*, rocblas_int **info*) rocblas_status **rocsolver_ssytrf**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, rocblas_int **ipiv*, rocblas_int **info*)

SYTRF computes the factorization of a symmetric indefinite matrix A using Bunch-Kaufman diagonal pivoting.

(This is the blocked version of the algorithm).

The factorization has the form

$$\begin{array}{cc} A = UDU^T & \text{or} \\ A = LDL^T & \end{array}$$

where U or L is a product of permutation and unit upper/lower triangular matrices (depending on the value of uplo), and D is a symmetric block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks $D(k)$.

Specifically, U and L are computed as

$$\begin{array}{cc} U = P(n)U(n)\cdots P(k)U(k)\cdots & \text{and} \\ L = P(1)L(1)\cdots P(k)L(k)\cdots & \end{array}$$

where k decreases from n to 1 (increases from 1 to n) in steps of 1 or 2, depending on the order of block $D(k)$, and $P(k)$ is a permutation matrix defined by $ipiv[k]$. If we let s denote the order of block $D(k)$, then $U(k)$ and $L(k)$ are unit upper/lower triangular matrices defined as

$$U(k) = \left[\begin{array}{ccc} I_{k-s} & v & 0 \\ 0 & I_s & 0 \\ 0 & 0 & I_{n-k} \end{array}\right]$$

and

$$L(k) = \left[\begin{array}{ccc} I_{k-1} & 0 & 0 \\ 0 & I_s & 0 \\ 0 & v & I_{n-k-s+1} \end{array}\right].$$

If $s = 1$, then $D(k)$ is stored in $A[k,k]$ and v is stored in the upper/lower part of column k of A. If $s = 2$ and uplo is upper, then $D(k)$ is stored in $A[k-1,k-1]$, $A[k-1,k]$, and $A[k,k]$, and v is stored in the upper parts of columns $k-1$ and k of A. If $s = 2$ and uplo is lower, then $D(k)$ is stored in $A[k,k]$, $A[k+1,k]$, and $A[k+1,k+1]$, and v is stored in the lower parts of columns k and $k+1$ of A.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower part of the matrix A is stored. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.
- `[in] n`: rocblas_int. n >= 0. The number of rows and columns of the matrix A.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the symmetric matrix A to be factored. On exit, the block diagonal matrix D and the multipliers needed to compute U or L.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of A.

- `[out] ipiv`: pointer to rocblas_int. Array on the GPU of dimension n. The vector of pivot indices. Elements of ipiv are 1-based indices. For 1 <= k <= n, if ipiv[k] > 0 then rows and columns k and ipiv[k] were interchanged and D[k,k] is a 1-by-1 diagonal block. If, instead, ipiv[k] = ipiv[k-1] < 0 and uplo is upper (or ipiv[k] = ipiv[k+1] < 0 and uplo is lower), then rows and columns k-1 and -ipiv[k] (or rows and columns k+1 and -ipiv[k]) were interchanged and D[k-1,k-1] to D[k,k] (or D[k,k] to D[k+1,k+1]) is a 2-by-2 diagonal block.
- `[out] info`: pointer to a rocblas_int on the GPU. If info = 0, successful exit. If info[i] = j > 0, D is singular. D[j,j] is the first diagonal zero.

rocsolver_<type>sytrf_batched()

rocblas_status **rocsolver_zsytrf_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_double_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_csytrf_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_float_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dsytrf_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, double ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_ssytrf_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, float ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_int **info*, **const** rocblas_int *batch_count*)

SYTRF_BATCHED computes the factorization of a batch of symmetric indefinite matrices using Bunch-Kaufman diagonal pivoting.

(This is the blocked version of the algorithm).

The factorization has the form

$$\begin{array}{cl} A_i = U_i D_i U_i^T & \text{or} \\ A_i = L_i D_i L_i^T & \end{array}$$

where U_i or L_i is a product of permutation and unit upper/lower triangular matrices (depending on the value of uplo), and D_i is a symmetric block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks $D_i(k)$.

Specifically, U_i and L_i are computed as

$$\begin{array}{cl} U_i = P_i(n)U_i(n)\cdots P_i(k)U_i(k)\cdots & \text{and} \\ L_i = P_i(1)L_i(1)\cdots P_i(k)L_i(k)\cdots & \end{array}$$

where k decreases from n to 1 (increases from 1 to n) in steps of 1 or 2, depending on the order of block $D_i(k)$, and $P_i(k)$ is a permutation matrix defined by $ipiv_i[k]$. If we let s denote the order of block $D_i(k)$, then $U_i(k)$ and $L_i(k)$ are unit upper/lower triangular matrices defined as

$$U_i(k) = \begin{bmatrix} I_{k-s} & v & 0 \\ 0 & I_s & 0 \\ 0 & 0 & I_{n-k} \end{bmatrix}$$

and

$$L_i(k) = \begin{bmatrix} I_{k-1} & 0 & 0 \\ 0 & I_s & 0 \\ 0 & v & I_{n-k-s+1} \end{bmatrix}.$$

If $s = 1$, then $D_i(k)$ is stored in $A_i[k, k]$ and v is stored in the upper/lower part of column k of A_i. If $s = 2$ and uplo is upper, then $D_i(k)$ is stored in $A_i[k-1, k-1]$, $A_i[k-1, k]$, and $A_i[k, k]$, and v is stored in the upper parts of columns $k-1$ and k of A_i. If $s = 2$ and uplo is lower, then $D_i(k)$ is stored in $A_i[k, k]$, $A_i[k+1, k]$, and $A_i[k+1, k+1]$, and v is stored in the lower parts of columns k and $k+1$ of A_i.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower part of the matrices A_i are stored. If uplo indicates lower (or upper), then the upper (or lower) part of A_i is not used.
- `[in] n`: rocblas_int. n >= 0. The number of rows and columns of all matrices A_i in the batch.
- `[inout] A`: array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n. On entry, the symmetric matrices A_i to be factored. On exit, the block diagonal matrices D_i and the multipliers needed to compute U_i or L_i.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of matrices A_i.
- `[out] ipiv`: pointer to rocblas_int. Array on the GPU of dimension n. The vector of pivot indices. Elements of ipiv are 1-based indices. For 1 <= k <= n, if ipiv_i[k] > 0 then rows and columns k and ipiv_i[k] were interchanged and D_i[k,k] is a 1-by-1 diagonal block. If, instead, ipiv_i[k] = ipiv_i[k-1] < 0 and uplo is upper (or ipiv_i[k] = ipiv_i[k+1] < 0 and uplo is lower), then rows and columns k-1 and -ipiv_i[k] (or rows and columns k+1 and -ipiv_i[k]) were interchanged and D_i[k-1,k-1] to D_i[k,k] (or D_i[k,k] to D_i[k+1,k+1]) is a 2-by-2 diagonal block.
- `[in] strideP`: rocblas_stride. Stride from the start of one vector ipiv_i to the next one ipiv_(i+1). There is no restriction for the value of strideP. Normal use case is strideP >= n.
- `[out] info`: pointer to rocblas_int. Array of batch_count integers on the GPU. If info[i] = 0, successful exit for factorization of A_i. If info[i] = j > 0, D_i is singular. D_i[j,j] is the first diagonal zero.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>sytrf_strided_batched()

rocblas_status **rocsolver_zsytrf_strided_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_csytrf_strided_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dsytrf_strided_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_ssytrf_strided_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_int **info*, **const** rocblas_int *batch_count*)

SYTRF_STRIDED_BATCHED computes the factorization of a batch of symmetric indefinite matrices using Bunch-Kaufman diagonal pivoting.

(This is the blocked version of the algorithm).

The factorization has the form

$$\begin{aligned} A_i &= U_i D_i U_i^T \quad \text{or} \\ A_i &= L_i D_i L_i^T \end{aligned}$$

where U_i or L_i is a product of permutation and unit upper/lower triangular matrices (depending on the value of uplo), and D_i is a symmetric block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks $D_i(k)$.

Specifically, U_i and L_i are computed as

$$\begin{aligned} U_i &= P_i(n)U_i(n)\cdots P_i(k)U_i(k)\cdots \quad \text{and} \\ L_i &= P_i(1)L_i(1)\cdots P_i(k)L_i(k)\cdots \end{aligned}$$

where k decreases from n to 1 (increases from 1 to n) in steps of 1 or 2, depending on the order of block $D_i(k)$, and $P_i(k)$ is a permutation matrix defined by $ipiv_i[k]$. If we let s denote the order of block $D_i(k)$, then $U_i(k)$ and $L_i(k)$ are unit upper/lower triangular matrices defined as

$$U_i(k) = \begin{bmatrix} I_{k-s} & v & 0 \\ 0 & I_s & 0 \\ 0 & 0 & I_{n-k} \end{bmatrix}$$

and

$$L_i(k) = \begin{bmatrix} I_{k-1} & 0 & 0 \\ 0 & I_s & 0 \\ 0 & v & I_{n-k-s+1} \end{bmatrix}.$$

If $s = 1$, then $D_i(k)$ is stored in $A_i[k, k]$ and v is stored in the upper/lower part of column k of A_i. If $s = 2$ and uplo is upper, then $D_i(k)$ is stored in $A_i[k-1, k-1]$, $A_i[k-1, k]$, and $A_i[k, k]$, and v is stored in the upper parts of columns $k-1$ and k of A_i. If $s = 2$ and uplo is lower, then $D_i(k)$ is stored in $A_i[k, k]$, $A_i[k+1, k]$, and $A_i[k+1, k+1]$, and v is stored in the lower parts of columns k and $k+1$ of A_i.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower part of the matrices A_i are stored. If uplo indicates lower (or upper), then the upper (or lower) part of A_i is not used.
- `[in] n`: rocblas_int. n >= 0. The number of rows and columns of all matrices A_i in the batch.
- `[inout] A`: pointer to type. Array on the GPU (the size depends on the value of strideA). On entry, the symmetric matrices A_i to be factored. On exit, the block diagonal matrices D_i and the multipliers needed to compute U_i or L_i.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of matrices A_i.
- `[in] strideA`: rocblas_stride. Stride from the start of one matrix A_i to the next one A_(i+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n
- `[out] ipiv`: pointer to rocblas_int. Array on the GPU of dimension n. The vector of pivot indices. Elements of ipiv are 1-based indices. For 1 <= k <= n, if ipiv_i[k] > 0 then rows and columns k and ipiv_i[k] were interchanged and D_i[k,k] is a 1-by-1 diagonal block. If, instead, ipiv_i[k] = ipiv_i[k-1] < 0 and uplo is upper (or ipiv_i[k] = ipiv_i[k+1] < 0 and uplo is lower), then rows and columns k-1 and -ipiv_i[k] (or rows and columns k+1 and -ipiv_i[k]) were interchanged and D_i[k-1,k-1] to D_i[k,k] (or D_i[k,k] to D_i[k+1,k+1]) is a 2-by-2 diagonal block.
- `[in] strideP`: rocblas_stride. Stride from the start of one vector ipiv_i to the next one ipiv_(i+1). There is no restriction for the value of strideP. Normal use case is strideP >= n.
- `[out] info`: pointer to rocblas_int. Array of batch_count integers on the GPU. If info[i] = 0, successful exit for factorization of A_i. If info[i] = j > 0, D_i is singular. D_i[j,j] is the first diagonal zero.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

3.3.2 Orthogonal factorizations

List of orthogonal factorizations

- *rocsolver_<type>geqr2()*
- *rocsolver_<type>geqr2_batched()*
- *rocsolver_<type>geqr2_strided_batched()*
- *rocsolver_<type>geqrf()*

- *rocsolver_<type>geqrf_batched()*
- *rocsolver_<type>geqrf_strided_batched()*
- *rocsolver_<type>gerq2()*
- *rocsolver_<type>gerq2_batched()*
- *rocsolver_<type>gerq2_strided_batched()*
- *rocsolver_<type>gerqf()*
- *rocsolver_<type>gerqf_batched()*
- *rocsolver_<type>gerqf_strided_batched()*
- *rocsolver_<type>geql2()*
- *rocsolver_<type>geql2_batched()*
- *rocsolver_<type>geql2_strided_batched()*
- *rocsolver_<type>geqlf()*
- *rocsolver_<type>geqlf_batched()*
- *rocsolver_<type>geqlf_strided_batched()*
- *rocsolver_<type>gelq2()*
- *rocsolver_<type>gelq2_batched()*
- *rocsolver_<type>gelq2_strided_batched()*
- *rocsolver_<type>gelqf()*
- *rocsolver_<type>gelqf_batched()*
- *rocsolver_<type>gelqf_strided_batched()*

rocsolver_<type>geqr2()

rocblas_status **rocsolver_zgeqr2** (rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, rocblas_double_complex **ipiv*)

rocblas_status **rocsolver_cgeqr2** (rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, rocblas_float_complex **ipiv*)

rocblas_status **rocsolver_dgeqr2** (rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, double **ipiv*)

rocblas_status **rocsolver_sgeqr2** (rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, float **ipiv*)

GEQR2 computes a QR factorization of a general m-by-n matrix A.

(This is the unblocked version of the algorithm).

The factorization has the form

$$A = Q \begin{bmatrix} R \\ 0 \end{bmatrix}$$

where R is upper triangular (upper trapezoidal if m < n), and Q is a m-by-m orthogonal/unitary matrix represented as the product of Householder matrices

$$Q = H_1 H_2 \cdots H_k, \quad \text{with } k = \min(m, n)$$

Each Householder matrix H_i is given by

$$H_i = I - \text{ipiv}[i] \cdot v_i v_i'$$

where the first i-1 elements of the Householder vector v_i are zero, and $v_i[i] = 1$.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. m >= 0. The number of rows of the matrix A.
- `[in] n`: rocblas_int. n >= 0. The number of columns of the matrix A.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the m-by-n matrix to be factored. On exit, the elements on and above the diagonal contain the factor R; the elements below the diagonal are the last m - i elements of Householder vector v_i.
- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of A.
- `[out] ipiv`: pointer to type. Array on the GPU of dimension min(m,n). The Householder scalars.

rocsolver_<type>geqr2_batched()

rocblas_status **rocsolver_zgeqr2_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_double_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_double_complex **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cgeqr2_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_float_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_float_complex **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dgeqr2_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, double ***const** *A*[], **const** rocblas_int *lda*, double **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_sgeqr2_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, float ***const** *A*[], **const** rocblas_int *lda*, float **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

GEQR2_BATCHED computes the QR factorization of a batch of general m-by-n matrices.

(This is the unblocked version of the algorithm).

The factorization of matrix A_j in the batch has the form

$$A_j = Q_j \begin{bmatrix} R_j \\ 0 \end{bmatrix}$$

where R_j is upper triangular (upper trapezoidal if m < n), and Q_j is a m-by-m orthogonal/unitary matrix represented as the product of Householder matrices

$$Q_j = H_{j_1} H_{j_2} \cdots H_{j_k}, \quad \text{with } k = \text{min}(m, n)$$

Each Householder matrix H_{j_i} is given by

$$H_{j_i} = I - \text{ipiv}_j[i] \cdot v_{j_i} v_{j_i}'$$

where the first i-1 elements of Householder vector v_{j_i} are zero, and $v_{j_i}[i] = 1$.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. m >= 0. The number of rows of all the matrices A_j in the batch.
- `[in] n`: rocblas_int. n >= 0. The number of columns of all the matrices A_j in the batch.
- `[inout] A`: Array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n. On entry, the m-by-n matrices A_j to be factored. On exit, the elements on and above the diagonal contain the factor R_j. The elements below the diagonal are the last m - i elements of Householder vector v_(j_i).
- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of matrices A_j.
- `[out] ipiv`: pointer to type. Array on the GPU (the size depends on the value of strideP). Contains the vectors ipiv_j of corresponding Householder scalars.
- `[in] strideP`: rocblas_stride. Stride from the start of one vector ipiv_j to the next one ipiv_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= min(m,n).
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>geqr2_strided_batched()

rocblas_status **rocsolver_zgeqr2_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_double_complex **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cgeqr2_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_float_complex **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*) rocblas_status **rocsolver_dgeqr2_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, double **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_sgeqr2_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, float **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

GEQR2_STRIDED_BATCHED computes the QR factorization of a batch of general m-by-n matrices.

(This is the unblocked version of the algorithm).

The factorization of matrix A_j in the batch has the form

$$A_j = Q_j \begin{bmatrix} R_j \\ 0 \end{bmatrix}$$

where R_j is upper triangular (upper trapezoidal if m < n), and Q_j is a m-by-m orthogonal/unitary matrix represented as the product of Householder matrices

$$Q_j = H_{j_1} H_{j_2} \cdots H_{j_k}, \quad \text{with } k = \text{min}(m, n)$$

Each Householder matrix H_{j_i} is given by

$$H_{j_i} = I - \text{ipiv}_j[i] \cdot v_{j_i} v_{j_i}'$$

where the first i-1 elements of Householder vector v_{j_i} are zero, and $v_{j_i}[i] = 1$.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. m >= 0. The number of rows of all the matrices A_j in the batch.
- `[in] n`: rocblas_int. n >= 0. The number of columns of all the matrices A_j in the batch.
- `[inout] A`: pointer to type. Array on the GPU (the size depends on the value of strideA). On entry, the m-by-n matrices A_j to be factored. On exit, the elements on and above the diagonal contain the factor R_j. The elements below the diagonal are the last m - i elements of Householder vector v_(j_i).
- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of matrices A_j.
- `[in] strideA`: rocblas_stride. Stride from the start of one matrix A_j to the next one A_(j+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n.
- `[out] ipiv`: pointer to type. Array on the GPU (the size depends on the value of strideP). Contains the vectors ipiv_j of corresponding Householder scalars.
- `[in] strideP`: rocblas_stride. Stride from the start of one vector ipiv_j to the next one ipiv_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= min(m,n).
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>geqrf()

rocblas_status **rocsolver_zgeqrf** (rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, rocblas_double_complex **ipiv*)

rocblas_status **rocsolver_cgeqrf** (rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, rocblas_float_complex **ipiv*)

rocblas_status **rocsolver_dgeqrf** (rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, double **ipiv*)

rocblas_status **rocsolver_sgeqrf** (rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, float **ipiv*)

GEQRF computes a QR factorization of a general m-by-n matrix A.

(This is the blocked version of the algorithm).

The factorization has the form

$$A = Q \begin{bmatrix} R \\ 0 \end{bmatrix}$$

where R is upper triangular (upper trapezoidal if m < n), and Q is a m-by-m orthogonal/unitary matrix represented as the product of Householder matrices

$$Q = H_1 H_2 \cdots H_k, \quad \text{with } k = \text{min}(m, n)$$

Each Householder matrix H_i is given by

$$H_i = I - \text{ipiv}[i] \cdot v_i v_i'$$

where the first i-1 elements of the Householder vector v_i are zero, and $v_i[i] = 1$.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. m >= 0. The number of rows of the matrix A.
- `[in] n`: rocblas_int. n >= 0. The number of columns of the matrix A.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the m-by-n matrix to be factored. On exit, the elements on and above the diagonal contain the factor R; the elements below the diagonal are the last m - i elements of Householder vector v_i.
- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of A.
- `[out] ipiv`: pointer to type. Array on the GPU of dimension min(m,n). The Householder scalars.

rocsolver_<type>geqrf_batched()

rocblas_status **rocsolver_zgeqrf_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_double_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_double_complex **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cgeqrf_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_float_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_float_complex **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dgeqrf_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, double ***const** *A*[], **const** rocblas_int *lda*, double **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_sgeqrf_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, float ***const** *A*[], **const** rocblas_int *lda*, float **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

GEQRF_BATCHED computes the QR factorization of a batch of general m-by-n matrices.

(This is the blocked version of the algorithm).

The factorization of matrix A_j in the batch has the form

$$A_j = Q_j \left[\begin{array}{c} R_j \\ 0 \end{array} \right]$$

where R_j is upper triangular (upper trapezoidal if m < n), and Q_j is a m-by-m orthogonal/unitary matrix represented as the product of Householder matrices

$$Q_j = H_{j_1} H_{j_2} \cdots H_{j_k}, \quad \text{with } k = \text{min}(m,n)$$

Each Householder matrix H_{j_i} is given by

$$H_{j_i} = I - \text{ipiv}_j[i] \cdot v_{j_i} v_{j_i}'$$

where the first i-1 elements of Householder vector v_{j_i} are zero, and $v_{j_i}[i] = 1$.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. m >= 0. The number of rows of all the matrices A_j in the batch.
- `[in] n`: rocblas_int. n >= 0. The number of columns of all the matrices A_j in the batch.
- `[inout] A`: Array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n. On entry, the m-by-n matrices A_j to be factored. On exit, the elements on and above the diagonal contain the factor R_j. The elements below the diagonal are the last m - i elements of Householder vector v_(j_i).

- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of matrices A_j.
- `[out] ipiv`: pointer to type. Array on the GPU (the size depends on the value of strideP). Contains the vectors ipiv_j of corresponding Householder scalars.
- `[in] strideP`: rocblas_stride. Stride from the start of one vector ipiv_j to the next one ipiv_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= min(m,n).
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>geqrf_strided_batched()

rocblas_status **rocsolver_zgeqrf_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_double_complex **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cgeqrf_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_float_complex **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dgeqrf_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, double **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_sgeqrf_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, float **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

GEQRF_STRIDED_BATCHED computes the QR factorization of a batch of general m-by-n matrices.

(This is the blocked version of the algorithm).

The factorization of matrix A_j in the batch has the form

$$A_j = Q_j \left[\begin{array}{c} R_j \\ 0 \end{array} \right]$$

where R_j is upper triangular (upper trapezoidal if m < n), and Q_j is a m-by-m orthogonal/unitary matrix represented as the product of Householder matrices

$$Q_j = H_{j_1} H_{j_2} \cdots H_{j_k}, \quad \text{with } k = \text{min}(m, n)$$

Each Householder matrix H_{j_i} is given by

$$H_{j_i} = I - \text{ipiv}_j[i] \cdot v_{j_i} v_{j_i}'$$

where the first i-1 elements of Householder vector v_{j_i} are zero, and $v_{j_i}[i] = 1$.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. m >= 0. The number of rows of all the matrices A_j in the batch.
- `[in] n`: rocblas_int. n >= 0. The number of columns of all the matrices A_j in the batch.
- `[inout] A`: pointer to type. Array on the GPU (the size depends on the value of strideA). On entry, the m-by-n matrices A_j to be factored. On exit, the elements on and above the diagonal contain the factor R_j. The elements below the diagonal are the last m - i elements of Householder vector v_(j_i).
- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of matrices A_j.
- `[in] strideA`: rocblas_stride. Stride from the start of one matrix A_j to the next one A_(j+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n.
- `[out] ipiv`: pointer to type. Array on the GPU (the size depends on the value of strideP). Contains the vectors ipiv_j of corresponding Householder scalars.
- `[in] strideP`: rocblas_stride. Stride from the start of one vector ipiv_j to the next one ipiv_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= min(m,n).
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>gerq2()

rocblas_status **rocsolver_zgerq2**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, rocblas_double_complex **ipiv*)

rocblas_status **rocsolver_cgerq2**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, rocblas_float_complex **ipiv*)

rocblas_status **rocsolver_dgerq2**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, double **ipiv*)

rocblas_status **rocsolver_sgerq2**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, float **ipiv*)

GERQ2 computes a RQ factorization of a general m-by-n matrix A.

(This is the unblocked version of the algorithm).

The factorization has the form

$$A = \begin{bmatrix} 0 & R \end{bmatrix} Q$$

where R is upper triangular (upper trapezoidal if m > n), and Q is a n-by-n orthogonal/unitary matrix represented as the product of Householder matrices

$$Q = H_1' H_2' \cdots H_k', \quad \text{with } k = \text{min}(m, n).$$

Each Householder matrix H_i is given by

$$H_i = I - \text{ipiv}[i] \cdot v_i v_i'$$

where the last n-i elements of the Householder vector v_i are zero, and $v_i[i] = 1$.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. m >= 0. The number of rows of the matrix A.
- `[in] n`: rocblas_int. n >= 0. The number of columns of the matrix A.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the m-by-n matrix to be factored. On exit, the elements on and above the (m-n)-th subdiagonal (when m >= n) or the (n-m)-th superdiagonal (when n > m) contain the factor R; the elements below the sub/superdiagonal are the first i - 1 elements of Householder vector v_i.
- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of A.
- `[out] ipiv`: pointer to type. Array on the GPU of dimension min(m,n). The Householder scalars.

rocsolver_<type>gerq2_batched()

rocblas_status **rocsolver_zgerq2_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_double_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_double_complex **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cgerq2_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_float_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_float_complex **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dgerq2_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, double ***const** *A*[], **const** rocblas_int *lda*, double **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_sgerq2_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, float ***const** *A*[], **const** rocblas_int *lda*, float **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

GERQ2_BATCHED computes the RQ factorization of a batch of general m-by-n matrices.

(This is the unblocked version of the algorithm).

The factorization of matrix A_j in the batch has the form

$$A_j = \begin{bmatrix} 0 & R_j \end{bmatrix} Q_j$$

where R_j is upper triangular (upper trapezoidal if m > n), and Q_j is a n-by-n orthogonal/unitary matrix represented as the product of Householder matrices

$$Q_j = H'_{j_1} H'_{j_2} \cdots H'_{j_k}, \quad \text{with } k = \text{min}(m, n).$$

Each Householder matrices H_{j_i} is given by

$$H_{j_i} = I - \text{ipiv}_j[i] \cdot v_{j_i} v'_{j_i}$$

where the last n-i elements of Householder vector v_{j_i} are zero, and $v_{j_i}[i] = 1$.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. m >= 0. The number of rows of all the matrices A_j in the batch.
- `[in] n`: rocblas_int. n >= 0. The number of columns of all the matrices A_j in the batch.
- `[inout] A`: Array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n. On entry, the m-by-n matrices A_j to be factored. On exit, the elements on and above the (m-n)-th subdiagonal (when m >= n) or the (n-m)-th superdiagonal (when n > m) contain the factor R_j; the elements below the sub/superdiagonal are the first i - 1 elements of Householder vector v_(j_i).
- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of matrices A_j.
- `[out] ipiv`: pointer to type. Array on the GPU (the size depends on the value of strideP). Contains the vectors ipiv_j of corresponding Householder scalars.
- `[in] strideP`: rocblas_stride. Stride from the start of one vector ipiv_j to the next one ipiv_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= min(m,n).
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>gerq2_strided_batched()

rocblas_status **rocsolver_zgerq2_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_double_complex **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cgerq2_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_float_complex **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dgerq2_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, double **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*) rocblas_status **rocsolver_sgerq2_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, float **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

GERQ2_STRIDED_BATCHED computes the RQ factorization of a batch of general m-by-n matrices.

(This is the unblocked version of the algorithm).

The factorization of matrix A_j in the batch has the form

$$A_j = \begin{bmatrix} 0 & R_j \end{bmatrix} Q_j$$

where R_j is upper triangular (upper trapezoidal if m > n), and Q_j is a n-by-n orthogonal/unitary matrix represented as the product of Householder matrices

$$Q_j = H_{j_1}' H_{j_2}' \cdots H_{j_k}', \quad \text{with } k = \text{min}(m, n).$$

Each Householder matrices H_{j_i} is given by

$$H_{j_i} = I - \text{ipiv}_j[i] \cdot v_{j_i} v_{j_i}'$$

where the last n-i elements of Householder vector v_{j_i} are zero, and $v_{j_i}[i] = 1$.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. m >= 0. The number of rows of all the matrices A_j in the batch.
- `[in] n`: rocblas_int. n >= 0. The number of columns of all the matrices A_j in the batch.
- `[inout] A`: pointer to type. Array on the GPU (the size depends on the value of strideA). On entry, the m-by-n matrices A_j to be factored. On exit, the elements on and above the (m-n)-th subdiagonal (when m >= n) or the (n-m)-th superdiagonal (when n > m) contain the factor R_j; the elements below the sub/superdiagonal are the first i - 1 elements of Householder vector v_(j_i).
- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of matrices A_j.
- `[in] strideA`: rocblas_stride. Stride from the start of one matrix A_j to the next one A_(j+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n.
- `[out] ipiv`: pointer to type. Array on the GPU (the size depends on the value of strideP). Contains the vectors ipiv_j of corresponding Householder scalars.
- `[in] strideP`: rocblas_stride. Stride from the start of one vector ipiv_j to the next one ipiv_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= min(m,n).
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>gerqf()

rocblas_status **rocsolver_zgerqf** (rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, rocblas_double_complex **ipiv*)

rocblas_status **rocsolver_cgerqf** (rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, rocblas_float_complex **ipiv*)

rocblas_status **rocsolver_dgerqf** (rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, double **ipiv*)

rocblas_status **rocsolver_sgerqf** (rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, float **ipiv*)

GERQF computes a RQ factorization of a general m-by-n matrix A.

(This is the blocked version of the algorithm).

The factorization has the form

$$A = \left[\begin{array}{cc} 0 & R \end{array}\right] Q$$

where R is upper triangular (upper trapezoidal if m > n), and Q is a n-by-n orthogonal/unitary matrix represented as the product of Householder matrices

$$Q = H_1' H_2' \cdots H_k', \quad \text{with } k = \text{min}(m,n).$$

Each Householder matrix H_i is given by

$$H_i = I - \text{ipiv}[i] \cdot v_i v_i'$$

where the last n-i elements of the Householder vector v_i are zero, and $v_i[i] = 1$.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. m >= 0. The number of rows of the matrix A.
- `[in] n`: rocblas_int. n >= 0. The number of columns of the matrix A.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the m-by-n matrix to be factored. On exit, the elements on and above the (m-n)-th subdiagonal (when m >= n) or the (n-m)-th superdiagonal (when n > m) contain the factor R; the elements below the sub/superdiagonal are the first i - 1 elements of Householder vector v_i.
- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of A.
- `[out] ipiv`: pointer to type. Array on the GPU of dimension min(m,n). The Householder scalars.

rocsolver_<type>gerqf_batched()

rocblas_status **rocsolver_zgerqf_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_double_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_double_complex **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cgerqf_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_float_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_float_complex **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dgerqf_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, double ***const** *A*[], **const** rocblas_int *lda*, double **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_sgerqf_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, float ***const** *A*[], **const** rocblas_int *lda*, float **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

GERQF_BATCHED computes the RQ factorization of a batch of general m-by-n matrices.

(This is the blocked version of the algorithm).

The factorization of matrix A_j in the batch has the form

$$A_j = \begin{bmatrix} 0 & R_j \end{bmatrix} Q_j$$

where R_j is upper triangular (upper trapezoidal if m > n), and Q_j is a n-by-n orthogonal/unitary matrix represented as the product of Householder matrices

$$Q_j = H_{j_1}' H_{j_2}' \cdots H_{j_k}', \quad \text{with } k = \text{min}(m, n).$$

Each Householder matrices H_{j_i} is given by

$$H_{j_i} = I - \text{ipiv}_j[i] \cdot v_{j_i} v_{j_i}'$$

where the last n-i elements of Householder vector v_{j_i} are zero, and $v_{j_i}[i] = 1$.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. m >= 0. The number of rows of all the matrices A_j in the batch.
- `[in] n`: rocblas_int. n >= 0. The number of columns of all the matrices A_j in the batch.
- `[inout] A`: Array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n. On entry, the m-by-n matrices A_j to be factored. On exit, the elements on and above the (m-n)-th subdiagonal (when m >= n) or the (n-m)-th superdiagonal (when n > m) contain the factor R_j; the elements below the sub/superdiagonal are the first i - 1 elements of Householder vector v_(j_i).

- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of matrices A_j.
- `[out] ipiv`: pointer to type. Array on the GPU (the size depends on the value of strideP). Contains the vectors ipiv_j of corresponding Householder scalars.
- `[in] strideP`: rocblas_stride. Stride from the start of one vector ipiv_j to the next one ipiv_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= min(m,n).
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>gerqf_strided_batched()

rocblas_status **rocsolver_zgerqf_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_double_complex **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cgerqf_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_float_complex **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dgerqf_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, double **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_sgerqf_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, float **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

GERQF_STRIDED_BATCHED computes the RQ factorization of a batch of general m-by-n matrices.

(This is the blocked version of the algorithm).

The factorization of matrix A_j in the batch has the form

$$A_j = \begin{bmatrix} 0 & R_j \end{bmatrix} Q_j$$

where R_j is upper triangular (upper trapezoidal if m > n), and Q_j is a n-by-n orthogonal/unitary matrix represented as the product of Householder matrices

$$Q_j = H_{j_1}' H_{j_2}' \cdots H_{j_k}', \quad \text{with } k = \text{min}(m, n).$$

Each Householder matrices H_{j_i} is given by

$$H_{j_i} = I - \text{ipiv}_j[i] \cdot v_{j_i} v_{j_i}'$$

where the last n-i elements of Householder vector v_{j_i} are zero, and $v_{j_i}[i] = 1$.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. m >= 0. The number of rows of all the matrices A_j in the batch.
- `[in] n`: rocblas_int. n >= 0. The number of columns of all the matrices A_j in the batch.
- `[inout] A`: pointer to type. Array on the GPU (the size depends on the value of strideA). On entry, the m-by-n matrices A_j to be factored. On exit, the elements on and above the (m-n)-th subdiagonal (when m >= n) or the (n-m)-th superdiagonal (when n > m) contain the factor R_j; the elements below the sub/superdiagonal are the first i - 1 elements of Householder vector v_(j_i).
- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of matrices A_j.
- `[in] strideA`: rocblas_stride. Stride from the start of one matrix A_j to the next one A_(j+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n.
- `[out] ipiv`: pointer to type. Array on the GPU (the size depends on the value of strideP). Contains the vectors ipiv_j of corresponding Householder scalars.
- `[in] strideP`: rocblas_stride. Stride from the start of one vector ipiv_j to the next one ipiv_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= min(m,n).
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>geql2()

rocblas_status **rocsolver_zgeql2**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, rocblas_double_complex **ipiv*)

rocblas_status **rocsolver_cgeql2**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, rocblas_float_complex **ipiv*)

rocblas_status **rocsolver_dgeql2**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, double **ipiv*)

rocblas_status **rocsolver_sgeql2**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, float **ipiv*)

GEQL2 computes a QL factorization of a general m-by-n matrix A.

(This is the unblocked version of the algorithm).

The factorization has the form

$$A = Q \begin{bmatrix} 0 \\ L \end{bmatrix}$$

where L is lower triangular (lower trapezoidal if m < n), and Q is a m-by-m orthogonal/unitary matrix represented as the product of Householder matrices

$$Q = H_k H_{k-1} \cdots H_1, \quad \text{with } k = \text{min}(m, n)$$

Each Householder matrix H_i is given by

$$H_i = I - \text{ipiv}[i] \cdot v_i v_i'$$

where the last m-i elements of the Householder vector v_i are zero, and $v_i[i] = 1$.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. m >= 0. The number of rows of the matrix A.
- `[in] n`: rocblas_int. n >= 0. The number of columns of the matrix A.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the m-by-n matrix to be factored. On exit, the elements on and below the (m-n)-th subdiagonal (when m >= n) or the (n-m)-th superdiagonal (when n > m) contain the factor L; the elements above the sub/superdiagonal are the first i - 1 elements of Householder vector v_i.
- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of A.
- `[out] ipiv`: pointer to type. Array on the GPU of dimension min(m,n). The Householder scalars.

rocsolver_<type>geql2_batched()

rocblas_status **rocsolver_zgeql2_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_double_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_double_complex **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cgeql2_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_float_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_float_complex **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dgeql2_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, double ***const** *A*[], **const** rocblas_int *lda*, double **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_sgeql2_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, float ***const** *A*[], **const** rocblas_int *lda*, float **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

GEQL2_BATCHED computes the QL factorization of a batch of general m-by-n matrices.

(This is the unblocked version of the algorithm).

The factorization of matrix A_j in the batch has the form

$$A_j = Q_j \begin{bmatrix} 0 \\ L_j \end{bmatrix}$$

where L_j is lower triangular (lower trapezoidal if m < n), and Q_j is a m-by-m orthogonal/unitary matrix represented as the product of Householder matrices

$$Q = H_{j_k} H_{j_{k-1}} \cdots H_{j_1}, \quad \text{with } k = \min(m, n)$$

Each Householder matrix H_{j_i} is given by

$$H_{j_i} = I - \text{ipiv}_j[i] \cdot v_{j_i} v_{j_i}'$$

where the last m-i elements of the Householder vector v_{j_i} are zero, and $v_{j_i}[i] = 1$.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. m >= 0. The number of rows of all the matrices A_j in the batch.
- `[in] n`: rocblas_int. n >= 0. The number of columns of all the matrices A_j in the batch.
- `[inout] A`: Array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n. On entry, the m-by-n matrices A_j to be factored. On exit, the elements on and below the (m-n)-th subdiagonal (when m >= n) or the (n-m)-th superdiagonal (when n > m) contain the factor L_j; the elements above the sub/superdiagonal are the first i - 1 elements of Householder vector v_(j_i).
- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of matrices A_j.
- `[out] ipiv`: pointer to type. Array on the GPU (the size depends on the value of strideP). Contains the vectors ipiv_j of corresponding Householder scalars.
- `[in] strideP`: rocblas_stride. Stride from the start of one vector ipiv_j to the next one ipiv_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= min(m,n).
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>geql2_strided_batched()

rocblas_status **`rocsolver_zgeql2_strided_batched`**(rocblas_handle *handle*, **`const`** rocblas_int *m*, **`const`** rocblas_int *n*, rocblas_double_complex **A*, **`const`** rocblas_int *lda*, **`const`** rocblas_stride *strideA*, rocblas_double_complex **ipiv*, **`const`** rocblas_stride *strideP*, **`const`** rocblas_int *batch_count*)

rocblas_status **`rocsolver_cgeql2_strided_batched`**(rocblas_handle *handle*, **`const`** rocblas_int *m*, **`const`** rocblas_int *n*, rocblas_float_complex **A*, **`const`** rocblas_int *lda*, **`const`** rocblas_stride *strideA*, rocblas_float_complex **ipiv*, **`const`** rocblas_stride *strideP*, **`const`** rocblas_int *batch_count*)

rocblas_status **`rocsolver_dgeql2_strided_batched`**(rocblas_handle *handle*, **`const`** rocblas_int *m*, **`const`** rocblas_int *n*, double **A*, **`const`** rocblas_int *lda*, **`const`** rocblas_stride *strideA*, double **ipiv*, **`const`** rocblas_stride *strideP*, **`const`** rocblas_int *batch_count*) rocblas_status **rocsolver_sgeql2_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, float **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

GEQL2_STRIDED_BATCHED computes the QL factorization of a batch of general m-by-n matrices.

(This is the unblocked version of the algorithm).

The factorization of matrix A_j in the batch has the form

$$A_j = Q_j \left[\begin{array}{c} 0 \\ L_j \end{array} \right]$$

where L_j is lower triangular (lower trapezoidal if m < n), and Q_j is a m-by-m orthogonal/unitary matrix represented as the product of Householder matrices

$$Q = H_{j_k} H_{j_{k-1}} \cdots H_{j_1}, \quad \text{with } k = \text{min}(m, n)$$

Each Householder matrix H_{j_i} is given by

$$H_{j_i} = I - \text{ipiv}_j[i] \cdot v_{j_i} v_{j_i}'$$

where the last m-i elements of the Householder vector v_{j_i} are zero, and $v_{j_i}[i] = 1$.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. m >= 0. The number of rows of all the matrices A_j in the batch.
- `[in] n`: rocblas_int. n >= 0. The number of columns of all the matrices A_j in the batch.
- `[inout] A`: pointer to type. Array on the GPU (the size depends on the value of strideA). On entry, the m-by-n matrices A_j to be factored. On exit, the elements on and below the (m-n)-th subdiagonal (when m >= n) or the (n-m)-th superdiagonal (when n > m) contain the factor L_j; the elements above the sub/superdiagonal are the first i - 1 elements of Householder vector v_(j_i).
- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of matrices A_j.
- `[in] strideA`: rocblas_stride. Stride from the start of one matrix A_j to the next one A_(j+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n.
- `[out] ipiv`: pointer to type. Array on the GPU (the size depends on the value of strideP). Contains the vectors ipiv_j of corresponding Householder scalars.
- `[in] strideP`: rocblas_stride. Stride from the start of one vector ipiv_j to the next one ipiv_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= min(m,n).
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>geqlf()

rocblas_status **rocsolver_zgeqlf**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, rocblas_double_complex **ipiv*)

rocblas_status **rocsolver_cgeqlf**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, rocblas_float_complex **ipiv*)

rocblas_status **rocsolver_dgeqlf**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, double **ipiv*)

rocblas_status **rocsolver_sgeqlf**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, float **ipiv*)

GEQLF computes a QL factorization of a general m-by-n matrix A.

(This is the blocked version of the algorithm).

The factorization has the form

$$A = Q \begin{bmatrix} 0 \\ L \end{bmatrix}$$

where L is lower triangular (lower trapezoidal if m < n), and Q is a m-by-m orthogonal/unitary matrix represented as the product of Householder matrices

$$Q = H_k H_{k-1} \cdots H_1, \quad \text{with } k = \text{min}(m, n)$$

Each Householder matrix H_i is given by

$$H_i = I - \text{ipiv}[i] \cdot v_i v_i'$$

where the last m-i elements of the Householder vector v_i are zero, and $v_i[i] = 1$.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. m >= 0. The number of rows of the matrix A.
- `[in] n`: rocblas_int. n >= 0. The number of columns of the matrix A.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the m-by-n matrix to be factored. On exit, the elements on and below the (m-n)-th subdiagonal (when m >= n) or the (n-m)-th superdiagonal (when n > m) contain the factor L; the elements above the sub/superdiagonal are the first i - 1 elements of Householder vector v_i.
- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of A.
- `[out] ipiv`: pointer to type. Array on the GPU of dimension min(m,n). The Householder scalars.

rocsolver_<type>geqlf_batched()

rocblas_status **rocsolver_zgeqlf_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_double_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_double_complex **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cgeqlf_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_float_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_float_complex **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dgeqlf_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, double ***const** *A*[], **const** rocblas_int *lda*, double **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_sgeqlf_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, float ***const** *A*[], **const** rocblas_int *lda*, float **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

GEQLF_BATCHED computes the QL factorization of a batch of general m-by-n matrices.

(This is the blocked version of the algorithm).

The factorization of matrix A_j in the batch has the form

$$A_j = Q_j \begin{bmatrix} 0 \\ L_j \end{bmatrix}$$

where L_j is lower triangular (lower trapezoidal if m < n), and Q_j is a m-by-m orthogonal/unitary matrix represented as the product of Householder matrices

$$Q = H_{j_k} H_{j_{k-1}} \cdots H_{j_1}, \quad \text{with } k = \text{min}(m, n)$$

Each Householder matrix H_{j_i} is given by

$$H_{j_i} = I - \text{ipiv}_j[i] \cdot v_{j_i} v_{j_i}'$$

where the last m-i elements of the Householder vector v_{j_i} are zero, and $v_{j_i}[i] = 1$.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. m >= 0. The number of rows of all the matrices A_j in the batch.
- `[in] n`: rocblas_int. n >= 0. The number of columns of all the matrices A_j in the batch.
- `[inout] A`: Array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n. On entry, the m-by-n matrices A_j to be factored. On exit, the elements on and below the (m-n)-th subdiagonal (when m >= n) or the (n-m)-th superdiagonal (when n > m) contain the factor L_j; the elements above the sub/superdiagonal are the first i - 1 elements of Householder vector v_(j_i).

- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of matrices A_j.
- `[out] ipiv`: pointer to type. Array on the GPU (the size depends on the value of strideP). Contains the vectors ipiv_j of corresponding Householder scalars.
- `[in] strideP`: rocblas_stride. Stride from the start of one vector ipiv_j to the next one ipiv_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= min(m,n).
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>geqlf_strided_batched()

rocblas_status **rocsolver_zgeqlf_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_double_complex **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cgeqlf_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_float_complex **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dgeqlf_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, double **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_sgeqlf_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, float **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

GEQLF_STRIDED_BATCHED computes the QL factorization of a batch of general m-by-n matrices.

(This is the blocked version of the algorithm).

The factorization of matrix A_j in the batch has the form

$$A_j = Q_j \left[\begin{array}{c} 0 \\ L_j \end{array} \right]$$

where L_j is lower triangular (lower trapezoidal if m < n), and Q_j is a m-by-m orthogonal/unitary matrix represented as the product of Householder matrices

$$Q = H_{j_k} H_{j_{k-1}} \cdots H_{j_1}, \quad \text{with } k = \min(m, n)$$

Each Householder matrix H_{j_i} is given by

$$H_{j_i} = I - \text{ipiv}_j[i] \cdot v_{j_i} v'_{j_i}$$

where the last m-i elements of the Householder vector v_{j_i} are zero, and $v_{j_i}[i] = 1$.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. m >= 0. The number of rows of all the matrices A_j in the batch.
- `[in] n`: rocblas_int. n >= 0. The number of columns of all the matrices A_j in the batch.
- `[inout] A`: pointer to type. Array on the GPU (the size depends on the value of strideA). On entry, the m-by-n matrices A_j to be factored. On exit, the elements on and below the (m-n)-th subdiagonal (when m >= n) or the (n-m)-th superdiagonal (when n > m) contain the factor L_j; the elements above the sub/superdiagonal are the first i - 1 elements of Householder vector v_(j_i).
- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of matrices A_j.
- `[in] strideA`: rocblas_stride. Stride from the start of one matrix A_j to the next one A_(j+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n.
- `[out] ipiv`: pointer to type. Array on the GPU (the size depends on the value of strideP). Contains the vectors ipiv_j of corresponding Householder scalars.
- `[in] strideP`: rocblas_stride. Stride from the start of one vector ipiv_j to the next one ipiv_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= min(m,n).
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>gelq2()

rocblas_status **rocsolver_zgelq2** (rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, rocblas_double_complex **ipiv*)

rocblas_status **rocsolver_cgelq2** (rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, rocblas_float_complex **ipiv*)

rocblas_status **rocsolver_dgelq2** (rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, double **ipiv*)

rocblas_status **rocsolver_sgelq2** (rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, float **ipiv*)

GELQ2 computes a LQ factorization of a general m-by-n matrix A.

(This is the unblocked version of the algorithm).

The factorization has the form

$$A = \begin{bmatrix} L & 0 \end{bmatrix} Q$$

where L is lower triangular (lower trapezoidal if m > n), and Q is a n-by-n orthogonal/unitary matrix represented as the product of Householder matrices

$$Q = H_k' H_{k-1}' \cdots H_1', \quad \text{with } k = \text{min}(m, n).$$

Each Householder matrix H_i is given by

$$H_i = I - \text{ipiv}[i] \cdot v_i' v_i$$

where the first i-1 elements of the Householder vector v_i are zero, and $v_i[i] = 1$.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. m >= 0. The number of rows of the matrix A.
- `[in] n`: rocblas_int. n >= 0. The number of columns of the matrix A.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the m-by-n matrix to be factored. On exit, the elements on and below the diagonal contain the factor L; the elements above the diagonal are the last n - i elements of Householder vector v_i.
- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of A.
- `[out] ipiv`: pointer to type. Array on the GPU of dimension min(m,n). The Householder scalars.

rocsolver_<type>gelq2_batched()

rocblas_status **rocsolver_zgelq2_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_double_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_double_complex **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cgelq2_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_float_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_float_complex **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dgelq2_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, double ***const** *A*[], **const** rocblas_int *lda*, double **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_sgelq2_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, float ***const** *A*[], **const** rocblas_int *lda*, float **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

GELQ2_BATCHED computes the LQ factorization of a batch of general m-by-n matrices.

(This is the unblocked version of the algorithm).

The factorization of matrix A_j in the batch has the form

$$A_j = \begin{bmatrix} L_j & 0 \end{bmatrix} Q_j$$

where L_j is lower triangular (lower trapezoidal if m > n), and Q_j is a n-by-n orthogonal/unitary matrix represented as the product of Householder matrices

$$Q_j = H_{j_k}' H_{j_{k-1}}' \cdots H_{j_1}', \quad \text{with } k = \text{min}(m, n).$$

Each Householder matrices H_{j_i} is given by

$$H_{j_i} = I - \text{ipiv}_j[i] \cdot v_{j_i}' v_{j_i}$$

where the first i-1 elements of Householder vector v_{j_i} are zero, and $v_{j_i}[i] = 1$.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. m >= 0. The number of rows of all the matrices A_j in the batch.
- `[in] n`: rocblas_int. n >= 0. The number of columns of all the matrices A_j in the batch.
- `[inout] A`: Array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n. On entry, the m-by-n matrices A_j to be factored. On exit, the elements on and below the diagonal contain the factor L_j. The elements above the diagonal are the last n - i elements of Householder vector v_(j_i).
- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of matrices A_j.
- `[out] ipiv`: pointer to type. Array on the GPU (the size depends on the value of strideP). Contains the vectors ipiv_j of corresponding Householder scalars.
- `[in] strideP`: rocblas_stride. Stride from the start of one vector ipiv_j to the next one ipiv_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= min(m,n).
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>gelq2_strided_batched()

rocblas_status **rocsolver_zgelq2_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_double_complex **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cgelq2_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_float_complex **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dgelq2_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, double **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_sgelq2_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, float **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

GELQ2_STRIDED_BATCHED computes the LQ factorization of a batch of general m-by-n matrices.

(This is the unblocked version of the algorithm).

The factorization of matrix A_j in the batch has the form

$$A_j = \left[\begin{array}{cc} L_j & 0 \end{array} \right] Q_j$$

where L_j is lower triangular (lower trapezoidal if m > n), and Q_j is a n-by-n orthogonal/unitary matrix represented as the product of Householder matrices

$$Q_j = H_{j_k}' H_{j_{k-1}}' \cdots H_{j_1}', \quad \text{with } k = \text{min}(m, n).$$

Each Householder matrices H_{j_i} is given by

$$H_{j_i} = I - \text{ipiv}_j[i] \cdot v_{j_i}' v_{j_i}$$

where the first i-1 elements of Householder vector v_{j_i} are zero, and $v_{j_i}[i] = 1$.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. m >= 0. The number of rows of all the matrices A_j in the batch.
- `[in] n`: rocblas_int. n >= 0. The number of columns of all the matrices A_j in the batch.
- `[inout] A`: pointer to type. Array on the GPU (the size depends on the value of strideA). On entry, the m-by-n matrices A_j to be factored. On exit, the elements on and below the diagonal contain the factor L_j. The elements above the diagonal are the last n - i elements of Householder vector v_(j_i).
- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of matrices A_j.
- `[in] strideA`: rocblas_stride. Stride from the start of one matrix A_j to the next one A_(j+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n.
- `[out] ipiv`: pointer to type. Array on the GPU (the size depends on the value of strideP). Contains the vectors ipiv_j of corresponding Householder scalars.
- `[in] strideP`: rocblas_stride. Stride from the start of one vector ipiv_j to the next one ipiv_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= min(m,n).
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>gelqf()

rocblas_status **rocsolver_zgelqf**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, rocblas_double_complex **ipiv*)

rocblas_status **rocsolver_cgelqf**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, rocblas_float_complex **ipiv*)

rocblas_status **rocsolver_dgelqf**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, double **ipiv*) rocblas_status **rocsolver_sgelqf**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, float **ipiv*)

GELQF computes a LQ factorization of a general m-by-n matrix A.

(This is the blocked version of the algorithm).

The factorization has the form

$$A = \begin{bmatrix} L & 0 \end{bmatrix} Q$$

where L is lower triangular (lower trapezoidal if m > n), and Q is a n-by-n orthogonal/unitary matrix represented as the product of Householder matrices

$$Q = H_k' H_{k-1}' \cdots H_1', \quad \text{with } k = \text{min}(m, n).$$

Each Householder matrix H_i is given by

$$H_i = I - \text{ipiv}[i] \cdot v_i' v_i$$

where the first i-1 elements of the Householder vector v_i are zero, and $v_i[i] = 1$.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. m >= 0. The number of rows of the matrix A.
- `[in] n`: rocblas_int. n >= 0. The number of columns of the matrix A.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the m-by-n matrix to be factored. On exit, the elements on and below the diagonal contain the factor L; the elements above the diagonal are the last n - i elements of Householder vector v_i.
- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of A.
- `[out] ipiv`: pointer to type. Array on the GPU of dimension min(m,n). The Householder scalars.

rocsolver_<type>gelqf_batched()

rocblas_status **rocsolver_zgelqf_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_double_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_double_complex **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cgelqf_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_float_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_float_complex **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dgelqf_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, double ***const** *A*[], **const** rocblas_int *lda*, double **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*) rocblas_status **rocsolver_sgelqf_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, float ***const** *A*[], **const** rocblas_int *lda*, float **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

GELQF_BATCHED computes the LQ factorization of a batch of general m-by-n matrices.

(This is the blocked version of the algorithm).

The factorization of matrix A_j in the batch has the form

$$A_j = \begin{bmatrix} L_j & 0 \end{bmatrix} Q_j$$

where L_j is lower triangular (lower trapezoidal if m > n), and Q_j is a n-by-n orthogonal/unitary matrix represented as the product of Householder matrices

$$Q_j = H_{j_k}' H_{j_{k-1}}' \cdots H_{j_1}', \quad \text{with } k = \text{min}(m, n).$$

Each Householder matrices H_{j_i} is given by

$$H_{j_i} = I - \text{ipiv}_j[i] \cdot v_{j_i}' v_{j_i}$$

where the first i-1 elements of Householder vector v_{j_i} are zero, and $v_{j_i}[i] = 1$.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. m >= 0. The number of rows of all the matrices A_j in the batch.
- `[in] n`: rocblas_int. n >= 0. The number of columns of all the matrices A_j in the batch.
- `[inout] A`: Array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n. On entry, the m-by-n matrices A_j to be factored. On exit, the elements on and below the diagonal contain the factor L_j. The elements above the diagonal are the last n - i elements of Householder vector v_(j_i).
- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of matrices A_j.
- `[out] ipiv`: pointer to type. Array on the GPU (the size depends on the value of strideP). Contains the vectors ipiv_j of corresponding Householder scalars.
- `[in] strideP`: rocblas_stride. Stride from the start of one vector ipiv_j to the next one ipiv_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= min(m,n).
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>gelqf_strided_batched()

rocblas_status **rocsolver_zgelqf_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_double_complex **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cgelqf_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_float_complex **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dgelqf_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, double **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_sgelqf_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, float **ipiv*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

GELQF_STRIDED_BATCHED computes the LQ factorization of a batch of general m-by-n matrices.

(This is the blocked version of the algorithm).

The factorization of matrix A_j in the batch has the form

$$A_j = \left[\begin{array}{cc} L_j & 0 \end{array}\right] Q_j$$

where L_j is lower triangular (lower trapezoidal if m > n), and Q_j is a n-by-n orthogonal/unitary matrix represented as the product of Householder matrices

$$Q_j = H_{j_k}' H_{j_{k-1}}' \cdots H_{j_1}', \quad \text{with } k = \text{min}(m, n).$$

Each Householder matrices H_{j_i} is given by

$$H_{j_i} = I - \text{ipiv}_j[i] \cdot v_{j_i}' v_{j_i}$$

where the first i-1 elements of Householder vector v_{j_i} are zero, and $v_{j_i}[i] = 1$.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. m >= 0. The number of rows of all the matrices A_j in the batch.

- `[in] n`: rocblas_int. n >= 0. The number of columns of all the matrices A_j in the batch.
- `[inout] A`: pointer to type. Array on the GPU (the size depends on the value of strideA). On entry, the m-by-n matrices A_j to be factored. On exit, the elements on and below the diagonal contain the factor L_j. The elements above the diagonal are the last n - i elements of Householder vector v_(j_i).
- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of matrices A_j.
- `[in] strideA`: rocblas_stride. Stride from the start of one matrix A_j to the next one A_(j+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n.
- `[out] ipiv`: pointer to type. Array on the GPU (the size depends on the value of strideP). Contains the vectors ipiv_j of corresponding Householder scalars.
- `[in] strideP`: rocblas_stride. Stride from the start of one vector ipiv_j to the next one ipiv_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= min(m,n).
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

3.3.3 Problem and matrix reductions

List of reductions

- *rocsolver_<type>gebd2()*
- *rocsolver_<type>gebd2_batched()*
- *rocsolver_<type>gebd2_strided_batched()*
- *rocsolver_<type>gebrd()*
- *rocsolver_<type>gebrd_batched()*
- *rocsolver_<type>gebrd_strided_batched()*
- *rocsolver_<type>sytd2()*
- *rocsolver_<type>sytd2_batched()*
- *rocsolver_<type>sytd2_strided_batched()*
- *rocsolver_<type>hetd2()*
- *rocsolver_<type>hetd2_batched()*
- *rocsolver_<type>hetd2_strided_batched()*
- *rocsolver_<type>sytrd()*
- *rocsolver_<type>sytrd_batched()*
- *rocsolver_<type>sytrd_strided_batched()*
- *rocsolver_<type>hetrd()*
- *rocsolver_<type>hetrd_batched()*
- *rocsolver_<type>hetrd_strided_batched()*
- *rocsolver_<type>sygs2()*
- *rocsolver_<type>sygs2_batched()*
- *rocsolver_<type>sygs2_strided_batched()*

- *rocsolver_<type>hegs2()*
- *rocsolver_<type>hegs2_batched()*
- *rocsolver_<type>hegs2_strided_batched()*
- *rocsolver_<type>sygst()*
- *rocsolver_<type>sygst_batched()*
- *rocsolver_<type>sygst_strided_batched()*
- *rocsolver_<type>hegst()*
- *rocsolver_<type>hegst_batched()*
- *rocsolver_<type>hegst_strided_batched()*

rocsolver_<type>gebd2()

rocblas_status **rocsolver_zgebd2** (rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, double **D*, double **E*, rocblas_double_complex **tauq*, rocblas_double_complex **taup*)

rocblas_status **rocsolver_cgebd2** (rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, float **D*, float **E*, rocblas_float_complex **tauq*, rocblas_float_complex **taup*)

rocblas_status **rocsolver_dgebd2** (rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, double **D*, double **E*, double **tauq*, double **taup*)

rocblas_status **rocsolver_sgebd2** (rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, float **D*, float **E*, float **tauq*, float **taup*)

GEBD2 computes the bidiagonal form of a general m-by-n matrix A.

(This is the unblocked version of the algorithm).

The bidiagonal form is given by:

$$B = Q'AP$$

where B is upper bidiagonal if m >= n and lower bidiagonal if m < n, and Q and P are orthogonal/unitary matrices represented as the product of Householder matrices

$$\begin{array}{ll} Q = H_1 H_2 \cdots H_n \text{ and } P = G_1 G_2 \cdots G_{n-1}, & \text{if } m >= n, \text{ or} \\ Q = H_1 H_2 \cdots H_{m-1} \text{ and } P = G_1 G_2 \cdots G_m, & \text{if } m < n. \end{array}$$

Each Householder matrix H_i and G_i is given by

$$\begin{array}{ll} H_i = I - \text{tauq}[i] \cdot v_i v_i', & \text{and} \\ G_i = I - \text{taup}[i] \cdot u_i' u_i. & \end{array}$$

If m >= n, the first i-1 elements of the Householder vector v_i are zero, and $v_i[i] = 1$; while the first i elements of the Householder vector u_i are zero, and $u_i[i+1] = 1$. If m < n, the first i elements of the Householder vector v_i are zero, and $v_i[i+1] = 1$; while the first i-1 elements of the Householder vector u_i are zero, and $u_i[i] = 1$.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. m >= 0. The number of rows of the matrix A.
- `[in] n`: rocblas_int. n >= 0. The number of columns of the matrix A.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the m-by-n matrix to be factored. On exit, the elements on the diagonal and superdiagonal (if m >= n), or subdiagonal (if m < n) contain the bidiagonal form B. If m >= n, the elements below the diagonal are the last m - i elements of Householder vector v_i, and the elements above the superdiagonal are the last n - i - 1 elements of Householder vector u_i. If m < n, the elements below the subdiagonal are the last m - i - 1 elements of Householder vector v_i, and the elements above the diagonal are the last n - i elements of Householder vector u_i.
- `[in] lda`: rocblas_int. lda >= m. specifies the leading dimension of A.
- `[out] D`: pointer to real type. Array on the GPU of dimension min(m,n). The diagonal elements of B.
- `[out] E`: pointer to real type. Array on the GPU of dimension min(m,n)-1. The off-diagonal elements of B.
- `[out] tauq`: pointer to type. Array on the GPU of dimension min(m,n). The Householder scalars associated with matrix Q.
- `[out] taup`: pointer to type. Array on the GPU of dimension min(m,n). The Householder scalars associated with matrix P.

rocsolver_<type>gebd2_batched()

rocblas_status **rocsolver_zgebd2_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_double_complex ***const** *A*[], **const** rocblas_int *lda*, double **D*, **const** rocblas_stride *strideD*, double **E*, **const** rocblas_stride *strideE*, rocblas_double_complex **tauq*, **const** rocblas_stride *strideQ*, rocblas_double_complex **taup*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cgebd2_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_float_complex ***const** *A*[], **const** rocblas_int *lda*, float **D*, **const** rocblas_stride *strideD*, float **E*, **const** rocblas_stride *strideE*, rocblas_float_complex **tauq*, **const** rocblas_stride *strideQ*, rocblas_float_complex **taup*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dgebd2_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, double ***const** *A*[], **const** rocblas_int *lda*, double **D*, **const** rocblas_stride *strideD*, double **E*, **const** rocblas_stride *strideE*, double **tauq*, **const** rocblas_stride *strideQ*, double **taup*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*) rocblas_status **rocsolver_sgebd2_batched** (rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, float ***const** *A*[], **const** rocblas_int *lda*, float **D*, **const** rocblas_stride *strideD*, float **E*, **const** rocblas_stride *strideE*, float **tauq*, **const** rocblas_stride *strideQ*, float **taup*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

GEBD2_BATCHED computes the bidiagonal form of a batch of general m-by-n matrices.

(This is the unblocked version of the algorithm).

For each instance in the batch, the bidiagonal form is given by:

$$B_j = Q_j' A_j P_j$$

where B_j is upper bidiagonal if m >= n and lower bidiagonal if m < n, and Q_j and P_j are orthogonal/unitary matrices represented as the product of Householder matrices

$$\begin{array}{cl} Q_j = H_{j_1} H_{j_2} \cdots H_{j_n} \text{ and } P_j = G_{j_1} G_{j_2} \cdots G_{j_{n-1}}, & \text{if } m >= n, \text{ or} \\ Q_j = H_{j_1} H_{j_2} \cdots H_{j_{m-1}} \text{ and } P_j = G_{j_1} G_{j_2} \cdots G_{j_m}, & \text{if } m < n. \end{array}$$

Each Householder matrix H_{j_i} and G_{j_i} is given by

$$\begin{array}{cl} H_{j_i} = I - \text{tauq}_j[i] \cdot v_{j_i} v_{j_i}', & \text{and} \\ G_{j_i} = I - \text{taup}_j[i] \cdot u_{j_i}' u_{j_i}. & \end{array}$$

If m >= n, the first i-1 elements of the Householder vector v_{j_i} are zero, and $v_{j_i}[i] = 1$; while the first i elements of the Householder vector u_{j_i} are zero, and $u_{j_i}[i+1] = 1$. If m < n, the first i elements of the Householder vector v_{j_i} are zero, and $v_{j_i}[i+1] = 1$; while the first i-1 elements of the Householder vector u_{j_i} are zero, and $u_{j_i}[i] = 1$.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. m >= 0. The number of rows of all the matrices A_j in the batch.
- `[in] n`: rocblas_int. n >= 0. The number of columns of all the matrices A_j in the batch.
- `[inout] A`: Array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n. On entry, the m-by-n matrices A_j to be factored. On exit, the elements on the diagonal and superdiagonal (if m >= n), or subdiagonal (if m < n) contain the bidiagonal form B_j. If m >= n, the elements below the diagonal are the last m - i elements of Householder vector v_(j_i), and the elements above the superdiagonal are the last n - i - 1 elements of Householder vector u_(j_i). If m < n, the elements below the subdiagonal are the last m - i - 1 elements of Householder vector v_(j_i), and the elements above the diagonal are the last n - i elements of Householder vector u_(j_i).
- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of matrices A_j.
- `[out] D`: pointer to real type. Array on the GPU (the size depends on the value of strideD). The diagonal elements of B_j.
- `[in] strideD`: rocblas_stride. Stride from the start of one vector D_j to the next one D_(j+1). There is no restriction for the value of strideD. Normal use case is strideD >= min(m,n).

- `[out] E`: pointer to real type. Array on the GPU (the size depends on the value of strideE). The off-diagonal elements of B_j.
- `[in] strideE`: rocblas_stride. Stride from the start of one vector E_j to the next one E_(j+1). There is no restriction for the value of strideE. Normal use case is strideE >= min(m,n)-1.
- `[out] tauq`: pointer to type. Array on the GPU (the size depends on the value of strideQ). Contains the vectors tauq_j of Householder scalars associated with matrices Q_j.
- `[in] strideQ`: rocblas_stride. Stride from the start of one vector tauq_j to the next one tauq_(j+1). There is no restriction for the value of strideQ. Normal use is strideQ >= min(m,n).
- `[out] taup`: pointer to type. Array on the GPU (the size depends on the value of strideP). Contains the vectors taup_j of Householder scalars associated with matrices P_j.
- `[in] strideP`: rocblas_stride. Stride from the start of one vector taup_j to the next one taup_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= min(m,n).
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>gebd2_strided_batched()

rocblas_status **rocsolver_zgebd2_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, double **D*, **const** rocblas_stride *strideD*, double **E*, **const** rocblas_stride *strideE*, rocblas_double_complex **tauq*, **const** rocblas_stride *strideQ*, rocblas_double_complex **taup*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cgebd2_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, float **D*, **const** rocblas_stride *strideD*, float **E*, **const** rocblas_stride *strideE*, rocblas_float_complex **tauq*, **const** rocblas_stride *strideQ*, rocblas_float_complex **taup*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dgebd2_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, double **D*, **const** rocblas_stride *strideD*, double **E*, **const** rocblas_stride *strideE*, double **tauq*, **const** rocblas_stride *strideQ*, double **taup*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*) rocblas_status **rocsolver_sgebd2_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, float **D*, **const** rocblas_stride *strideD*, float **E*, **const** rocblas_stride *strideE*, float **tauq*, **const** rocblas_stride *strideQ*, float **taup*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

GEBD2_STRIDED_BATCHED computes the bidiagonal form of a batch of general m-by-n matrices.

(This is the unblocked version of the algorithm).

For each instance in the batch, the bidiagonal form is given by:

$$B_j = Q_j' A_j P_j$$

where B_j is upper bidiagonal if m >= n and lower bidiagonal if m < n, and Q_j and P_j are orthogonal/unitary matrices represented as the product of Householder matrices

$$\begin{array}{cl} Q_j = H_{j_1} H_{j_2} \cdots H_{j_n} \text{ and } P_j = G_{j_1} G_{j_2} \cdots G_{j_{n-1}}, & \text{if } m >= n, \text{ or} \\ Q_j = H_{j_1} H_{j_2} \cdots H_{j_{m-1}} \text{ and } P_j = G_{j_1} G_{j_2} \cdots G_{j_m}, & \text{if } m < n. \end{array}$$

Each Householder matrix H_{j_i} and G_{j_i} is given by

$$\begin{array}{cl} H_{j_i} = I - \text{tauq}_j[i] \cdot v_{j_i} v_{j_i}', & \text{and} \\ G_{j_i} = I - \text{taup}_j[i] \cdot u_{j_i}' u_{j_i}. & \end{array}$$

If m >= n, the first i-1 elements of the Householder vector v_{j_i} are zero, and $v_{j_i}[i] = 1$; while the first i elements of the Householder vector u_{j_i} are zero, and $u_{j_i}[i+1] = 1$. If m < n, the first i elements of the Householder vector v_{j_i} are zero, and $v_{j_i}[i+1] = 1$; while the first i-1 elements of the Householder vector u_{j_i} are zero, and $u_{j_i}[i] = 1$.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. m >= 0. The number of rows of all the matrices A_j in the batch.
- `[in] n`: rocblas_int. n >= 0. The number of columns of all the matrices A_j in the batch.
- `[inout] A`: pointer to type. Array on the GPU (the size depends on the value of strideA). On entry, the m-by-n matrices A_j to be factored. On exit, the elements on the diagonal and superdiagonal (if m >= n), or subdiagonal (if m < n) contain the bidiagonal form B_j. If m >= n, the elements below the diagonal are the last m - i elements of Householder vector v_(j_i), and the elements above the superdiagonal are the last n - i - 1 elements of Householder vector u_(j_i). If m < n, the elements below the subdiagonal are the last m - i - 1 elements of Householder vector v_(j_i), and the elements above the diagonal are the last n - i elements of Householder vector u_(j_i).
- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of matrices A_j.
- `[in] strideA`: rocblas_stride. Stride from the start of one matrix A_j to the next one A_(j+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n.

- `[out] D`: pointer to real type. Array on the GPU (the size depends on the value of strideD). The diagonal elements of B_j.
- `[in] strideD`: rocblas_stride. Stride from the start of one vector D_j to the next one D_(j+1). There is no restriction for the value of strideD. Normal use case is strideD >= min(m,n).
- `[out] E`: pointer to real type. Array on the GPU (the size depends on the value of strideE). The off-diagonal elements of B_j.
- `[in] strideE`: rocblas_stride. Stride from the start of one vector E_j to the next one E_(j+1). There is no restriction for the value of strideE. Normal use case is strideE >= min(m,n)-1.
- `[out] tauq`: pointer to type. Array on the GPU (the size depends on the value of strideQ). Contains the vectors tauq_j of Householder scalars associated with matrices Q_j.
- `[in] strideQ`: rocblas_stride. Stride from the start of one vector tauq_j to the next one tauq_(j+1). There is no restriction for the value of strideQ. Normal use is strideQ >= min(m,n).
- `[out] taup`: pointer to type. Array on the GPU (the size depends on the value of strideP). Contains the vectors taup_j of Householder scalars associated with matrices P_j.
- `[in] strideP`: rocblas_stride. Stride from the start of one vector taup_j to the next one taup_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= min(m,n).
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>gebrd()

rocblas_status **rocsolver_zgebrd**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, double **D*, double **E*, rocblas_double_complex **tauq*, rocblas_double_complex **taup*)

rocblas_status **rocsolver_cgebrd**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, float **D*, float **E*, rocblas_float_complex **tauq*, rocblas_float_complex **taup*)

rocblas_status **rocsolver_dgebrd**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, double **D*, double **E*, double **tauq*, double **taup*)

rocblas_status **rocsolver_sgebrd**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, float **D*, float **E*, float **tauq*, float **taup*)

GEBRD computes the bidiagonal form of a general m-by-n matrix A.

(This is the blocked version of the algorithm).

The bidiagonal form is given by:

$$B = Q'AP$$

where B is upper bidiagonal if m >= n and lower bidiagonal if m < n, and Q and P are orthogonal/unitary matrices represented as the product of Householder matrices

$$\begin{array}{cl} Q = H_1H_2\cdots H_n \text{ and } P = G_1G_2\cdots G_{n-1}, & \text{if } m >= n, \text{ or} \\ Q = H_1H_2\cdots H_{m-1} \text{ and } P = G_1G_2\cdots G_m, & \text{if } m < n. \end{array}$$

Each Householder matrix H_i and G_i is given by

$$H_i = I - \text{tauq}[i] \cdot v_i v_i', \quad \text{and}$$
$$G_i = I - \text{taup}[i] \cdot u_i' u_i.$$

If m >= n, the first i-1 elements of the Householder vector v_i are zero, and $v_i[i] = 1$; while the first i elements of the Householder vector u_i are zero, and $u_i[i+1] = 1$. If m < n, the first i elements of the Householder vector v_i are zero, and $v_i[i+1] = 1$; while the first i-1 elements of the Householder vector u_i are zero, and $u_i[i] = 1$.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. m >= 0. The number of rows of the matrix A.
- `[in] n`: rocblas_int. n >= 0. The number of columns of the matrix A.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the m-by-n matrix to be factored. On exit, the elements on the diagonal and superdiagonal (if m >= n), or subdiagonal (if m < n) contain the bidiagonal form B. If m >= n, the elements below the diagonal are the last m - i elements of Householder vector v_i, and the elements above the superdiagonal are the last n - i - 1 elements of Householder vector u_i. If m < n, the elements below the subdiagonal are the last m - i - 1 elements of Householder vector v_i, and the elements above the diagonal are the last n - i elements of Householder vector u_i.
- `[in] lda`: rocblas_int. lda >= m. specifies the leading dimension of A.
- `[out] D`: pointer to real type. Array on the GPU of dimension min(m,n). The diagonal elements of B.
- `[out] E`: pointer to real type. Array on the GPU of dimension min(m,n)-1. The off-diagonal elements of B.
- `[out] tauq`: pointer to type. Array on the GPU of dimension min(m,n). The Householder scalars associated with matrix Q.
- `[out] taup`: pointer to type. Array on the GPU of dimension min(m,n). The Householder scalars associated with matrix P.

rocsolver_<type>gebrd_batched()

rocblas_status **rocsolver_zgebrd_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_double_complex ***const** *A*[], **const** rocblas_int *lda*, double **D*, **const** rocblas_stride *strideD*, double **E*, **const** rocblas_stride *strideE*, rocblas_double_complex **tauq*, **const** rocblas_stride *strideQ*, rocblas_double_complex **taup*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cgebrd_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_float_complex ***const** *A*[], **const** rocblas_int *lda*, float **D*, **const** rocblas_stride *strideD*, float **E*, **const** rocblas_stride *strideE*, rocblas_float_complex **tauq*, **const** rocblas_stride *strideQ*, rocblas_float_complex **taup*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*) rocblas_status **rocsolver_dgebrd_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, double ***const** *A*[], **const** rocblas_int *lda*, double **D*, **const** rocblas_stride *strideD*, double **E*, **const** rocblas_stride *strideE*, double **tauq*, **const** rocblas_stride *strideQ*, double **taup*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_sgebrd_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, float ***const** *A*[], **const** rocblas_int *lda*, float **D*, **const** rocblas_stride *strideD*, float **E*, **const** rocblas_stride *strideE*, float **tauq*, **const** rocblas_stride *strideQ*, float **taup*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

GEBRD_BATCHED computes the bidiagonal form of a batch of general m-by-n matrices.

(This is the blocked version of the algorithm).

For each instance in the batch, the bidiagonal form is given by:

$$B_j = Q_j' A_j P_j$$

where B_j is upper bidiagonal if m >= n and lower bidiagonal if m < n, and Q_j and P_j are orthogonal/unitary matrices represented as the product of Householder matrices

$$\begin{array}{cl} Q_j = H_{j_1} H_{j_2} \cdots H_{j_n} \text{ and } P_j = G_{j_1} G_{j_2} \cdots G_{j_{n-1}}, & \text{if } m >= n, \text{ or} \\ Q_j = H_{j_1} H_{j_2} \cdots H_{j_{m-1}} \text{ and } P_j = G_{j_1} G_{j_2} \cdots G_{j_m}, & \text{if } m < n. \end{array}$$

Each Householder matrix H_{j_i} and G_{j_i} is given by

$$\begin{array}{cl} H_{j_i} = I - \text{tauq}_j[i] \cdot v_{j_i} v_{j_i}', & \text{and} \\ G_{j_i} = I - \text{taup}_j[i] \cdot u_{j_i}' u_{j_i}. & \end{array}$$

If m >= n, the first i-1 elements of the Householder vector v_{j_i} are zero, and $v_{j_i}[i] = 1$; while the first i elements of the Householder vector u_{j_i} are zero, and $u_{j_i}[i+1] = 1$. If m < n, the first i elements of the Householder vector v_{j_i} are zero, and $v_{j_i}[i+1] = 1$; while the first i-1 elements of the Householder vector u_{j_i} are zero, and $u_{j_i}[i] = 1$.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. m >= 0. The number of rows of all the matrices A_j in the batch.
- `[in] n`: rocblas_int. n >= 0. The number of columns of all the matrices A_j in the batch.
- `[inout] A`: Array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n. On entry, the m-by-n matrices A_j to be factored. On exit, the elements on the diagonal and superdiagonal (if m >= n), or subdiagonal (if m < n) contain the bidiagonal form B_j. If m >= n, the elements below the diagonal are the last m - i elements of Householder vector v_(j_i), and the elements above the superdiagonal are the last n - i - 1 elements of Householder vector u_(j_i). If m < n, the elements below the subdiagonal are the last m - i - 1 elements of Householder vector v_(j_i), and the elements above the diagonal are the last n - i elements of Householder vector u_(j_i).

- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of matrices A_j.
- `[out] D`: pointer to real type. Array on the GPU (the size depends on the value of strideD). The diagonal elements of B_j.
- `[in] strideD`: rocblas_stride. Stride from the start of one vector D_j to the next one D_(j+1). There is no restriction for the value of strideD. Normal use case is strideD >= min(m,n).
- `[out] E`: pointer to real type. Array on the GPU (the size depends on the value of strideE). The off-diagonal elements of B_j.
- `[in] strideE`: rocblas_stride. Stride from the start of one vector E_j to the next one E_(j+1). There is no restriction for the value of strideE. Normal use case is strideE >= min(m,n)-1.
- `[out] tauq`: pointer to type. Array on the GPU (the size depends on the value of strideQ). Contains the vectors tauq_j of Householder scalars associated with matrices Q_j.
- `[in] strideQ`: rocblas_stride. Stride from the start of one vector tauq_j to the next one tauq_(j+1). There is no restriction for the value of strideQ. Normal use is strideQ >= min(m,n).
- `[out] taup`: pointer to type. Array on the GPU (the size depends on the value of strideP). Contains the vectors taup_j of Householder scalars associated with matrices P_j.
- `[in] strideP`: rocblas_stride. Stride from the start of one vector taup_j to the next one taup_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= min(m,n).
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>gebrd_strided_batched()

rocblas_status **rocsolver_zgebrd_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, double **D*, **const** rocblas_stride *strideD*, double **E*, **const** rocblas_stride *strideE*, rocblas_double_complex **tauq*, **const** rocblas_stride *strideQ*, rocblas_double_complex **taup*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cgebrd_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, float **D*, **const** rocblas_stride *strideD*, float **E*, **const** rocblas_stride *strideE*, rocblas_float_complex **tauq*, **const** rocblas_stride *strideQ*, rocblas_float_complex **taup*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*) rocblas_status **rocsolver_dgebrd_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, double **D*, **const** rocblas_stride *strideD*, double **E*, **const** rocblas_stride *strideE*, double **tauq*, **const** rocblas_stride *strideQ*, double **taup*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_sgebrd_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, float **D*, **const** rocblas_stride *strideD*, float **E*, **const** rocblas_stride *strideE*, float **tauq*, **const** rocblas_stride *strideQ*, float **taup*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

GEBRD_STRIDED_BATCHED computes the bidiagonal form of a batch of general m-by-n matrices.

(This is the blocked version of the algorithm).

For each instance in the batch, the bidiagonal form is given by:

$$B_j = Q_j' A_j P_j$$

where B_j is upper bidiagonal if m >= n and lower bidiagonal if m < n, and Q_j and P_j are orthogonal/unitary matrices represented as the product of Householder matrices

$$\begin{array}{cl} Q_j = H_{j_1} H_{j_2} \cdots H_{j_n} \text{ and } P_j = G_{j_1} G_{j_2} \cdots G_{j_{n-1}}, & \text{if } m >= n, \text{ or} \\ Q_j = H_{j_1} H_{j_2} \cdots H_{j_{m-1}} \text{ and } P_j = G_{j_1} G_{j_2} \cdots G_{j_m}, & \text{if } m < n. \end{array}$$

Each Householder matrix H_{j_i} and G_{j_i} is given by

$$\begin{array}{cl} H_{j_i} = I - \text{tauq}_j[i] \cdot v_{j_i} v_{j_i}', & \text{and} \\ G_{j_i} = I - \text{taup}_j[i] \cdot u_{j_i}' u_{j_i}. & \end{array}$$

If m >= n, the first i-1 elements of the Householder vector v_{j_i} are zero, and $v_{j_i}[i] = 1$; while the first i elements of the Householder vector u_{j_i} are zero, and $u_{j_i}[i+1] = 1$. If m < n, the first i elements of the Householder vector v_{j_i} are zero, and $v_{j_i}[i+1] = 1$; while the first i-1 elements of the Householder vector u_{j_i} are zero, and $u_{j_i}[i] = 1$.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. m >= 0. The number of rows of all the matrices A_j in the batch.
- `[in] n`: rocblas_int. n >= 0. The number of columns of all the matrices A_j in the batch.
- `[inout] A`: pointer to type. Array on the GPU (the size depends on the value of strideA). On entry, the m-by-n matrices A_j to be factored. On exit, the elements on the diagonal and superdiagonal (if m >= n), or subdiagonal (if m < n) contain the bidiagonal form B_j. If m >= n, the elements below

the diagonal are the last m - i elements of Householder vector v_(j_i), and the elements above the superdiagonal are the last n - i - 1 elements of Householder vector u_(j_i). If m < n, the elements below the subdiagonal are the last m - i - 1 elements of Householder vector v_(j_i), and the elements above the diagonal are the last n - i elements of Householder vector u_(j_i).

- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of matrices A_j.
- `[in] strideA`: rocblas_stride. Stride from the start of one matrix A_j to the next one A_(j+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n.
- `[out] D`: pointer to real type. Array on the GPU (the size depends on the value of strideD). The diagonal elements of B_j.
- `[in] strideD`: rocblas_stride. Stride from the start of one vector D_j to the next one D_(j+1). There is no restriction for the value of strideD. Normal use case is strideD >= min(m,n).
- `[out] E`: pointer to real type. Array on the GPU (the size depends on the value of strideE). The off-diagonal elements of B_j.
- `[in] strideE`: rocblas_stride. Stride from the start of one vector E_j to the next one E_(j+1). There is no restriction for the value of strideE. Normal use case is strideE >= min(m,n)-1.
- `[out] tauq`: pointer to type. Array on the GPU (the size depends on the value of strideQ). Contains the vectors tauq_j of Householder scalars associated with matrices Q_j.
- `[in] strideQ`: rocblas_stride. Stride from the start of one vector tauq_j to the next one tauq_(j+1). There is no restriction for the value of strideQ. Normal use is strideQ >= min(m,n).
- `[out] taup`: pointer to type. Array on the GPU (the size depends on the value of strideP). Contains the vectors taup_j of Householder scalars associated with matrices P_j.
- `[in] strideP`: rocblas_stride. Stride from the start of one vector taup_j to the next one taup_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= min(m,n).
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>sytd2()

rocblas_status **rocsolver_dsytd2** (rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, double **D*, double **E*, double **tau*)

rocblas_status **rocsolver_ssytd2** (rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, float **D*, float **E*, float **tau*)

SYTD2 computes the tridiagonal form of a real symmetric matrix A.

(This is the unblocked version of the algorithm).

The tridiagonal form is given by:

$$T = Q'AQ$$

where T is symmetric tridiagonal and Q is an orthogonal matrix represented as the product of Householder matrices

$$\begin{array}{cl} Q = H_1 H_2 \cdots H_{n-1} & \text{if uplo indicates lower, or} \\ Q = H_{n-1} H_{n-2} \cdots H_1 & \text{if uplo indicates upper.} \end{array}$$

Each Householder matrix H_i is given by

$$H_i = I - \text{tau}[i] \cdot v_i v_i'$$

where tau[i] is the corresponding Householder scalar. When uplo indicates lower, the first i elements of the Householder vector v_i are zero, and $v_i[i+1] = 1$. If uplo indicates upper, the last n-i elements of the Householder vector v_i are zero, and $v_i[i] = 1$.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower part of the symmetric matrix A is stored. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.
- `[in] n`: rocblas_int. n >= 0. The number of rows and columns of the matrix A.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the matrix to be factored. On exit, if upper, then the elements on the diagonal and superdiagonal contain the tridiagonal form T; the elements above the superdiagonal contain the first i-1 elements of the Householder vectors v_i stored as columns. If lower, then the elements on the diagonal and subdiagonal contain the tridiagonal form T; the elements below the subdiagonal contain the last n-i-1 elements of the Householder vectors v_i stored as columns.
- `[in] lda`: rocblas_int. lda >= n. The leading dimension of A.
- `[out] D`: pointer to type. Array on the GPU of dimension n. The diagonal elements of T.
- `[out] E`: pointer to type. Array on the GPU of dimension n-1. The off-diagonal elements of T.
- `[out] tau`: pointer to type. Array on the GPU of dimension n-1. The Householder scalars.

rocsolver_<type>sytd2_batched()

rocblas_status **rocsolver_dsytd2_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, double ***const** *A*[], **const** rocblas_int *lda*, double **D*, **const** rocblas_stride *strideD*, double **E*, **const** rocblas_stride *strideE*, double **tau*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_ssytd2_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, float ***const** *A*[], **const** rocblas_int *lda*, float **D*, **const** rocblas_stride *strideD*, float **E*, **const** rocblas_stride *strideE*, float **tau*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

SYTD2_BATCHED computes the tridiagonal form of a batch of real symmetric matrices A_j.

(This is the unblocked version of the algorithm).

The tridiagonal form of A_j is given by:

$$T_j = Q_j' A_j Q_j$$

where T_j is symmetric tridiagonal and Q_j is an orthogonal matrix represented as the product of Householder matrices

$$Q_j = H_{j_1} H_{j_2} \cdots H_{j_{n-1}} \quad \text{if uplo indicates lower, or}$$
$$Q_j = H_{j_{n-1}} H_{j_{n-2}} \cdots H_{j_1} \quad \text{if uplo indicates upper.}$$

Each Householder matrix H_{j_i} is given by

$$H_{j_i} = I - \text{tau}_j[i] \cdot v_{j_i} v_{j_i}'$$

where $\text{tau}_j[i]$ is the corresponding Householder scalar. When uplo indicates lower, the first i elements of the Householder vector v_{j_i} are zero, and $v_{j_i}[i+1] = 1$. If uplo indicates upper, the last n-i elements of the Householder vector v_{j_i} are zero, and $v_{j_i}[i] = 1$.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower part of the symmetric matrix A_j is stored. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.
- `[in] n`: rocblas_int. n >= 0. The number of rows and columns of the matrices A_j.
- `[inout] A`: array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n. On entry, the matrices A_j to be factored. On exit, if upper, then the elements on the diagonal and superdiagonal contain the tridiagonal form T_j; the elements above the superdiagonal contain the first i-1 elements of the Householder vectors v_(j_i) stored as columns. If lower, then the elements on the diagonal and subdiagonal contain the tridiagonal form T_j; the elements below the subdiagonal contain the last n-i-1 elements of the Householder vectors v_(j_i) stored as columns.
- `[in] lda`: rocblas_int. lda >= n. The leading dimension of A_j.
- `[out] D`: pointer to type. Array on the GPU (the size depends on the value of strideD). The diagonal elements of T_j.
- `[in] strideD`: rocblas_stride. Stride from the start of one vector D_j to the next one D_(j+1). There is no restriction for the value of strideD. Normal use case is strideD >= n.
- `[out] E`: pointer to type. Array on the GPU (the size depends on the value of strideE). The off-diagonal elements of T_j.
- `[in] strideE`: rocblas_stride. Stride from the start of one vector E_j to the next one E_(j+1). There is no restriction for the value of strideE. Normal use case is strideE >= n-1.
- `[out] tau`: pointer to type. Array on the GPU (the size depends on the value of strideP). Contains the vectors tau_j of corresponding Householder scalars.
- `[in] strideP`: rocblas_stride. Stride from the start of one vector tau_j to the next one tau_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= n-1.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>sytd2_strided_batched()

rocblas_status **rocsolver_dsytd2_strided_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, double **D*, **const** rocblas_stride *strideD*, double **E*, **const** rocblas_stride *strideE*, double **tau*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_ssytd2_strided_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, float **D*, **const** rocblas_stride *strideD*, float **E*, **const** rocblas_stride *strideE*, float **tau*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

SYTD2_STRIDED_BATCHED computes the tridiagonal form of a batch of real symmetric matrices A_j.

(This is the unblocked version of the algorithm).

The tridiagonal form of A_j is given by:

$$T_j = Q_j' A_j Q_j$$

where T_j is symmetric tridiagonal and Q_j is an orthogonal matrix represented as the product of Householder matrices

$$\begin{array}{cl} Q_j = H_{j_1} H_{j_2} \cdots H_{j_{n-1}} & \text{if uplo indicates lower, or} \\ Q_j = H_{j_{n-1}} H_{j_{n-2}} \cdots H_{j_1} & \text{if uplo indicates upper.} \end{array}$$

Each Householder matrix H_{j_i} is given by

$$H_{j_i} = I - \text{tau}_j[i] \cdot v_{j_i} v_{j_i}'$$

where $\text{tau}_j[i]$ is the corresponding Householder scalar. When uplo indicates lower, the first i elements of the Householder vector v_{j_i} are zero, and $v_{j_i}[i+1] = 1$. If uplo indicates upper, the last n-i elements of the Householder vector v_{j_i} are zero, and $v_{j_i}[i] = 1$.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower part of the symmetric matrix A_j is stored. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.
- `[in] n`: rocblas_int. n >= 0. The number of rows and columns of the matrices A_j.
- `[inout] A`: pointer to type. Array on the GPU (the size depends on the value of strideA). On entry, the matrices A_j to be factored. On exit, if upper, then the elements on the diagonal and superdiagonal contain the tridiagonal form T_j; the elements above the superdiagonal contain the first i-1 elements of the Householder vectors v_(j_i) stored as columns. If lower, then the elements on the diagonal and subdiagonal contain the tridiagonal form T_j; the elements below the subdiagonal contain the last n-i-1 elements of the Householder vectors v_(j_i) stored as columns.

- `[in] lda`: rocblas_int. lda >= n. The leading dimension of A_j.
- `[in] strideA`: rocblas_stride. Stride from the start of one matrix A_j to the next one A_(j+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n.
- `[out] D`: pointer to type. Array on the GPU (the size depends on the value of strideD). The diagonal elements of T_j.
- `[in] strideD`: rocblas_stride. Stride from the start of one vector D_j to the next one D_(j+1). There is no restriction for the value of strideD. Normal use case is strideD >= n.
- `[out] E`: pointer to type. Array on the GPU (the size depends on the value of strideE). The off-diagonal elements of T_j.
- `[in] strideE`: rocblas_stride. Stride from the start of one vector E_j to the next one E_(j+1). There is no restriction for the value of strideE. Normal use case is strideE >= n-1.
- `[out] tau`: pointer to type. Array on the GPU (the size depends on the value of strideP). Contains the vectors tau_j of corresponding Householder scalars.
- `[in] strideP`: rocblas_stride. Stride from the start of one vector tau_j to the next one tau_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= n-1.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>hetd2()

rocblas_status **rocsolver_zhetd2** (rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, double **D*, double **E*, rocblas_double_complex **tau*)

rocblas_status **rocsolver_chetd2** (rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, float **D*, float **E*, rocblas_float_complex **tau*)

HETD2 computes the tridiagonal form of a complex hermitian matrix A.

(This is the unblocked version of the algorithm).

The tridiagonal form is given by:

$$T = Q'AQ$$

where T is hermitian tridiagonal and Q is an unitary matrix represented as the product of Householder matrices

$$\begin{array}{cl} Q = H_1 H_2 \cdots H_{n-1} & \text{if uplo indicates lower, or} \\ Q = H_{n-1} H_{n-2} \cdots H_1 & \text{if uplo indicates upper.} \end{array}$$

Each Householder matrix H_i is given by

$$H_i = I - \text{tau}[i] \cdot v_i v_i'$$

where tau[i] is the corresponding Householder scalar. When uplo indicates lower, the first i elements of the Householder vector v_i are zero, and $v_i[i+1] = 1$. If uplo indicates upper, the last n-i elements of the Householder vector v_i are zero, and $v_i[i] = 1$.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower part of the hermitian matrix A is stored. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.
- `[in] n`: rocblas_int. n >= 0. The number of rows and columns of the matrix A.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the matrix to be factored. On exit, if upper, then the elements on the diagonal and superdiagonal contain the tridiagonal form T; the elements above the superdiagonal contain the first i-1 elements of the Householders vector v_i stored as columns. If lower, then the elements on the diagonal and subdiagonal contain the tridiagonal form T; the elements below the subdiagonal contain the last n-i-1 elements of the Householder vectors v_i stored as columns.
- `[in] lda`: rocblas_int. lda >= n. The leading dimension of A.
- `[out] D`: pointer to real type. Array on the GPU of dimension n. The diagonal elements of T.
- `[out] E`: pointer to real type. Array on the GPU of dimension n-1. The off-diagonal elements of T.
- `[out] tau`: pointer to type. Array on the GPU of dimension n-1. The Householder scalars.

rocsolver_<type>hetd2_batched()

rocblas_status **rocsolver_zhetd2_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_double_complex ***const** *A*[], **const** rocblas_int *lda*, double **D*, **const** rocblas_stride *strideD*, double **E*, **const** rocblas_stride *strideE*, rocblas_double_complex **tau*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_chetd2_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_float_complex ***const** *A*[], **const** rocblas_int *lda*, float **D*, **const** rocblas_stride *strideD*, float **E*, **const** rocblas_stride *strideE*, rocblas_float_complex **tau*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

HETD2_BATCHED computes the tridiagonal form of a batch of complex hermitian matrices A_j.

(This is the unblocked version of the algorithm).

The tridiagonal form of A_j is given by:

$$T_j = Q_j' A_j Q_j$$

where T_j is Hermitian tridiagonal and Q_j is a unitary matrix represented as the product of Householder matrices

$$\begin{array}{cl} Q_j = H_{j_1} H_{j_2} \cdots H_{j_{n-1}} & \text{if uplo indicates lower, or} \\ Q_j = H_{j_{n-1}} H_{j_{n-2}} \cdots H_{j_1} & \text{if uplo indicates upper.} \end{array}$$

Each Householder matrix H_{j_i} is given by

$$H_{j_i} = I - \text{tau}_j[i] \cdot v_{j_i} v_{j_i}'$$

where $\text{tau}_j[i]$ is the corresponding Householder scalar. When uplo indicates lower, the first i elements of the Householder vector v_{j_i} are zero, and $v_{j_i}[i+1] = 1$. If uplo indicates upper, the last n-i elements of the Householder vector v_{j_i} are zero, and $v_{j_i}[i] = 1$.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower part of the hermitian matrix A_j is stored. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.
- `[in] n`: rocblas_int. n >= 0. The number of rows and columns of the matrices A_j.
- `[inout] A`: array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n. On entry, the matrices A_j to be factored. On exit, if upper, then the elements on the diagonal and superdiagonal contain the tridiagonal form T_j; the elements above the superdiagonal contain the first i-1 elements of the Householder vectors v_(j_i) stored as columns. If lower, then the elements on the diagonal and subdiagonal contain the tridiagonal form T_j; the elements below the subdiagonal contain the last n-i-1 elements of the Householder vectors v_(j_i) stored as columns.
- `[in] lda`: rocblas_int. lda >= n. The leading dimension of A_j.
- `[out] D`: pointer to real type. Array on the GPU (the size depends on the value of strideD). The diagonal elements of T_j.
- `[in] strideD`: rocblas_stride. Stride from the start of one vector D_j to the next one D_(j+1). There is no restriction for the value of strideD. Normal use case is strideD >= n.
- `[out] E`: pointer to real type. Array on the GPU (the size depends on the value of strideE). The off-diagonal elements of T_j.
- `[in] strideE`: rocblas_stride. Stride from the start of one vector E_j to the next one E_(j+1). There is no restriction for the value of strideE. Normal use case is strideE >= n-1.
- `[out] tau`: pointer to type. Array on the GPU (the size depends on the value of strideP). Contains the vectors tau_j of corresponding Householder scalars.
- `[in] strideP`: rocblas_stride. Stride from the start of one vector tau_j to the next one tau_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= n-1.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>hetd2_strided_batched()

rocblas_status **rocsolver_zhetd2_strided_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, double **D*, **const** rocblas_stride *strideD*, double **E*, **const** rocblas_stride *strideE*, rocblas_double_complex **tau*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*) rocblas_status **rocsolver_chetd2_strided_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, float **D*, **const** rocblas_stride *strideD*, float **E*, **const** rocblas_stride *strideE*, rocblas_float_complex **tau*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

HETD2_STRIDED_BATCHED computes the tridiagonal form of a batch of complex hermitian matrices A_j.

(This is the unblocked version of the algorithm).

The tridiagonal form of A_j is given by:

$$T_j = Q_j' A_j Q_j$$

where T_j is Hermitian tridiagonal and Q_j is a unitary matrix represented as the product of Householder matrices

$$\begin{array}{cl} Q_j = H_{j_1} H_{j_2} \cdots H_{j_{n-1}} & \text{if uplo indicates lower, or} \\ Q_j = H_{j_{n-1}} H_{j_{n-2}} \cdots H_{j_1} & \text{if uplo indicates upper.} \end{array}$$

Each Householder matrix H_{j_i} is given by

$$H_{j_i} = I - \text{tau}_j[i] \cdot v_{j_i} v_{j_i}'$$

where $\text{tau}_j[i]$ is the corresponding Householder scalar. When uplo indicates lower, the first i elements of the Householder vector v_{j_i} are zero, and $v_{j_i}[i+1] = 1$. If uplo indicates upper, the last n-i elements of the Householder vector v_{j_i} are zero, and $v_{j_i}[i] = 1$.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower part of the hermitian matrix A_j is stored. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.
- `[in] n`: rocblas_int. n >= 0. The number of rows and columns of the matrices A_j.
- `[inout] A`: pointer to type. Array on the GPU (the size depends on the value of strideA). On entry, the matrices A_j to be factored. On exit, if upper, then the elements on the diagonal and superdiagonal contain the tridiagonal form T_j; the elements above the superdiagonal contain the first i-1 elements of the Householder vectors v_(j_i) stored as columns. If lower, then the elements on the diagonal and subdiagonal contain the tridiagonal form T_j; the elements below the subdiagonal contain the last n-i-1 elements of the Householder vectors v_(j_i) stored as columns.
- `[in] lda`: rocblas_int. lda >= n. The leading dimension of A_j.
- `[in] strideA`: rocblas_stride. Stride from the start of one matrix A_j to the next one A_(j+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n.
- `[out] D`: pointer to real type. Array on the GPU (the size depends on the value of strideD). The diagonal elements of T_j.

- [in] strideD: rocblas_stride. Stride from the start of one vector D_j to the next one D_(j+1). There is no restriction for the value of strideD. Normal use case is strideD >= n.
- [out] E: pointer to real type. Array on the GPU (the size depends on the value of strideE). The off-diagonal elements of T_j.
- [in] strideE: rocblas_stride. Stride from the start of one vector E_j to the next one E_(j+1). There is no restriction for the value of strideE. Normal use case is strideE >= n-1.
- [out] tau: pointer to type. Array on the GPU (the size depends on the value of strideP). Contains the vectors tau_j of corresponding Householder scalars.
- [in] strideP: rocblas_stride. Stride from the start of one vector tau_j to the next one tau_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= n-1.
- [in] batch_count: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>sytrd()

rocblas_status **rocsolver_dsytrd**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, double **D*, double **E*, double **tau*)

rocblas_status **rocsolver_ssytrd**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, float **D*, float **E*, float **tau*)

SYTRD computes the tridiagonal form of a real symmetric matrix A.

(This is the blocked version of the algorithm).

The tridiagonal form is given by:

$$T = Q'AQ$$

where T is symmetric tridiagonal and Q is an orthogonal matrix represented as the product of Householder matrices

$$\begin{array}{cl} Q = H_1H_2\cdots H_{n-1} & \text{if uplo indicates lower, or} \\ Q = H_{n-1}H_{n-2}\cdots H_1 & \text{if uplo indicates upper.} \end{array}$$

Each Householder matrix H_i is given by

$$H_i = I - \text{tau}[i] \cdot v_i v_i'$$

where tau[i] is the corresponding Householder scalar. When uplo indicates lower, the first i elements of the Householder vector v_i are zero, and $v_i[i+1] = 1$. If uplo indicates upper, the last n-i elements of the Householder vector v_i are zero, and $v_i[i] = 1$.

Parameters

- [in] handle: rocblas_handle.
- [in] uplo: rocblas_fill. Specifies whether the upper or lower part of the symmetric matrix A is stored. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.

- `[in] n`: rocblas_int. n >= 0. The number of rows and columns of the matrix A.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the matrix to be factored. On exit, if upper, then the elements on the diagonal and superdiagonal contain the tridiagonal form T; the elements above the superdiagonal contain the first i-1 elements of the Householder vectors v_i stored as columns. If lower, then the elements on the diagonal and subdiagonal contain the tridiagonal form T; the elements below the subdiagonal contain the last n-i-1 elements of the Householder vectors v_i stored as columns.
- `[in] lda`: rocblas_int. lda >= n. The leading dimension of A.
- `[out] D`: pointer to type. Array on the GPU of dimension n. The diagonal elements of T.
- `[out] E`: pointer to type. Array on the GPU of dimension n-1. The off-diagonal elements of T.
- `[out] tau`: pointer to type. Array on the GPU of dimension n-1. The Householder scalars.

rocsolver_<type>sytrd_batched()

rocblas_status **rocsolver_dsytrd_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, double ***const** *A*[], **const** rocblas_int *lda*, double **D*, **const** rocblas_stride *strideD*, double **E*, **const** rocblas_stride *strideE*, double **tau*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_ssytrd_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, float ***const** *A*[], **const** rocblas_int *lda*, float **D*, **const** rocblas_stride *strideD*, float **E*, **const** rocblas_stride *strideE*, float **tau*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

SYTRD_BATCHED computes the tridiagonal form of a batch of real symmetric matrices A_j.

(This is the blocked version of the algorithm).

The tridiagonal form of A_j is given by:

$$T_j = Q_j' A_j Q_j$$

where T_j is symmetric tridiagonal and Q_j is an orthogonal matrix represented as the product of Householder matrices

$$\begin{array}{cl} Q_j = H_{j_1} H_{j_2} \cdots H_{j_{n-1}} & \text{if uplo indicates lower, or} \\ Q_j = H_{j_{n-1}} H_{j_{n-2}} \cdots H_{j_1} & \text{if uplo indicates upper.} \end{array}$$

Each Householder matrix H_{j_i} is given by

$$H_{j_i} = I - \text{tau}_j[i] \cdot v_{j_i} v_{j_i}'$$

where $\text{tau}_j[i]$ is the corresponding Householder scalar. When uplo indicates lower, the first i elements of the Householder vector v_{j_i} are zero, and $v_{j_i}[i+1] = 1$. If uplo indicates upper, the last n-i elements of the Householder vector v_{j_i} are zero, and $v_{j_i}[i] = 1$.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower part of the symmetric matrix A_j is stored. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.
- `[in] n`: rocblas_int. n >= 0. The number of rows and columns of the matrices A_j.
- `[inout] A`: array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n. On entry, the matrices A_j to be factored. On exit, if upper, then the elements on the diagonal and superdiagonal contain the tridiagonal form T_j; the elements above the superdiagonal contain the first i-1 elements of the Householder vectors v_(j_i) stored as columns. If lower, then the elements on the diagonal and subdiagonal contain the tridiagonal form T_j; the elements below the subdiagonal contain the last n-i-1 elements of the Householder vectors v_(j_i) stored as columns.
- `[in] lda`: rocblas_int. lda >= n. The leading dimension of A_j.
- `[out] D`: pointer to type. Array on the GPU (the size depends on the value of strideD). The diagonal elements of T_j.
- `[in] strideD`: rocblas_stride. Stride from the start of one vector D_j to the next one D_(j+1). There is no restriction for the value of strideD. Normal use case is strideD >= n.
- `[out] E`: pointer to type. Array on the GPU (the size depends on the value of strideE). The off-diagonal elements of T_j.
- `[in] strideE`: rocblas_stride. Stride from the start of one vector E_j to the next one E_(j+1). There is no restriction for the value of strideE. Normal use case is strideE >= n-1.
- `[out] tau`: pointer to type. Array on the GPU (the size depends on the value of strideP). Contains the vectors tau_j of corresponding Householder scalars.
- `[in] strideP`: rocblas_stride. Stride from the start of one vector tau_j to the next one tau_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= n-1.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>sytrd_strided_batched()

rocblas_status **rocsolver_dsytrd_strided_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, double **D*, **const** rocblas_stride *strideD*, double **E*, **const** rocblas_stride *strideE*, double **tau*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_ssytrd_strided_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, float **D*, **const** rocblas_stride *strideD*, float **E*, **const** rocblas_stride *strideE*, float **tau*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

SYTRD_STRIDED_BATCHED computes the tridiagonal form of a batch of real symmetric matrices A_j.

(This is the blocked version of the algorithm).

The tridiagonal form of A_j is given by:

$$T_j = Q_j' A_j Q_j$$

where T_j is symmetric tridiagonal and Q_j is an orthogonal matrix represented as the product of Householder matrices

$$\begin{array}{cl} Q_j = H_{j_1} H_{j_2} \cdots H_{j_{n-1}} & \text{if uplo indicates lower, or} \\ Q_j = H_{j_{n-1}} H_{j_{n-2}} \cdots H_{j_1} & \text{if uplo indicates upper.} \end{array}$$

Each Householder matrix H_{j_i} is given by

$$H_{j_i} = I - \text{tau}_j[i] \cdot v_{j_i} v_{j_i}'$$

where $\text{tau}_j[i]$ is the corresponding Householder scalar. When uplo indicates lower, the first i elements of the Householder vector v_{j_i} are zero, and $v_{j_i}[i+1] = 1$. If uplo indicates upper, the last n-i elements of the Householder vector v_{j_i} are zero, and $v_{j_i}[i] = 1$.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower part of the symmetric matrix A_j is stored. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.
- `[in] n`: rocblas_int. n >= 0. The number of rows and columns of the matrices A_j.
- `[inout] A`: pointer to type. Array on the GPU (the size depends on the value of strideA). On entry, the matrices A_j to be factored. On exit, if upper, then the elements on the diagonal and superdiagonal contain the tridiagonal form T_j; the elements above the superdiagonal contain the first i-1 elements of the Householder vectors v_(j_i) stored as columns. If lower, then the elements on the diagonal and subdiagonal contain the tridiagonal form T_j; the elements below the subdiagonal contain the last n-i-1 elements of the Householder vectors v_(j_i) stored as columns.
- `[in] lda`: rocblas_int. lda >= n. The leading dimension of A_j.
- `[in] strideA`: rocblas_stride. Stride from the start of one matrix A_j to the next one A_(j+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n.
- `[out] D`: pointer to type. Array on the GPU (the size depends on the value of strideD). The diagonal elements of T_j.
- `[in] strideD`: rocblas_stride. Stride from the start of one vector D_j to the next one D_(j+1). There is no restriction for the value of strideD. Normal use case is strideD >= n.
- `[out] E`: pointer to type. Array on the GPU (the size depends on the value of strideE). The off-diagonal elements of T_j.
- `[in] strideE`: rocblas_stride. Stride from the start of one vector E_j to the next one E_(j+1). There is no restriction for the value of strideE. Normal use case is strideE >= n-1.
- `[out] tau`: pointer to type. Array on the GPU (the size depends on the value of strideP). Contains the vectors tau_j of corresponding Householder scalars.
- `[in] strideP`: rocblas_stride. Stride from the start of one vector tau_j to the next one tau_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= n-1.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>hetrd()

rocblas_status **rocsolver_zhetrd**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, double **D*, double **E*, rocblas_double_complex **tau*)

rocblas_status **rocsolver_chetrd**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, float **D*, float **E*, rocblas_float_complex **tau*)

HETRD computes the tridiagonal form of a complex hermitian matrix A.

(This is the blocked version of the algorithm).

The tridiagonal form is given by:

$$T = Q'AQ$$

where T is hermitian tridiagonal and Q is an unitary matrix represented as the product of Householder matrices

$$\begin{array}{cl} Q = H_1 H_2 \cdots H_{n-1} & \text{if uplo indicates lower, or} \\ Q = H_{n-1} H_{n-2} \cdots H_1 & \text{if uplo indicates upper.} \end{array}$$

Each Householder matrix H_i is given by

$$H_i = I - \text{tau}[i] \cdot v_i v_i'$$

where tau[i] is the corresponding Householder scalar. When uplo indicates lower, the first i elements of the Householder vector v_i are zero, and $v_i[i+1] = 1$. If uplo indicates upper, the last n-i elements of the Householder vector v_i are zero, and $v_i[i] = 1$.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower part of the hermitian matrix A is stored. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.
- `[in] n`: rocblas_int. n >= 0. The number of rows and columns of the matrix A.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the matrix to be factored. On exit, if upper, then the elements on the diagonal and superdiagonal contain the tridiagonal form T; the elements above the superdiagonal contain the first i-1 elements of the Householder vectors v_i stored as columns. If lower, then the elements on the diagonal and subdiagonal contain the tridiagonal form T; the elements below the subdiagonal contain the last n-i-1 elements of the Householder vectors v_i stored as columns.
- `[in] lda`: rocblas_int. lda >= n. The leading dimension of A.
- `[out] D`: pointer to real type. Array on the GPU of dimension n. The diagonal elements of T.
- `[out] E`: pointer to real type. Array on the GPU of dimension n-1. The off-diagonal elements of T.
- `[out] tau`: pointer to type. Array on the GPU of dimension n-1. The Householder scalars.

rocsolver_<type>hetrd_batched()

rocblas_status **rocsolver_zhetrd_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_double_complex ***const** *A*[], **const** rocblas_int *lda*, double **D*, **const** rocblas_stride *strideD*, double **E*, **const** rocblas_stride *strideE*, rocblas_double_complex **tau*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_chetrd_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_float_complex ***const** *A*[], **const** rocblas_int *lda*, float **D*, **const** rocblas_stride *strideD*, float **E*, **const** rocblas_stride *strideE*, rocblas_float_complex **tau*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

HETRD_BATCHED computes the tridiagonal form of a batch of complex hermitian matrices A_j.

(This is the blocked version of the algorithm).

The tridiagonal form of A_j is given by:

$$T_j = Q_j' A_j Q_j$$

where T_j is Hermitian tridiagonal and Q_j is a unitary matrix represented as the product of Householder matrices

$$\begin{array}{cl} Q_j = H_{j_1} H_{j_2} \cdots H_{j_{n-1}} & \text{if uplo indicates lower, or} \\ Q_j = H_{j_{n-1}} H_{j_{n-2}} \cdots H_{j_1} & \text{if uplo indicates upper.} \end{array}$$

Each Householder matrix H_{j_i} is given by

$$H_{j_i} = I - \text{tau}_j[i] \cdot v_{j_i} v_{j_i}'$$

where $\text{tau}_j[i]$ is the corresponding Householder scalar. When uplo indicates lower, the first i elements of the Householder vector v_{j_i} are zero, and $v_{j_i}[i+1] = 1$. If uplo indicates upper, the last n-i elements of the Householder vector v_{j_i} are zero, and $v_{j_i}[i] = 1$.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower part of the hermitian matrix A_j is stored. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.
- `[in] n`: rocblas_int. n >= 0. The number of rows and columns of the matrices A_j.
- `[inout] A`: array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n. On entry, the matrices A_j to be factored. On exit, if upper, then the elements on the diagonal and superdiagonal contain the tridiagonal form T_j; the elements above the superdiagonal contain the first i-1 elements of the Householder vectors v_(j_i) stored as columns. If lower, then the elements on the diagonal and subdiagonal contain the tridiagonal form T_j; the elements below the subdiagonal contain the last n-i-1 elements of the Householder vectors v_(j_i) stored as columns.
- `[in] lda`: rocblas_int. lda >= n. The leading dimension of A_j.

- `[out] D`: pointer to real type. Array on the GPU (the size depends on the value of strideD). The diagonal elements of T_j.
- `[in] strideD`: rocblas_stride. Stride from the start of one vector D_j to the next one D_(j+1). There is no restriction for the value of strideD. Normal use case is strideD >= n.
- `[out] E`: pointer to real type. Array on the GPU (the size depends on the value of strideE). The off-diagonal elements of T_j.
- `[in] strideE`: rocblas_stride. Stride from the start of one vector E_j to the next one E_(j+1). There is no restriction for the value of strideE. Normal use case is strideE >= n-1.
- `[out] tau`: pointer to type. Array on the GPU (the size depends on the value of strideP). Contains the vectors tau_j of corresponding Householder scalars.
- `[in] strideP`: rocblas_stride. Stride from the start of one vector tau_j to the next one tau_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= n-1.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>hetrd_strided_batched()

rocblas_status **rocsolver_zhetrd_strided_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, double **D*, **const** rocblas_stride *strideD*, double **E*, **const** rocblas_stride *strideE*, rocblas_double_complex **tau*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_chetrd_strided_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, float **D*, **const** rocblas_stride *strideD*, float **E*, **const** rocblas_stride *strideE*, rocblas_float_complex **tau*, **const** rocblas_stride *strideP*, **const** rocblas_int *batch_count*)

HETRD_STRIDED_BATCHED computes the tridiagonal form of a batch of complex hermitian matrices A_j.

(This is the blocked version of the algorithm).

The tridiagonal form of A_j is given by:

$$T_j = Q_j' A_j Q_j$$

where T_j is Hermitian tridiagonal and Q_j is a unitary matrix represented as the product of Householder matrices

$$\begin{array}{cl} Q_j = H_{j_1} H_{j_2} \cdots H_{j_{n-1}} & \text{if uplo indicates lower, or} \\ Q_j = H_{j_{n-1}} H_{j_{n-2}} \cdots H_{j_1} & \text{if uplo indicates upper.} \end{array}$$

Each Householder matrix H_{j_i} is given by

$$H_{j_i} = I - \text{tau}_j[i] \cdot v_{j_i} v_{j_i}'$$

where $\text{tau}_j[i]$ is the corresponding Householder scalar. When uplo indicates lower, the first i elements of the Householder vector v_{j_i} are zero, and $v_{j_i}[i+1] = 1$. If uplo indicates upper, the last n-i elements of the Householder vector v_{j_i} are zero, and $v_{j_i}[i] = 1$.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower part of the hermitian matrix A_j is stored. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.
- `[in] n`: rocblas_int. n >= 0. The number of rows and columns of the matrices A_j.
- `[inout] A`: pointer to type. Array on the GPU (the size depends on the value of strideA). On entry, the matrices A_j to be factored. On exit, if upper, then the elements on the diagonal and superdiagonal contain the tridiagonal form T_j; the elements above the superdiagonal contain the first i-1 elements of the Householder vectors v_(j_i) stored as columns. If lower, then the elements on the diagonal and subdiagonal contain the tridiagonal form T_j; the elements below the subdiagonal contain the last n-i-1 elements of the Householder vectors v_(j_i) stored as columns.
- `[in] lda`: rocblas_int. lda >= n. The leading dimension of A_j.
- `[in] strideA`: rocblas_stride. Stride from the start of one matrix A_j to the next one A_(j+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n.
- `[out] D`: pointer to real type. Array on the GPU (the size depends on the value of strideD). The diagonal elements of T_j.
- `[in] strideD`: rocblas_stride. Stride from the start of one vector D_j to the next one D_(j+1). There is no restriction for the value of strideD. Normal use case is strideD >= n.
- `[out] E`: pointer to real type. Array on the GPU (the size depends on the value of strideE). The off-diagonal elements of T_j.
- `[in] strideE`: rocblas_stride. Stride from the start of one vector E_j to the next one E_(j+1). There is no restriction for the value of strideE. Normal use case is strideE >= n-1.
- `[out] tau`: pointer to type. Array on the GPU (the size depends on the value of strideP). Contains the vectors tau_j of corresponding Householder scalars.
- `[in] strideP`: rocblas_stride. Stride from the start of one vector tau_j to the next one tau_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= n-1.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>sygs2()

rocblas_status **rocsolver_dsygs2**(rocblas_handle *handle*, **const** *rocblas_eform* *itype*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, double **B*, **const** rocblas_int *ldb*)

rocblas_status **rocsolver_ssygs2**(rocblas_handle *handle*, **const** *rocblas_eform* *itype*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, float **B*, **const** rocblas_int *ldb*)

SYGS2 reduces a real symmetric-definite generalized eigenproblem to standard form.

(This is the unblocked version of the algorithm).

The problem solved by this function is either of the form

$$\begin{array}{ll} AX = \lambda BX & \text{1st form,} \\ ABX = \lambda X & \text{2nd form, or} \\ BAX = \lambda X & \text{3rd form,} \end{array}$$

depending on the value of itype.

If the problem is of the 1st form, then A is overwritten with

$$\begin{array}{ll} U^{-T}AU^{-1}, & \text{or} \\ L^{-1}AL^{-T}, & \end{array}$$

where the symmetric-definite matrix B has been factorized as either U^TU or LL^T as returned by *POTRF*, depending on the value of uplo.

If the problem is of the 2nd or 3rd form, then A is overwritten with

$$\begin{array}{ll} UAU^T, & \text{or} \\ L^TAL, & \end{array}$$

also depending on the value of uplo.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] itype`: *rocblas_eform*. Specifies the form of the generalized eigenproblem.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower part of the matrix A is stored, and whether the factorization applied to B was upper or lower triangular. If uplo indicates lower (or upper), then the upper (or lower) parts of A and B are not used.
- `[in] n`: rocblas_int. n >= 0. The matrix dimensions.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the matrix A. On exit, the transformed matrix associated with the equivalent standard eigenvalue problem.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of A.
- `[out] B`: pointer to type. Array on the GPU of dimension ldb*n. The triangular factor of the matrix B, as returned by *POTRF*.
- `[in] ldb`: rocblas_int. ldb >= n. Specifies the leading dimension of B.

rocsolver_<type>sygs2_batched()

rocblas_status **rocsolver_dsygs2_batched**(rocblas_handle *handle*, **const** *rocblas_eform* *itype*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, double ***const** *A*[], **const** rocblas_int *lda*, double ***const** *B*[], **const** rocblas_int *ldb*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_ssygs2_batched**(rocblas_handle *handle*, **const** *rocblas_eform* *itype*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, float ***const** *A*[], **const** rocblas_int *lda*, float ***const** *B*[], **const** rocblas_int *ldb*, **const** rocblas_int *batch_count*)

SYGS2_BATCHED reduces a batch of real symmetric-definite generalized eigenproblems to standard form.

(This is the unblocked version of the algorithm).

For each instance in the batch, the problem solved by this function is either of the form

$$\begin{array}{cl} A_iX_i = \lambda B_iX_i & \text{1st form,} \\ A_iB_iX_i = \lambda X_i & \text{2nd form, or} \\ B_iA_iX_i = \lambda X_i & \text{3rd form,} \end{array}$$

depending on the value of itype.

If the problem is of the 1st form, then A_i is overwritten with

$$\begin{array}{cl} U_i^{-T}A_iU_i^{-1}, & \text{or} \\ L_i^{-1}A_iL_i^{-T}, & \end{array}$$

where the symmetric-definite matrix B_i has been factorized as either $U_i^TU_i$ or $L_iL_i^T$ as returned by *POTRF*, depending on the value of uplo.

If the problem is of the 2nd or 3rd form, then A is overwritten with

$$\begin{array}{cl} U_iA_iU_i^T, & \text{or} \\ L_i^TA_iL_i, & \end{array}$$

also depending on the value of uplo.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] itype`: *rocblas_eform*. Specifies the form of the generalized eigenproblems.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower part of the matrices A_i are stored, and whether the factorization applied to B_i was upper or lower triangular. If uplo indicates lower (or upper), then the upper (or lower) parts of A_i and B_i are not used.
- `[in] n`: rocblas_int. n >= 0. The matrix dimensions.
- `[inout] A`: array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n. On entry, the matrices A_i. On exit, the transformed matrices associated with the equivalent standard eigenvalue problems.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of A_i.

- `[out] B`: array of pointers to type. Each pointer points to an array on the GPU of dimension ldb*n. The triangular factors of the matrices B_i, as returned by *POTRF_BATCHED*.
- `[in] ldb`: rocblas_int. ldb >= n. Specifies the leading dimension of B_i.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>sygs2_strided_batched()

rocblas_status **rocsolver_dsygs2_strided_batched**(rocblas_handle *handle*, **const** *rocblas_eform* *itype*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, double **B*, **const** rocblas_int *ldb*, **const** rocblas_stride *strideB*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_ssygs2_strided_batched**(rocblas_handle *handle*, **const** *rocblas_eform* *itype*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, float **B*, **const** rocblas_int *ldb*, **const** rocblas_stride *strideB*, **const** rocblas_int *batch_count*)

SYGS2_STRIDED_BATCHED reduces a batch of real symmetric-definite generalized eigenproblems to standard form.

(This is the unblocked version of the algorithm).

For each instance in the batch, the problem solved by this function is either of the form

$$\begin{array}{ll} A_i X_i = \lambda B_i X_i & \text{1st form,} \\ A_i B_i X_i = \lambda X_i & \text{2nd form, or} \\ B_i A_i X_i = \lambda X_i & \text{3rd form,} \end{array}$$

depending on the value of itype.

If the problem is of the 1st form, then A_i is overwritten with

$$\begin{array}{ll} U_i^{-T} A_i U_i^{-1}, & \text{or} \\ L_i^{-1} A_i L_i^{-T}, & \end{array}$$

where the symmetric-definite matrix B_i has been factorized as either $U_i^T U_i$ or $L_i L_i^T$ as returned by *POTRF*, depending on the value of uplo.

If the problem is of the 2nd or 3rd form, then A is overwritten with

$$\begin{array}{ll} U_i A_i U_i^T, & \text{or} \\ L_i^T A_i L_i, & \end{array}$$

also depending on the value of uplo.

Parameters

- `[in] handle`: rocblas_handle.

- [in] itype: *rocblas_eform*. Specifies the form of the generalized eigenproblems.
- [in] uplo: rocblas_fill. Specifies whether the upper or lower part of the matrices A_i are stored, and whether the factorization applied to B_i was upper or lower triangular. If uplo indicates lower (or upper), then the upper (or lower) parts of A_i and B_i are not used.
- [in] n: rocblas_int. n >= 0. The matrix dimensions.
- [inout] A: pointer to type. Array on the GPU (the size depends on the value of strideA). On entry, the matrices A_i. On exit, the transformed matrices associated with the equivalent standard eigenvalue problems.
- [in] lda: rocblas_int. lda >= n. Specifies the leading dimension of A_i.
- [in] strideA: rocblas_stride. Stride from the start of one matrix A_i to the next one A_(i+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n.
- [out] B: pointer to type. Array on the GPU (the size depends on the value of strideB). The triangular factors of the matrices B_i, as returned by *POTRF_STRIDED_BATCHED*.
- [in] ldb: rocblas_int. ldb >= n. Specifies the leading dimension of B_i.
- [in] strideB: rocblas_stride. Stride from the start of one matrix B_i to the next one B_(i+1). There is no restriction for the value of strideB. Normal use case is strideB >= ldb*n.
- [in] batch_count: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>hegs2()

rocblas_status **rocsolver_zhegs2**(rocblas_handle *handle*, **const** *rocblas_eform* *itype*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, rocblas_double_complex **B*, **const** rocblas_int *ldb*)

rocblas_status **rocsolver_chegs2**(rocblas_handle *handle*, **const** *rocblas_eform* *itype*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, rocblas_float_complex **B*, **const** rocblas_int *ldb*)

HEGS2 reduces a hermitian-definite generalized eigenproblem to standard form.

(This is the unblocked version of the algorithm).

The problem solved by this function is either of the form

$$\begin{array}{ll} AX = \lambda BX & \text{1st form,} \\ ABX = \lambda X & \text{2nd form, or} \\ BAX = \lambda X & \text{3rd form,} \end{array}$$

depending on the value of itype.

If the problem is of the 1st form, then A is overwritten with

$$\begin{array}{ll} U^{-H}AU^{-1}, & \text{or} \\ L^{-1}AL^{-H}, & \end{array}$$

where the hermitian-definite matrix B has been factorized as either U^HU or LL^H as returned by *POTRF*, depending on the value of uplo.

If the problem is of the 2nd or 3rd form, then A is overwritten with

$$\begin{array}{ll} UAU^H, & \text{or} \\ L^H AL, & \end{array}$$

also depending on the value of uplo.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] itype`: *rocblas_eform*. Specifies the form of the generalized eigenproblem.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower part of the matrix A is stored, and whether the factorization applied to B was upper or lower triangular. If uplo indicates lower (or upper), then the upper (or lower) parts of A and B are not used.
- `[in] n`: rocblas_int. n >= 0. The matrix dimensions.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the matrix A. On exit, the transformed matrix associated with the equivalent standard eigenvalue problem.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of A.
- `[out] B`: pointer to type. Array on the GPU of dimension ldb*n. The triangular factor of the matrix B, as returned by *POTRF*.
- `[in] ldb`: rocblas_int. ldb >= n. Specifies the leading dimension of B.

rocsolver_<type>hegs2_batched()

rocblas_status **rocsolver_zhegs2_batched**(rocblas_handle *handle*, **const** *rocblas_eform* *itype*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_double_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_double_complex ***const** *B*[], **const** rocblas_int *ldb*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_chegs2_batched**(rocblas_handle *handle*, **const** *rocblas_eform* *itype*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_float_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_float_complex ***const** *B*[], **const** rocblas_int *ldb*, **const** rocblas_int *batch_count*)

HEGS2_BATCHED reduces a batch of hermitian-definite generalized eigenproblems to standard form.

(This is the unblocked version of the algorithm).

For each instance in the batch, the problem solved by this function is either of the form

$$\begin{array}{ll} A_i X_i = \lambda B_i X_i & \text{1st form,} \\ A_i B_i X_i = \lambda X_i & \text{2nd form, or} \\ B_i A_i X_i = \lambda X_i & \text{3rd form,} \end{array}$$

depending on the value of itype.

If the problem is of the 1st form, then A_i is overwritten with

$$U_i^{-H} A_i U_i^{-1}, \quad \text{or}$$
$$L_i^{-1} A_i L_i^{-H},$$

where the hermitian-definite matrix B_i has been factorized as either $U_i^H U_i$ or $L_i L_i^H$ as returned by *POTRF*, depending on the value of uplo.

If the problem is of the 2nd or 3rd form, then A is overwritten with

$$U_i A_i U_i^H, \quad \text{or}$$
$$L_i^H A_i L_i,$$

also depending on the value of uplo.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] itype`: *rocblas_eform*. Specifies the form of the generalized eigenproblems.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower part of the matrices A_i are stored, and whether the factorization applied to B_i was upper or lower triangular. If uplo indicates lower (or upper), then the upper (or lower) parts of A_i and B_i are not used.
- `[in] n`: rocblas_int. n >= 0. The matrix dimensions.
- `[inout] A`: array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n. On entry, the matrices A_i. On exit, the transformed matrices associated with the equivalent standard eigenvalue problems.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of A_i.
- `[out] B`: array of pointers to type. Each pointer points to an array on the GPU of dimension ldb*n. The triangular factors of the matrices B_i, as returned by *POTRF_BATCHED*.
- `[in] ldb`: rocblas_int. ldb >= n. Specifies the leading dimension of B_i.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>hegs2_strided_batched()

rocblas_status **rocsolver_zhegs2_strided_batched**(rocblas_handle *handle*, **const** *rocblas_eform* *itype*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_double_complex **B*, **const** rocblas_int *ldb*, **const** rocblas_stride *strideB*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_chegs2_strided_batched**(rocblas_handle *handle*, **const** *rocblas_eform* *itype*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_float_complex **B*, **const** rocblas_int *ldb*, **const** rocblas_stride *strideB*, **const** rocblas_int *batch_count*)

HEGS2_STRIDED_BATCHED reduces a batch of hermitian-definite generalized eigenproblems to standard form.

(This is the unblocked version of the algorithm).

For each instance in the batch, the problem solved by this function is either of the form

$$\begin{array}{cl} A_i X_i = \lambda B_i X_i & \text{1st form,} \\ A_i B_i X_i = \lambda X_i & \text{2nd form, or} \\ B_i A_i X_i = \lambda X_i & \text{3rd form,} \end{array}$$

depending on the value of itype.

If the problem is of the 1st form, then A_i is overwritten with

$$\begin{array}{cl} U_i^{-H} A_i U_i^{-1}, & \text{or} \\ L_i^{-1} A_i L_i^{-H}, & \end{array}$$

where the hermitian-definite matrix B_i has been factorized as either $U_i^H U_i$ or $L_i L_i^H$ as returned by *POTRF*, depending on the value of uplo.

If the problem is of the 2nd or 3rd form, then A is overwritten with

$$\begin{array}{cl} U_i A_i U_i^H, & \text{or} \\ L_i^H A_i L_i, & \end{array}$$

also depending on the value of uplo.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] itype`: *rocblas_eform*. Specifies the form of the generalized eigenproblems.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower part of the matrices A_i are stored, and whether the factorization applied to B_i was upper or lower triangular. If uplo indicates lower (or upper), then the upper (or lower) parts of A_i and B_i are not used.
- `[in] n`: rocblas_int. n >= 0. The matrix dimensions.
- `[inout] A`: pointer to type. Array on the GPU (the size depends on the value of strideA). On entry, the matrices A_i. On exit, the transformed matrices associated with the equivalent standard eigenvalue problems.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of A_i.
- `[in] strideA`: rocblas_stride. Stride from the start of one matrix A_i to the next one A_(i+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n.
- `[out] B`: pointer to type. Array on the GPU (the size depends on the value of strideB). The triangular factors of the matrices B_i, as returned by *POTRF_STRIDED_BATCHED*.
- `[in] ldb`: rocblas_int. ldb >= n. Specifies the leading dimension of B_i.
- `[in] strideB`: rocblas_stride. Stride from the start of one matrix B_i to the next one B_(i+1). There is no restriction for the value of strideB. Normal use case is strideB >= ldb*n.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>sygst()

rocblas_status **rocsolver_dsygst**(rocblas_handle *handle*, **const** *rocblas_eform* *itype*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, double **B*, **const** rocblas_int *ldb*)

rocblas_status **rocsolver_ssygst**(rocblas_handle *handle*, **const** *rocblas_eform* *itype*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, float **B*, **const** rocblas_int *ldb*)

SYGST reduces a real symmetric-definite generalized eigenproblem to standard form.

(This is the blocked version of the algorithm).

The problem solved by this function is either of the form

$$\begin{array}{ll} AX = \lambda BX & \text{1st form,} \\ ABX = \lambda X & \text{2nd form, or} \\ BAX = \lambda X & \text{3rd form,} \end{array}$$

depending on the value of itype.

If the problem is of the 1st form, then A is overwritten with

$$\begin{array}{ll} U^{-T}AU^{-1}, & \text{or} \\ L^{-1}AL^{-T}, & \end{array}$$

where the symmetric-definite matrix B has been factorized as either U^TU or LL^T as returned by *POTRF*, depending on the value of uplo.

If the problem is of the 2nd or 3rd form, then A is overwritten with

$$\begin{array}{ll} UAU^T, & \text{or} \\ L^TAL, & \end{array}$$

also depending on the value of uplo.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] itype`: *rocblas_eform*. Specifies the form of the generalized eigenproblem.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower part of the matrix A is stored, and whether the factorization applied to B was upper or lower triangular. If uplo indicates lower (or upper), then the upper (or lower) parts of A and B are not used.
- `[in] n`: rocblas_int. n >= 0. The matrix dimensions.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the matrix A. On exit, the transformed matrix associated with the equivalent standard eigenvalue problem.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of A.
- `[out] B`: pointer to type. Array on the GPU of dimension ldb*n. The triangular factor of the matrix B, as returned by *POTRF*.
- `[in] ldb`: rocblas_int. ldb >= n. Specifies the leading dimension of B.

rocsolver_<type>sygst_batched()

rocblas_status **rocsolver_dsygst_batched**(rocblas_handle *handle*, **const** *rocblas_eform* *itype*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, double ***const** *A*[], **const** rocblas_int *lda*, double ***const** *B*[], **const** rocblas_int *ldb*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_ssygst_batched**(rocblas_handle *handle*, **const** *rocblas_eform* *itype*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, float ***const** *A*[], **const** rocblas_int *lda*, float ***const** *B*[], **const** rocblas_int *ldb*, **const** rocblas_int *batch_count*)

SYGST_BATCHED reduces a batch of real symmetric-definite generalized eigenproblems to standard form.

(This is the blocked version of the algorithm).

For each instance in the batch, the problem solved by this function is either of the form

$$\begin{array}{ll} A_iX_i = \lambda B_iX_i & \text{1st form,} \\ A_iB_iX_i = \lambda X_i & \text{2nd form, or} \\ B_iA_iX_i = \lambda X_i & \text{3rd form,} \end{array}$$

depending on the value of itype.

If the problem is of the 1st form, then A_i is overwritten with

$$\begin{array}{ll} U_i^{-T}A_iU_i^{-1}, & \text{or} \\ L_i^{-1}A_iL_i^{-T}, & \end{array}$$

where the symmetric-definite matrix B_i has been factorized as either $U_i^TU_i$ or $L_iL_i^T$ as returned by *POTRF*, depending on the value of uplo.

If the problem is of the 2nd or 3rd form, then A is overwritten with

$$\begin{array}{ll} U_iA_iU_i^T, & \text{or} \\ L_i^TA_iL_i, & \end{array}$$

also depending on the value of uplo.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] itype`: *rocblas_eform*. Specifies the form of the generalized eigenproblems.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower part of the matrices A_i are stored, and whether the factorization applied to B_i was upper or lower triangular. If uplo indicates lower (or upper), then the upper (or lower) parts of A_i and B_i are not used.
- `[in] n`: rocblas_int. n >= 0. The matrix dimensions.
- `[inout] A`: array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n. On entry, the matrices A_i. On exit, the transformed matrices associated with the equivalent standard eigenvalue problems.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of A_i.

- `[out] B`: array of pointers to type. Each pointer points to an array on the GPU of dimension ldb*n. The triangular factors of the matrices B_i, as returned by *POTRF_BATCHED*.
- `[in] ldb`: rocblas_int. ldb >= n. Specifies the leading dimension of B_i.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>sygst_strided_batched()

rocblas_status **rocsolver_dsygst_strided_batched**(rocblas_handle *handle*, **const** *rocblas_eform* *itype*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, double **B*, **const** rocblas_int *ldb*, **const** rocblas_stride *strideB*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_ssygst_strided_batched**(rocblas_handle *handle*, **const** *rocblas_eform* *itype*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, float **B*, **const** rocblas_int *ldb*, **const** rocblas_stride *strideB*, **const** rocblas_int *batch_count*)

SYGST_STRIDED_BATCHED reduces a batch of real symmetric-definite generalized eigenproblems to standard form.

(This is the blocked version of the algorithm).

For each instance in the batch, the problem solved by this function is either of the form

$$\begin{array}{ll} A_iX_i = \lambda B_iX_i & \text{1st form,} \\ A_iB_iX_i = \lambda X_i & \text{2nd form, or} \\ B_iA_iX_i = \lambda X_i & \text{3rd form,} \end{array}$$

depending on the value of itype.

If the problem is of the 1st form, then A_i is overwritten with

$$\begin{array}{ll} U_i^{-T}A_iU_i^{-1}, & \text{or} \\ L_i^{-1}A_iL_i^{-T}, & \end{array}$$

where the symmetric-definite matrix B_i has been factorized as either $U_i^TU_i$ or $L_iL_i^T$ as returned by *POTRF*, depending on the value of uplo.

If the problem is of the 2nd or 3rd form, then A is overwritten with

$$\begin{array}{ll} U_iA_iU_i^T, & \text{or} \\ L_i^TA_iL_i, & \end{array}$$

also depending on the value of uplo.

Parameters

- `[in] handle`: rocblas_handle.

- `[in] itype`: *rocblas_eform*. Specifies the form of the generalized eigenproblems.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower part of the matrices A_i are stored, and whether the factorization applied to B_i was upper or lower triangular. If uplo indicates lower (or upper), then the upper (or lower) parts of A_i and B_i are not used.
- `[in] n`: rocblas_int. n >= 0. The matrix dimensions.
- `[inout] A`: pointer to type. Array on the GPU (the size depends on the value of strideA). On entry, the matrices A_i. On exit, the transformed matrices associated with the equivalent standard eigenvalue problems.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of A_i.
- `[in] strideA`: rocblas_stride. Stride from the start of one matrix A_i to the next one A_(i+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n.
- `[out] B`: pointer to type. Array on the GPU (the size depends on the value of strideB). The triangular factors of the matrices B_i, as returned by *POTRF_STRIDED_BATCHED*.
- `[in] ldb`: rocblas_int. ldb >= n. Specifies the leading dimension of B_i.
- `[in] strideB`: rocblas_stride. Stride from the start of one matrix B_i to the next one B_(i+1). There is no restriction for the value of strideB. Normal use case is strideB >= ldb*n.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>hegst()

rocblas_status **`rocsolver_zhegst`** (rocblas_handle *handle*, **`const`** *rocblas_eform* *itype*, **`const`** rocblas_fill *uplo*, **`const`** rocblas_int *n*, rocblas_double_complex **A*, **`const`** rocblas_int *lda*, rocblas_double_complex **B*, **`const`** rocblas_int *ldb*)

rocblas_status **`rocsolver_chegst`** (rocblas_handle *handle*, **`const`** *rocblas_eform* *itype*, **`const`** rocblas_fill *uplo*, **`const`** rocblas_int *n*, rocblas_float_complex **A*, **`const`** rocblas_int *lda*, rocblas_float_complex **B*, **`const`** rocblas_int *ldb*)

HEGST reduces a hermitian-definite generalized eigenproblem to standard form.

(This is the blocked version of the algorithm).

The problem solved by this function is either of the form

$$\begin{array}{cl} AX = \lambda BX & \text{1st form,} \\ ABX = \lambda X & \text{2nd form, or} \\ BAX = \lambda X & \text{3rd form,} \end{array}$$

depending on the value of itype.

If the problem is of the 1st form, then A is overwritten with

$$\begin{array}{cl} U^{-H}AU^{-1}, & \text{or} \\ L^{-1}AL^{-H}, & \end{array}$$

where the hermitian-definite matrix B has been factorized as either U^HU or LL^H as returned by *POTRF*, depending on the value of uplo.

If the problem is of the 2nd or 3rd form, then A is overwritten with

$$\begin{array}{cc} UAU^H, & \text{or} \\ L^H AL, & \end{array}$$

also depending on the value of uplo.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] itype`: *rocblas_eform*. Specifies the form of the generalized eigenproblem.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower part of the matrix A is stored, and whether the factorization applied to B was upper or lower triangular. If uplo indicates lower (or upper), then the upper (or lower) parts of A and B are not used.
- `[in] n`: rocblas_int. n >= 0. The matrix dimensions.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the matrix A. On exit, the transformed matrix associated with the equivalent standard eigenvalue problem.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of A.
- `[out] B`: pointer to type. Array on the GPU of dimension ldb*n. The triangular factor of the matrix B, as returned by *POTRF*.
- `[in] ldb`: rocblas_int. ldb >= n. Specifies the leading dimension of B.

rocsolver_<type>hegst_batched()

rocblas_status **rocsolver_zhegst_batched**(rocblas_handle *handle*, **const** *rocblas_eform* *itype*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_double_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_double_complex ***const** *B*[], **const** rocblas_int *ldb*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_chegst_batched**(rocblas_handle *handle*, **const** *rocblas_eform* *itype*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_float_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_float_complex ***const** *B*[], **const** rocblas_int *ldb*, **const** rocblas_int *batch_count*)

HEGST_BATCHED reduces a batch of hermitian-definite generalized eigenproblems to standard form.

(This is the blocked version of the algorithm).

For each instance in the batch, the problem solved by this function is either of the form

$$\begin{array}{cl} A_i X_i = \lambda B_i X_i & \text{1st form,} \\ A_i B_i X_i = \lambda X_i & \text{2nd form, or} \\ B_i A_i X_i = \lambda X_i & \text{3rd form,} \end{array}$$

depending on the value of itype.

If the problem is of the 1st form, then A_i is overwritten with

$$\begin{array}{ll} U_i^{-H} A_i U_i^{-1}, & \text{or} \\ L_i^{-1} A_i L_i^{-H}, & \end{array}$$

where the hermitian-definite matrix B_i has been factorized as either $U_i^H U_i$ or $L_i L_i^H$ as returned by *POTRF*, depending on the value of uplo.

If the problem is of the 2nd or 3rd form, then A is overwritten with

$$\begin{array}{ll} U_i A_i U_i^H, & \text{or} \\ L_i^H A_i L_i, & \end{array}$$

also depending on the value of uplo.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] itype`: *rocblas_eform*. Specifies the form of the generalized eigenproblems.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower part of the matrices A_i are stored, and whether the factorization applied to B_i was upper or lower triangular. If uplo indicates lower (or upper), then the upper (or lower) parts of A_i and B_i are not used.
- `[in] n`: rocblas_int. n >= 0. The matrix dimensions.
- `[inout] A`: array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n. On entry, the matrices A_i. On exit, the transformed matrices associated with the equivalent standard eigenvalue problems.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of A_i.
- `[out] B`: array of pointers to type. Each pointer points to an array on the GPU of dimension ldb*n. The triangular factors of the matrices B_i, as returned by *POTRF_BATCHED*.
- `[in] ldb`: rocblas_int. ldb >= n. Specifies the leading dimension of B_i.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>hegst_strided_batched()

rocblas_status **rocsolver_zhegst_strided_batched**(rocblas_handle *handle*, **const** *rocblas_eform* *itype*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_double_complex **B*, **const** rocblas_int *ldb*, **const** rocblas_stride *strideB*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_chegst_strided_batched**(rocblas_handle *handle*, **const** *rocblas_eform* *itype*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_float_complex **B*, **const** rocblas_int *ldb*, **const** rocblas_stride *strideB*, **const** rocblas_int *batch_count*)

HEGST_STRIDED_BATCHED reduces a batch of hermitian-definite generalized eigenproblems to standard form.

(This is the blocked version of the algorithm).

For each instance in the batch, the problem solved by this function is either of the form

$$\begin{array}{cl} A_i X_i = \lambda B_i X_i & \text{1st form,} \\ A_i B_i X_i = \lambda X_i & \text{2nd form, or} \\ B_i A_i X_i = \lambda X_i & \text{3rd form,} \end{array}$$

depending on the value of itype.

If the problem is of the 1st form, then A_i is overwritten with

$$\begin{array}{cl} U_i^{-H} A_i U_i^{-1}, & \text{or} \\ L_i^{-1} A_i L_i^{-H}, & \end{array}$$

where the hermitian-definite matrix B_i has been factorized as either $U_i^H U_i$ or $L_i L_i^H$ as returned by *POTRF*, depending on the value of uplo.

If the problem is of the 2nd or 3rd form, then A is overwritten with

$$\begin{array}{cl} U_i A_i U_i^H, & \text{or} \\ L_i^H A_i L_i, & \end{array}$$

also depending on the value of uplo.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] itype`: *rocblas_eform*. Specifies the form of the generalized eigenproblems.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower part of the matrices A_i are stored, and whether the factorization applied to B_i was upper or lower triangular. If uplo indicates lower (or upper), then the upper (or lower) parts of A_i and B_i are not used.
- `[in] n`: rocblas_int. n >= 0. The matrix dimensions.
- `[inout] A`: pointer to type. Array on the GPU (the size depends on the value of strideA). On entry, the matrices A_i. On exit, the transformed matrices associated with the equivalent standard eigenvalue problems.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of A_i.
- `[in] strideA`: rocblas_stride. Stride from the start of one matrix A_i to the next one A_(i+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n.
- `[out] B`: pointer to type. Array on the GPU (the size depends on the value of strideB). The triangular factors of the matrices B_i, as returned by *POTRF_STRIDED_BATCHED*.
- `[in] ldb`: rocblas_int. ldb >= n. Specifies the leading dimension of B_i.
- `[in] strideB`: rocblas_stride. Stride from the start of one matrix B_i to the next one B_(i+1). There is no restriction for the value of strideB. Normal use case is strideB >= ldb*n.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

3.3.4 Linear-systems solvers

List of linear solvers

- *rocsolver_<type>trtri()*
- *rocsolver_<type>trtri_batched()*
- *rocsolver_<type>trtri_strided_batched()*
- *rocsolver_<type>getri()*
- *rocsolver_<type>getri_batched()*
- *rocsolver_<type>getri_strided_batched()*
- *rocsolver_<type>getrs()*
- *rocsolver_<type>getrs_batched()*
- *rocsolver_<type>getrs_strided_batched()*
- *rocsolver_<type>gesv()*
- *rocsolver_<type>gesv_batched()*
- *rocsolver_<type>gesv_strided_batched()*
- *rocsolver_<type>potri()*
- *rocsolver_<type>potri_batched()*
- *rocsolver_<type>potri_strided_batched()*
- *rocsolver_<type>potrs()*
- *rocsolver_<type>potrs_batched()*
- *rocsolver_<type>potrs_strided_batched()*
- *rocsolver_<type>posv()*
- *rocsolver_<type>posv_batched()*
- *rocsolver_<type>posv_strided_batched()*

rocsolver_<type>trtri()

rocblas_status **rocsolver_ztrtri** (rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_diagonal *diag*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, rocblas_int **info*)

rocblas_status **rocsolver_ctrtri** (rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_diagonal *diag*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, rocblas_int **info*)

rocblas_status **rocsolver_dtrtri** (rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_diagonal *diag*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, rocblas_int **info*)

rocblas_status **rocsolver_strtri** (rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_diagonal *diag*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, rocblas_int **info*)

TRTRI inverts a triangular n-by-n matrix A.

A can be upper or lower triangular, depending on the value of uplo, and unit or non-unit triangular, depending on the value of diag.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower part of the matrix A is stored. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.
- `[in] diag`: rocblas_diagonal. If diag indicates unit, then the diagonal elements of A are not referenced and assumed to be one.
- `[in] n`: rocblas_int. n >= 0. The number of rows and columns of the matrix A.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the triangular matrix. On exit, the inverse of A if info = 0.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of A.
- `[out] info`: pointer to a rocblas_int on the GPU. If info = 0, successful exit. If info = i > 0, A is singular. A[i,i] is the first zero element in the diagonal.

rocsolver_<type>trtri_batched()

rocblas_status **rocsolver_ztrtri_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_diagonal *diag*, **const** rocblas_int *n*, rocblas_double_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_ctrtri_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_diagonal *diag*, **const** rocblas_int *n*, rocblas_float_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dtrtri_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_diagonal *diag*, **const** rocblas_int *n*, double ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_strtri_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_diagonal *diag*, **const** rocblas_int *n*, float ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **info*, **const** rocblas_int *batch_count*)

TRTRI_BATCHED inverts a batch of triangular n-by-n matrices A_j.

A_j can be upper or lower triangular, depending on the value of uplo, and unit or non-unit triangular, depending on the value of diag.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower part of the matrices A_j are stored. If uplo indicates lower (or upper), then the upper (or lower) part of A_j is not used.
- `[in] diag`: rocblas_diagonal. If diag indicates unit, then the diagonal elements of matrices A_j are not referenced and assumed to be one.

- `[in] n`: rocblas_int. n >= 0. The number of rows and columns of all matrices A_j in the batch.
- `[inout] A`: array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n. On entry, the triangular matrices A_j. On exit, the inverses of A_j if info[j] = 0.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of matrices A_j.
- `[out] info`: pointer to rocblas_int. Array of batch_count integers on the GPU. If info[j] = 0, successful exit for inversion of A_j. If info[j] = i > 0, A_j is singular. A_j[i,i] is the first zero element in the diagonal.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>trtri_strided_batched()

rocblas_status **rocsolver_ztrtri_strided_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_diagonal *diag*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_ctrtri_strided_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_diagonal *diag*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dtrtri_strided_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_diagonal *diag*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_strtri_strided_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_diagonal *diag*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **info*, **const** rocblas_int *batch_count*)

TRTRI_STRIDED_BATCHED inverts a batch of triangular n-by-n matrices A_j.

A_j can be upper or lower triangular, depending on the value of uplo, and unit or non-unit triangular, depending on the value of diag.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower part of the matrices A_j are stored. If uplo indicates lower (or upper), then the upper (or lower) part of A_j is not used.
- `[in] diag`: rocblas_diagonal. If diag indicates unit, then the diagonal elements of matrices A_j are not referenced and assumed to be one.
- `[in] n`: rocblas_int. n >= 0. The number of rows and columns of all matrices A_j in the batch.
- `[inout] A`: pointer to type. Array on the GPU (the size depends on the value of strideA). On entry, the triangular matrices A_j. On exit, the inverses of A_j if info[j] = 0.

- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of matrices A_j.
- `[in] strideA`: rocblas_stride. Stride from the start of one matrix A_j to the next one A_(j+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n
- `[out] info`: pointer to rocblas_int. Array of batch_count integers on the GPU. If info[j] = 0, successful exit for inversion of A_j. If info[j] = i > 0, A_j is singular. A_j[i,i] is the first zero element in the diagonal.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>getri()

rocblas_status **rocsolver_zgetri**(rocblas_handle *handle*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, rocblas_int **ipiv*, rocblas_int **info*)

rocblas_status **rocsolver_cgetri**(rocblas_handle *handle*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, rocblas_int **ipiv*, rocblas_int **info*)

rocblas_status **rocsolver_dgetri**(rocblas_handle *handle*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, rocblas_int **ipiv*, rocblas_int **info*)

rocblas_status **rocsolver_sgetri**(rocblas_handle *handle*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, rocblas_int **ipiv*, rocblas_int **info*)

GETRI inverts a general n-by-n matrix A using the LU factorization computed by *GETRF*.

The inverse is computed by solving the linear system

$$A^{-1}L = U^{-1}$$

where L is the lower triangular factor of A with unit diagonal elements, and U is the upper triangular factor.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] n`: rocblas_int. n >= 0. The number of rows and columns of the matrix A.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the factors L and U of the factorization A = P*L*U returned by *GETRF*. On exit, the inverse of A if info = 0; otherwise undefined.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of A.
- `[in] ipiv`: pointer to rocblas_int. Array on the GPU of dimension n. The pivot indices returned by *GETRF*.
- `[out] info`: pointer to a rocblas_int on the GPU. If info = 0, successful exit. If info = i > 0, U is singular. U[i,i] is the first zero pivot.

rocsolver_<type>getri_batched()

rocblas_status **rocsolver_zgetri_batched**(rocblas_handle *handle*, **const** rocblas_int *n*, rocblas_double_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cgetri_batched**(rocblas_handle *handle*, **const** rocblas_int *n*, rocblas_float_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dgetri_batched**(rocblas_handle *handle*, **const** rocblas_int *n*, double ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_sgetri_batched**(rocblas_handle *handle*, **const** rocblas_int *n*, float ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_int **info*, **const** rocblas_int *batch_count*)

GETRI_BATCHED inverts a batch of general n-by-n matrices using the LU factorization computed by *GETRF_BATCHED*.

The inverse of matrix A_j in the batch is computed by solving the linear system

$$A_j^{-1} L_j = U_j^{-1}$$

where L_j is the lower triangular factor of A_j with unit diagonal elements, and U_j is the upper triangular factor.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] n`: rocblas_int. n >= 0. The number of rows and columns of all matrices A_j in the batch.
- `[inout] A`: array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n. On entry, the factors L_j and U_j of the factorization A = P_j*L_j*U_j returned by *GETRF_BATCHED*. On exit, the inverses of A_j if info[j] = 0; otherwise undefined.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of matrices A_j.
- `[in] ipiv`: pointer to rocblas_int. Array on the GPU (the size depends on the value of strideP). The pivot indices returned by *GETRF_BATCHED*.
- `[in] strideP`: rocblas_stride. Stride from the start of one vector ipiv_j to the next one ipiv_(i+j). There is no restriction for the value of strideP. Normal use case is strideP >= n.
- `[out] info`: pointer to rocblas_int. Array of batch_count integers on the GPU. If info[j] = 0, successful exit for inversion of A_j. If info[j] = i > 0, U_j is singular. U_j[i,i] is the first zero pivot.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>getri_strided_batched()

rocblas_status **rocsolver_zgetri_strided_batched** (rocblas_handle *handle*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cgetri_strided_batched** (rocblas_handle *handle*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dgetri_strided_batched** (rocblas_handle *handle*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_sgetri_strided_batched** (rocblas_handle *handle*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_int **info*, **const** rocblas_int *batch_count*)

GETRI_STRIDED_BATCHED inverts a batch of general n-by-n matrices using the LU factorization computed by *GETRF_STRIDED_BATCHED*.

The inverse of matrix A_j in the batch is computed by solving the linear system

$$A_j^{-1} L_j = U_j^{-1}$$

where L_j is the lower triangular factor of A_j with unit diagonal elements, and U_j is the upper triangular factor.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] n`: rocblas_int. n >= 0. The number of rows and columns of all matrices A_j in the batch.
- `[inout] A`: pointer to type. Array on the GPU (the size depends on the value of strideA). On entry, the factors L_j and U_j of the factorization A_j = P_j*L_j*U_j returned by *GETRF_STRIDED_BATCHED*. On exit, the inverses of A_j if info[j] = 0; otherwise undefined.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of matrices A_j.
- `[in] strideA`: rocblas_stride. Stride from the start of one matrix A_j to the next one A_(j+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n
- `[in] ipiv`: pointer to rocblas_int. Array on the GPU (the size depends on the value of strideP). The pivot indices returned by *GETRF_STRIDED_BATCHED*.
- `[in] strideP`: rocblas_stride. Stride from the start of one vector ipiv_j to the next one ipiv_(j+1). There is no restriction for the value of strideP. Normal use case is strideP >= n.

- [out] info: pointer to rocblas_int. Array of batch_count integers on the GPU. If info[j] = 0, successful exit for inversion of A_j. If info[j] = i > 0, U_j is singular. U_j[i,i] is the first zero pivot.
- [in] batch_count: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>getrs()

rocblas_status **rocsolver_zgetrs**(rocblas_handle *handle*, **const** rocblas_operation *trans*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, rocblas_double_complex **A*, **const** rocblas_int *lda*, **const** rocblas_int **ipiv*, rocblas_double_complex **B*, **const** rocblas_int *ldb*)

rocblas_status **rocsolver_cgetrs**(rocblas_handle *handle*, **const** rocblas_operation *trans*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, rocblas_float_complex **A*, **const** rocblas_int *lda*, **const** rocblas_int **ipiv*, rocblas_float_complex **B*, **const** rocblas_int *ldb*)

rocblas_status **rocsolver_dgetrs**(rocblas_handle *handle*, **const** rocblas_operation *trans*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, double **A*, **const** rocblas_int *lda*, **const** rocblas_int **ipiv*, double **B*, **const** rocblas_int *ldb*)

rocblas_status **rocsolver_sgetrs**(rocblas_handle *handle*, **const** rocblas_operation *trans*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, float **A*, **const** rocblas_int *lda*, **const** rocblas_int **ipiv*, float **B*, **const** rocblas_int *ldb*)

GETRS solves a system of n linear equations on n variables in its factorized form.

It solves one of the following systems, depending on the value of trans:

$$\begin{array}{cl} AX = B & \text{not transposed,} \\ A^T X = B & \text{transposed, or} \\ A^H X = B & \text{conjugate transposed.} \end{array}$$

Matrix A is defined by its triangular factors as returned by *GETRF*.

Parameters

- [in] handle: rocblas_handle.
- [in] trans: rocblas_operation. Specifies the form of the system of equations.
- [in] n: rocblas_int. n >= 0. The order of the system, i.e. the number of columns and rows of A.
- [in] nrhs: rocblas_int. nrhs >= 0. The number of right hand sides, i.e., the number of columns of the matrix B.
- [in] A: pointer to type. Array on the GPU of dimension lda*n. The factors L and U of the factorization A = P*L*U returned by *GETRF*.
- [in] lda: rocblas_int. lda >= n. The leading dimension of A.
- [in] ipiv: pointer to rocblas_int. Array on the GPU of dimension n. The pivot indices returned by *GETRF*.
- [inout] B: pointer to type. Array on the GPU of dimension ldb*nrhs. On entry, the right hand side matrix B. On exit, the solution matrix X.
- [in] ldb: rocblas_int. ldb >= n. The leading dimension of B.

rocsolver_<type>getrs_batched()

rocblas_status **rocsolver_zgetrs_batched**(rocblas_handle *handle*, **const** rocblas_operation *trans*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, rocblas_double_complex ***const** *A*[], **const** rocblas_int *lda*, **const** rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_double_complex ***const** *B*[], **const** rocblas_int *ldb*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cgetrs_batched**(rocblas_handle *handle*, **const** rocblas_operation *trans*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, rocblas_float_complex ***const** *A*[], **const** rocblas_int *lda*, **const** rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_float_complex ***const** *B*[], **const** rocblas_int *ldb*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dgetrs_batched**(rocblas_handle *handle*, **const** rocblas_operation *trans*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, double ***const** *A*[], **const** rocblas_int *lda*, **const** rocblas_int **ipiv*, **const** rocblas_stride *strideP*, double ***const** *B*[], **const** rocblas_int *ldb*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_sgetrs_batched**(rocblas_handle *handle*, **const** rocblas_operation *trans*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, float ***const** *A*[], **const** rocblas_int *lda*, **const** rocblas_int **ipiv*, **const** rocblas_stride *strideP*, float ***const** *B*[], **const** rocblas_int *ldb*, **const** rocblas_int *batch_count*)

GETRS_BATCHED solves a batch of systems of n linear equations on n variables in its factorized forms.

For each instance j in the batch, it solves one of the following systems, depending on the value of trans:

$$\begin{array}{ll} A_j X_j = B_j & \text{not transposed,} \\ A_j^T X_j = B_j & \text{transposed, or} \\ A_j^H X_j = B_j & \text{conjugate transposed.} \end{array}$$

Matrix A_j is defined by its triangular factors as returned by *GETRF_BATCHED*.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] trans`: rocblas_operation. Specifies the form of the system of equations of each instance in the batch.
- `[in] n`: rocblas_int. n >= 0. The order of the system, i.e. the number of columns and rows of all A_j matrices.
- `[in] nrhs`: rocblas_int. nrhs >= 0. The number of right hand sides, i.e., the number of columns of all the matrices B_j.
- `[in] A`: Array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n. The factors L_j and U_j of the factorization A_j = P_j*L_j*U_j returned by *GETRF_BATCHED*.
- `[in] lda`: rocblas_int. lda >= n. The leading dimension of matrices A_j.
- `[in] ipiv`: pointer to rocblas_int. Array on the GPU (the size depends on the value of strideP). Contains the vectors ipiv_j of pivot indices returned by *GETRF_BATCHED*.

- `[in] strideP`: rocblas_stride. Stride from the start of one vector ipiv_j to the next one ipiv_(j+1). There is no restriction for the value of strideP. Normal use case is strideP >= n.
- `[inout] B`: Array of pointers to type. Each pointer points to an array on the GPU of dimension ldb*nrhs. On entry, the right hand side matrices B_j. On exit, the solution matrix X_j of each system in the batch.
- `[in] ldb`: rocblas_int. ldb >= n. The leading dimension of matrices B_j.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of instances (systems) in the batch.

rocsolver_<type>getrs_strided_batched()

rocblas_status **rocsolver_zgetrs_strided_batched**(rocblas_handle *handle*, **const** rocblas_operation *trans*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, rocblas_double_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, **const** rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_double_complex **B*, **const** rocblas_int *ldb*, **const** rocblas_stride *strideB*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cgetrs_strided_batched**(rocblas_handle *handle*, **const** rocblas_operation *trans*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, rocblas_float_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, **const** rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_float_complex **B*, **const** rocblas_int *ldb*, **const** rocblas_stride *strideB*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dgetrs_strided_batched**(rocblas_handle *handle*, **const** rocblas_operation *trans*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, double **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, **const** rocblas_int **ipiv*, **const** rocblas_stride *strideP*, double **B*, **const** rocblas_int *ldb*, **const** rocblas_stride *strideB*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_sgetrs_strided_batched**(rocblas_handle *handle*, **const** rocblas_operation *trans*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, float **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, **const** rocblas_int **ipiv*, **const** rocblas_stride *strideP*, float **B*, **const** rocblas_int *ldb*, **const** rocblas_stride *strideB*, **const** rocblas_int *batch_count*)

GETRS_STRIDED_BATCHED solves a batch of systems of n linear equations on n variables in its factorized forms.

For each instance j in the batch, it solves one of the following systems, depending on the value of trans:

$$\begin{array}{ll} A_j X_j = B_j & \text{not transposed,} \\ A_j^T X_j = B_j & \text{transposed, or} \\ A_j^H X_j = B_j & \text{conjugate transposed.} \end{array}$$

Matrix A_j is defined by its triangular factors as returned by *GETRF_STRIDED_BATCHED*.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] trans`: rocblas_operation. Specifies the form of the system of equations of each instance in the batch.
- `[in] n`: rocblas_int. n >= 0. The order of the system, i.e. the number of columns and rows of all A_j matrices.
- `[in] nrhs`: rocblas_int. nrhs >= 0. The number of right hand sides, i.e., the number of columns of all the matrices B_j.
- `[in] A`: pointer to type. Array on the GPU (the size depends on the value of strideA). The factors L_j and U_j of the factorization A_j = P_j*L_j*U_j returned by *GETRF_STRIDED_BATCHED*.
- `[in] lda`: rocblas_int. lda >= n. The leading dimension of matrices A_j.
- `[in] strideA`: rocblas_stride. Stride from the start of one matrix A_j to the next one A_(j+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n.
- `[in] ipiv`: pointer to rocblas_int. Array on the GPU (the size depends on the value of strideP). Contains the vectors ipiv_j of pivot indices returned by *GETRF_STRIDED_BATCHED*.
- `[in] strideP`: rocblas_stride. Stride from the start of one vector ipiv_j to the next one ipiv_(j+1). There is no restriction for the value of strideP. Normal use case is strideP >= n.
- `[inout] B`: pointer to type. Array on the GPU (size depends on the value of strideB). On entry, the right hand side matrices B_j. On exit, the solution matrix X_j of each system in the batch.
- `[in] ldb`: rocblas_int. ldb >= n. The leading dimension of matrices B_j.
- `[in] strideB`: rocblas_stride. Stride from the start of one matrix B_j to the next one B_(j+1). There is no restriction for the value of strideB. Normal use case is strideB >= ldb*nrhs.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of instances (systems) in the batch.

rocsolver_<type>gesv()

rocblas_status **rocsolver_zgesv**(rocblas_handle *handle*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, rocblas_double_complex **A*, **const** rocblas_int *lda*, rocblas_int **ipiv*, rocblas_double_complex **B*, **const** rocblas_int *ldb*, rocblas_int **info*)

rocblas_status **rocsolver_cgesv**(rocblas_handle *handle*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, rocblas_float_complex **A*, **const** rocblas_int *lda*, rocblas_int **ipiv*, rocblas_float_complex **B*, **const** rocblas_int *ldb*, rocblas_int **info*)

rocblas_status **rocsolver_dgesv**(rocblas_handle *handle*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, double **A*, **const** rocblas_int *lda*, rocblas_int **ipiv*, double **B*, **const** rocblas_int *ldb*, rocblas_int **info*) rocblas_status **rocsolver_sgesv**(rocblas_handle *handle*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, float **A*, **const** rocblas_int *lda*, rocblas_int **ipiv*, float **B*, **const** rocblas_int *ldb*, rocblas_int **info*)

GESV solves a general system of n linear equations on n variables.

The linear system is of the form

$$AX = B$$

where A is a general n-by-n matrix. Matrix A is first factorized in triangular factors L and U using *GETRF*; then, the solution is computed with *GETRS*.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] n`: rocblas_int. n >= 0. The order of the system, i.e. the number of columns and rows of A.
- `[in] nrhs`: rocblas_int. nrhs >= 0. The number of right hand sides, i.e., the number of columns of the matrix B.
- `[in] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the matrix A. On exit, if info = 0, the factors L and U of the LU decomposition of A returned by *GETRF*.
- `[in] lda`: rocblas_int. lda >= n. The leading dimension of A.
- `[out] ipiv`: pointer to rocblas_int. Array on the GPU of dimension n. The pivot indices returned by *GETRF*.
- `[inout] B`: pointer to type. Array on the GPU of dimension ldb*nrhs. On entry, the right hand side matrix B. On exit, the solution matrix X.
- `[in] ldb`: rocblas_int. ldb >= n. The leading dimension of B.
- `[out] info`: pointer to a rocblas_int on the GPU. If info = 0, successful exit. If info = i > 0, U is singular, and the solution could not be computed. U[i,i] is the first zero element in the diagonal.

rocsolver_<type>gesv_batched()

rocblas_status **rocsolver_zgesv_batched**(rocblas_handle *handle*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, rocblas_double_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_double_complex ***const** *B*[], **const** rocblas_int *ldb*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cgesv_batched**(rocblas_handle *handle*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, rocblas_float_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_float_complex ***const** *B*[], **const** rocblas_int *ldb*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dgesv_batched**(rocblas_handle *handle*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, double ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, double ***const** *B*[], **const** rocblas_int *ldb*, rocblas_int **info*, **const** rocblas_int *batch_count*) rocblas_status **rocsolver_sgesv_batched** (rocblas_handle *handle*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, float ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, float ***const** *B*[], **const** rocblas_int *ldb*, rocblas_int **info*, **const** rocblas_int *batch_count*)

GESV_BATCHED solves a batch of general systems of n linear equations on n variables.

The linear systems are of the form

$$A_j X_j = B_j$$

where A_j is a general n-by-n matrix. Matrix A_j is first factorized in triangular factors L_j and U_j using *GETRF_BATCHED*; then, the solutions are computed with *GETRS_BATCHED*.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] n`: rocblas_int. n >= 0. The order of the system, i.e. the number of columns and rows of all A_j matrices.
- `[in] nrhs`: rocblas_int. nrhs >= 0. The number of right hand sides, i.e., the number of columns of all the matrices B_j.
- `[in] A`: Array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n. On entry, the matrices A_j. On exit, if info_j = 0, the factors L_j and U_j of the LU decomposition of A_j returned by *GETRF_BATCHED*.
- `[in] lda`: rocblas_int. lda >= n. The leading dimension of matrices A_j.
- `[out] ipiv`: pointer to rocblas_int. Array on the GPU (the size depends on the value of strideP). The vectors ipiv_j of pivot indices returned by *GETRF_BATCHED*.
- `[in] strideP`: rocblas_stride. Stride from the start of one vector ipiv_j to the next one ipiv_(j+1). There is no restriction for the value of strideP. Normal use case is strideP >= n.
- `[inout] B`: Array of pointers to type. Each pointer points to an array on the GPU of dimension ldb*nrhs. On entry, the right hand side matrices B_j. On exit, the solution matrix X_j of each system in the batch.
- `[in] ldb`: rocblas_int. ldb >= n. The leading dimension of matrices B_j.
- `[out] info`: pointer to rocblas_int. Array of batch_count integers on the GPU. If info[j] = 0, successful exit for A_j. If info[i] = j > 0, U_i is singular, and the solution could not be computed. U_j[i,i] is the first zero element in the diagonal.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of instances (systems) in the batch.

rocsolver_<type>gesv_strided_batched()

rocblas_status **rocsolver_zgesv_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, rocblas_double_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_double_complex **B*, **const** rocblas_int *ldb*, **const** rocblas_stride *strideB*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cgesv_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, rocblas_float_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_float_complex **B*, **const** rocblas_int *ldb*, **const** rocblas_stride *strideB*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dgesv_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, double **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, double **B*, **const** rocblas_int *ldb*, **const** rocblas_stride *strideB*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_sgesv_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, float **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, float **B*, **const** rocblas_int *ldb*, **const** rocblas_stride *strideB*, rocblas_int **info*, **const** rocblas_int *batch_count*)

GESV_STRIDED_BATCHED solves a batch of general systems of n linear equations on n variables.

The linear systems are of the form

$$A_j X_j = B_j$$

where A_j is a general n-by-n matrix. Matrix A_j is first factorized in triangular factors L_j and U_j using *GETRF_STRIDED_BATCHED*; then, the solutions are computed with *GETRS_STRIDED_BATCHED*.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] n`: rocblas_int. n >= 0. The order of the system, i.e. the number of columns and rows of all A_j matrices.
- `[in] nrhs`: rocblas_int. nrhs >= 0. The number of right hand sides, i.e., the number of columns of all the matrices B_j.

- `[in] A`: pointer to type. Array on the GPU (the size depends on the value of strideA). On entry, the matrices A_j. On exit, if info_j = 0, the factors L_j and U_j of the LU decomposition of A_j returned by *GETRF_STRIDED_BATCHED*.
- `[in] lda`: rocblas_int. lda >= n. The leading dimension of matrices A_j.
- `[in] strideA`: rocblas_stride. Stride from the start of one matrix A_j to the next one A_(j+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n.
- `[out] ipiv`: pointer to rocblas_int. Array on the GPU (the size depends on the value of strideP). The vectors ipiv_j of pivot indices returned by *GETRF_STRIDED_BATCHED*.
- `[in] strideP`: rocblas_stride. Stride from the start of one vector ipiv_j to the next one ipiv_(j+1). There is no restriction for the value of strideP. Normal use case is strideP >= n.
- `[inout] B`: pointer to type. Array on the GPU (size depends on the value of strideB). On entry, the right hand side matrices B_j. On exit, the solution matrix X_j of each system in the batch.
- `[in] ldb`: rocblas_int. ldb >= n. The leading dimension of matrices B_j.
- `[in] strideB`: rocblas_stride. Stride from the start of one matrix B_j to the next one B_(j+1). There is no restriction for the value of strideB. Normal use case is strideB >= ldb*nrhs.
- `[out] info`: pointer to rocblas_int. Array of batch_count integers on the GPU. If info[j] = 0, successful exit for A_j. If info[i] = j > 0, U_i is singular, and the solution could not be computed. U_j[i,i] is the first zero element in the diagonal.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of instances (systems) in the batch.

rocsolver_<type>potri()

rocblas_status **rocsolver_zpotri** (rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, rocblas_int **info*)

rocblas_status **rocsolver_cpotri** (rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, rocblas_int **info*)

rocblas_status **rocsolver_dpotri** (rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, rocblas_int **info*)

rocblas_status **rocsolver_spotri** (rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, rocblas_int **info*)

POTRI inverts a symmetric/hermitian positive definite matrix A.

The inverse of matrix A is computed as

$$\begin{array}{cl} A^{-1} = U^{-1}U^{-1'} & \text{if uplo is upper, or} \\ A^{-1} = L^{-1'}L^{-1} & \text{if uplo is lower.} \end{array}$$

where U or L is the triangular factor of the Cholesky factorization of A returned by *POTRF*.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] uplo`: rocblas_fill. Specifies whether the factorization is upper or lower triangular. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.
- `[in] n`: rocblas_int. n >= 0. The number of rows and columns of matrix A.

- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the factor L or U of the Cholesky factorization of A returned by *POTRF*. On exit, the inverses of A if info = 0.
- `[in] lda`: rocblas_int. lda >= n. specifies the leading dimension of A.
- `[out] info`: pointer to a rocblas_int on the GPU. If info = 0, successful exit for inversion of A. If info = j > 0, A is singular. L[j,j] or U[j,j] is zero.

rocsolver_<type>potri_batched()

rocblas_status **rocsolver_zpotri_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_double_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cpotri_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_float_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dpotri_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, double ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_spotri_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, float ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **info*, **const** rocblas_int *batch_count*)

POTRI_BATCHED inverts a batch of symmetric/hermitian positive definite matrices A_i.

The inverse of matrix A_i in the batch is computed as

$$\begin{array}{cl} A_i^{-1} = U_i^{-1} {U_i^{-1}}' & \text{if uplo is upper, or} \\ A_i^{-1} = {L_i^{-1}}' L_i^{-1} & \text{if uplo is lower.} \end{array}$$

where U_i or L_i is the triangular factor of the Cholesky factorization of A_i returned by *POTRF_BATCHED*.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] uplo`: rocblas_fill. Specifies whether the factorization is upper or lower triangular. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.
- `[in] n`: rocblas_int. n >= 0. The number of rows and columns of matrix A_i.
- `[inout] A`: array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n. On entry, the factor L_i or U_i of the Cholesky factorization of A_i returned by *POTRF_BATCHED*. On exit, the inverses of A_i if info[i] = 0.
- `[in] lda`: rocblas_int. lda >= n. specifies the leading dimension of A_i.
- `[out] info`: pointer to rocblas_int. Array of batch_count integers on the GPU. If info[i] = 0, successful exit for inversion of A_i. If info[i] = j > 0, A_i is singular. L_i[j,j] or U_i[j,j] is zero.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>potri_strided_batched()

rocblas_status **rocsolver_zpotri_strided_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cpotri_strided_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dpotri_strided_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_spotri_strided_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **info*, **const** rocblas_int *batch_count*)

POTRI_STRIDED_BATCHED inverts a batch of symmetric/hermitian positive definite matrices A_i.

The inverse of matrix A_i in the batch is computed as

$$\begin{array}{cl} A_i^{-1} = U_i^{-1} {U_i^{-1}}' & \text{if uplo is upper, or} \\ A_i^{-1} = {L_i^{-1}}' L_i^{-1} & \text{if uplo is lower.} \end{array}$$

where U_i or L_i is the triangular factor of the Cholesky factorization of A_i returned by *POTRF_STRIDED_BATCHED*.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] uplo`: rocblas_fill. Specifies whether the factorization is upper or lower triangular. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.
- `[in] n`: rocblas_int. n >= 0. The number of rows and columns of matrix A_i.
- `[inout] A`: pointer to type. Array on the GPU (the size depends on the value of strideA). On entry, the factor L_i or U_i of the Cholesky factorization of A_i returned by *POTRF_STRIDED_BATCHED*. On exit, the inverses of A_i if info[i] = 0.
- `[in] lda`: rocblas_int. lda >= n. specifies the leading dimension of A_i.
- `[in] strideA`: rocblas_stride. Stride from the start of one matrix A_i to the next one A_(i+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n.
- `[out] info`: pointer to rocblas_int. Array of batch_count integers on the GPU. If info[i] = 0, successful exit for inversion of A_i. If info[i] = j > 0, A_i is singular. L_i[j,j] or U_i[j,j] is zero.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>potrs()

rocblas_status **rocsolver_zpotrs** (rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, rocblas_double_complex **A*, **const** rocblas_int *lda*, rocblas_double_complex **B*, **const** rocblas_int *ldb*)

rocblas_status **rocsolver_cpotrs** (rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, rocblas_float_complex **A*, **const** rocblas_int *lda*, rocblas_float_complex **B*, **const** rocblas_int *ldb*)

rocblas_status **rocsolver_dpotrs** (rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, double **A*, **const** rocblas_int *lda*, double **B*, **const** rocblas_int *ldb*)

rocblas_status **rocsolver_spotrs** (rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, float **A*, **const** rocblas_int *lda*, float **B*, **const** rocblas_int *ldb*)

POTRS solves a symmetric/hermitian system of n linear equations on n variables in its factorized form.

It solves the system

$$AX = B$$

where A is a real symmetric (complex hermitian) positive definite matrix defined by its triangular factor

$$\begin{array}{cl} A = U'U & \text{if uplo is upper, or} \\ A = LL' & \text{if uplo is lower.} \end{array}$$

as returned by *POTRF*.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] uplo`: rocblas_fill. Specifies whether the factorization is upper or lower triangular. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.
- `[in] n`: rocblas_int. n >= 0. The order of the system, i.e. the number of columns and rows of A.
- `[in] nrhs`: rocblas_int. nrhs >= 0. The number of right hand sides, i.e., the number of columns of the matrix B.
- `[in] A`: pointer to type. Array on the GPU of dimension lda*n. The factor L or U of the Cholesky factorization of A returned by *POTRF*.
- `[in] lda`: rocblas_int. lda >= n. The leading dimension of A.
- `[inout] B`: pointer to type. Array on the GPU of dimension ldb*nrhs. On entry, the right hand side matrix B. On exit, the solution matrix X.
- `[in] ldb`: rocblas_int. ldb >= n. The leading dimension of B.

rocsolver_<type>potrs_batched()

rocblas_status **rocsolver_zpotrs_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, rocblas_double_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_double_complex ***const** *B*[], **const** rocblas_int *ldb*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cpotrs_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, rocblas_float_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_float_complex ***const** *B*[], **const** rocblas_int *ldb*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dpotrs_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, double ***const** *A*[], **const** rocblas_int *lda*, double ***const** *B*[], **const** rocblas_int *ldb*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_spotrs_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, float ***const** *A*[], **const** rocblas_int *lda*, float ***const** *B*[], **const** rocblas_int *ldb*, **const** rocblas_int *batch_count*)

POTRS_BATCHED solves a batch of symmetric/hermitian systems of n linear equations on n variables in its factorized forms.

For each instance j in the batch, it solves the system

$$A_j X_j = B_j$$

where A_j is a real symmetric (complex hermitian) positive definite matrix defined by its triangular factor

$$\begin{array}{cl} A_j = U_j' U_j & \text{if uplo is upper, or} \\ A_j = L_j L_j' & \text{if uplo is lower.} \end{array}$$

as returned by *POTRF_BATCHED*.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] uplo`: rocblas_fill. Specifies whether the factorization is upper or lower triangular. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.
- `[in] n`: rocblas_int. n >= 0. The order of the system, i.e. the number of columns and rows of all A_j matrices.
- `[in] nrhs`: rocblas_int. nrhs >= 0. The number of right hand sides, i.e., the number of columns of all the matrices B_j.
- `[in] A`: Array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n. The factor L_j or U_j of the Cholesky factorization of A_j returned by *POTRF_BATCHED*.
- `[in] lda`: rocblas_int. lda >= n. The leading dimension of matrices A_j.

- `[inout] B`: Array of pointers to type. Each pointer points to an array on the GPU of dimension ldb*nrhs. On entry, the right hand side matrices B_j. On exit, the solution matrix X_j of each system in the batch.
- `[in] ldb`: rocblas_int. ldb >= n. The leading dimension of matrices B_j.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of instances (systems) in the batch.

rocsolver_<type>potrs_strided_batched()

rocblas_status **rocsolver_zpotrs_strided_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, rocblas_double_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_double_complex **B*, **const** rocblas_int *ldb*, **const** rocblas_stride *strideB*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cpotrs_strided_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, rocblas_float_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_float_complex **B*, **const** rocblas_int *ldb*, **const** rocblas_stride *strideB*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dpotrs_strided_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, double **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, double **B*, **const** rocblas_int *ldb*, **const** rocblas_stride *strideB*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_spotrs_strided_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, float **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, float **B*, **const** rocblas_int *ldb*, **const** rocblas_stride *strideB*, **const** rocblas_int *batch_count*)

POTRS_STRIDED_BATCHED solves a batch of symmetric/hermitian systems of n linear equations on n variables in its factorized forms.

For each instance j in the batch, it solves the system

$$A_j X_j = B_j$$

where A_j is a real symmetric (complex hermitian) positive definite matrix defined by its triangular factor

$$\begin{array}{cl} A_j = U_j' U_j & \text{if uplo is upper, or} \\ A_j = L_j L_j' & \text{if uplo is lower.} \end{array}$$

as returned by *POTRF_STRIDED_BATCHED*.

Parameters

- [in] handle: rocblas_handle.
- [in] uplo: rocblas_fill. Specifies whether the factorization is upper or lower triangular. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.
- [in] n: rocblas_int. n >= 0. The order of the system, i.e. the number of columns and rows of all A_j matrices.
- [in] nrhs: rocblas_int. nrhs >= 0. The number of right hand sides, i.e., the number of columns of all the matrices B_j.
- [in] A: pointer to type. Array on the GPU (the size depends on the value of strideA). The factor L_j or U_j of the Cholesky factorization of A_j returned by *POTRF_STRIDED_BATCHED*.
- [in] lda: rocblas_int. lda >= n. The leading dimension of matrices A_j.
- [in] strideA: rocblas_stride. Stride from the start of one matrix A_j to the next one A_(j+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n.
- [inout] B: pointer to type. Array on the GPU (size depends on the value of strideB). On entry, the right hand side matrices B_j. On exit, the solution matrix X_j of each system in the batch.
- [in] ldb: rocblas_int. ldb >= n. The leading dimension of matrices B_j.
- [in] strideB: rocblas_stride. Stride from the start of one matrix B_j to the next one B_(j+1). There is no restriction for the value of strideB. Normal use case is strideB >= ldb*nrhs.
- [in] batch_count: rocblas_int. batch_count >= 0. Number of instances (systems) in the batch.

rocsolver_<type>posv()

rocblas_status **rocsolver_zposv**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, rocblas_double_complex **A*, **const** rocblas_int *lda*, rocblas_double_complex **B*, **const** rocblas_int *ldb*, rocblas_int **info*)

rocblas_status **rocsolver_cposv**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, rocblas_float_complex **A*, **const** rocblas_int *lda*, rocblas_float_complex **B*, **const** rocblas_int *ldb*, rocblas_int **info*)

rocblas_status **rocsolver_dposv**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, double **A*, **const** rocblas_int *lda*, double **B*, **const** rocblas_int *ldb*, rocblas_int **info*)

rocblas_status **rocsolver_sposv**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, float **A*, **const** rocblas_int *lda*, float **B*, **const** rocblas_int *ldb*, rocblas_int **info*)

POSV solves a symmetric/hermitian system of n linear equations on n variables.

It solves the system

$$AX = B$$

where A is a real symmetric (complex hermitian) positive definite matrix. Matrix A is first factorized as $A = LL'$ or $A = U'U$, depending on the value of uplo, using *POTRF*; then, the solution is computed with *POTRS*.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] uplo`: rocblas_fill. Specifies whether the factorization is upper or lower triangular. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.
- `[in] n`: rocblas_int. n >= 0. The order of the system, i.e. the number of columns and rows of A.
- `[in] nrhs`: rocblas_int. nrhs >= 0. The number of right hand sides, i.e., the number of columns of the matrix B.
- `[in] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the symmetric/hermitian matrix A. On exit, if info = 0, the factor L or U of the Cholesky factorization of A returned by *POTRF*.
- `[in] lda`: rocblas_int. lda >= n. The leading dimension of A.
- `[inout] B`: pointer to type. Array on the GPU of dimension ldb*nrhs. On entry, the right hand side matrix B. On exit, the solution matrix X.
- `[in] ldb`: rocblas_int. ldb >= n. The leading dimension of B.
- `[out] info`: pointer to a rocblas_int on the GPU. If info = 0, successful exit. If info = j > 0, the leading minor of order j of A is not positive definite. The solution could not be computed.

rocsolver_<type>posv_batched()

rocblas_status **rocsolver_zposv_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, rocblas_double_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_double_complex ***const** *B*[], **const** rocblas_int *ldb*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cposv_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, rocblas_float_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_float_complex ***const** *B*[], **const** rocblas_int *ldb*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dposv_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, double ***const** *A*[], **const** rocblas_int *lda*, double ***const** *B*[], **const** rocblas_int *ldb*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_sposv_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, float ***const** *A*[], **const** rocblas_int *lda*, float ***const** *B*[], **const** rocblas_int *ldb*, rocblas_int **info*, **const** rocblas_int *batch_count*)

POSV_BATCHED solves a batch of symmetric/hermitian systems of n linear equations on n variables.

For each instance j in the batch, it solves the system

$$A_j X_j = B_j$$

where A_j is a real symmetric (complex hermitian) positive definite matrix. Matrix A_j is first factorized as $A_j = L_j L_j'$ or $A_j = U_j' U_j$, depending on the value of uplo, using *POTRF_BATCHED*; then, the solution is computed with *POTRS_BATCHED*.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] uplo`: rocblas_fill. Specifies whether the factorization is upper or lower triangular. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.
- `[in] n`: rocblas_int. n >= 0. The order of the system, i.e. the number of columns and rows of all A_j matrices.
- `[in] nrhs`: rocblas_int. nrhs >= 0. The number of right hand sides, i.e., the number of columns of all the matrices B_j.
- `[in] A`: Array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n. On entry, the symmetric/hermitian matrices A_j. On exit, if info[j] = 0, the factor L_j or U_j of the Cholesky factorization of A_j returned by *POTRF_BATCHED*.
- `[in] lda`: rocblas_int. lda >= n. The leading dimension of matrices A_j.
- `[inout] B`: Array of pointers to type. Each pointer points to an array on the GPU of dimension ldb*nrhs. On entry, the right hand side matrices B_j. On exit, the solution matrix X_j of each system in the batch.
- `[in] ldb`: rocblas_int. ldb >= n. The leading dimension of matrices B_j.
- `[out] info`: pointer to rocblas_int. Array of batch_count integers on the GPU. If info[j] = 0, successful exit. If info[j] = i > 0, the leading minor of order i of A_j is not positive definite. The j-th solution could not be computed.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of instances (systems) in the batch.

rocsolver_<type>posv_strided_batched()

rocblas_status **rocsolver_zposv_strided_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, rocblas_double_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_double_complex **B*, **const** rocblas_int *ldb*, **const** rocblas_stride *strideB*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cposv_strided_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, rocblas_float_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_float_complex **B*, **const** rocblas_int *ldb*, **const** rocblas_stride *strideB*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dposv_strided_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, double **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, double **B*, **const** rocblas_int *ldb*, **const** rocblas_stride *strideB*, rocblas_int **info*, **const** rocblas_int *batch_count*) rocblas_status **rocsolver_sposv_strided_batched**(rocblas_handle *handle*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, float **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, float **B*, **const** rocblas_int *ldb*, **const** rocblas_stride *strideB*, rocblas_int **info*, **const** rocblas_int *batch_count*)

POSV_STRIDED_BATCHED solves a batch of symmetric/hermitian systems of n linear equations on n variables.

For each instance j in the batch, it solves the system

$$A_j X_j = B_j$$

where A_j is a real symmetric (complex hermitian) positive definite matrix. Matrix A_j is first factorized as $A_j = L_j L_j'$ or $A_j = U_j' U_j$, depending on the value of uplo, using *POTRF_STRIDED_BATCHED*; then, the solution is computed with *POTRS_STRIDED_BATCHED*.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] uplo`: rocblas_fill. Specifies whether the factorization is upper or lower triangular. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.
- `[in] n`: rocblas_int. n >= 0. The order of the system, i.e. the number of columns and rows of all A_j matrices.
- `[in] nrhs`: rocblas_int. nrhs >= 0. The number of right hand sides, i.e., the number of columns of all the matrices B_j.
- `[in] A`: pointer to type. Array on the GPU (the size depends on the value of strideA). On entry, the symmetric/hermitian matrices A_j. On exit, if info[j] = 0, the factor L_j or U_j of the Cholesky factorization of A_j returned by *POTRF_STRIDED_BATCHED*.
- `[in] lda`: rocblas_int. lda >= n. The leading dimension of matrices A_j.
- `[in] strideA`: rocblas_stride. Stride from the start of one matrix A_j to the next one A_(j+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n.
- `[inout] B`: pointer to type. Array on the GPU (size depends on the value of strideB). On entry, the right hand side matrices B_j. On exit, the solution matrix X_j of each system in the batch.
- `[in] ldb`: rocblas_int. ldb >= n. The leading dimension of matrices B_j.
- `[in] strideB`: rocblas_stride. Stride from the start of one matrix B_j to the next one B_(j+1). There is no restriction for the value of strideB. Normal use case is strideB >= ldb*nrhs.
- `[out] info`: pointer to rocblas_int. Array of batch_count integers on the GPU. If info[j] = 0, successful exit. If info[j] = i > 0, the leading minor of order i of A_j is not positive definite. The j-th solution could not be computed.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of instances (systems) in the batch.

3.3.5 Least-squares solvers

List of least-squares solvers

- *rocsolver_<type>gels()*
- *rocsolver_<type>gels_batched()*
- *rocsolver_<type>gels_strided_batched()*

rocsolver_<type>gels()

rocblas_status **rocsolver_zgels**(rocblas_handle *handle*, rocblas_operation *trans*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, rocblas_double_complex **A*, **const** rocblas_int *lda*, rocblas_double_complex **B*, **const** rocblas_int *ldb*, rocblas_int **info*)

rocblas_status **rocsolver_cgels**(rocblas_handle *handle*, rocblas_operation *trans*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, rocblas_float_complex **A*, **const** rocblas_int *lda*, rocblas_float_complex **B*, **const** rocblas_int *ldb*, rocblas_int **info*)

rocblas_status **rocsolver_dgels**(rocblas_handle *handle*, rocblas_operation *trans*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, double **A*, **const** rocblas_int *lda*, double **B*, **const** rocblas_int *ldb*, rocblas_int **info*)

rocblas_status **rocsolver_sgels**(rocblas_handle *handle*, rocblas_operation *trans*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, float **A*, **const** rocblas_int *lda*, float **B*, **const** rocblas_int *ldb*, rocblas_int **info*)

GELS solves an overdetermined (or underdetermined) linear system defined by an m-by-n matrix A, and a corresponding matrix B, using the QR factorization computed by *GEQRF* (or the LQ factorization computed by *GELQF*).

Depending on the value of trans, the problem solved by this function is either of the form

$$\begin{array}{cl} AX = B & \text{not transposed, or} \\ A'X = B & \text{transposed if real, or conjugate transposed if complex} \end{array}$$

If m >= n (or m < n in the case of transpose/conjugate transpose), the system is overdetermined and a least-squares solution approximating X is found by minimizing

$$||B - AX|| \quad \text{(or } ||B - A'X||\text{)}$$

If m < n (or m >= n in the case of transpose/conjugate transpose), the system is underdetermined and a unique solution for X is chosen such that $||X||$ is minimal.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] trans`: rocblas_operation. Specifies the form of the system of equations.
- `[in] m`: rocblas_int. m >= 0. The number of rows of matrix A.

- `[in] n`: rocblas_int. n >= 0. The number of columns of matrix A.
- `[in] nrhs`: rocblas_int. nrhs >= 0. The number of columns of matrices B and X; i.e., the columns on the right hand side.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the matrix A. On exit, the QR (or LQ) factorization of A as returned by *GEQRF* (or *GELQF*).
- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of matrix A.
- `[inout] B`: pointer to type. Array on the GPU of dimension ldb*nrhs. On entry, the matrix B. On exit, when info = 0, B is overwritten by the solution vectors (and the residuals in the overdetermined cases) stored as columns.
- `[in] ldb`: rocblas_int. ldb >= max(m,n). Specifies the leading dimension of matrix B.
- `[out] info`: pointer to rocblas_int on the GPU. If info = 0, successful exit. If info = j > 0, the solution could not be computed because input matrix A is rank deficient; the j-th diagonal element of its triangular factor is zero.

rocsolver_<type>gels_batched()

rocblas_status **rocsolver_zgels_batched**(rocblas_handle *handle*, rocblas_operation *trans*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, rocblas_double_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_double_complex ***const** *B*[], **const** rocblas_int *ldb*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cgels_batched**(rocblas_handle *handle*, rocblas_operation *trans*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, rocblas_float_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_float_complex ***const** *B*[], **const** rocblas_int *ldb*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dgels_batched**(rocblas_handle *handle*, rocblas_operation *trans*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, double ***const** *A*[], **const** rocblas_int *lda*, double ***const** *B*[], **const** rocblas_int *ldb*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_sgels_batched**(rocblas_handle *handle*, rocblas_operation *trans*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, float ***const** *A*[], **const** rocblas_int *lda*, float ***const** *B*[], **const** rocblas_int *ldb*, rocblas_int **info*, **const** rocblas_int *batch_count*)

GELS_BATCHED solves a batch of overdetermined (or underdetermined) linear systems defined by a set of m-by-n matrices A_i, and corresponding matrices B_i, using the QR factorizations computed by *GEQRF_BATCHED* (or the LQ factorizations computed by *GELQF_BATCHED*).

For each instance in the batch, depending on the value of trans, the problem solved by this function is either of the form

$$\begin{array}{ll} A_i X_i = B_i & \text{not transposed, or} \\ A_i' X_i = B_i & \text{transposed if real, or conjugate transposed if complex} \end{array}$$

If m >= n (or m < n in the case of transpose/conjugate transpose), the system is overdetermined and a least-squares solution approximating X_i is found by minimizing

$$||B_i - A_i X_i|| \quad (\text{or } ||B_i - A_i' X_i||)$$

If m < n (or m >= n in the case of transpose/conjugate transpose), the system is underdetermined and a unique solution for X_i is chosen such that $||X_i||$ is minimal.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] trans`: rocblas_operation. Specifies the form of the system of equations.
- `[in] m`: rocblas_int. m >= 0. The number of rows of all matrices A_i in the batch.
- `[in] n`: rocblas_int. n >= 0. The number of columns of all matrices A_i in the batch.
- `[in] nrhs`: rocblas_int. nrhs >= 0. The number of columns of all matrices B_i and X_i in the batch; i.e., the columns on the right hand side.
- `[inout] A`: array of pointer to type. Each pointer points to an array on the GPU of dimension lda*n. On entry, the matrices A_i. On exit, the QR (or LQ) factorizations of A_i as returned by *GEQRF_BATCHED* (or *GELQF_BATCHED*).
- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of matrices A_i.
- `[inout] B`: array of pointer to type. Each pointer points to an array on the GPU of dimension ldb*nrhs. On entry, the matrices B_i. On exit, when info[i] = 0, B_i is overwritten by the solution vectors (and the residuals in the overdetermined cases) stored as columns.
- `[in] ldb`: rocblas_int. ldb >= max(m,n). Specifies the leading dimension of matrices B_i.
- `[out] info`: pointer to rocblas_int. Array of batch_count integers on the GPU. If info[i] = 0, successful exit for solution of A_i. If info[i] = j > 0, the solution of A_i could not be computed because input matrix A_i is rank deficient; the j-th diagonal element of its triangular factor is zero.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>gels_strided_batched()

rocblas_status **rocsolver_zgels_strided_batched**(rocblas_handle *handle*, rocblas_operation *trans*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, rocblas_double_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_double_complex **B*, **const** rocblas_int *ldb*, **const** rocblas_stride *strideB*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cgels_strided_batched**(rocblas_handle *handle*, rocblas_operation *trans*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, rocblas_float_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_float_complex **B*, **const** rocblas_int *ldb*, **const** rocblas_stride *strideB*, rocblas_int **info*, **const** rocblas_int *batch_count*) rocblas_status **rocsolver_dgels_strided_batched** (rocblas_handle *handle*, rocblas_operation *trans*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, double **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, double **B*, **const** rocblas_int *ldb*, **const** rocblas_stride *strideB*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_sgels_strided_batched** (rocblas_handle *handle*, rocblas_operation *trans*, **const** rocblas_int *m*, **const** rocblas_int *n*, **const** rocblas_int *nrhs*, float **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, float **B*, **const** rocblas_int *ldb*, **const** rocblas_stride *strideB*, rocblas_int **info*, **const** rocblas_int *batch_count*)

GELS_STRIDED_BATCHED solves a batch of overdetermined (or underdetermined) linear systems defined by a set of m-by-n matrices A_i, and corresponding matrices B_i, using the QR factorizations computed by *GEQRF_STRIDED_BATCHED* (or the LQ factorizations computed by *GELQF_STRIDED_BATCHED*).

For each instance in the batch, depending on the value of trans, the problem solved by this function is either of the form

$$\begin{array}{cl} A_i X_i = B_i & \text{not transposed, or} \\ A_i' X_i = B_i & \text{transposed if real, or conjugate transposed if complex} \end{array}$$

If m >= n (or m < n in the case of transpose/conjugate transpose), the system is overdetermined and a least-squares solution approximating X_i is found by minimizing

$$||B_i - A_i X_i|| \quad \text{(or } ||B_i - A_i' X_i||\text{)}$$

If m < n (or m >= n in the case of transpose/conjugate transpose), the system is underdetermined and a unique solution for X_i is chosen such that $||X_i||$ is minimal.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] trans`: rocblas_operation. Specifies the form of the system of equations.
- `[in] m`: rocblas_int. m >= 0. The number of rows of all matrices A_i in the batch.
- `[in] n`: rocblas_int. n >= 0. The number of columns of all matrices A_i in the batch.
- `[in] nrhs`: rocblas_int. nrhs >= 0. The number of columns of all matrices B_i and X_i in the batch; i.e., the columns on the right hand side.
- `[inout] A`: pointer to type. Array on the GPU (the size depends on the value of strideA). On entry, the matrices A_i. On exit, the QR (or LQ) factorizations of A_i as returned by *GEQRF_STRIDED_BATCHED* (or *GELQF_STRIDED_BATCHED*).
- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of matrices A_i.
- `[in] strideA`: rocblas_stride. Stride from the start of one matrix A_i to the next one A_(i+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n

- `[inout] B`: pointer to type. Array on the GPU (the size depends on the value of strideB). On entry, the matrices B_i. On exit, when info = 0, each B_i is overwritten by the solution vectors (and the residuals in the overdetermined cases) stored as columns.
- `[in] ldb`: rocblas_int. ldb >= max(m,n). Specifies the leading dimension of matrices B_i.
- `[in] strideB`: rocblas_stride. Stride from the start of one matrix B_i to the next one B_(i+1). There is no restriction for the value of strideB. Normal use case is strideB >= ldb*nrhs
- `[out] info`: pointer to rocblas_int. Array of batch_count integers on the GPU. If info[i] = 0, successful exit for solution of A_i. If info[i] = j > 0, the solution of A_i could not be computed because input matrix A_i is rank deficient; the j-th diagonal element of its triangular factor is zero.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

3.3.6 Symmetric eigensolvers

List of symmetric eigensolvers

- *rocsolver_<type>syev()*
- *rocsolver_<type>syev_batched()*
- *rocsolver_<type>syev_strided_batched()*
- *rocsolver_<type>heev()*
- *rocsolver_<type>heev_batched()*
- *rocsolver_<type>heev_strided_batched()*
- *rocsolver_<type>syevd()*
- *rocsolver_<type>syevd_batched()*
- *rocsolver_<type>syevd_strided_batched()*
- *rocsolver_<type>heevd()*
- *rocsolver_<type>heevd_batched()*
- *rocsolver_<type>heevd_strided_batched()*
- *rocsolver_<type>sygv()*
- *rocsolver_<type>sygv_batched()*
- *rocsolver_<type>sygv_strided_batched()*
- *rocsolver_<type>hegv()*
- *rocsolver_<type>hegv_batched()*
- *rocsolver_<type>hegv_strided_batched()*
- *rocsolver_<type>sygvd()*
- *rocsolver_<type>sygvd_batched()*
- *rocsolver_<type>sygvd_strided_batched()*
- *rocsolver_<type>hegvd()*
- *rocsolver_<type>hegvd_batched()*

- *rocsolver_<type>hegvd_strided_batched()*

rocsolver_<type>syev()

rocblas_status **rocsolver_dsyev**(rocblas_handle *handle*, **const** *rocblas_evect evect*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, double **D*, double **E*, rocblas_int **info*)

rocblas_status **rocsolver_ssyev**(rocblas_handle *handle*, **const** *rocblas_evect evect*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, float **D*, float **E*, rocblas_int **info*)

SYEV computes the eigenvalues and optionally the eigenvectors of a real symmetric matrix A.

The eigenvalues are returned in ascending order. The eigenvectors are computed depending on the value of evect. The computed eigenvectors are orthonormal.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] evect`: *rocblas_evect*. Specifies whether the eigenvectors are to be computed. If evect is rocblas_evect_original, then the eigenvectors are computed. rocblas_evect_tridiagonal is not supported.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower part of the symmetric matrix A is stored. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.
- `[in] n`: rocblas_int. n >= 0. Number of rows and columns of matrix A.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the matrix A. On exit, the eigenvectors of A if they were computed and the algorithm converged; otherwise the contents of A are destroyed.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of matrix A.
- `[out] D`: pointer to type. Array on the GPU of dimension n. The eigenvalues of A in increasing order.
- `[out] E`: pointer to type. Array on the GPU of dimension n. This array is used to work internally with the tridiagonal matrix T associated with A. On exit, if info > 0, it contains the unconverged off-diagonal elements of T (or properly speaking, a tridiagonal matrix equivalent to T). The diagonal elements of this matrix are in D; those that converged correspond to a subset of the eigenvalues of A (not necessarily ordered).
- `[out] info`: pointer to a rocblas_int on the GPU. If info = 0, successful exit. If info = i > 0, the algorithm did not converge. i elements of E did not converge to zero.

rocsolver_<type>syev_batched()

rocblas_status **rocsolver_dsyev_batched**(rocblas_handle *handle*, **const** *rocblas_evect evect*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, double ***const** *A*[], **const** rocblas_int *lda*, double **D*, **const** rocblas_stride *strideD*, double **E*, **const** rocblas_stride *strideE*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_ssyev_batched**(rocblas_handle *handle*, **const** *rocblas_evect evect*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, float ***const** *A*[], **const** rocblas_int *lda*, float **D*, **const** rocblas_stride *strideD*, float **E*, **const** rocblas_stride *strideE*, rocblas_int **info*, **const** rocblas_int *batch_count*)

SYEV_BATCHED computes the eigenvalues and optionally the eigenvectors of a batch of real symmetric matrices A_j.

The eigenvalues are returned in ascending order. The eigenvectors are computed depending on the value of evect. The computed eigenvectors are orthonormal.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] evect`: *rocblas_evect*. Specifies whether the eigenvectors are to be computed. If evect is rocblas_evect_original, then the eigenvectors are computed. rocblas_evect_tridiagonal is not supported.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower part of the symmetric matrices A_j is stored. If uplo indicates lower (or upper), then the upper (or lower) part of A_j is not used.
- `[in] n`: rocblas_int. n >= 0. Number of rows and columns of matrices A_j.
- `[inout] A`: Array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n. On entry, the matrices A_j. On exit, the eigenvectors of A_j if they were computed and the algorithm converged; otherwise the contents of A_j are destroyed.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of matrices A_j.
- `[out] D`: pointer to type. Array on the GPU (the size depends on the value of strideD). The eigenvalues of A_j in increasing order.
- `[in] strideD`: rocblas_stride. Stride from the start of one vector D_j to the next one D_(j+1). There is no restriction for the value of strideD. Normal use case is strideD >= n.
- `[out] E`: pointer to type. Array on the GPU (the size depends on the value of strideE). This array is used to work internally with the tridiagonal matrix T_j associated with A_j. On exit, if info[j] > 0, E_j contains the unconverged off-diagonal elements of T_j (or properly speaking, a tridiagonal matrix equivalent to T_j). The diagonal elements of this matrix are in D_j; those that converged correspond to a subset of the eigenvalues of A_j (not necessarily ordered).
- `[in] strideE`: rocblas_stride. Stride from the start of one vector E_j to the next one E_(j+1). There is no restriction for the value of strideE. Normal use case is strideE >= n.
- `[out] info`: pointer to rocblas_int. Array of batch_count integers on the GPU. If info[j] = 0, successful exit for matrix A_j. If info[j] = i > 0, the algorithm did not converge. i elements of E_j did not converge to zero.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>syev_strided_batched()

rocblas_status **rocsolver_dsyev_strided_batched**(rocblas_handle *handle*, **const** *rocblas_evect evect*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, double **D*, **const** rocblas_stride *strideD*, double **E*, **const** rocblas_stride *strideE*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_ssyev_strided_batched**(rocblas_handle *handle*, **const** *rocblas_evect evect*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, float **D*, **const** rocblas_stride *strideD*, float **E*, **const** rocblas_stride *strideE*, rocblas_int **info*, **const** rocblas_int *batch_count*)

SYEV_STRIDED_BATCHED computes the eigenvalues and optionally the eigenvectors of a batch of real symmetric matrices A_j.

The eigenvalues are returned in ascending order. The eigenvectors are computed depending on the value of evect. The computed eigenvectors are orthonormal.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] evect`: *rocblas_evect*. Specifies whether the eigenvectors are to be computed. If evect is rocblas_evect_original, then the eigenvectors are computed. rocblas_evect_tridiagonal is not supported.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower part of the symmetric matrices A_j is stored. If uplo indicates lower (or upper), then the upper (or lower) part of A_j is not used.
- `[in] n`: rocblas_int. n >= 0. Number of rows and columns of matrices A_j.
- `[inout] A`: pointer to type. Array on the GPU (the size depends on the value of strideA). On entry, the matrices A_j. On exit, the eigenvectors of A_j if they were computed and the algorithm converged; otherwise the contents of A_j are destroyed.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of matrices A_j.
- `[in] strideA`: rocblas_stride. Stride from the start of one matrix A_j to the next one A_(j+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n.
- `[out] D`: pointer to type. Array on the GPU (the size depends on the value of strideD). The eigenvalues of A_j in increasing order.
- `[in] strideD`: rocblas_stride. Stride from the start of one vector D_j to the next one D_(j+1). There is no restriction for the value of strideD. Normal use case is strideD >= n.
- `[out] E`: pointer to type. Array on the GPU (the size depends on the value of strideE). This array is used to work internally with the tridiagonal matrix T_j associated with A_j. On exit, if info[j] > 0, E_j contains the unconverged off-diagonal elements of T_j (or properly speaking, a tridiagonal matrix equivalent to T_j). The diagonal elements of this matrix are in D_j; those that converged correspond to a subset of the eigenvalues of A_j (not necessarily ordered).
- `[in] strideE`: rocblas_stride. Stride from the start of one vector E_j to the next one E_(j+1). There is no restriction for the value of strideE. Normal use case is strideE >= n.

- `[out] info`: pointer to rocblas_int. Array of batch_count integers on the GPU. If info[j] = 0, successful exit for matrix A_j. If info[j] = i > 0, the algorithm did not converge. i elements of E_j did not converge to zero.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>heev()

rocblas_status **rocsolver_zheev**(rocblas_handle *handle*, **const** *rocblas_evect* *evect*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, double **D*, double **E*, rocblas_int **info*)

rocblas_status **rocsolver_cheev**(rocblas_handle *handle*, **const** *rocblas_evect* *evect*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, float **D*, float **E*, rocblas_int **info*)

HEEV computes the eigenvalues and optionally the eigenvectors of a Hermitian matrix A.

The eigenvalues are returned in ascending order. The eigenvectors are computed depending on the value of evect. The computed eigenvectors are orthonormal.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] evect`: *rocblas_evect*. Specifies whether the eigenvectors are to be computed. If evect is rocblas_evect_original, then the eigenvectors are computed. rocblas_evect_tridiagonal is not supported.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower part of the Hermitian matrix A is stored. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.
- `[in] n`: rocblas_int. n >= 0. Number of rows and columns of matrix A.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the matrix A. On exit, the eigenvectors of A if they were computed and the algorithm converged; otherwise the contents of A are destroyed.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of matrix A.
- `[out] D`: pointer to real type. Array on the GPU of dimension n. The eigenvalues of A in increasing order.
- `[out] E`: pointer to real type. Array on the GPU of dimension n. This array is used to work internally with the tridiagonal matrix T associated with A. On exit, if info > 0, it contains the unconverged off-diagonal elements of T (or properly speaking, a tridiagonal matrix equivalent to T). The diagonal elements of this matrix are in D; those that converged correspond to a subset of the eigenvalues of A (not necessarily ordered).
- `[out] info`: pointer to a rocblas_int on the GPU. If info = 0, successful exit. If info = i > 0, the algorithm did not converge. i elements of E did not converge to zero.

rocsolver_<type>heev_batched()

rocblas_status **rocsolver_zheev_batched**(rocblas_handle *handle*, **const** *rocblas_evect* *evect*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_double_complex ***const** *A*[], **const** rocblas_int *lda*, double **D*, **const** rocblas_stride *strideD*, double **E*, **const** rocblas_stride *strideE*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cheev_batched**(rocblas_handle *handle*, **const** *rocblas_evect* *evect*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_float_complex ***const** *A*[], **const** rocblas_int *lda*, float **D*, **const** rocblas_stride *strideD*, float **E*, **const** rocblas_stride *strideE*, rocblas_int **info*, **const** rocblas_int *batch_count*)

HEEV_BATCHED computes the eigenvalues and optionally the eigenvectors of a batch of Hermitian matrices A_j.

The eigenvalues are returned in ascending order. The eigenvectors are computed depending on the value of evect. The computed eigenvectors are orthonormal.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] evect`: *rocblas_evect*. Specifies whether the eigenvectors are to be computed. If evect is rocblas_evect_original, then the eigenvectors are computed. rocblas_evect_tridiagonal is not supported.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower part of the Hermitian matrices A_j is stored. If uplo indicates lower (or upper), then the upper (or lower) part of A_j is not used.
- `[in] n`: rocblas_int. n >= 0. Number of rows and columns of matrices A_j.
- `[inout] A`: Array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n. On entry, the matrices A_j. On exit, the eigenvectors of A_j if they were computed and the algorithm converged; otherwise the contents of A_j are destroyed.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of matrices A_j.
- `[out] D`: pointer to real type. Array on the GPU (the size depends on the value of strideD). The eigenvalues of A_j in increasing order.
- `[in] strideD`: rocblas_stride. Stride from the start of one vector D_j to the next one D_(j+1). There is no restriction for the value of strideD. Normal use case is strideD >= n.
- `[out] E`: pointer to real type. Array on the GPU (the size depends on the value of strideE). This array is used to work internally with the tridiagonal matrix T_j associated with A_j. On exit, if info[j] > 0, E_j contains the unconverged off-diagonal elements of T_j (or properly speaking, a tridiagonal matrix equivalent to T_j). The diagonal elements of this matrix are in D_j; those that converged correspond to a subset of the eigenvalues of A_j (not necessarily ordered).
- `[in] strideE`: rocblas_stride. Stride from the start of one vector E_j to the next one E_(j+1). There is no restriction for the value of strideE. Normal use case is strideE >= n.
- `[out] info`: pointer to rocblas_int. Array of batch_count integers on the GPU. If info[j] = 0, successful exit for matrix A_j. If info[j] = i > 0, the algorithm did not converge. i elements of E_j did not converge to zero.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>heev_strided_batched()

rocblas_status **rocsolver_zheev_strided_batched**(rocblas_handle *handle*, **const** *rocblas_evect* *evect*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, double **D*, **const** rocblas_stride *strideD*, double **E*, **const** rocblas_stride *strideE*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cheev_strided_batched**(rocblas_handle *handle*, **const** *rocblas_evect* *evect*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, float **D*, **const** rocblas_stride *strideD*, float **E*, **const** rocblas_stride *strideE*, rocblas_int **info*, **const** rocblas_int *batch_count*)

HEEV_STRIDED_BATCHED computes the eigenvalues and optionally the eigenvectors of a batch of Hermitian matrices A_j.

The eigenvalues are returned in ascending order. The eigenvectors are computed depending on the value of evect. The computed eigenvectors are orthonormal.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] evect`: *rocblas_evect*. Specifies whether the eigenvectors are to be computed. If evect is rocblas_evect_original, then the eigenvectors are computed. rocblas_evect_tridiagonal is not supported.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower part of the Hermitian matrices A_j is stored. If uplo indicates lower (or upper), then the upper (or lower) part of A_j is not used.
- `[in] n`: rocblas_int. n >= 0. Number of rows and columns of matrices A_j.
- `[inout] A`: pointer to type. Array on the GPU (the size depends on the value of strideA). On entry, the matrices A_j. On exit, the eigenvectors of A_j if they were computed and the algorithm converged; otherwise the contents of A_j are destroyed.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of matrices A_j.
- `[in] strideA`: rocblas_stride. Stride from the start of one matrix A_j to the next one A_(j+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n.
- `[out] D`: pointer to real type. Array on the GPU (the size depends on the value of strideD). The eigenvalues of A_j in increasing order.
- `[in] strideD`: rocblas_stride. Stride from the start of one vector D_j to the next one D_(j+1). There is no restriction for the value of strideD. Normal use case is strideD >= n.
- `[out] E`: pointer to real type. Array on the GPU (the size depends on the value of strideE). This array is used to work internally with the tridiagonal matrix T_j associated with A_j. On exit, if info[j] > 0, E_j contains the unconverged off-diagonal elements of T_j (or properly speaking, a tridiagonal matrix equivalent to T_j). The diagonal elements of this matrix are in D_j; those that converged correspond to a subset of the eigenvalues of A_j (not necessarily ordered).

- `[in] strideE`: rocblas_stride. Stride from the start of one vector E_j to the next one E_(j+1). There is no restriction for the value of strideE. Normal use case is strideE >= n.
- `[out] info`: pointer to rocblas_int. Array of batch_count integers on the GPU. If info[j] = 0, successful exit for matrix A_j. If info[j] = i > 0, the algorithm did not converge. i elements of E_j did not converge to zero.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>syevd()

rocblas_status **rocsolver_dsyevd**(rocblas_handle *handle*, **const** *rocblas_evect* *evect*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, double **D*, double **E*, rocblas_int **info*)

rocblas_status **rocsolver_ssyevd**(rocblas_handle *handle*, **const** *rocblas_evect* *evect*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, float **D*, float **E*, rocblas_int **info*)

SYEVD computes the eigenvalues and optionally the eigenvectors of a real symmetric matrix A.

The eigenvalues are returned in ascending order. The eigenvectors are computed using a divide-and-conquer algorithm, depending on the value of evect. The computed eigenvectors are orthonormal.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] evect`: *rocblas_evect*. Specifies whether the eigenvectors are to be computed. If evect is rocblas_evect_original, then the eigenvectors are computed. rocblas_evect_tridiagonal is not supported.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower part of the symmetric matrix A is stored. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.
- `[in] n`: rocblas_int. n >= 0. Number of rows and columns of matrix A.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the matrix A. On exit, the eigenvectors of A if they were computed and the algorithm converged; otherwise the contents of A are destroyed.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of matrix A.
- `[out] D`: pointer to type. Array on the GPU of dimension n. The eigenvalues of A in increasing order.
- `[out] E`: pointer to type. Array on the GPU of dimension n. This array is used to work internally with the tridiagonal matrix T associated with A. On exit, if info > 0, it contains the unconverged off-diagonal elements of T (or properly speaking, a tridiagonal matrix equivalent to T). The diagonal elements of this matrix are in D; those that converged correspond to a subset of the eigenvalues of A (not necessarily ordered).
- `[out] info`: pointer to a rocblas_int on the GPU. If info = 0, successful exit. If info = i > 0 and evect is rocblas_evect_none, the algorithm did not converge. i elements of E did not converge to zero. If info = i > 0 and evect is rocblas_evect_original, the algorithm failed to compute an eigenvalue in the submatrix from [i/(n+1), i/(n+1)] to [i%(n+1), i%(n+1)].

rocsolver_<type>syevd_batched()

rocblas_status **rocsolver_dsyevd_batched**(rocblas_handle *handle*, **const** *rocblas_evect* *evect*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, double **const* *A*[], **const** rocblas_int *lda*, double **D*, **const** rocblas_stride *strideD*, double **E*, **const** rocblas_stride *strideE*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_ssyevd_batched**(rocblas_handle *handle*, **const** *rocblas_evect* *evect*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, float **const* *A*[], **const** rocblas_int *lda*, float **D*, **const** rocblas_stride *strideD*, float **E*, **const** rocblas_stride *strideE*, rocblas_int **info*, **const** rocblas_int *batch_count*)

SYEVD_BATCHED computes the eigenvalues and optionally the eigenvectors of a batch of real symmetric matrices A_j.

The eigenvalues are returned in ascending order. The eigenvectors are computed using a divide-and-conquer algorithm, depending on the value of evect. The computed eigenvectors are orthonormal.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] evect`: *rocblas_evect*. Specifies whether the eigenvectors are to be computed. If evect is rocblas_evect_original, then the eigenvectors are computed. rocblas_evect_tridiagonal is not supported.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower part of the symmetric matrices A_j is stored. If uplo indicates lower (or upper), then the upper (or lower) part of A_j is not used.
- `[in] n`: rocblas_int. n >= 0. Number of rows and columns of matrices A_j.
- `[inout] A`: Array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n. On entry, the matrices A_j. On exit, the eigenvectors of A_j if they were computed and the algorithm converged; otherwise the contents of A_j are destroyed.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of matrices A_j.
- `[out] D`: pointer to type. Array on the GPU (the size depends on the value of strideD). The eigenvalues of A_j in increasing order.
- `[in] strideD`: rocblas_stride. Stride from the start of one vector D_j to the next one D_(j+1). There is no restriction for the value of strideD. Normal use case is strideD >= n.
- `[out] E`: pointer to type. Array on the GPU (the size depends on the value of strideE). This array is used to work internally with the tridiagonal matrix T_j associated with A_j. On exit, if info[j] > 0, E_j contains the unconverged off-diagonal elements of T_j (or properly speaking, a tridiagonal matrix equivalent to T_j). The diagonal elements of this matrix are in D_j; those that converged correspond to a subset of the eigenvalues of A_j (not necessarily ordered).
- `[in] strideE`: rocblas_stride. Stride from the start of one vector E_j to the next one E_(j+1). There is no restriction for the value of strideE. Normal use case is strideE >= n.
- `[out] info`: pointer to rocblas_int. Array of batch_count integers on the GPU. If info[j] = 0, successful exit for matrix A_j. If info[j] = i > 0 and evect is rocblas_evect_none, the algorithm did not converge. i elements of E_j did not converge to zero. If info[j] = i > 0 and evect is rocblas_evect_original, the algorithm failed to compute an eigenvalue in the submatrix from [i/(n+1), i/(n+1)] to [i%(n+1), i%(n+1)].

- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>syevd_strided_batched()

rocblas_status **rocsolver_dsyevd_strided_batched**(rocblas_handle *handle*, **const** *rocblas_evect evect*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, double **D*, **const** rocblas_stride *strideD*, double **E*, **const** rocblas_stride *strideE*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_ssyevd_strided_batched**(rocblas_handle *handle*, **const** *rocblas_evect evect*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, float **D*, **const** rocblas_stride *strideD*, float **E*, **const** rocblas_stride *strideE*, rocblas_int **info*, **const** rocblas_int *batch_count*)

SYEVD_STRIDED_BATCHED computes the eigenvalues and optionally the eigenvectors of a batch of real symmetric matrices A_j.

The eigenvalues are returned in ascending order. The eigenvectors are computed using a divide-and-conquer algorithm, depending on the value of evect. The computed eigenvectors are orthonormal.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] evect`: *rocblas_evect*. Specifies whether the eigenvectors are to be computed. If evect is rocblas_evect_original, then the eigenvectors are computed. rocblas_evect_tridiagonal is not supported.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower part of the symmetric matrices A_j is stored. If uplo indicates lower (or upper), then the upper (or lower) part of A_j is not used.
- `[in] n`: rocblas_int. n >= 0. Number of rows and columns of matrices A_j.
- `[inout] A`: pointer to type. Array on the GPU (the size depends on the value of strideA). On entry, the matrices A_j. On exit, the eigenvectors of A_j if they were computed and the algorithm converged; otherwise the contents of A_j are destroyed.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of matrices A_j.
- `[in] strideA`: rocblas_stride. Stride from the start of one matrix A_j to the next one A_(j+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n.
- `[out] D`: pointer to type. Array on the GPU (the size depends on the value of strideD). The eigenvalues of A_j in increasing order.
- `[in] strideD`: rocblas_stride. Stride from the start of one vector D_j to the next one D_(j+1). There is no restriction for the value of strideD. Normal use case is strideD >= n.
- `[out] E`: pointer to type. Array on the GPU (the size depends on the value of strideE). This array is used to work internally with the tridiagonal matrix T_j associated with A_j. On exit, if info[j] > 0, E_j contains the unconverged off-diagonal elements of T_j (or properly speaking, a tridiagonal matrix equivalent to T_j). The diagonal elements of this matrix are in D_j; those that converged correspond to a subset of the eigenvalues of A_j (not necessarily ordered).

- `[in] strideE`: rocblas_stride. Stride from the start of one vector E_j to the next one E_(j+1). There is no restriction for the value of strideE. Normal use case is strideE >= n.
- `[out] info`: pointer to rocblas_int. Array of batch_count integers on the GPU. If info[j] = 0, successful exit for matrix A_j. If info[j] = i > 0 and evect is rocblas_evect_none, the algorithm did not converge. i elements of E_j did not converge to zero. If info[j] = i > 0 and evect is rocblas_evect_original, the algorithm failed to compute an eigenvalue in the submatrix from [i/(n+1), i/(n+1)] to [i%(n+1), i%(n+1)].
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>heevd()

rocblas_status **rocsolver_zheevd**(rocblas_handle *handle*, **const** *rocblas_evect* *evect*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, double **D*, double **E*, rocblas_int **info*)

rocblas_status **rocsolver_cheevd**(rocblas_handle *handle*, **const** *rocblas_evect* *evect*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, float **D*, float **E*, rocblas_int **info*)

HEEVD computes the eigenvalues and optionally the eigenvectors of a Hermitian matrix A.

The eigenvalues are returned in ascending order. The eigenvectors are computed using a divide-and-conquer algorithm, depending on the value of evect. The computed eigenvectors are orthonormal.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] evect`: *rocblas_evect*. Specifies whether the eigenvectors are to be computed. If evect is rocblas_evect_original, then the eigenvectors are computed. rocblas_evect_tridiagonal is not supported.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower part of the Hermitian matrix A is stored. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.
- `[in] n`: rocblas_int. n >= 0. Number of rows and columns of matrix A.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the matrix A. On exit, the eigenvectors of A if they were computed and the algorithm converged; otherwise the contents of A are destroyed.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of matrix A.
- `[out] D`: pointer to real type. Array on the GPU of dimension n. The eigenvalues of A in increasing order.
- `[out] E`: pointer to real type. Array on the GPU of dimension n. This array is used to work internally with the tridiagonal matrix T associated with A. On exit, if info > 0, it contains the unconverged off-diagonal elements of T (or properly speaking, a tridiagonal matrix equivalent to T). The diagonal elements of this matrix are in D; those that converged correspond to a subset of the eigenvalues of A (not necessarily ordered).
- `[out] info`: pointer to a rocblas_int on the GPU. If info = 0, successful exit. If info = i > 0 and evect is rocblas_evect_none, the algorithm did not converge. i elements of E did not converge to zero. If info = i > 0 and evect is rocblas_evect_original, the algorithm failed to compute an eigenvalue in the submatrix from [i/(n+1), i/(n+1)] to [i%(n+1), i%(n+1)].

rocsolver_<type>heevd_batched()

rocblas_status **rocsolver_zheevd_batched**(rocblas_handle *handle*, **const** *rocblas_evect* *evect*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_double_complex ***const** *A*[], **const** rocblas_int *lda*, double **D*, **const** rocblas_stride *strideD*, double **E*, **const** rocblas_stride *strideE*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cheevd_batched**(rocblas_handle *handle*, **const** *rocblas_evect* *evect*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_float_complex ***const** *A*[], **const** rocblas_int *lda*, float **D*, **const** rocblas_stride *strideD*, float **E*, **const** rocblas_stride *strideE*, rocblas_int **info*, **const** rocblas_int *batch_count*)

HEEVD_BATCHED computes the eigenvalues and optionally the eigenvectors of a batch of Hermitian matrices A_j.

The eigenvalues are returned in ascending order. The eigenvectors are computed using a divide-and-conquer algorithm, depending on the value of evect. The computed eigenvectors are orthonormal.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] evect`: *rocblas_evect*. Specifies whether the eigenvectors are to be computed. If evect is rocblas_evect_original, then the eigenvectors are computed. rocblas_evect_tridiagonal is not supported.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower part of the Hermitian matrices A_j is stored. If uplo indicates lower (or upper), then the upper (or lower) part of A_j is not used.
- `[in] n`: rocblas_int. n >= 0. Number of rows and columns of matrices A_j.
- `[inout] A`: Array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n. On entry, the matrices A_j. On exit, the eigenvectors of A_j if they were computed and the algorithm converged; otherwise the contents of A_j are destroyed.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of matrices A_j.
- `[out] D`: pointer to real type. Array on the GPU (the size depends on the value of strideD). The eigenvalues of A_j in increasing order.
- `[in] strideD`: rocblas_stride. Stride from the start of one vector D_j to the next one D_(j+1). There is no restriction for the value of strideD. Normal use case is strideD >= n.
- `[out] E`: pointer to real type. Array on the GPU (the size depends on the value of strideE). This array is used to work internally with the tridiagonal matrix T_j associated with A_j. On exit, if info[j] > 0, E_j contains the unconverged off-diagonal elements of T_j (or properly speaking, a tridiagonal matrix equivalent to T_j). The diagonal elements of this matrix are in D_j; those that converged correspond to a subset of the eigenvalues of A_j (not necessarily ordered).
- `[in] strideE`: rocblas_stride. Stride from the start of one vector E_j to the next one E_(j+1). There is no restriction for the value of strideE. Normal use case is strideE >= n.
- `[out] info`: pointer to rocblas_int. Array of batch_count integers on the GPU. If info[j] = 0, successful exit for matrix A_j. If info[j] = i > 0 and evect is rocblas_evect_none, the algorithm did not converge. i elements of E_j did not converge to zero. If info[j] = i > 0 and evect is rocblas_evect_original, the algorithm failed to compute an eigenvalue in the submatrix from [i/(n+1), i/(n+1)] to [i%(n+1), i%(n+1)].

- [in] batch_count: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>heevd_strided_batched()

rocblas_status **rocsolver_zheevd_strided_batched**(rocblas_handle *handle*, **const** *rocblas_evect evect*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, double **D*, **const** rocblas_stride *strideD*, double **E*, **const** rocblas_stride *strideE*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cheevd_strided_batched**(rocblas_handle *handle*, **const** *rocblas_evect evect*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, float **D*, **const** rocblas_stride *strideD*, float **E*, **const** rocblas_stride *strideE*, rocblas_int **info*, **const** rocblas_int *batch_count*)

HEEVD_STRIDED_BATCHED computes the eigenvalues and optionally the eigenvectors of a batch of Hermitian matrices A_j.

The eigenvalues are returned in ascending order. The eigenvectors are computed using a divide-and-conquer algorithm, depending on the value of evect. The computed eigenvectors are orthonormal.

Parameters

- [in] handle: rocblas_handle.
- [in] evect: *rocblas_evect*. Specifies whether the eigenvectors are to be computed. If evect is rocblas_evect_original, then the eigenvectors are computed. rocblas_evect_tridiagonal is not supported.
- [in] uplo: rocblas_fill. Specifies whether the upper or lower part of the Hermitian matrices A_j is stored. If uplo indicates lower (or upper), then the upper (or lower) part of A_j is not used.
- [in] n: rocblas_int. n >= 0. Number of rows and columns of matrices A_j.
- [inout] A: pointer to type. Array on the GPU (the size depends on the value of strideA). On entry, the matrices A_j. On exit, the eigenvectors of A_j if they were computed and the algorithm converged; otherwise the contents of A_j are destroyed.
- [in] lda: rocblas_int. lda >= n. Specifies the leading dimension of matrices A_j.
- [in] strideA: rocblas_stride. Stride from the start of one matrix A_j to the next one A_(j+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n.
- [out] D: pointer to real type. Array on the GPU (the size depends on the value of strideD). The eigenvalues of A_j in increasing order.
- [in] strideD: rocblas_stride. Stride from the start of one vector D_j to the next one D_(j+1). There is no restriction for the value of strideD. Normal use case is strideD >= n.
- [out] E: pointer to real type. Array on the GPU (the size depends on the value of strideE). This array is used to work internally with the tridiagonal matrix T_j associated with A_j. On exit, if info[j] > 0, E_j contains the unconverged off-diagonal elements of T_j (or properly speaking, a tridiagonal

matrix equivalent to T_j). The diagonal elements of this matrix are in D_j; those that converged correspond to a subset of the eigenvalues of A_j (not necessarily ordered).

- `[in] strideE`: rocblas_stride. Stride from the start of one vector E_j to the next one E_(j+1). There is no restriction for the value of strideE. Normal use case is strideE >= n.
- `[out] info`: pointer to rocblas_int. Array of batch_count integers on the GPU. If info[j] = 0, successful exit for matrix A_j. If info[j] = i > 0 and evect is rocblas_evect_none, the algorithm did not converge. i elements of E_j did not converge to zero. If info[j] = i > 0 and evect is rocblas_evect_original, the algorithm failed to compute an eigenvalue in the submatrix from [i/(n+1), i/(n+1)] to [i%(n+1), i%(n+1)].
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>sygv()

rocblas_status **rocsolver_dsygv**(rocblas_handle *handle*, **const** *rocblas_eform* *itype*, **const** *rocblas_evect* *evect*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, double **B*, **const** rocblas_int *ldb*, double **D*, double **E*, rocblas_int **info*)

rocblas_status **rocsolver_ssygv**(rocblas_handle *handle*, **const** *rocblas_eform* *itype*, **const** *rocblas_evect* *evect*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, float **B*, **const** rocblas_int *ldb*, float **D*, float **E*, rocblas_int **info*)

SYGV computes the eigenvalues and (optionally) eigenvectors of a real generalized symmetric-definite eigenproblem.

The problem solved by this function is either of the form

$$\begin{array}{ll} AX = \lambda BX & \text{1st form,} \\ ABX = \lambda X & \text{2nd form, or} \\ BAX = \lambda X & \text{3rd form,} \end{array}$$

depending on the value of itype. The eigenvectors are computed depending on the value of evect.

When computed, the matrix Z of eigenvectors is normalized as follows:

$$\begin{array}{ll} Z^T BZ = I & \text{if 1st or 2nd form, or} \\ Z^T B^{-1} Z = I & \text{if 3rd form.} \end{array}$$

Parameters

- `[in] handle`: rocblas_handle.
- `[in] itype`: *rocblas_eform*. Specifies the form of the generalized eigenproblem.
- `[in] evect`: *rocblas_evect*. Specifies whether the eigenvectors are to be computed. If evect is rocblas_evect_original, then the eigenvectors are computed. rocblas_evect_tridiagonal is not supported.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower parts of the matrices A and B are stored. If uplo indicates lower (or upper), then the upper (or lower) parts of A and B are not used.
- `[in] n`: rocblas_int. n >= 0. The matrix dimensions.

- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the symmetric matrix A. On exit, if evect is original, the normalized matrix Z of eigenvectors. If evect is none, then the upper or lower triangular part of the matrix A (including the diagonal) is destroyed, depending on the value of uplo.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of A.
- `[out] B`: pointer to type. Array on the GPU of dimension ldb*n. On entry, the symmetric positive definite matrix B. On exit, the triangular factor of B as returned by *POTRF*.
- `[in] ldb`: rocblas_int. ldb >= n. Specifies the leading dimension of B.
- `[out] D`: pointer to type. Array on the GPU of dimension n. On exit, the eigenvalues in increasing order.
- `[out] E`: pointer to type. Array on the GPU of dimension n. This array is used to work internally with the tridiagonal matrix T associated with the reduced eigenvalue problem. On exit, if 0 < info <= n, it contains the unconverged off-diagonal elements of T (or properly speaking, a tridiagonal matrix equivalent to T). The diagonal elements of this matrix are in D; those that converged correspond to a subset of the eigenvalues (not necessarily ordered).
- `[out] info`: pointer to a rocblas_int on the GPU. If info = 0, successful exit. If info = j <= n, j off-diagonal elements of an intermediate tridiagonal form did not converge to zero. If info = n + j, the leading minor of order j of B is not positive definite.

rocsolver_<type>sygv_batched()

rocblas_status **rocsolver_dsygv_batched**(rocblas_handle *handle*, **const** *rocblas_eform* *itype*, **const** *rocblas_evect* *evect*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, double ***const** *A*[], **const** rocblas_int *lda*, double ***const** *B*[], **const** rocblas_int *ldb*, double **D*, **const** rocblas_stride *strideD*, double **E*, **const** rocblas_stride *strideE*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_ssygv_batched**(rocblas_handle *handle*, **const** *rocblas_eform* *itype*, **const** *rocblas_evect* *evect*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, float ***const** *A*[], **const** rocblas_int *lda*, float ***const** *B*[], **const** rocblas_int *ldb*, float **D*, **const** rocblas_stride *strideD*, float **E*, **const** rocblas_stride *strideE*, rocblas_int **info*, **const** rocblas_int *batch_count*)

SYGV_BATCHED computes the eigenvalues and (optionally) eigenvectors of a batch of real generalized symmetric-definite eigenproblems.

For each instance in the batch, the problem solved by this function is either of the form

$$\begin{array}{ll} A_iX_i = \lambda B_iX_i & \text{1st form,} \\ A_iB_iX_i = \lambda X_i & \text{2nd form, or} \\ B_iA_iX_i = \lambda X_i & \text{3rd form,} \end{array}$$

depending on the value of itype. The eigenvectors are computed depending on the value of evect.

When computed, the matrix Z_i of eigenvectors is normalized as follows:

$$\begin{array}{ll} Z_i^T B_i Z_i = I & \text{if 1st or 2nd form, or} \\ Z_i^T B_i^{-1} Z_i = I & \text{if 3rd form.} \end{array}$$

Parameters

- `[in] handle`: rocblas_handle.
- `[in] itype`: *rocblas_eform*. Specifies the form of the generalized eigenproblems.
- `[in] evect`: *rocblas_evect*. Specifies whether the eigenvectors are to be computed. If evect is rocblas_evect_original, then the eigenvectors are computed. rocblas_evect_tridiagonal is not supported.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower parts of the matrices A_i and B_i are stored. If uplo indicates lower (or upper), then the upper (or lower) parts of A_i and B_i are not used.
- `[in] n`: rocblas_int. n >= 0. The matrix dimensions.
- `[inout] A`: array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n. On entry, the symmetric matrices A_i. On exit, if evect is original, the normalized matrix Z_i of eigenvectors. If evect is none, then the upper or lower triangular part of the matrices A_i (including the diagonal) are destroyed, depending on the value of uplo.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of A_i.
- `[out] B`: array of pointers to type. Each pointer points to an array on the GPU of dimension ldb*n. On entry, the symmetric positive definite matrices B_i. On exit, the triangular factor of B_i as returned by *POTRF_BATCHED*.
- `[in] ldb`: rocblas_int. ldb >= n. Specifies the leading dimension of B_i.
- `[out] D`: pointer to type. Array on the GPU (the size depends on the value of strideD). On exit, the eigenvalues in increasing order.
- `[in] strideD`: rocblas_stride. Stride from the start of one vector D_i to the next one D_(i+1). There is no restriction for the value of strideD. Normal use is strideD >= n.
- `[out] E`: pointer to type. Array on the GPU (the size depends on the value of strideE). This array is used to work internally with the tridiagonal matrix T_i associated with the ith reduced eigenvalue problem. On exit, if 0 < info[i] <= n, E_i contains the unconverged off-diagonal elements of T_i (or properly speaking, a tridiagonal matrix equivalent to T_i). The diagonal elements of this matrix are in D_i; those that converged correspond to a subset of the eigenvalues (not necessarily ordered).
- `[in] strideE`: rocblas_stride. Stride from the start of one vector E_i to the next one E_(i+1). There is no restriction for the value of strideE. Normal use is strideE >= n.
- `[out] info`: pointer to rocblas_int. Array of batch_count integers on the GPU. If info[i] = 0, successful exit of batch instance i. If info[i] = j <= n, j off-diagonal elements of an intermediate tridiagonal form did not converge to zero. If info[i] = n + j, the leading minor of order j of B_i is not positive definite.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>sygv_strided_batched()

rocblas_status **rocsolver_dsygv_strided_batched**(rocblas_handle *handle*, **const** *rocblas_eform itype*, **const** *rocblas_evect evect*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, double **B*, **const** rocblas_int *ldb*, **const** rocblas_stride *strideB*, double **D*, **const** rocblas_stride *strideD*, double **E*, **const** rocblas_stride *strideE*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_ssygv_strided_batched**(rocblas_handle *handle*, **const** *rocblas_eform itype*, **const** *rocblas_evect evect*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, float **B*, **const** rocblas_int *ldb*, **const** rocblas_stride *strideB*, float **D*, **const** rocblas_stride *strideD*, float **E*, **const** rocblas_stride *strideE*, rocblas_int **info*, **const** rocblas_int *batch_count*)

SYGV_STRIDED_BATCHED computes the eigenvalues and (optionally) eigenvectors of a batch of real generalized symmetric-definite eigenproblems.

For each instance in the batch, the problem solved by this function is either of the form

$$\begin{array}{ll} A_iX_i = \lambda B_iX_i & \text{1st form,} \\ A_iB_iX_i = \lambda X_i & \text{2nd form, or} \\ B_iA_iX_i = \lambda X_i & \text{3rd form,} \end{array}$$

depending on the value of itype. The eigenvectors are computed depending on the value of evect.

When computed, the matrix Z_i of eigenvectors is normalized as follows:

$$\begin{array}{ll} Z_i^T B_i Z_i = I & \text{if 1st or 2nd form, or} \\ Z_i^T B_i^{-1} Z_i = I & \text{if 3rd form.} \end{array}$$

Parameters

- `[in] handle`: rocblas_handle.
- `[in] itype`: *rocblas_eform*. Specifies the form of the generalized eigenproblems.
- `[in] evect`: *rocblas_evect*. Specifies whether the eigenvectors are to be computed. If evect is rocblas_evect_original, then the eigenvectors are computed. rocblas_evect_tridiagonal is not supported.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower parts of the matrices A_i and B_i are stored. If uplo indicates lower (or upper), then the upper (or lower) parts of A_i and B_i are not used.
- `[in] n`: rocblas_int. n >= 0. The matrix dimensions.
- `[inout] A`: pointer to type. Array on the GPU (the size depends on the value of strideA). On entry, the symmetric matrices A_i. On exit, if evect is original, the normalized matrix Z_i of eigenvectors.

If evect is none, then the upper or lower triangular part of the matrices A_i (including the diagonal) are destroyed, depending on the value of uplo.

- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of A_i.
- `[in] strideA`: rocblas_stride. Stride from the start of one matrix A_i to the next one A_(i+1). There is no restriction for the value of strideA. Normal use is strideA >= lda*n.
- `[out] B`: pointer to type. Array on the GPU (the size depends on the value of strideB). On entry, the symmetric positive definite matrices B_i. On exit, the triangular factor of B_i as returned by *POTRF_STRIDED_BATCHED*.
- `[in] ldb`: rocblas_int. ldb >= n. Specifies the leading dimension of B_i.
- `[in] strideB`: rocblas_stride. Stride from the start of one matrix B_i to the next one B_(i+1). There is no restriction for the value of strideB. Normal use is strideB >= ldb*n.
- `[out] D`: pointer to type. Array on the GPU (the size depends on the value of strideD). On exit, the eigenvalues in increasing order.
- `[in] strideD`: rocblas_stride. Stride from the start of one vector D_i to the next one D_(i+1). There is no restriction for the value of strideD. Normal use is strideD >= n.
- `[out] E`: pointer to type. Array on the GPU (the size depends on the value of strideE). This array is used to work internally with the tridiagonal matrix T_i associated with the ith reduced eigenvalue problem. On exit, if 0 < info[i] <= n, it contains the unconverged off-diagonal elements of T_i (or properly speaking, a tridiagonal matrix equivalent to T_i). The diagonal elements of this matrix are in D_i; those that converged correspond to a subset of the eigenvalues (not necessarily ordered).
- `[in] strideE`: rocblas_stride. Stride from the start of one vector E_i to the next one E_(i+1). There is no restriction for the value of strideE. Normal use is strideE >= n.
- `[out] info`: pointer to rocblas_int. Array of batch_count integers on the GPU. If info[i] = 0, successful exit of batch i. If info[i] = j <= n, j off-diagonal elements of an intermediate tridiagonal form did not converge to zero. If info[i] = n + j, the leading minor of order j of B_i is not positive definite.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>hegv()

rocblas_status **rocsolver_zhegv**(rocblas_handle *handle*, **const** *rocblas_eform* *itype*, **const** *rocblas_evect* *evect*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, rocblas_double_complex **B*, **const** rocblas_int *ldb*, double **D*, double **E*, rocblas_int **info*)

rocblas_status **rocsolver_chegv**(rocblas_handle *handle*, **const** *rocblas_eform* *itype*, **const** *rocblas_evect* *evect*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, rocblas_float_complex **B*, **const** rocblas_int *ldb*, float **D*, float **E*, rocblas_int **info*)

HEGV computes the eigenvalues and (optionally) eigenvectors of a complex generalized hermitian-definite eigenproblem.

The problem solved by this function is either of the form

$$\begin{array}{ll} AX = \lambda BX & \text{1st form,} \\ ABX = \lambda X & \text{2nd form, or} \\ BAX = \lambda X & \text{3rd form,} \end{array}$$

depending on the value of itype. The eigenvectors are computed depending on the value of evect.

When computed, the matrix Z of eigenvectors is normalized as follows:

$$\begin{array}{cl} Z^H B Z = I & \text{if 1st or 2nd form, or} \\ Z^H B^{-1} Z = I & \text{if 3rd form.} \end{array}$$

Parameters

- `[in] handle`: rocblas_handle.
- `[in] itype`: *rocblas_eform*. Specifies the form of the generalized eigenproblem.
- `[in] evect`: *rocblas_evect*. Specifies whether the eigenvectors are to be computed. If evect is rocblas_evect_original, then the eigenvectors are computed. rocblas_evect_tridiagonal is not supported.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower parts of the matrices A and B are stored. If uplo indicates lower (or upper), then the upper (or lower) parts of A and B are not used.
- `[in] n`: rocblas_int. n >= 0. The matrix dimensions.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the hermitian matrix A. On exit, if evect is original, the normalized matrix Z of eigenvectors. If evect is none, then the upper or lower triangular part of the matrix A (including the diagonal) is destroyed, depending on the value of uplo.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of A.
- `[out] B`: pointer to type. Array on the GPU of dimension ldb*n. On entry, the hermitian positive definite matrix B. On exit, the triangular factor of B as returned by *POTRF*.
- `[in] ldb`: rocblas_int. ldb >= n. Specifies the leading dimension of B.
- `[out] D`: pointer to real type. Array on the GPU of dimension n. On exit, the eigenvalues in increasing order.
- `[out] E`: pointer to real type. Array on the GPU of dimension n. This array is used to work internally with the tridiagonal matrix T associated with the reduced eigenvalue problem. On exit, if 0 < info <= n, it contains the unconverged off-diagonal elements of T (or properly speaking, a tridiagonal matrix equivalent to T). The diagonal elements of this matrix are in D; those that converged correspond to a subset of the eigenvalues (not necessarily ordered).
- `[out] info`: pointer to a rocblas_int on the GPU. If info = 0, successful exit. If info = j <= n, j off-diagonal elements of an intermediate tridiagonal form did not converge to zero. If info = n + j, the leading minor of order j of B is not positive definite.

rocsolver_<type>hegv_batched()

rocblas_status **rocsolver_zhegv_batched**(rocblas_handle *handle*, **const** *rocblas_eform* *itype*, **const** *rocblas_evect* *evect*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_double_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_double_complex ***const** *B*[], **const** rocblas_int *ldb*, double **D*, **const** rocblas_stride *strideD*, double **E*, **const** rocblas_stride *strideE*, rocblas_int **info*, **const** rocblas_int *batch_count*) rocblas_status **rocsolver_chegv_batched** (rocblas_handle *handle*, **const** *rocblas_eform* *itype*, **const** *rocblas_evect* *evect*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_float_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_float_complex ***const** *B*[], **const** rocblas_int *ldb*, float **D*, **const** rocblas_stride *strideD*, float **E*, **const** rocblas_stride *strideE*, rocblas_int **info*, **const** rocblas_int *batch_count*)

HEGV_BATCHED computes the eigenvalues and (optionally) eigenvectors of a batch of complex generalized hermitian-definite eigenproblems.

For each instance in the batch, the problem solved by this function is either of the form

$$\begin{array}{ll} A_iX_i = \lambda B_iX_i & \text{1st form,} \\ A_iB_iX_i = \lambda X_i & \text{2nd form, or} \\ B_iA_iX_i = \lambda X_i & \text{3rd form,} \end{array}$$

depending on the value of itype. The eigenvectors are computed depending on the value of evect.

When computed, the matrix Z_i of eigenvectors is normalized as follows:

$$\begin{array}{ll} Z_i^H B_i Z_i = I & \text{if 1st or 2nd form, or} \\ Z_i^H B_i^{-1} Z_i = I & \text{if 3rd form.} \end{array}$$

Parameters

- `[in] handle`: rocblas_handle.
- `[in] itype`: *rocblas_eform*. Specifies the form of the generalized eigenproblems.
- `[in] evect`: *rocblas_evect*. Specifies whether the eigenvectors are to be computed. If evect is rocblas_evect_original, then the eigenvectors are computed. rocblas_evect_tridiagonal is not supported.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower parts of the matrices A_i and B_i are stored. If uplo indicates lower (or upper), then the upper (or lower) parts of A_i and B_i are not used.
- `[in] n`: rocblas_int. n >= 0. The matrix dimensions.
- `[inout] A`: array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n. On entry, the hermitian matrices A_i. On exit, if evect is original, the normalized matrix Z_i of eigenvectors. If evect is none, then the upper or lower triangular part of the matrices A_i (including the diagonal) are destroyed, depending on the value of uplo.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of A_i.
- `[out] B`: array of pointers to type. Each pointer points to an array on the GPU of dimension ldb*n. On entry, the hermitian positive definite matrices B_i. On exit, the triangular factor of B_i as returned by *POTRF_BATCHED*.
- `[in] ldb`: rocblas_int. ldb >= n. Specifies the leading dimension of B_i.
- `[out] D`: pointer to real type. Array on the GPU (the size depends on the value of strideD). On exit, the eigenvalues in increasing order.
- `[in] strideD`: rocblas_stride. Stride from the start of one vector D_i to the next one D_(i+1). There is no restriction for the value of strideD. Normal use is strideD >= n.

- `[out] E`: pointer to real type. Array on the GPU (the size depends on the value of strideE). This array is used to work internally with the tridiagonal matrix T_i associated with the ith reduced eigenvalue problem. On exit, if 0 < info[i] <= n, it contains the unconverged off-diagonal elements of T_i (or properly speaking, a tridiagonal matrix equivalent to T_i). The diagonal elements of this matrix are in D_i; those that converged correspond to a subset of the eigenvalues (not necessarily ordered).
- `[in] strideE`: rocblas_stride. Stride from the start of one vector E_i to the next one E_(i+1). There is no restriction for the value of strideE. Normal use is strideE >= n.
- `[out] info`: pointer to rocblas_int. Array of batch_count integers on the GPU. If info[i] = 0, successful exit of batch i. If info[i] = j <= n, j off-diagonal elements of an intermediate tridiagonal form did not converge to zero. If info[i] = n + j, the leading minor of order j of B_i is not positive definite.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>hegv_strided_batched()

rocblas_status **rocsolver_zhegv_strided_batched**(rocblas_handle *handle*, **const** *rocblas_eform itype*, **const** *rocblas_evect evect*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_double_complex **B*, **const** rocblas_int *ldb*, **const** rocblas_stride *strideB*, double **D*, **const** rocblas_stride *strideD*, double **E*, **const** rocblas_stride *strideE*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_chegv_strided_batched**(rocblas_handle *handle*, **const** *rocblas_eform itype*, **const** *rocblas_evect evect*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_float_complex **B*, **const** rocblas_int *ldb*, **const** rocblas_stride *strideB*, float **D*, **const** rocblas_stride *strideD*, float **E*, **const** rocblas_stride *strideE*, rocblas_int **info*, **const** rocblas_int *batch_count*)

HEGV_STRIDED_BATCHED computes the eigenvalues and (optionally) eigenvectors of a batch of complex generalized hermitian-definite eigenproblems.

For each instance in the batch, the problem solved by this function is either of the form

$$\begin{array}{ll} A_i X_i = \lambda B_i X_i & \text{1st form,} \\ A_i B_i X_i = \lambda X_i & \text{2nd form, or} \\ B_i A_i X_i = \lambda X_i & \text{3rd form,} \end{array}$$

depending on the value of itype. The eigenvectors are computed depending on the value of evect.

When computed, the matrix Z_i of eigenvectors is normalized as follows:

$$\begin{array}{ll} Z_i^H B_i Z_i = I & \text{if 1st or 2nd form, or} \\ Z_i^H B_i^{-1} Z_i = I & \text{if 3rd form.} \end{array}$$

Parameters

- `[in] handle`: rocblas_handle.
- `[in] itype`: *rocblas_eform*. Specifies the form of the generalized eigenproblems.
- `[in] evect`: *rocblas_evect*. Specifies whether the eigenvectors are to be computed. If evect is rocblas_evect_original, then the eigenvectors are computed. rocblas_evect_tridiagonal is not supported.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower parts of the matrices A_i and B_i are stored. If uplo indicates lower (or upper), then the upper (or lower) parts of A_i and B_i are not used.
- `[in] n`: rocblas_int. n >= 0. The matrix dimensions.
- `[inout] A`: pointer to type. Array on the GPU (the size depends on the value of strideA). On entry, the hermitian matrices A_i. On exit, if evect is original, the normalized matrix Z_i of eigenvectors. If evect is none, then the upper or lower triangular part of the matrices A_i (including the diagonal) are destroyed, depending on the value of uplo.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of A_i.
- `[in] strideA`: rocblas_stride. Stride from the start of one matrix A_i to the next one A_(i+1). There is no restriction for the value of strideA. Normal use is strideA >= lda*n.
- `[out] B`: pointer to type. Array on the GPU (the size depends on the value of strideB). On entry, the hermitian positive definite matrices B_i. On exit, the triangular factor of B_i as returned by *POTRF_STRIDED_BATCHED*.
- `[in] ldb`: rocblas_int. ldb >= n. Specifies the leading dimension of B_i.
- `[in] strideB`: rocblas_stride. Stride from the start of one matrix B_i to the next one B_(i+1). There is no restriction for the value of strideB. Normal use is strideB >= ldb*n.
- `[out] D`: pointer to real type. Array on the GPU (the size depends on the value of strideD). On exit, the eigenvalues in increasing order.
- `[in] strideD`: rocblas_stride. Stride from the start of one vector D_i to the next one D_(i+1). There is no restriction for the value of strideD. Normal use is strideD >= n.
- `[out] E`: pointer to real type. Array on the GPU (the size depends on the value of strideE). This array is used to work internally with the tridiagonal matrix T_i associated with the ith reduced eigenvalue problem. On exit, if 0 < info[i] <= n, it contains the unconverged off-diagonal elements of T_i (or properly speaking, a tridiagonal matrix equivalent to T_i). The diagonal elements of this matrix are in D_i; those that converged correspond to a subset of the eigenvalues (not necessarily ordered).
- `[in] strideE`: rocblas_stride. Stride from the start of one vector E_i to the next one E_(i+1). There is no restriction for the value of strideE. Normal use is strideE >= n.
- `[out] info`: pointer to rocblas_int. Array of batch_count integers on the GPU. If info[i] = 0, successful exit of batch i. If info[i] = j <= n, j off-diagonal elements of an intermediate tridiagonal form did not converge to zero. If info[i] = n + j, the leading minor of order j of B_i is not positive definite.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>sygvd()

rocblas_status **rocsolver_dsygvd**(rocblas_handle *handle*, **const** *rocblas_eform* *itype*, **const** *rocblas_evect* *evect*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, double **B*, **const** rocblas_int *ldb*, double **D*, double **E*, rocblas_int **info*)

rocblas_status **rocsolver_ssygvd**(rocblas_handle *handle*, **const** *rocblas_eform* *itype*, **const** *rocblas_evect* *evect*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, float **B*, **const** rocblas_int *ldb*, float **D*, float **E*, rocblas_int **info*)

SYGVD computes the eigenvalues and (optionally) eigenvectors of a real generalized symmetric-definite eigenproblem.

The problem solved by this function is either of the form

$$\begin{array}{ll} AX = \lambda BX & \text{1st form,} \\ ABX = \lambda X & \text{2nd form, or} \\ BAX = \lambda X & \text{3rd form,} \end{array}$$

depending on the value of itype. The eigenvectors are computed using a divide-and-conquer algorithm, depending on the value of evect.

When computed, the matrix Z of eigenvectors is normalized as follows:

$$\begin{array}{ll} Z^T BZ = I & \text{if 1st or 2nd form, or} \\ Z^T B^{-1} Z = I & \text{if 3rd form.} \end{array}$$

Parameters

- `[in] handle`: rocblas_handle.
- `[in] itype`: *rocblas_eform*. Specifies the form of the generalized eigenproblem.
- `[in] evect`: *rocblas_evect*. Specifies whether the eigenvectors are to be computed. If evect is rocblas_evect_original, then the eigenvectors are computed. rocblas_evect_tridiagonal is not supported.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower parts of the matrices A and B are stored. If uplo indicates lower (or upper), then the upper (or lower) parts of A and B are not used.
- `[in] n`: rocblas_int. n >= 0. The matrix dimensions.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the symmetric matrix A. On exit, if evect is original, the normalized matrix Z of eigenvectors. If evect is none, then the upper or lower triangular part of the matrix A (including the diagonal) is destroyed, depending on the value of uplo.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of A.
- `[out] B`: pointer to type. Array on the GPU of dimension ldb*n. On entry, the symmetric positive definite matrix B. On exit, the triangular factor of B as returned by *POTRF*.
- `[in] ldb`: rocblas_int. ldb >= n. Specifies the leading dimension of B.
- `[out] D`: pointer to type. Array on the GPU of dimension n. On exit, the eigenvalues in increasing order.

- `[out] E`: pointer to type. Array on the GPU of dimension n. This array is used to work internally with the tridiagonal matrix T associated with the reduced eigenvalue problem. On exit, if 0 < info <= n, it contains the unconverged off-diagonal elements of T (or properly speaking, a tridiagonal matrix equivalent to T). The diagonal elements of this matrix are in D; those that converged correspond to a subset of the eigenvalues (not necessarily ordered).
- `[out] info`: pointer to a rocblas_int on the GPU. If info = 0, successful exit. If info = j <= n and evect is rocblas_evect_none, j off-diagonal elements of an intermediate tridiagonal form did not converge to zero. If info = j <= n and evect is rocblas_evect_original, the algorithm failed to compute an eigenvalue in the submatrix from [j/(n+1), j/(n+1)] to [j%(n+1), j%(n+1)]. If info = n + j, the leading minor of order j of B is not positive definite.

rocsolver_<type>sygvd_batched()

rocblas_status **rocsolver_dsygvd_batched**(rocblas_handle *handle*, **const** *rocblas_eform* *itype*, **const** *rocblas_evect* *evect*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, double ***const** *A*[], **const** rocblas_int *lda*, double ***const** *B*[], **const** rocblas_int *ldb*, double **D*, **const** rocblas_stride *strideD*, double **E*, **const** rocblas_stride *strideE*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_ssygvd_batched**(rocblas_handle *handle*, **const** *rocblas_eform* *itype*, **const** *rocblas_evect* *evect*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, float ***const** *A*[], **const** rocblas_int *lda*, float ***const** *B*[], **const** rocblas_int *ldb*, float **D*, **const** rocblas_stride *strideD*, float **E*, **const** rocblas_stride *strideE*, rocblas_int **info*, **const** rocblas_int *batch_count*)

SYGVD_BATCHED computes the eigenvalues and (optionally) eigenvectors of a batch of real generalized symmetric-definite eigenproblems.

For each instance in the batch, the problem solved by this function is either of the form

$$\begin{array}{ll} A_i X_i = \lambda B_i X_i & \text{1st form,} \\ A_i B_i X_i = \lambda X_i & \text{2nd form, or} \\ B_i A_i X_i = \lambda X_i & \text{3rd form,} \end{array}$$

depending on the value of itype. The eigenvectors are computed using a divide-and-conquer algorithm, depending on the value of evect.

When computed, the matrix Z_i of eigenvectors is normalized as follows:

$$\begin{array}{ll} Z_i^T B_i Z_i = I & \text{if 1st or 2nd form, or} \\ Z_i^T B_i^{-1} Z_i = I & \text{if 3rd form.} \end{array}$$

Parameters

- `[in] handle`: rocblas_handle.
- `[in] itype`: *rocblas_eform*. Specifies the form of the generalized eigenproblems.

- `[in] evect`: *rocblas_evect*. Specifies whether the eigenvectors are to be computed. If evect is rocblas_evect_original, then the eigenvectors are computed. rocblas_evect_tridiagonal is not supported.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower parts of the matrices A_i and B_i are stored. If uplo indicates lower (or upper), then the upper (or lower) parts of A_i and B_i are not used.
- `[in] n`: rocblas_int. n >= 0. The matrix dimensions.
- `[inout] A`: array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n. On entry, the symmetric matrices A_i. On exit, if evect is original, the normalized matrix Z_i of eigenvectors. If evect is none, then the upper or lower triangular part of the matrices A_i (including the diagonal) are destroyed, depending on the value of uplo.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of A_i.
- `[out] B`: array of pointers to type. Each pointer points to an array on the GPU of dimension ldb*n. On entry, the symmetric positive definite matrices B_i. On exit, the triangular factor of B_i as returned by *POTRF_BATCHED*.
- `[in] ldb`: rocblas_int. ldb >= n. Specifies the leading dimension of B_i.
- `[out] D`: pointer to type. Array on the GPU (the size depends on the value of strideD). On exit, the eigenvalues in increasing order.
- `[in] strideD`: rocblas_stride. Stride from the start of one vector D_i to the next one D_(i+1). There is no restriction for the value of strideD. Normal use is strideD >= n.
- `[out] E`: pointer to type. Array on the GPU (the size depends on the value of strideE). This array is used to work internally with the tridiagonal matrix T_i associated with the ith reduced eigenvalue problem. On exit, if 0 < info[i] <= n, it contains the unconverged off-diagonal elements of T_i (or properly speaking, a tridiagonal matrix equivalent to T_i). The diagonal elements of this matrix are in D_i; those that converged correspond to a subset of the eigenvalues (not necessarily ordered).
- `[in] strideE`: rocblas_stride. Stride from the start of one vector E_i to the next one E_(i+1). There is no restriction for the value of strideE. Normal use is strideE >= n.
- `[out] info`: pointer to rocblas_int. Array of batch_count integers on the GPU. If info[i] = 0, successful exit of batch i. If info[i] = j <= n and evect is rocblas_evect_none, j off-diagonal elements of an intermediate tridiagonal form did not converge to zero. If info[i] = j <= n and evect is rocblas_evect_original, the algorithm failed to compute an eigenvalue in the submatrix from [j/(n+1), j/(n+1)] to [j%(n+1), j%(n+1)]. If info[i] = n + j, the leading minor of order j of B_i is not positive definite.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>sygvd_strided_batched()

rocblas_status **rocsolver_dsygvd_strided_batched**(rocblas_handle *handle*, **const** *rocblas_eform* *itype*, **const** *rocblas_evect* *evect*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, double **B*, **const** rocblas_int *ldb*, **const** rocblas_stride *strideB*, double **D*, **const** rocblas_stride *strideD*, double **E*, **const** rocblas_stride *strideE*, rocblas_int **info*, **const** rocblas_int *batch_count*) rocblas_status **rocsolver_ssygvd_strided_batched**(rocblas_handle *handle*, **const** *rocblas_eform* *itype*, **const** *rocblas_evect* *evect*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, float **B*, **const** rocblas_int *ldb*, **const** rocblas_stride *strideB*, float **D*, **const** rocblas_stride *strideD*, float **E*, **const** rocblas_stride *strideE*, rocblas_int **info*, **const** rocblas_int *batch_count*)

SYGVD_STRIDED_BATCHED computes the eigenvalues and (optionally) eigenvectors of a batch of real generalized symmetric-definite eigenproblems.

For each instance in the batch, the problem solved by this function is either of the form

$$\begin{array}{ll} A_i X_i = \lambda B_i X_i & \text{1st form,} \\ A_i B_i X_i = \lambda X_i & \text{2nd form, or} \\ B_i A_i X_i = \lambda X_i & \text{3rd form,} \end{array}$$

depending on the value of itype. The eigenvectors are computed using a divide-and-conquer algorithm, depending on the value of evect.

When computed, the matrix Z_i of eigenvectors is normalized as follows:

$$\begin{array}{ll} Z_i^T B_i Z_i = I & \text{if 1st or 2nd form, or} \\ Z_i^T B_i^{-1} Z_i = I & \text{if 3rd form.} \end{array}$$

Parameters

- `[in] handle`: rocblas_handle.
- `[in] itype`: *rocblas_eform*. Specifies the form of the generalized eigenproblems.
- `[in] evect`: *rocblas_evect*. Specifies whether the eigenvectors are to be computed. If evect is rocblas_evect_original, then the eigenvectors are computed. rocblas_evect_tridiagonal is not supported.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower parts of the matrices A_i and B_i are stored. If uplo indicates lower (or upper), then the upper (or lower) parts of A_i and B_i are not used.
- `[in] n`: rocblas_int. n >= 0. The matrix dimensions.
- `[inout] A`: pointer to type. Array on the GPU (the size depends on the value of strideA). On entry, the symmetric matrices A_i. On exit, if evect is original, the normalized matrix Z_i of eigenvectors. If evect is none, then the upper or lower triangular part of the matrices A_i (including the diagonal) are destroyed, depending on the value of uplo.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of A_i.
- `[in] strideA`: rocblas_stride. Stride from the start of one matrix A_i to the next one A_(i+1). There is no restriction for the value of strideA. Normal use is strideA >= lda*n.
- `[out] B`: pointer to type. Array on the GPU (the size depends on the value of strideB). On entry, the symmetric positive definite matrices B_i. On exit, the triangular factor of B_i as returned by *POTRF_STRIDED_BATCHED*.
- `[in] ldb`: rocblas_int. ldb >= n. Specifies the leading dimension of B_i.

- `[in] strideB`: rocblas_stride. Stride from the start of one matrix B_i to the next one B_(i+1). There is no restriction for the value of strideB. Normal use is strideB >= ldb*n.
- `[out] D`: pointer to type. Array on the GPU (the size depends on the value of strideD). On exit, the eigenvalues in increasing order.
- `[in] strideD`: rocblas_stride. Stride from the start of one vector D_i to the next one D_(i+1). There is no restriction for the value of strideD. Normal use is strideD >= n.
- `[out] E`: pointer to type. Array on the GPU (the size depends on the value of strideE). This array is used to work internally with the tridiagonal matrix T_i associated with the ith reduced eigenvalue problem. On exit, if 0 < info[i] <= n, it contains the unconverged off-diagonal elements of T_i (or properly speaking, a tridiagonal matrix equivalent to T_i). The diagonal elements of this matrix are in D_i; those that converged correspond to a subset of the eigenvalues (not necessarily ordered).
- `[in] strideE`: rocblas_stride. Stride from the start of one vector E_i to the next one E_(i+1). There is no restriction for the value of strideE. Normal use is strideE >= n.
- `[out] info`: pointer to rocblas_int. Array of batch_count integers on the GPU. If info[i] = 0, successful exit of batch i. If info[i] = j <= n and evect is rocblas_evect_none, j off-diagonal elements of an intermediate tridiagonal form did not converge to zero. If info[i] = j <= n and evect is rocblas_evect_original, the algorithm failed to compute an eigenvalue in the submatrix from [j/(n+1), j/(n+1)] to [j%(n+1), j%(n+1)]. If info[i] = n + j, the leading minor of order j of B_i is not positive definite.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>hegvd()

rocblas_status **rocsolver_zhegvd**(rocblas_handle *handle*, **const** *rocblas_eform* *itype*, **const** *rocblas_evect* *evect*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, rocblas_double_complex **B*, **const** rocblas_int *ldb*, double **D*, double **E*, rocblas_int **info*)

rocblas_status **rocsolver_chegvd**(rocblas_handle *handle*, **const** *rocblas_eform* *itype*, **const** *rocblas_evect* *evect*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, rocblas_float_complex **B*, **const** rocblas_int *ldb*, float **D*, float **E*, rocblas_int **info*)

HEGVD computes the eigenvalues and (optionally) eigenvectors of a complex generalized hermitian-definite eigenproblem.

The problem solved by this function is either of the form

$$\begin{array}{ll} AX = \lambda BX & \text{1st form,} \\ ABX = \lambda X & \text{2nd form, or} \\ BAX = \lambda X & \text{3rd form,} \end{array}$$

depending on the value of itype. The eigenvectors are computed using a divide-and-conquer algorithm, depending on the value of evect.

When computed, the matrix Z of eigenvectors is normalized as follows:

$$\begin{array}{ll} Z^H B Z = I & \text{if 1st or 2nd form, or} \\ Z^H B^{-1} Z = I & \text{if 3rd form.} \end{array}$$

Parameters

- `[in] handle`: rocblas_handle.
- `[in] itype`: *rocblas_eform*. Specifies the form of the generalized eigenproblem.
- `[in] evect`: *rocblas_evect*. Specifies whether the eigenvectors are to be computed. If evect is rocblas_evect_original, then the eigenvectors are computed. rocblas_evect_tridiagonal is not supported.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower parts of the matrices A and B are stored. If uplo indicates lower (or upper), then the upper (or lower) parts of A and B are not used.
- `[in] n`: rocblas_int. n >= 0. The matrix dimensions.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the hermitian matrix A. On exit, if evect is original, the normalized matrix Z of eigenvectors. If evect is none, then the upper or lower triangular part of the matrix A (including the diagonal) is destroyed, depending on the value of uplo.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of A.
- `[out] B`: pointer to type. Array on the GPU of dimension ldb*n. On entry, the hermitian positive definite matrix B. On exit, the triangular factor of B as returned by *POTRF*.
- `[in] ldb`: rocblas_int. ldb >= n. Specifies the leading dimension of B.
- `[out] D`: pointer to real type. Array on the GPU of dimension n. On exit, the eigenvalues in increasing order.
- `[out] E`: pointer to real type. Array on the GPU of dimension n. This array is used to work internally with the tridiagonal matrix T associated with the reduced eigenvalue problem. On exit, if 0 < info <= n, it contains the unconverged off-diagonal elements of T (or properly speaking, a tridiagonal matrix equivalent to T). The diagonal elements of this matrix are in D; those that converged correspond to a subset of the eigenvalues (not necessarily ordered).
- `[out] info`: pointer to a rocblas_int on the GPU. If info = 0, successful exit. If info = j <= n and evect is rocblas_evect_none, j off-diagonal elements of an intermediate tridiagonal form did not converge to zero. If info = j <= n and evect is rocblas_evect_original, the algorithm failed to compute an eigenvalue in the submatrix from [j/(n+1), j/(n+1)] to [j%(n+1), j%(n+1)]. If info = n + j, the leading minor of order j of B is not positive definite.

rocsolver_<type>hegvd_batched()

rocblas_status **rocsolver_zhegvd_batched**(rocblas_handle *handle*, **const** *rocblas_eform* *itype*, **const** *rocblas_evect* *evect*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_double_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_double_complex ***const** *B*[], **const** rocblas_int *ldb*, double **D*, **const** rocblas_stride *strideD*, double **E*, **const** rocblas_stride *strideE*, rocblas_int **info*, **const** rocblas_int *batch_count*) rocblas_status **rocsolver_chegvd_batched**(rocblas_handle *handle*, **const** *rocblas_eform* *itype*, **const** *rocblas_evect* *evect*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_float_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_float_complex ***const** *B*[], **const** rocblas_int *ldb*, float **D*, **const** rocblas_stride *strideD*, float **E*, **const** rocblas_stride *strideE*, rocblas_int **info*, **const** rocblas_int *batch_count*)

HEGVD_BATCHED computes the eigenvalues and (optionally) eigenvectors of a batch of complex generalized hermitian-definite eigenproblems.

For each instance in the batch, the problem solved by this function is either of the form

$$\begin{array}{rl} A_i X_i = \lambda B_i X_i & \text{1st form,} \\ A_i B_i X_i = \lambda X_i & \text{2nd form, or} \\ B_i A_i X_i = \lambda X_i & \text{3rd form,} \end{array}$$

depending on the value of itype. The eigenvectors are computed using a divide-and-conquer algorithm, depending on the value of evect.

When computed, the matrix Z_i of eigenvectors is normalized as follows:

$$\begin{array}{rl} Z_i^H B_i Z_i = I & \text{if 1st or 2nd form, or} \\ Z_i^H B_i^{-1} Z_i = I & \text{if 3rd form.} \end{array}$$

Parameters

- `[in] handle`: rocblas_handle.
- `[in] itype`: *rocblas_eform*. Specifies the form of the generalized eigenproblems.
- `[in] evect`: *rocblas_evect*. Specifies whether the eigenvectors are to be computed. If evect is rocblas_evect_original, then the eigenvectors are computed. rocblas_evect_tridiagonal is not supported.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower parts of the matrices A_i and B_i are stored. If uplo indicates lower (or upper), then the upper (or lower) parts of A_i and B_i are not used.
- `[in] n`: rocblas_int. n >= 0. The matrix dimensions.
- `[inout] A`: array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n. On entry, the hermitian matrices A_i. On exit, if evect is original, the normalized matrix Z_i of eigenvectors. If evect is none, then the upper or lower triangular part of the matrices A_i (including the diagonal) are destroyed, depending on the value of uplo.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of A_i.
- `[out] B`: array of pointers to type. Each pointer points to an array on the GPU of dimension ldb*n. On entry, the hermitian positive definite matrices B_i. On exit, the triangular factor of B_i as returned by *POTRF_BATCHED*.
- `[in] ldb`: rocblas_int. ldb >= n. Specifies the leading dimension of B_i.
- `[out] D`: pointer to real type. Array on the GPU (the size depends on the value of strideD). On exit, the eigenvalues in increasing order.

- `[in] strideD`: rocblas_stride. Stride from the start of one vector D_i to the next one D_(i+1). There is no restriction for the value of strideD. Normal use is strideD >= n.
- `[out] E`: pointer to real type. Array on the GPU (the size depends on the value of strideE). This array is used to work internally with the tridiagonal matrix T_i associated with the ith reduced eigenvalue problem. On exit, if 0 < info[i] <= n, it contains the unconverged off-diagonal elements of T_i (or properly speaking, a tridiagonal matrix equivalent to T_i). The diagonal elements of this matrix are in D_i; those that converged correspond to a subset of the eigenvalues (not necessarily ordered).
- `[in] strideE`: rocblas_stride. Stride from the start of one vector E_i to the next one E_(i+1). There is no restriction for the value of strideE. Normal use is strideE >= n.
- `[out] info`: pointer to rocblas_int. Array of batch_count integers on the GPU. If info[i] = 0, successful exit of batch i. If info[i] = j <= n and evect is rocblas_evect_none, j off-diagonal elements of an intermediate tridiagonal form did not converge to zero. If info[i] = j <= n and evect is rocblas_evect_original, the algorithm failed to compute an eigenvalue in the submatrix from [j/(n+1), j/(n+1)] to [j%(n+1), j%(n+1)]. If info[i] = n + j, the leading minor of order j of B_i is not positive definite.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>hegvd_strided_batched()

rocblas_status **rocsolver_zhegvd_strided_batched**(rocblas_handle *handle*, **const** *rocblas_eform* *itype*, **const** *rocblas_evect* *evect*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_double_complex **B*, **const** rocblas_int *ldb*, **const** rocblas_stride *strideB*, double **D*, **const** rocblas_stride *strideD*, double **E*, **const** rocblas_stride *strideE*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_chegvd_strided_batched**(rocblas_handle *handle*, **const** *rocblas_eform* *itype*, **const** *rocblas_evect* *evect*, **const** rocblas_fill *uplo*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_float_complex **B*, **const** rocblas_int *ldb*, **const** rocblas_stride *strideB*, float **D*, **const** rocblas_stride *strideD*, float **E*, **const** rocblas_stride *strideE*, rocblas_int **info*, **const** rocblas_int *batch_count*)

HEGVD_STRIDED_BATCHED computes the eigenvalues and (optionally) eigenvectors of a batch of complex generalized hermitian-definite eigenproblems.

For each instance in the batch, the problem solved by this function is either of the form

$$\begin{array}{ll} A_iX_i = \lambda B_iX_i & \text{1st form,} \\ A_iB_iX_i = \lambda X_i & \text{2nd form, or} \\ B_iA_iX_i = \lambda X_i & \text{3rd form,} \end{array}$$

depending on the value of itype. The eigenvectors are computed using a divide-and-conquer algorithm, depending on the value of evect.

When computed, the matrix Z_i of eigenvectors is normalized as follows:

$$\begin{array}{cl} Z_i^H B_i Z_i = I & \text{if 1st or 2nd form, or} \\ Z_i^H B_i^{-1} Z_i = I & \text{if 3rd form.} \end{array}$$

Parameters

- `[in] handle`: rocblas_handle.
- `[in] itype`: *rocblas_eform*. Specifies the form of the generalized eigenproblems.
- `[in] evect`: *rocblas_evect*. Specifies whether the eigenvectors are to be computed. If evect is rocblas_evect_original, then the eigenvectors are computed. rocblas_evect_tridiagonal is not supported.
- `[in] uplo`: rocblas_fill. Specifies whether the upper or lower parts of the matrices A_i and B_i are stored. If uplo indicates lower (or upper), then the upper (or lower) parts of A_i and B_i are not used.
- `[in] n`: rocblas_int. n >= 0. The matrix dimensions.
- `[inout] A`: pointer to type. Array on the GPU (the size depends on the value of strideA). On entry, the hermitian matrices A_i. On exit, if evect is original, the normalized matrix Z_i of eigenvectors. If evect is none, then the upper or lower triangular part of the matrices A_i (including the diagonal) are destroyed, depending on the value of uplo.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of A_i.
- `[in] strideA`: rocblas_stride. Stride from the start of one matrix A_i to the next one A_(i+1). There is no restriction for the value of strideA. Normal use is strideA >= lda*n.
- `[out] B`: pointer to type. Array on the GPU (the size depends on the value of strideB). On entry, the hermitian positive definite matrices B_i. On exit, the triangular factor of B_i as returned by *POTRF_STRIDED_BATCHED*.
- `[in] ldb`: rocblas_int. ldb >= n. Specifies the leading dimension of B_i.
- `[in] strideB`: rocblas_stride. Stride from the start of one matrix B_i to the next one B_(i+1). There is no restriction for the value of strideB. Normal use is strideB >= ldb*n.
- `[out] D`: pointer to real type. Array on the GPU (the size depends on the value of strideD). On exit, the eigenvalues in increasing order.
- `[in] strideD`: rocblas_stride. Stride from the start of one vector D_i to the next one D_(i+1). There is no restriction for the value of strideD. Normal use is strideD >= n.
- `[out] E`: pointer to real type. Array on the GPU (the size depends on the value of strideE). This array is used to work internally with the tridiagonal matrix T_i associated with the ith reduced eigenvalue problem. On exit, if 0 < info[i] <= n, it contains the unconverged off-diagonal elements of T_i (or properly speaking, a tridiagonal matrix equivalent to T_i). The diagonal elements of this matrix are in D_i; those that converged correspond to a subset of the eigenvalues (not necessarily ordered).
- `[in] strideE`: rocblas_stride. Stride from the start of one vector E_i to the next one E_(i+1). There is no restriction for the value of strideE. Normal use is strideE >= n.
- `[out] info`: pointer to rocblas_int. Array of batch_count integers on the GPU. If info[i] = 0, successful exit of batch i. If info[i] = j <= n and evect is rocblas_evect_none, j off-diagonal elements of an intermediate tridiagonal form did not converge to zero. If info[i] = j <= n and evect is rocblas_evect_original, the algorithm failed to compute an eigenvalue in the submatrix from [j/(n+1), j/(n+1)] to [j%(n+1), j%(n+1)]. If info[i] = n + j, the leading minor of order j of B_i is not positive definite.

- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

3.3.7 Singular value decomposition

List of SVD related functions

- *rocsolver_<type>gesvd()*
- *rocsolver_<type>gesvd_batched()*
- *rocsolver_<type>gesvd_strided_batched()*

rocsolver_<type>gesvd()

rocblas_status **rocsolver_zgesvd**(rocblas_handle *handle*, **const** *rocblas_svect* *left_svect*, **const** *rocblas_svect* *right_svect*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, double **S*, rocblas_double_complex **U*, **const** rocblas_int *ldu*, rocblas_double_complex **V*, **const** rocblas_int *ldv*, double **E*, **const** *rocblas_workmode* *fast_alg*, rocblas_int **info*)

rocblas_status **rocsolver_cgesvd**(rocblas_handle *handle*, **const** *rocblas_svect* *left_svect*, **const** *rocblas_svect* *right_svect*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, float **S*, rocblas_float_complex **U*, **const** rocblas_int *ldu*, rocblas_float_complex **V*, **const** rocblas_int *ldv*, float **E*, **const** *rocblas_workmode* *fast_alg*, rocblas_int **info*)

rocblas_status **rocsolver_dgesvd**(rocblas_handle *handle*, **const** *rocblas_svect* *left_svect*, **const** *rocblas_svect* *right_svect*, **const** rocblas_int *m*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, double **S*, double **U*, **const** rocblas_int *ldu*, double **V*, **const** rocblas_int *ldv*, double **E*, **const** *rocblas_workmode* *fast_alg*, rocblas_int **info*)

rocblas_status **rocsolver_sgesvd**(rocblas_handle *handle*, **const** *rocblas_svect* *left_svect*, **const** *rocblas_svect* *right_svect*, **const** rocblas_int *m*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, float **S*, float **U*, **const** rocblas_int *ldu*, float **V*, **const** rocblas_int *ldv*, float **E*, **const** *rocblas_workmode* *fast_alg*, rocblas_int **info*)

GESVD computes the singular values and optionally the singular vectors of a general m-by-n matrix A (Singular Value Decomposition).

The SVD of matrix A is given by:

$$A = USV'$$

where the m-by-n matrix S is zero except, possibly, for its min(m,n) diagonal elements, which are the singular values of A. U and V are orthogonal (unitary) matrices. The first min(m,n) columns of U and V are the left and right singular vectors of A, respectively.

The computation of the singular vectors is optional and it is controlled by the function arguments left_svect and right_svect as described below. When computed, this function returns the transpose (or transpose conjugate) of the right singular vectors, i.e. the rows of V'.

left_svect and right_svect are *rocblas_svect* enums that can take the following values:

- rocblas_svect_all: the entire matrix U (or V') is computed,
- rocblas_svect_singular: only the singular vectors (first min(m,n) columns of U or rows of V') are computed,
- rocblas_svect_overwrite: the first columns (or rows) of A are overwritten with the singular vectors, or
- rocblas_svect_none: no columns (or rows) of U (or V') are computed, i.e. no singular vectors.

left_svect and right_svect cannot both be set to overwrite. When neither is set to overwrite, the contents of A are destroyed by the time the function returns.

Note When m >> n (or n >> m) the algorithm could be sped up by compressing the matrix A via a QR (or LQ) factorization, and working with the triangular factor afterwards (thin-SVD). If the singular vectors are also requested, its computation could be sped up as well via executing some intermediate operations out-of-place, and relying more on matrix multiplications (GEMMs); this will require, however, a larger memory workspace. The parameter fast_alg controls whether the fast algorithm is executed or not. For more details, see the "Tuning rocSOLVER performance" and "Memory model" sections of the documentation.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] left_svect`: *rocblas_svect*. Specifies how the left singular vectors are computed.
- `[in] right_svect`: *rocblas_svect*. Specifies how the right singular vectors are computed.
- `[in] m`: rocblas_int. m >= 0. The number of rows of matrix A.
- `[in] n`: rocblas_int. n >= 0. The number of columns of matrix A.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the matrix A. On exit, if left_svect (or right_svect) is equal to overwrite, the first columns (or rows) contain the left (or right) singular vectors; otherwise, the contents of A are destroyed.
- `[in] lda`: rocblas_int. lda >= m. The leading dimension of A.
- `[out] S`: pointer to real type. Array on the GPU of dimension min(m,n). The singular values of A in decreasing order.
- `[out] U`: pointer to type. Array on the GPU of dimension ldu*min(m,n) if left_svect is set to singular, or ldu*m when left_svect is equal to all. The matrix of left singular vectors stored as columns. Not referenced if left_svect is set to overwrite or none.
- `[in] ldu`: rocblas_int. ldu >= m if left_svect is all or singular; ldu >= 1 otherwise. The leading dimension of U.
- `[out] V`: pointer to type. Array on the GPU of dimension ldv*n. The matrix of right singular vectors stored as rows (transposed / conjugate-transposed). Not referenced if right_svect is set to overwrite or none.
- `[in] ldv`: rocblas_int. ldv >= n if right_svect is all; ldv >= min(m,n) if right_svect is set to singular; or ldv >= 1 otherwise. The leading dimension of V.
- `[out] E`: pointer to real type. Array on the GPU of dimension min(m,n)-1. This array is used to work internally with the bidiagonal matrix B associated with A (using *BDSQR*). On exit, if info > 0, it contains the unconverged off-diagonal elements of B (or properly speaking, a bidiagonal matrix orthogonally equivalent to B). The diagonal elements of this matrix are in S; those that converged correspond to a subset of the singular values of A (not necessarily ordered).

- `[in] fast_alg`: *rocblas_workmode*. If set to rocblas_outofplace, the function will execute the fast thin-SVD version of the algorithm when possible.
- `[out] info`: pointer to a rocblas_int on the GPU. If info = 0, successful exit. If info = i > 0, *BDSQR* did not converge. i elements of E did not converge to zero.

rocsolver_<type>gesvd_batched()

rocblas_status **rocsolver_zgesvd_batched**(rocblas_handle *handle*, **const** *rocblas_svect* *left_svect*, **const** *rocblas_svect* *right_svect*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_double_complex ***const** *A*[], **const** rocblas_int *lda*, double **S*, **const** rocblas_stride *strideS*, rocblas_double_complex **U*, **const** rocblas_int *ldu*, **const** rocblas_stride *strideU*, rocblas_double_complex **V*, **const** rocblas_int *ldv*, **const** rocblas_stride *strideV*, double **E*, **const** rocblas_stride *strideE*, **const** *rocblas_workmode* *fast_alg*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cgesvd_batched**(rocblas_handle *handle*, **const** *rocblas_svect* *left_svect*, **const** *rocblas_svect* *right_svect*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_float_complex ***const** *A*[], **const** rocblas_int *lda*, float **S*, **const** rocblas_stride *strideS*, rocblas_float_complex **U*, **const** rocblas_int *ldu*, **const** rocblas_stride *strideU*, rocblas_float_complex **V*, **const** rocblas_int *ldv*, **const** rocblas_stride *strideV*, float **E*, **const** rocblas_stride *strideE*, **const** *rocblas_workmode* *fast_alg*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dgesvd_batched**(rocblas_handle *handle*, **const** *rocblas_svect* *left_svect*, **const** *rocblas_svect* *right_svect*, **const** rocblas_int *m*, **const** rocblas_int *n*, double ***const** *A*[], **const** rocblas_int *lda*, double **S*, **const** rocblas_stride *strideS*, double **U*, **const** rocblas_int *ldu*, **const** rocblas_stride *strideU*, double **V*, **const** rocblas_int *ldv*, **const** rocblas_stride *strideV*, double **E*, **const** rocblas_stride *strideE*, **const** *rocblas_workmode* *fast_alg*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_sgesvd_batched**(rocblas_handle *handle*, **const** *rocblas_svect* *left_svect*, **const** *rocblas_svect* *right_svect*, **const** rocblas_int *m*, **const** rocblas_int *n*, float ***const** *A*[], **const** rocblas_int *lda*, float **S*, **const** rocblas_stride *strideS*, float **U*, **const** rocblas_int *ldu*, **const** rocblas_stride *strideU*, float **V*, **const** rocblas_int *ldv*, **const** rocblas_stride *strideV*, float **E*, **const** rocblas_stride *strideE*, **const** *rocblas_workmode* *fast_alg*, rocblas_int **info*, **const** rocblas_int *batch_count*)

GESVD_BATCHED computes the singular values and optionally the singular vectors of a batch of general m-by-n matrix A (Singular Value Decomposition).

The SVD of matrix A_j in the batch is given by:

$$A_j = U_j S_j V_j'$$

where the m-by-n matrix S_j is zero except, possibly, for its min(m,n) diagonal elements, which are the singular values of A_j. U_j and V_j are orthogonal (unitary) matrices. The first min(m,n) columns of U_j and V_j are the left and right singular vectors of A_j, respectively.

The computation of the singular vectors is optional and it is controlled by the function arguments left_svect and right_svect as described below. When computed, this function returns the transpose (or transpose conjugate) of the right singular vectors, i.e. the rows of V_j'.

left_svect and right_svect are *rocblas_svect* enums that can take the following values:

- rocblas_svect_all: the entire matrix U_j (or V_j') is computed,
- rocblas_svect_singular: only the singular vectors (first min(m,n) columns of U_j or rows of V_j') are computed,
- rocblas_svect_overwrite: the first columns (or rows) of A_j are overwritten with the singular vectors, or
- rocblas_svect_none: no columns (or rows) of U_j (or V_j') are computed, i.e. no singular vectors.

left_svect and right_svect cannot both be set to overwrite. When neither is set to overwrite, the contents of A_j are destroyed by the time the function returns.

Note When m >> n (or n >> m) the algorithm could be sped up by compressing the matrix A_j via a QR (or LQ) factorization, and working with the triangular factor afterwards (thin-SVD). If the singular vectors are also requested, its computation could be sped up as well via executing some intermediate operations out-of-place, and relying more on matrix multiplications (GEMMs); this will require, however, a larger memory workspace. The parameter fast_alg controls whether the fast algorithm is executed or not. For more details, see the "Tuning rocSOLVER performance" and "Memory model" sections of the documentation.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] left_svect`: *rocblas_svect*. Specifies how the left singular vectors are computed.
- `[in] right_svect`: *rocblas_svect*. Specifies how the right singular vectors are computed.
- `[in] m`: rocblas_int. m >= 0. The number of rows of all matrices A_j in the batch.
- `[in] n`: rocblas_int. n >= 0. The number of columns of all matrices A_j in the batch.
- `[inout] A`: Array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n. On entry, the matrices A_j. On exit, if left_svect (or right_svect) is equal to overwrite, the first columns (or rows) of A_j contain the left (or right) corresponding singular vectors; otherwise, the contents of A_j are destroyed.
- `[in] lda`: rocblas_int. lda >= m. The leading dimension of A_j.
- `[out] S`: pointer to real type. Array on the GPU (the size depends on the value of strideS). The singular values of A_j in decreasing order.
- `[in] strideS`: rocblas_stride. Stride from the start of one vector S_j to the next one S_(j+1). There is no restriction for the value of strideS. Normal use case is strideS >= min(m,n).
- `[out] U`: pointer to type. Array on the GPU (the side depends on the value of strideU). The matrices U_j of left singular vectors stored as columns. Not referenced if left_svect is set to overwrite or none.

- `[in] ldu`: rocblas_int. ldu >= m if left_svect is all or singular; ldu >= 1 otherwise. The leading dimension of U_j.
- `[in] strideU`: rocblas_stride. Stride from the start of one matrix U_j to the next one U_(j+1). There is no restriction for the value of strideU. Normal use case is strideU >= ldu*min(m,n) if left_svect is set to singular, or strideU >= ldu*m when left_svect is equal to all.
- `[out] V`: pointer to type. Array on the GPU (the size depends on the value of strideV). The matrices V_j of right singular vectors stored as rows (transposed / conjugate-transposed). Not referenced if right_svect is set to overwrite or none.
- `[in] ldv`: rocblas_int. ldv >= n if right_svect is all; ldv >= min(m,n) if right_svect is set to singular; or ldv >= 1 otherwise. The leading dimension of V.
- `[in] strideV`: rocblas_stride. Stride from the start of one matrix V_j to the next one V_(j+1). There is no restriction for the value of strideV. Normal use case is strideV >= ldv*n.
- `[out] E`: pointer to real type. Array on the GPU (the size depends on the value of strideE). This array is used to work internally with the bidiagonal matrix B_j associated with A_j (using *BDSQR*). On exit, if info[j] > 0, E_j contains the unconverged off-diagonal elements of B_j (or properly speaking, a bidiagonal matrix orthogonally equivalent to B_j). The diagonal elements of this matrix are in S_j; those that converged correspond to a subset of the singular values of A_j (not necessarily ordered).
- `[in] strideE`: rocblas_stride. Stride from the start of one vector E_j to the next one E_(j+1). There is no restriction for the value of strideE. Normal use case is strideE >= min(m,n)-1.
- `[in] fast_alg`: *rocblas_workmode*. If set to rocblas_outofplace, the function will execute the fast thin-SVD version of the algorithm when possible.
- `[out] info`: pointer to a rocblas_int on the GPU. If info[j] = 0, successful exit. If info[j] = i > 0, *BDSQR* did not converge. i elements of E_j did not converge to zero.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>gesvd_strided_batched()

rocblas_status **rocsolver_zgesvd_strided_batched**(rocblas_handle *handle*, **const** *rocblas_svect* *left_svect*, **const** *rocblas_svect* *right_svect*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, double **S*, **const** rocblas_stride *strideS*, rocblas_double_complex **U*, **const** rocblas_int *ldu*, **const** rocblas_stride *strideU*, rocblas_double_complex **V*, **const** rocblas_int *ldv*, **const** rocblas_stride *strideV*, double **E*, **const** rocblas_stride *strideE*, **const** *rocblas_workmode* *fast_alg*, rocblas_int **info*, **const** rocblas_int *batch_count*) rocblas_status **rocsolver_cgesvd_strided_batched** (rocblas_handle *handle*, **const** *rocblas_svect left_svect*, **const** *rocblas_svect right_svect*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, float **S*, **const** rocblas_stride *strideS*, rocblas_float_complex **U*, **const** rocblas_int *ldu*, **const** rocblas_stride *strideU*, rocblas_float_complex **V*, **const** rocblas_int *ldv*, **const** rocblas_stride *strideV*, float **E*, **const** rocblas_stride *strideE*, **const** *rocblas_workmode fast_alg*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dgesvd_strided_batched** (rocblas_handle *handle*, **const** *rocblas_svect left_svect*, **const** *rocblas_svect right_svect*, **const** rocblas_int *m*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, double **S*, **const** rocblas_stride *strideS*, double **U*, **const** rocblas_int *ldu*, **const** rocblas_stride *strideU*, double **V*, **const** rocblas_int *ldv*, **const** rocblas_stride *strideV*, double **E*, **const** rocblas_stride *strideE*, **const** *rocblas_workmode fast_alg*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_sgesvd_strided_batched** (rocblas_handle *handle*, **const** *rocblas_svect left_svect*, **const** *rocblas_svect right_svect*, **const** rocblas_int *m*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, float **S*, **const** rocblas_stride *strideS*, float **U*, **const** rocblas_int *ldu*, **const** rocblas_stride *strideU*, float **V*, **const** rocblas_int *ldv*, **const** rocblas_stride *strideV*, float **E*, **const** rocblas_stride *strideE*, **const** *rocblas_workmode fast_alg*, rocblas_int **info*, **const** rocblas_int *batch_count*)

GESVD_STRIDED_BATCHED computes the singular values and optionally the singular vectors of a batch of general m-by-n matrix A (Singular Value Decomposition).

The SVD of matrix A_j in the batch is given by:

$$A_j = U_j S_j V_j'$$

where the m-by-n matrix S_j is zero except, possibly, for its min(m,n) diagonal elements, which are the singular values of A_j. U_j and V_j are orthogonal (unitary) matrices. The first min(m,n) columns of U_j and V_j are the left and right singular vectors of A_j, respectively.

The computation of the singular vectors is optional and it is controlled by the function arguments left_svect and right_svect as described below. When computed, this function returns the transpose (or transpose conjugate) of the right singular vectors, i.e. the rows of V_j'.

left_svect and right_svect are *rocblas_svect* enums that can take the following values:

- rocblas_svect_all: the entire matrix U_j (or V_j') is computed,
- rocblas_svect_singular: only the singular vectors (first min(m,n) columns of U_j or rows of V_j') are computed,
- rocblas_svect_overwrite: the first columns (or rows) of A_j are overwritten with the singular vectors, or
- rocblas_svect_none: no columns (or rows) of U_j (or V_j') are computed, i.e. no singular vectors.

left_svect and right_svect cannot both be set to overwrite. When neither is set to overwrite, the contents of A_j are destroyed by the time the function returns.

Note When m >> n (or n >> m) the algorithm could be sped up by compressing the matrix A_j via a QR (or LQ) factorization, and working with the triangular factor afterwards (thin-SVD). If the singular vectors are also requested, its computation could be sped up as well via executing some intermediate operations out-of-place, and relying more on matrix multiplications (GEMMs); this will require, however, a larger memory workspace. The parameter fast_alg controls whether the fast algorithm is executed or not. For more details, see the "Tuning rocSOLVER performance" and "Memory model" sections of the documentation.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] left_svect`: *rocblas_svect*. Specifies how the left singular vectors are computed.
- `[in] right_svect`: *rocblas_svect*. Specifies how the right singular vectors are computed.
- `[in] m`: rocblas_int. m >= 0. The number of rows of all matrices A_j in the batch.
- `[in] n`: rocblas_int. n >= 0. The number of columns of all matrices A_j in the batch.
- `[inout] A`: pointer to type. Array on the GPU (the size depends on the value of strideA). On entry, the matrices A_j. On exit, if left_svect (or right_svect) is equal to overwrite, the first columns (or rows) of A_j contain the left (or right) corresponding singular vectors; otherwise, the contents of A_j are destroyed.
- `[in] lda`: rocblas_int. lda >= m. The leading dimension of A_j.
- `[in] strideA`: rocblas_stride. Stride from the start of one matrix A_j to the next one A_(j+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n.
- `[out] S`: pointer to real type. Array on the GPU (the size depends on the value of strideS). The singular values of A_j in decreasing order.
- `[in] strideS`: rocblas_stride. Stride from the start of one vector S_j to the next one S_(j+1). There is no restriction for the value of strideS. Normal use case is strideS >= min(m,n).
- `[out] U`: pointer to type. Array on the GPU (the side depends on the value of strideU). The matrices U_j of left singular vectors stored as columns. Not referenced if left_svect is set to overwrite or none.
- `[in] ldu`: rocblas_int. ldu >= m if left_svect is all or singular; ldu >= 1 otherwise. The leading dimension of U_j.
- `[in] strideU`: rocblas_stride. Stride from the start of one matrix U_j to the next one U_(j+1). There is no restriction for the value of strideU. Normal use case is strideU >= ldu*min(m,n) if left_svect is set to singular, or strideU >= ldu*m when left_svect is equal to all.
- `[out] V`: pointer to type. Array on the GPU (the size depends on the value of strideV). The matrices V_j of right singular vectors stored as rows (transposed / conjugate-transposed). Not referenced if right_svect is set to overwrite or none.

- `[in] ldv`: rocblas_int. ldv >= n if right_svect is all; ldv >= min(m,n) if right_svect is set to singular; or ldv >= 1 otherwise. The leading dimension of V.
- `[in] strideV`: rocblas_stride. Stride from the start of one matrix V_j to the next one V_(j+1). There is no restriction for the value of strideV. Normal use case is strideV >= ldv*n.
- `[out] E`: pointer to real type. Array on the GPU (the size depends on the value of strideE). This array is used to work internally with the bidiagonal matrix B_j associated with A_j (using *BDSQR*). On exit, if info > 0, E_j contains the unconverged off-diagonal elements of B_j (or properly speaking, a bidiagonal matrix orthogonally equivalent to B_j). The diagonal elements of this matrix are in S_j; those that converged correspond to a subset of the singular values of A_j (not necessarily ordered).
- `[in] strideE`: rocblas_stride. Stride from the start of one vector E_j to the next one E_(j+1). There is no restriction for the value of strideE. Normal use case is strideE >= min(m,n)-1.
- `[in] fast_alg`: *rocblas_workmode*. If set to rocblas_outofplace, the function will execute the fast thin-SVD version of the algorithm when possible.
- `[out] info`: pointer to a rocblas_int on the GPU. If info[j] = 0, successful exit. If info[j] = i > 0, BDSQR did not converge. i elements of E_j did not converge to zero.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

3.4 Lapack-like Functions

Other Lapack-like routines provided by rocSOLVER. These are divided into the following subcategories:

- *Triangular factorizations*. Based on Gaussian elimination.
- *Linear-systems solvers*. Based on triangular factorizations.

Note: Throughout the APIs' descriptions, we use the following notations:

- x[i] stands for the i-th element of vector x, while A[i,j] represents the element in the i-th row and j-th column of matrix A. Indices are 1-based, i.e. x[1] is the first element of x.
- If X is a real vector or matrix, X^T indicates its transpose; if X is complex, then X^H represents its conjugate transpose. When X could be real or complex, we use X' to indicate X transposed or X conjugate transposed, accordingly.
- x_i = x_i; we sometimes use both notations, x_i when displaying mathematical equations, and x_i in the text describing the function parameters.

3.4.1 Triangular factorizations

List of Lapack-like triangular factorizations

- *rocsolver_<type>getf2_npvt()*
- *rocsolver_<type>getf2_npvt_batched()*
- *rocsolver_<type>getf2_npvt_strided_batched()*
- *rocsolver_<type>getrf_npvt()*
- *rocsolver_<type>getrf_npvt_batched()*

- *rocsolver_<type>getrf_npvt_strided_batched()*

rocsolver_<type>getf2_npvt()

rocblas_status **rocsolver_zgetf2_npvt** (rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, rocblas_int **info*)

rocblas_status **rocsolver_cgetf2_npvt** (rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, rocblas_int **info*)

rocblas_status **rocsolver_dgetf2_npvt** (rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, rocblas_int **info*)

rocblas_status **rocsolver_sgetf2_npvt** (rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, rocblas_int **info*)

GETF2_NPVT computes the LU factorization of a general m-by-n matrix A without partial pivoting.

(This is the unblocked Level-2-BLAS version of the algorithm. An optimized internal implementation without rocBLAS calls could be executed with small and mid-size matrices if optimizations are enabled (default option). For more details, see the "Tuning rocSOLVER performance" section of the Library Design Guide).

The factorization has the form

$$A = LU$$

where L is lower triangular with unit diagonal elements (lower trapezoidal if m > n), and U is upper triangular (upper trapezoidal if m < n).

Note: Although this routine can offer better performance, Gaussian elimination without pivoting is not backward stable. If numerical accuracy is compromised, use the legacy-LAPACK-like API *GETF2* routines instead.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. m >= 0. The number of rows of the matrix A.
- `[in] n`: rocblas_int. n >= 0. The number of columns of the matrix A.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the m-by-n matrix A to be factored. On exit, the factors L and U from the factorization. The unit diagonal elements of L are not stored.
- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of A.
- `[out] info`: pointer to a rocblas_int on the GPU. If info = 0, successful exit. If info = j > 0, U is singular. U[j,j] is the first zero element in the diagonal. The factorization from this point might be incomplete.

rocsolver_<type>getf2_npvt_batched()

rocblas_status **rocsolver_zgetf2_npvt_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_double_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cgetf2_npvt_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_float_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dgetf2_npvt_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, double ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_sgetf2_npvt_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, float ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **info*, **const** rocblas_int *batch_count*)

GETF2_NPVT_BATCHED computes the LU factorization of a batch of general m-by-n matrices without partial pivoting.

(This is the unblocked Level-2-BLAS version of the algorithm. An optimized internal implementation without rocBLAS calls could be executed with small and mid-size matrices if optimizations are enabled (default option). For more details, see the "Tuning rocSOLVER performance" section of the Library Design Guide).

The factorization of matrix A_i in the batch has the form

$$A_i = L_i U_i$$

where L_i is lower triangular with unit diagonal elements (lower trapezoidal if m > n), and U_i is upper triangular (upper trapezoidal if m < n).

Note: Although this routine can offer better performance, Gaussian elimination without pivoting is not backward stable. If numerical accuracy is compromised, use the legacy-LAPACK-like API *GETF2_BATCHED* routines instead.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. m >= 0. The number of rows of all matrices A_i in the batch.
- `[in] n`: rocblas_int. n >= 0. The number of columns of all matrices A_i in the batch.
- `[inout] A`: array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n. On entry, the m-by-n matrices A_i to be factored. On exit, the factors L_i and U_i from the factorizations. The unit diagonal elements of L_i are not stored.
- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of matrices A_i.
- `[out] info`: pointer to rocblas_int. Array of batch_count integers on the GPU. If info[i] = 0, successful exit for factorization of A_i. If info[i] = j > 0, U_i is singular. U_i[j,j] is the first zero element in the diagonal. The factorization from this point might be incomplete.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>getf2_npvt_strided_batched()

rocblas_status **rocsolver_zgetf2_npvt_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cgetf2_npvt_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dgetf2_npvt_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_sgetf2_npvt_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **info*, **const** rocblas_int *batch_count*)

GETF2_NPVT_STRIDED_BATCHED computes the LU factorization of a batch of general m-by-n matrices without partial pivoting.

(This is the unblocked Level-2-BLAS version of the algorithm. An optimized internal implementation without rocBLAS calls could be executed with small and mid-size matrices if optimizations are enabled (default option). For more details, see the "Tuning rocSOLVER performance" section of the Library Design Guide).

The factorization of matrix A_i in the batch has the form

$$A_i = L_i U_i$$

where L_i is lower triangular with unit diagonal elements (lower trapezoidal if m > n), and U_i is upper triangular (upper trapezoidal if m < n).

Note: Although this routine can offer better performance, Gaussian elimination without pivoting is not backward stable. If numerical accuracy is compromised, use the legacy-LAPACK-like API *GETF2_STRIDED_BATCHED* routines instead.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. m >= 0. The number of rows of all matrices A_i in the batch.
- `[in] n`: rocblas_int. n >= 0. The number of columns of all matrices A_i in the batch.
- `[inout] A`: pointer to type. Array on the GPU (the size depends on the value of strideA). On entry, the m-by-n matrices A_i to be factored. On exit, the factors L_i and U_i from the factorization. The unit diagonal elements of L_i are not stored.

- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of matrices A_i.
- `[in] strideA`: rocblas_stride. Stride from the start of one matrix A_i to the next one A_(i+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n
- `[out] info`: pointer to rocblas_int. Array of batch_count integers on the GPU. If info[i] = 0, successful exit for factorization of A_i. If info[i] = j > 0, U_i is singular. U_i[j,j] is the first zero element in the diagonal. The factorization from this point might be incomplete.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>getrf_npvt()

rocblas_status **rocsolver_zgetrf_npvt** (rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, rocblas_int **info*)

rocblas_status **rocsolver_cgetrf_npvt** (rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, rocblas_int **info*)

rocblas_status **rocsolver_dgetrf_npvt** (rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, rocblas_int **info*)

rocblas_status **rocsolver_sgetrf_npvt** (rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, rocblas_int **info*)

GETRF_NPVT computes the LU factorization of a general m-by-n matrix A without partial pivoting.

(This is the blocked Level-3-BLAS version of the algorithm. An optimized internal implementation without rocBLAS calls could be executed with mid-size matrices if optimizations are enabled (default option). For more details, see the "Tuning rocSOLVER performance" section of the Library Design Guide).

The factorization has the form

$$A = LU$$

where L is lower triangular with unit diagonal elements (lower trapezoidal if m > n), and U is upper triangular (upper trapezoidal if m < n).

Note: Although this routine can offer better performance, Gaussian elimination without pivoting is not backward stable. If numerical accuracy is compromised, use the legacy-LAPACK-like API *GETRF* routines instead.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. m >= 0. The number of rows of the matrix A.
- `[in] n`: rocblas_int. n >= 0. The number of columns of the matrix A.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the m-by-n matrix A to be factored. On exit, the factors L and U from the factorization. The unit diagonal elements of L are not stored.
- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of A.

- `[out] info`: pointer to a rocblas_int on the GPU. If info = 0, successful exit. If info = j > 0, U is singular. U[j,j] is the first zero element in the diagonal. The factorization from this point might be incomplete.

rocsolver_<type>getrf_npvt_batched()

rocblas_status **rocsolver_zgetrf_npvt_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_double_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cgetrf_npvt_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_float_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dgetrf_npvt_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, double ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_sgetrf_npvt_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, float ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **info*, **const** rocblas_int *batch_count*)

GETRF_NPVT_BATCHED computes the LU factorization of a batch of general m-by-n matrices without partial pivoting.

(This is the blocked Level-3-BLAS version of the algorithm. An optimized internal implementation without rocBLAS calls could be executed with mid-size matrices if optimizations are enabled (default option). For more details, see the “Tuning rocSOLVER performance” section of the Library Design Guide).

The factorization of matrix A_i in the batch has the form

$$A_i = L_i U_i$$

where L_i is lower triangular with unit diagonal elements (lower trapezoidal if m > n), and U_i is upper triangular (upper trapezoidal if m < n).

Note: Although this routine can offer better performance, Gaussian elimination without pivoting is not backward stable. If numerical accuracy is compromised, use the legacy-LAPACK-like API *GETRF_BATCHED* routines instead.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] m`: rocblas_int. m >= 0. The number of rows of all matrices A_i in the batch.
- `[in] n`: rocblas_int. n >= 0. The number of columns of all matrices A_i in the batch.
- `[inout] A`: array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n. On entry, the m-by-n matrices A_i to be factored. On exit, the factors L_i and U_i from the factorizations. The unit diagonal elements of L_i are not stored.
- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of matrices A_i.

- `[out] info`: pointer to rocblas_int. Array of batch_count integers on the GPU. If info[i] = 0, successful exit for factorization of A_i. If info[i] = j > 0, U_i is singular. U_i[j,j] is the first zero element in the diagonal. The factorization from this point might be incomplete.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>getrf_npvt_strided_batched()

rocblas_status **rocsolver_zgetrf_npvt_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cgetrf_npvt_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dgetrf_npvt_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_sgetrf_npvt_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *m*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **info*, **const** rocblas_int *batch_count*)

GETRF_NPVT_STRIDED_BATCHED computes the LU factorization of a batch of general m-by-n matrices without partial pivoting.

(This is the blocked Level-3-BLAS version of the algorithm. An optimized internal implementation without rocBLAS calls could be executed with mid-size matrices if optimizations are enabled (default option). For more details, see the "Tuning rocSOLVER performance" section of the Library Design Guide).

The factorization of matrix A_i in the batch has the form

$$A_i = L_i U_i$$

where L_i is lower triangular with unit diagonal elements (lower trapezoidal if m > n), and U_i is upper triangular (upper trapezoidal if m < n).

Note: Although this routine can offer better performance, Gaussian elimination without pivoting is not backward stable. If numerical accuracy is compromised, use the legacy-LAPACK-like API *GETRF_STRIDED_BATCHED* routines instead.

Parameters

- `[in] handle`: rocblas_handle.

- `[in] m`: rocblas_int. m >= 0. The number of rows of all matrices A_i in the batch.
- `[in] n`: rocblas_int. n >= 0. The number of columns of all matrices A_i in the batch.
- `[inout] A`: pointer to type. Array on the GPU (the size depends on the value of strideA). On entry, the m-by-n matrices A_i to be factored. On exit, the factors L_i and U_i from the factorization. The unit diagonal elements of L_i are not stored.
- `[in] lda`: rocblas_int. lda >= m. Specifies the leading dimension of matrices A_i.
- `[in] strideA`: rocblas_stride. Stride from the start of one matrix A_i to the next one A_(i+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n
- `[out] info`: pointer to rocblas_int. Array of batch_count integers on the GPU. If info[i] = 0, successful exit for factorization of A_i. If info[i] = j > 0, U_i is singular. U_i[j,j] is the first zero element in the diagonal. The factorization from this point might be incomplete.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

3.4.2 Linear-systems solvers

List of Lapack-like linear solvers

- *rocsolver_<type>getri_npvt()*
- *rocsolver_<type>getri_npvt_batched()*
- *rocsolver_<type>getri_npvt_strided_batched()*
- *rocsolver_<type>getri_outofplace()*
- *rocsolver_<type>getri_outofplace_batched()*
- *rocsolver_<type>getri_outofplace_strided_batched()*
- *rocsolver_<type>getri_npvt_outofplace()*
- *rocsolver_<type>getri_npvt_outofplace_batched()*
- *rocsolver_<type>getri_npvt_outofplace_strided_batched()*

rocsolver_<type>getri_npvt()

rocblas_status **rocsolver_zgetri_npvt** (rocblas_handle *handle*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, rocblas_int **info*)

rocblas_status **rocsolver_cgetri_npvt** (rocblas_handle *handle*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, rocblas_int **info*)

rocblas_status **rocsolver_dgetri_npvt** (rocblas_handle *handle*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, rocblas_int **info*)

rocblas_status **rocsolver_sgetri_npvt** (rocblas_handle *handle*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, rocblas_int **info*)

GETRI_NPVT inverts a general n-by-n matrix A using the LU factorization computed by *GETRF_NPVT*.

The inverse is computed by solving the linear system

$$A^{-1}L = U^{-1}$$

where L is the lower triangular factor of A with unit diagonal elements, and U is the upper triangular factor.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] n`: rocblas_int. n >= 0. The number of rows and columns of the matrix A.
- `[inout] A`: pointer to type. Array on the GPU of dimension lda*n. On entry, the factors L and U of the factorization A = L*U returned by *GETRF_NPVT*. On exit, the inverse of A if info = 0; otherwise undefined.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of A.
- `[out] info`: pointer to a rocblas_int on the GPU. If info = 0, successful exit. If info = i > 0, U is singular. U[i,i] is the first zero pivot.

rocsolver_<type>getri_npvt_batched()

rocblas_status **rocsolver_zgetri_npvt_batched**(rocblas_handle *handle*, **const** rocblas_int *n*, rocblas_double_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cgetri_npvt_batched**(rocblas_handle *handle*, **const** rocblas_int *n*, rocblas_float_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dgetri_npvt_batched**(rocblas_handle *handle*, **const** rocblas_int *n*, double ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_sgetri_npvt_batched**(rocblas_handle *handle*, **const** rocblas_int *n*, float ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **info*, **const** rocblas_int *batch_count*)

GETRI_NPVT_BATCHED inverts a batch of general n-by-n matrices using the LU factorization computed by *GETRF_NPVT_BATCHED*.

The inverse of matrix A_j in the batch is computed by solving the linear system

$$A_j^{-1}L_j = U_j^{-1}$$

where L_j is the lower triangular factor of A_j with unit diagonal elements, and U_j is the upper triangular factor.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] n`: rocblas_int. n >= 0. The number of rows and columns of all matrices A_j in the batch.

- `[inout] A`: array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n. On entry, the factors L_j and U_j of the factorization A = L_j*U_j returned by *GETRF_NPVT_BATCHED*. On exit, the inverses of A_j if info[j] = 0; otherwise undefined.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of matrices A_j.
- `[out] info`: pointer to rocblas_int. Array of batch_count integers on the GPU. If info[j] = 0, successful exit for inversion of A_j. If info[j] = i > 0, U_j is singular. U_j[i,i] is the first zero pivot.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>getri_npvt_strided_batched()

rocblas_status **rocsolver_zgetri_npvt_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cgetri_npvt_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dgetri_npvt_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_sgetri_npvt_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **info*, **const** rocblas_int *batch_count*)

GETRI_NPVT_STRIDED_BATCHED inverts a batch of general n-by-n matrices using the LU factorization computed by *GETRF_NPVT_STRIDED_BATCHED*.

The inverse of matrix A_j in the batch is computed by solving the linear system

$$A_j^{-1} L_j = U_j^{-1}$$

where L_j is the lower triangular factor of A_j with unit diagonal elements, and U_j is the upper triangular factor.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] n`: rocblas_int. n >= 0. The number of rows and columns of all matrices A_j in the batch.
- `[inout] A`: pointer to type. Array on the GPU (the size depends on the value of strideA). On entry, the factors L_j and U_j of the factorization A_j = L_j*U_j returned by *GETRF_NPVT_STRIDED_BATCHED*. On exit, the inverses of A_j if info[j] = 0; otherwise undefined.

- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of matrices A_j.
- `[in] strideA`: rocblas_stride. Stride from the start of one matrix A_j to the next one A_(j+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n
- `[out] info`: pointer to rocblas_int. Array of batch_count integers on the GPU. If info[j] = 0, successful exit for inversion of A_j. If info[j] = i > 0, U_j is singular. U_j[i,i] is the first zero pivot.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>getri_outofplace()

rocblas_status **rocsolver_zgetri_outofplace**(rocblas_handle *handle*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, rocblas_int **ipiv*, rocblas_double_complex **C*, **const** rocblas_int *ldc*, rocblas_int **info*)

rocblas_status **rocsolver_cgetri_outofplace**(rocblas_handle *handle*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, rocblas_int **ipiv*, rocblas_float_complex **C*, **const** rocblas_int *ldc*, rocblas_int **info*)

rocblas_status **rocsolver_dgetri_outofplace**(rocblas_handle *handle*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, rocblas_int **ipiv*, double **C*, **const** rocblas_int *ldc*, rocblas_int **info*)

rocblas_status **rocsolver_sgetri_outofplace**(rocblas_handle *handle*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, rocblas_int **ipiv*, float **C*, **const** rocblas_int *ldc*, rocblas_int **info*)

GETRI_OUTOFPLACE computes the inverse $C = A^{-1}$ of a general n-by-n matrix A.

The inverse is computed by solving the linear system

$$AC = I$$

where I is the identity matrix, and A is factorized as $A = PLU$ as given by *GETRF*.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] n`: rocblas_int. n >= 0. The number of rows and columns of the matrix A.
- `[in] A`: pointer to type. Array on the GPU of dimension lda*n. The factors L and U of the factorization A = P*L*U returned by *GETRF*.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of A.
- `[in] ipiv`: pointer to rocblas_int. Array on the GPU of dimension n. The pivot indices returned by *GETRF*.
- `[out] C`: pointer to type. Array on the GPU of dimension ldc*n. If info = 0, the inverse of A. Otherwise, undefined.
- `[in] ldc`: rocblas_int. ldc >= n. Specifies the leading dimension of C.
- `[out] info`: pointer to a rocblas_int on the GPU. If info = 0, successful exit. If info = i > 0, U is singular. U[i,i] is the first zero pivot.

rocsolver_<type>getri_outofplace_batched()

rocblas_status **rocsolver_zgetri_outofplace_batched**(rocblas_handle *handle*, **const** rocblas_int *n*, rocblas_double_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_double_complex ***const** *C*[], **const** rocblas_int *ldc*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cgetri_outofplace_batched**(rocblas_handle *handle*, **const** rocblas_int *n*, rocblas_float_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_float_complex ***const** *C*[], **const** rocblas_int *ldc*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dgetri_outofplace_batched**(rocblas_handle *handle*, **const** rocblas_int *n*, double ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, double ***const** *C*[], **const** rocblas_int *ldc*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_sgetri_outofplace_batched**(rocblas_handle *handle*, **const** rocblas_int *n*, float ***const** *A*[], **const** rocblas_int *lda*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, float ***const** *C*[], **const** rocblas_int *ldc*, rocblas_int **info*, **const** rocblas_int *batch_count*)

GETRI_OUTOFPLACE_BATCHED computes the inverse $C_j = A_j^{-1}$ of a batch of general n-by-n matrices A_j.

The inverse is computed by solving the linear system

$$A_j C_j = I$$

where I is the identity matrix, and A_j is factorized as $A_j = P_j L_j U_j$ as given by *GETRF_BATCHED*.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] n`: rocblas_int. n >= 0. The number of rows and columns of all matrices A_j in the batch.
- `[in] A`: array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n. The factors L_j and U_j of the factorization A_j = P_j*L_j*U_j returned by *GETRF_BATCHED*.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of matrices A_j.
- `[in] ipiv`: pointer to rocblas_int. Array on the GPU (the size depends on the value of strideP). The pivot indices returned by *GETRF_BATCHED*.
- `[in] strideP`: rocblas_stride. Stride from the start of one vector ipiv_j to the next one ipiv_(i+j). There is no restriction for the value of strideP. Normal use case is strideP >= n.

- `[out] C`: array of pointers to type. Each pointer points to an array on the GPU of dimension ldc*n. If info[j] = 0, the inverse of matrices A_j. Otherwise, undefined.
- `[in] ldc`: rocblas_int. ldc >= n. Specifies the leading dimension of C_j.
- `[out] info`: pointer to rocblas_int. Array of batch_count integers on the GPU. If info[j] = 0, successful exit for inversion of A_j. If info[j] = i > 0, U_j is singular. U_j[i,i] is the first zero pivot.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>getri_outofplace_strided_batched()

rocblas_status **rocsolver_zgetri_outofplace_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_double_complex **C*, **const** rocblas_int *ldc*, **const** rocblas_stride *strideC*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cgetri_outofplace_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, rocblas_float_complex **C*, **const** rocblas_int *ldc*, **const** rocblas_stride *strideC*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dgetri_outofplace_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, double **C*, **const** rocblas_int *ldc*, **const** rocblas_stride *strideC*, rocblas_int **info*, **const** rocblas_int *batch_count*) rocblas_status **rocsolver_sgetri_outofplace_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_int **ipiv*, **const** rocblas_stride *strideP*, float **C*, **const** rocblas_int *ldc*, **const** rocblas_stride *strideC*, rocblas_int **info*, **const** rocblas_int *batch_count*)

GETRI_OUTOFPLACE_STRIDED_BATCHED computes the inverse $C_j = A_j^{-1}$ of a batch of general n-by-n matrices A_j.

The inverse is computed by solving the linear system

$$A_j C_j = I$$

where I is the identity matrix, and A_j is factorized as $A_j = P_j L_j U_j$ as given by *GETRF_STRIDED_BATCHED*.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] n`: rocblas_int. n >= 0. The number of rows and columns of all matrices A_j in the batch.
- `[in] A`: pointer to type. Array on the GPU (the size depends on the value of strideA). The factors L_j and U_j of the factorization A_j = P_j*L_j*U_j returned by *GETRF_STRIDED_BATCHED*.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of matrices A_j.
- `[in] strideA`: rocblas_stride. Stride from the start of one matrix A_j to the next one A_(j+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n
- `[in] ipiv`: pointer to rocblas_int. Array on the GPU (the size depends on the value of strideP). The pivot indices returned by *GETRF_STRIDED_BATCHED*.
- `[in] strideP`: rocblas_stride. Stride from the start of one vector ipiv_j to the next one ipiv_(j+1). There is no restriction for the value of strideP. Normal use case is strideP >= n.
- `[out] C`: pointer to type. Array on the GPU (the size depends on the value of strideC). If info[j] = 0, the inverse of matrices A_j. Otherwise, undefined.
- `[in] ldc`: rocblas_int. ldc >= n. Specifies the leading dimension of C_j.
- `[in] strideC`: rocblas_stride. Stride from the start of one matrix C_j to the next one C_(j+1). There is no restriction for the value of strideC. Normal use case is strideC >= ldc*n
- `[out] info`: pointer to rocblas_int. Array of batch_count integers on the GPU. If info[j] = 0, successful exit for inversion of A_j. If info[j] = i > 0, U_j is singular. U_j[i,i] is the first zero pivot.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>getri_npvt_outofplace()

rocblas_status **rocsolver_zgetri_npvt_outofplace**(rocblas_handle *handle*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, rocblas_double_complex **C*, **const** rocblas_int *ldc*, rocblas_int **info*)

rocblas_status **rocsolver_cgetri_npvt_outofplace**(rocblas_handle *handle*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, rocblas_float_complex **C*, **const** rocblas_int *ldc*, rocblas_int **info*)

rocblas_status **rocsolver_dgetri_npvt_outofplace**(rocblas_handle *handle*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, double **C*, **const** rocblas_int *ldc*, rocblas_int **info*)

rocblas_status **rocsolver_sgetri_npvt_outofplace**(rocblas_handle *handle*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, float **C*, **const** rocblas_int *ldc*, rocblas_int **info*)

GETRI_NPVT_OUTOFPLACE computes the inverse $C = A^{-1}$ of a general n-by-n matrix A without partial pivoting.

The inverse is computed by solving the linear system

$$AC = I$$

where I is the identity matrix, and A is factorized as $A = LU$ as given by *GETRF_NPVT*.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] n`: rocblas_int. n >= 0. The number of rows and columns of the matrix A.
- `[in] A`: pointer to type. Array on the GPU of dimension lda*n. The factors L and U of the factorization A = L*U returned by *GETRF_NPVT*.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of A.
- `[out] C`: pointer to type. Array on the GPU of dimension ldc*n. If info = 0, the inverse of A. Otherwise, undefined.
- `[in] ldc`: rocblas_int. ldc >= n. Specifies the leading dimension of C.
- `[out] info`: pointer to a rocblas_int on the GPU. If info = 0, successful exit. If info = i > 0, U is singular. U[i,i] is the first zero pivot.

rocsolver_<type>getri_npvt_outofplace_batched()

rocblas_status **rocsolver_zgetri_npvt_outofplace_batched**(rocblas_handle *handle*, **const** rocblas_int *n*, rocblas_double_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_double_complex ***const** *C*[], **const** rocblas_int *ldc*, rocblas_int **info*, **const** rocblas_int *batch_count*) rocblas_status **rocsolver_cgetri_npvt_outofplace_batched**(rocblas_handle *handle*, **const** rocblas_int *n*, rocblas_float_complex ***const** *A*[], **const** rocblas_int *lda*, rocblas_float_complex ***const** *C*[], **const** rocblas_int *ldc*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dgetri_npvt_outofplace_batched**(rocblas_handle *handle*, **const** rocblas_int *n*, double ***const** *A*[], **const** rocblas_int *lda*, double ***const** *C*[], **const** rocblas_int *ldc*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_sgetri_npvt_outofplace_batched**(rocblas_handle *handle*, **const** rocblas_int *n*, float ***const** *A*[], **const** rocblas_int *lda*, float ***const** *C*[], **const** rocblas_int *ldc*, rocblas_int **info*, **const** rocblas_int *batch_count*)

GETRI_NPVT_OUTOFPLACE_BATCHED computes the inverse $C_j = A_j^{-1}$ of a batch of general n-by-n matrices A_j without partial pivoting.

The inverse is computed by solving the linear system

$$A_j C_j = I$$

where I is the identity matrix, and A_j is factorized as $A_j = L_j U_j$ as given by *GETRF_NPVT_BATCHED*.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] n`: rocblas_int. n >= 0. The number of rows and columns of all matrices A_j in the batch.
- `[in] A`: array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n. The factors L_j and U_j of the factorization A_j = L_j*U_j returned by *GETRF_NPVT_BATCHED*.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of matrices A_j.
- `[out] C`: array of pointers to type. Each pointer points to an array on the GPU of dimension ldc*n. If info[j] = 0, the inverse of matrices A_j. Otherwise, undefined.
- `[in] ldc`: rocblas_int. ldc >= n. Specifies the leading dimension of C_j.
- `[out] info`: pointer to rocblas_int. Array of batch_count integers on the GPU. If info[j] = 0, successful exit for inversion of A_j. If info[j] = i > 0, U_j is singular. U_j[i,i] is the first zero pivot.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

rocsolver_<type>getri_npvt_outofplace_strided_batched()

rocblas_status **rocsolver_zgetri_npvt_outofplace_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *n*, rocblas_double_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_double_complex **C*, **const** rocblas_int *ldc*, **const** rocblas_stride *strideC*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_cgetri_npvt_outofplace_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *n*, rocblas_float_complex **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, rocblas_float_complex **C*, **const** rocblas_int *ldc*, **const** rocblas_stride *strideC*, rocblas_int **info*, **const** rocblas_int *batch_count*)

rocblas_status **rocsolver_dgetri_npvt_outofplace_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *n*, double **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, double **C*, **const** rocblas_int *ldc*, **const** rocblas_stride *strideC*, rocblas_int **info*, **const** rocblas_int *batch_count*) rocblas_status **rocsolver_sgetri_npvt_outofplace_strided_batched**(rocblas_handle *handle*, **const** rocblas_int *n*, float **A*, **const** rocblas_int *lda*, **const** rocblas_stride *strideA*, float **C*, **const** rocblas_int *ldc*, **const** rocblas_stride *strideC*, rocblas_int **info*, **const** rocblas_int *batch_count*)

GETRI_NPVT_OUTOFPLACE_STRIDED_BATCHED computes the inverse $C_j = A_j^{-1}$ of a batch of general n-by-n matrices A_j without partial pivoting.

The inverse is computed by solving the linear system

$$A_j C_j = I$$

where I is the identity matrix, and A_j is factorized as $A_j = L_j U_j$ as given by *GETRF_NPVT_STRIDED_BATCHED*.

Parameters

- `[in] handle`: rocblas_handle.
- `[in] n`: rocblas_int. n >= 0. The number of rows and columns of all matrices A_j in the batch.
- `[in] A`: pointer to type. Array on the GPU (the size depends on the value of strideA). The factors L_j and U_j of the factorization A_j = L_j*U_j returned by *GETRF_NPVT_STRIDED_BATCHED*.
- `[in] lda`: rocblas_int. lda >= n. Specifies the leading dimension of matrices A_j.
- `[in] strideA`: rocblas_stride. Stride from the start of one matrix A_j to the next one A_(j+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n
- `[out] C`: pointer to type. Array on the GPU (the size depends on the value of strideC). If info[j] = 0, the inverse of matrices A_j. Otherwise, undefined.
- `[in] ldc`: rocblas_int. ldc >= n. Specifies the leading dimension of C_j.
- `[in] strideC`: rocblas_stride. Stride from the start of one matrix C_j to the next one C_(j+1). There is no restriction for the value of strideC. Normal use case is strideC >= ldc*n
- `[out] info`: pointer to rocblas_int. Array of batch_count integers on the GPU. If info[j] = 0, successful exit for inversion of A_j. If info[j] = i > 0, U_j is singular. U_j[i,i] is the first zero pivot.
- `[in] batch_count`: rocblas_int. batch_count >= 0. Number of matrices in the batch.

3.5 Logging Functions and Library Information

3.5.1 Logging functions

These functions control rocSOLVER's *Multi-level Logging* capabilities.

List of logging functions

- *rocsolver_log_begin()*
- *rocsolver_log_end()*
- *rocsolver_log_set_layer_mode()*
- *rocsolver_log_set_max_levels()*
- *rocsolver_log_restore_defaults()*
- *rocsolver_log_write_profile()*
- *rocsolver_log_flush_profile()*

rocsolver_log_begin()

rocblas_status **rocsolver_log_begin**(void)

LOG_BEGIN begins a rocSOLVER multi-level logging session.

Initializes the rocSOLVER logging environment with default values (no logging and one level depth). Default mode can be overridden by using the environment variables ROCSOLVER_LAYER and ROCSOLVER_LEVELS.

This function also sets the streams where the log results will be outputted. The default is STDERR for all the modes. This default can also be overridden using the environment variable ROCSOLVER_LOG_PATH, or specifically ROCSOLVER_LOG_TRACE_PATH, ROCSOLVER_LOG_BENCH_PATH, and/or ROCSOLVER_LOG_PROFILE_PATH.

rocsolver_log_end()

rocblas_status **rocsolver_log_end**(void)

LOG_END ends the multi-level rocSOLVER logging session.

If applicable, this function also prints the profile logging results before cleaning the logging environment.

rocsolver_log_set_layer_mode()

rocblas_status **rocsolver_log_set_layer_mode**(**const** rocblas_layer_mode_flags *layer_mode*)

LOG_SET_LAYER_MODE sets the logging mode for the rocSOLVER multi-level logging environment.

Parameters

- `[in] layer_mode`: rocblas_layer_mode_flags. Specifies the logging mode.

rocsolver_log_set_max_levels()

rocblas_status **rocsolver_log_set_max_levels**(**const** rocblas_int *max_levels*)

LOG_SET_MAX_LEVELS sets the maximum trace log depth for the rocSOLVER multi-level logging environment.

Parameters

- `[in] max_levels`: rocblas_int. max_levels >= 1. Specifies the maximum depth at which nested function calls will appear in the trace and profile logs.

rocsolver_log_restore_defaults()

rocblas_status **rocsolver_log_restore_defaults**(void)

LOG_RESTORE_DEFAULTS restores the default values of the rocSOLVER multi-level logging environment.

This function sets the logging mode and maximum trace log depth to their default values (no logging and one level depth).

rocsolver_log_write_profile()

rocblas_status **rocsolver_log_write_profile**(void)

LOG_WRITE_PROFILE prints the profile logging results.

rocsolver_log_flush_profile()

rocblas_status **rocsolver_log_flush_profile**(void)

LOG_FLUSH_PROFILE prints the profile logging results and clears the profile record.

3.5.2 Library information

List of library information functions

- *rocsolver_get_version_string()*
- *rocsolver_get_version_string_size()*

rocsolver_get_version_string()

rocblas_status **rocsolver_get_version_string**(char **buf*, size_t *len*)

GET_VERSION_STRING Queries the library version.

Parameters

- `[out] buf`: A buffer that the version string will be written into.
- `[in] len`: The size of the given buffer in bytes.

rocsolver_get_version_string_size()

rocblas_status **rocsolver_get_version_string_size**(size_t **len*)

GET_VERSION_STRING_SIZE Queries the minimum buffer size for a successful call to *rocsolver_get_version_string*.

Parameters

- `[out] len`: pointer to size_t. The minimum length of buffer to pass to *rocsolver_get_version_string*.

3.6 Deprecated

Originally, rocSOLVER maintained its own types and helpers as aliases to those of rocBLAS. These aliases are now deprecated. See the rocBLAS types and rocBLAS auxiliary functions documentation for information on the suggested replacements.

- Deprecated *Types*.
- Deprecated *Auxiliary functions*.

3.6.1 Types

List of deprecated types

- *rocsolver_int*
- *rocsolver_handle*
- *rocsolver_direction*
- *rocsolver_storev*
- *rocsolver_operation*
- *rocsolver_fill*
- *rocsolver_diagonal*
- *rocsolver_side*
- *rocsolver_status*

rocsolver_int

typedef rocblas_int **rocsolver_int**

Deprecated:
Use `rocblas_int`.

Deprecated since version 3.5: Use `rocblas_int`.

rocsolver_handle

typedef rocblas_handle **rocsolver_handle**

Deprecated:
Use rocblas_handle.

Deprecated since version 3.5: Use rocblas_handle.

rocsolver_direction

typedef *rocblas_direct* **rocsolver_direction**

Deprecated:
Use rocblas_direct

Deprecated since version 3.5: Use *rocblas_direct*.

rocsolver_storev

typedef *rocblas_storev* **rocsolver_storev**

Deprecated:
Use rocblas_storev.

Deprecated since version 3.5: Use *rocblas_storev*.

rocsolver_operation

typedef rocblas_operation **rocsolver_operation**

Deprecated:
Use rocblas_operation.

Deprecated since version 3.5: Use rocblas_operation.

rocsolver_fill

typedef rocblas_fill **rocsolver_fill**

Deprecated:
Use rocblas_fill.

Deprecated since version 3.5: Use rocblas_fill.

rocsolver_diagonal

typedef rocblas_diagonal **rocsolver_diagonal**

Deprecated:
Use rocblas_diagonal.

Deprecated since version 3.5: Use rocblas_diagonal.

rocsolver_side

typedef rocblas_side **rocsolver_side**

Deprecated:
Use `rocblas_stide`.

Deprecated since version 3.5: Use `rocblas_side`.

rocsolver_status

typedef rocblas_status **rocsolver_status**

Deprecated:
Use `rocblas_status`.

Deprecated since version 3.5: Use `rocblas_status`.

3.6.2 Auxiliary functions

List of deprecated helpers

- *rocsolver_create_handle()*
- *rocsolver_destroy_handle()*
- *rocsolver_set_stream()*
- *rocsolver_get_stream()*
- *rocsolver_set_vector()*
- *rocsolver_get_vector()*
- *rocsolver_set_matrix()*
- *rocsolver_get_matrix()*

rocsolver_create_handle()

rocsolver_status **rocsolver_create_handle**(*rocsolver_handle *handle*)

Deprecated:
Use `rocblas_create_handle`.

Deprecated since version 3.5: Use `rocblas_create_handle()`.

rocsolver_destroy_handle()

rocsolver_status **rocsolver_destroy_handle**(*rocsolver_handle handle*)

Deprecated:
Use `rocblas_destroy_handle`.

Deprecated since version 3.5: Use `rocblas_destroy_handle()`.

rocsolver_set_stream()

rocsolver_status **rocsolver_set_stream**(*rocsolver_handle handle*, hipStream_t *stream*)

Deprecated:
Use `rocblas_set_stream`.

Deprecated since version 3.5: Use `rocblas_set_stream()`.

rocsolver_get_stream()

rocsolver_status **rocsolver_get_stream**(*rocsolver_handle handle*, hipStream_t **stream*)

Deprecated:
Use `rocblas_get_stream`.

Deprecated since version 3.5: Use `rocblas_get_stream()`.

rocsolver_set_vector()

rocsolver_status **rocsolver_set_vector**(*rocsolver_int n*, *rocsolver_int elem_size*, **const** void **x*, *rocsolver_int incx*, void **y*, *rocsolver_int incy*)

Deprecated:
Use `rocblas_set_vector`.

Deprecated since version 3.5: Use `rocblas_set_vector()`.

rocsolver_get_vector()

rocsolver_status **rocsolver_get_vector**(*rocsolver_int n*, *rocsolver_int elem_size*, **const** void **x*, *rocsolver_int incx*, void **y*, *rocsolver_int incy*)

Deprecated:
Use `rocblas_get_vector`.

Deprecated since version 3.5: Use `rocblas_get_vector()`.

rocsolver_set_matrix()

rocsolver_status **rocsolver_set_matrix**(*rocsolver_int rows*, *rocsolver_int cols*, *rocsolver_int elem_size*, **const** void **a*, *rocsolver_int lda*, void **b*, *rocsolver_int ldb*)

Deprecated:

Use `rocblas_set_matrix`.

Deprecated since version 3.5: Use `rocblas_set_matrix()`.

rocsolver_get_matrix()

rocsolver_status **rocsolver_get_matrix**(*rocsolver_int rows*, *rocsolver_int cols*, *rocsolver_int elem_size*, **const** void **a*, *rocsolver_int lda*, void **b*, *rocsolver_int ldb*)

Deprecated:

Use `rocblas_get_matrix`.

Deprecated since version 3.5: Use `rocblas_get_matrix()`.

CHAPTER

FOUR

LICENSE & ATTRIBUTIONS

This product includes code derived from the LAPACK and MAGMA projects. Copyright holders for these projects are indicated below, and distributed under their license terms as specified.

– LAPACK –

– MAGMA –

INDEX

G

GEBRD_BLOCKSIZE (*C macro*), 33
GEBRD_GEBD2_SWITCHSIZE (*C macro*), 33
GEQxF_BLOCKSIZE (*C macro*), 30
GEQxF_GEQx2_SWITCHSIZE (*C macro*), 30
GExQF_BLOCKSIZE (*C macro*), 30
GExQF_GExQ2_SWITCHSIZE (*C macro*), 31

P

POTRF_BLOCKSIZE (*C macro*), 35
POTRF_POTF2_SWITCHSIZE (*C macro*), 36

R

rocblas_direct (*C enum*), 39
rocblas_direct.rocblas_backward_direction (*C enumerator*), 39
rocblas_direct.rocblas_forward_direction (*C enumerator*), 39
rocblas_eform (*C enum*), 41
rocblas_eform.rocblas_eform_abx (*C enumerator*), 41
rocblas_eform.rocblas_eform_ax (*C enumerator*), 41
rocblas_eform.rocblas_eform_bax (*C enumerator*), 41
rocblas_evect (*C enum*), 40
rocblas_evect.rocblas_evect_none (*C enumerator*), 40
rocblas_evect.rocblas_evect_original (*C enumerator*), 40
rocblas_evect.rocblas_evect_tridiagonal (*C enumerator*), 40
rocblas_storev (*C enum*), 40
rocblas_storev.rocblas_column_wise (*C enumerator*), 40
rocblas_storev.rocblas_row_wise (*C enumerator*), 40
rocblas_svect (*C enum*), 40
rocblas_svect.rocblas_svect_all (*C enumerator*), 40
rocblas_svect.rocblas_svect_none (*C enumerator*), 40
rocblas_svect.rocblas_svect_overwrite (*C enumerator*), 40
rocblas_svect.rocblas_svect_singular (*C enumerator*), 40
rocblas_workmode (*C enum*), 41
rocblas_workmode.rocblas_inplace (*C enumerator*), 41
rocblas_workmode.rocblas_outofplace (*C enumerator*), 41
rocsolver_cbdsqr (*C function*), 50
rocsolver_cgebd2 (*C function*), 140
rocsolver_cgebd2_batched (*C function*), 141
rocsolver_cgebd2_strided_batched (*C function*), 143
rocsolver_cgebrd (*C function*), 145
rocsolver_cgebrd_batched (*C function*), 146
rocsolver_cgebrd_strided_batched (*C function*), 148
rocsolver_cgelq2 (*C function*), 132
rocsolver_cgelq2_batched (*C function*), 133
rocsolver_cgelq2_strided_batched (*C function*), 134
rocsolver_cgelqf (*C function*), 135
rocsolver_cgelqf_batched (*C function*), 136
rocsolver_cgelqf_strided_batched (*C function*), 138
rocsolver_cgels (*C function*), 203
rocsolver_cgels_batched (*C function*), 204
rocsolver_cgels_strided_batched (*C function*), 205
rocsolver_cgeql2 (*C function*), 125
rocsolver_cgeql2_batched (*C function*), 126
rocsolver_cgeql2_strided_batched (*C function*), 127
rocsolver_cgeqlf (*C function*), 129
rocsolver_cgeqlf_batched (*C function*), 130
rocsolver_cgeqlf_strided_batched (*C function*), 131
rocsolver_cgeqr2 (*C function*), 111
rocsolver_cgeqr2_batched (*C function*), 112
rocsolver_cgeqr2_strided_batched (*C function*), 113

rocsolver_cgeqrf (*C function*), 115
rocsolver_cgeqrf_batched (*C function*), 116
rocsolver_cgeqrf_strided_batched (*C function*), 117
rocsolver_cgerq2 (*C function*), 118
rocsolver_cgerq2_batched (*C function*), 119
rocsolver_cgerq2_strided_batched (*C function*), 120
rocsolver_cgerqf (*C function*), 122
rocsolver_cgerqf_batched (*C function*), 123
rocsolver_cgerqf_strided_batched (*C function*), 124
rocsolver_cgesv (*C function*), 189
rocsolver_cgesv_batched (*C function*), 190
rocsolver_cgesv_strided_batched (*C function*), 192
rocsolver_cgesvd (*C function*), 238
rocsolver_cgesvd_batched (*C function*), 240
rocsolver_cgesvd_strided_batched (*C function*), 242
rocsolver_cgetf2 (*C function*), 94
rocsolver_cgetf2_batched (*C function*), 95
rocsolver_cgetf2_npvt (*C function*), 246
rocsolver_cgetf2_npvt_batched (*C function*), 247
rocsolver_cgetf2_npvt_strided_batched (*C function*), 248
rocsolver_cgetf2_strided_batched (*C function*), 96
rocsolver_cgetrf (*C function*), 97
rocsolver_cgetrf_batched (*C function*), 98
rocsolver_cgetrf_npvt (*C function*), 249
rocsolver_cgetrf_npvt_batched (*C function*), 250
rocsolver_cgetrf_npvt_strided_batched (*C function*), 251
rocsolver_cgetrf_strided_batched (*C function*), 99
rocsolver_cgetri (*C function*), 183
rocsolver_cgetri_batched (*C function*), 184
rocsolver_cgetri_npvt (*C function*), 252
rocsolver_cgetri_npvt_batched (*C function*), 253
rocsolver_cgetri_npvt_outofplace (*C function*), 259
rocsolver_cgetri_npvt_outofplace_batched (*C function*), 260
rocsolver_cgetri_npvt_outofplace_strided_batched (*C function*), 261
rocsolver_cgetri_npvt_strided_batched (*C function*), 254
rocsolver_cgetri_outofplace (*C function*), 255
rocsolver_cgetri_outofplace_batched (*C function*), 256
rocsolver_cgetri_outofplace_strided_batched (*C function*), 257
rocsolver_cgetri_strided_batched (*C function*), 185
rocsolver_cgetrs (*C function*), 186
rocsolver_cgetrs_batched (*C function*), 187
rocsolver_cgetrs_strided_batched (*C function*), 188
rocsolver_cheev (*C function*), 211
rocsolver_cheev_batched (*C function*), 212
rocsolver_cheev_strided_batched (*C function*), 213
rocsolver_cheevd (*C function*), 217
rocsolver_cheevd_batched (*C function*), 218
rocsolver_cheevd_strided_batched (*C function*), 219
rocsolver_chegs2 (*C function*), 169
rocsolver_chegs2_batched (*C function*), 170
rocsolver_chegs2_strided_batched (*C function*), 171
rocsolver_chegst (*C function*), 176
rocsolver_chegst_batched (*C function*), 177
rocsolver_chegst_strided_batched (*C function*), 178
rocsolver_chegv (*C function*), 224
rocsolver_chegv_batched (*C function*), 225
rocsolver_chegv_strided_batched (*C function*), 227
rocsolver_chegvd (*C function*), 233
rocsolver_chegvd_batched (*C function*), 234
rocsolver_chegvd_strided_batched (*C function*), 236
rocsolver_chetd2 (*C function*), 154
rocsolver_chetd2_batched (*C function*), 155
rocsolver_chetd2_strided_batched (*C function*), 156
rocsolver_chetrd (*C function*), 162
rocsolver_chetrd_batched (*C function*), 163
rocsolver_chetrd_strided_batched (*C function*), 164
rocsolver_clabrd (*C function*), 48
rocsolver_clacgv (*C function*), 42
rocsolver_clarf (*C function*), 45
rocsolver_clarfb (*C function*), 46
rocsolver_clarfg (*C function*), 43
rocsolver_clarft (*C function*), 44
rocsolver_claswp (*C function*), 42
rocsolver_clasyf (*C function*), 56
rocsolver_clatrd (*C function*), 51
rocsolver_cposv (*C function*), 199
rocsolver_cposv_batched (*C function*), 200
rocsolver_cposv_strided_batched (*C function*), 201

rocsolver_cpotf2 (*C function*), 88
rocsolver_cpotf2_batched (*C function*), 89
rocsolver_cpotf2_strided_batched (*C function*), 90
rocsolver_cpotrf (*C function*), 91
rocsolver_cpotrf_batched (*C function*), 92
rocsolver_cpotrf_strided_batched (*C function*), 93
rocsolver_cpotri (*C function*), 193
rocsolver_cpotri_batched (*C function*), 194
rocsolver_cpotri_strided_batched (*C function*), 195
rocsolver_cpotrs (*C function*), 196
rocsolver_cpotrs_batched (*C function*), 197
rocsolver_cpotrs_strided_batched (*C function*), 198
rocsolver_create_handle (*C function*), 267
rocsolver_cstedc (*C function*), 54
rocsolver_csteqr (*C function*), 53
rocsolver_csytf2 (*C function*), 100
rocsolver_csytf2_batched (*C function*), 102
rocsolver_csytf2_strided_batched (*C function*), 104
rocsolver_csytrf (*C function*), 105
rocsolver_csytrf_batched (*C function*), 107
rocsolver_csytrf_strided_batched (*C function*), 109
rocsolver_ctrtri (*C function*), 180
rocsolver_ctrtri_batched (*C function*), 181
rocsolver_ctrtri_strided_batched (*C function*), 182
rocsolver_cung2l (*C function*), 75
rocsolver_cung2r (*C function*), 72
rocsolver_cungbr (*C function*), 76
rocsolver_cungl2 (*C function*), 74
rocsolver_cunglq (*C function*), 74
rocsolver_cungql (*C function*), 76
rocsolver_cungqr (*C function*), 73
rocsolver_cungtr (*C function*), 77
rocsolver_cunm2l (*C function*), 83
rocsolver_cunm2r (*C function*), 78
rocsolver_cunmbr (*C function*), 85
rocsolver_cunml2 (*C function*), 80
rocsolver_cunmlq (*C function*), 82
rocsolver_cunmql (*C function*), 84
rocsolver_cunmqr (*C function*), 79
rocsolver_cunmtr (*C function*), 86
rocsolver_dbdsqr (*C function*), 50
rocsolver_destroy_handle (*C function*), 268
rocsolver_dgebd2 (*C function*), 140
rocsolver_dgebd2_batched (*C function*), 141
rocsolver_dgebd2_strided_batched (*C function*), 143
rocsolver_dgebrd (*C function*), 145
rocsolver_dgebrd_batched (*C function*), 146
rocsolver_dgebrd_strided_batched (*C function*), 148
rocsolver_dgelq2 (*C function*), 132
rocsolver_dgelq2_batched (*C function*), 133
rocsolver_dgelq2_strided_batched (*C function*), 134
rocsolver_dgelqf (*C function*), 135
rocsolver_dgelqf_batched (*C function*), 136
rocsolver_dgelqf_strided_batched (*C function*), 138
rocsolver_dgels (*C function*), 203
rocsolver_dgels_batched (*C function*), 204
rocsolver_dgels_strided_batched (*C function*), 205
rocsolver_dgeql2 (*C function*), 125
rocsolver_dgeql2_batched (*C function*), 126
rocsolver_dgeql2_strided_batched (*C function*), 127
rocsolver_dgeqlf (*C function*), 129
rocsolver_dgeqlf_batched (*C function*), 130
rocsolver_dgeqlf_strided_batched (*C function*), 131
rocsolver_dgeqr2 (*C function*), 111
rocsolver_dgeqr2_batched (*C function*), 112
rocsolver_dgeqr2_strided_batched (*C function*), 114
rocsolver_dgeqrf (*C function*), 115
rocsolver_dgeqrf_batched (*C function*), 116
rocsolver_dgeqrf_strided_batched (*C function*), 117
rocsolver_dgerq2 (*C function*), 118
rocsolver_dgerq2_batched (*C function*), 119
rocsolver_dgerq2_strided_batched (*C function*), 120
rocsolver_dgerqf (*C function*), 122
rocsolver_dgerqf_batched (*C function*), 123
rocsolver_dgerqf_strided_batched (*C function*), 124
rocsolver_dgesv (*C function*), 189
rocsolver_dgesv_batched (*C function*), 190
rocsolver_dgesv_strided_batched (*C function*), 192
rocsolver_dgesvd (*C function*), 238
rocsolver_dgesvd_batched (*C function*), 240
rocsolver_dgesvd_strided_batched (*C function*), 243
rocsolver_dgetf2 (*C function*), 94
rocsolver_dgetf2_batched (*C function*), 95
rocsolver_dgetf2_npvt (*C function*), 246
rocsolver_dgetf2_npvt_batched (*C function*), 247
rocsolver_dgetf2_npvt_strided_batched (*C function*), 248

rocsolver_dgetf2_strided_batched (*C function*), 96
rocsolver_dgetrf (*C function*), 97
rocsolver_dgetrf_batched (*C function*), 98
rocsolver_dgetrf_npvt (*C function*), 249
rocsolver_dgetrf_npvt_batched (*C function*), 250
rocsolver_dgetrf_npvt_strided_batched (*C function*), 251
rocsolver_dgetrf_strided_batched (*C function*), 99
rocsolver_dgetri (*C function*), 183
rocsolver_dgetri_batched (*C function*), 184
rocsolver_dgetri_npvt (*C function*), 252
rocsolver_dgetri_npvt_batched (*C function*), 253
rocsolver_dgetri_npvt_outofplace (*C function*), 259
rocsolver_dgetri_npvt_outofplace_batched (*C function*), 260
rocsolver_dgetri_npvt_outofplace_strided_batched (*C function*), 261
rocsolver_dgetri_npvt_strided_batched (*C function*), 254
rocsolver_dgetri_outofplace (*C function*), 255
rocsolver_dgetri_outofplace_batched (*C function*), 256
rocsolver_dgetri_outofplace_strided_batched (*C function*), 257
rocsolver_dgetri_strided_batched (*C function*), 185
rocsolver_dgetrs (*C function*), 186
rocsolver_dgetrs_batched (*C function*), 187
rocsolver_dgetrs_strided_batched (*C function*), 188
rocsolver_diagonal (*C type*), 266
rocsolver_direction (*C type*), 266
rocsolver_dlabrd (*C function*), 48
rocsolver_dlarf (*C function*), 46
rocsolver_dlarfb (*C function*), 46
rocsolver_dlarfg (*C function*), 43
rocsolver_dlarft (*C function*), 44
rocsolver_dlaswp (*C function*), 42
rocsolver_dlasyf (*C function*), 56
rocsolver_dlatrd (*C function*), 51
rocsolver_dorg2l (*C function*), 60
rocsolver_dorg2r (*C function*), 57
rocsolver_dorgbr (*C function*), 61
rocsolver_dorgl2 (*C function*), 59
rocsolver_dorglq (*C function*), 59
rocsolver_dorgql (*C function*), 61
rocsolver_dorgqr (*C function*), 58
rocsolver_dorgtr (*C function*), 62
rocsolver_dorm2l (*C function*), 67
rocsolver_dorm2r (*C function*), 63
rocsolver_dormbr (*C function*), 69
rocsolver_dorml2 (*C function*), 65
rocsolver_dormlq (*C function*), 66
rocsolver_dormql (*C function*), 68
rocsolver_dormqr (*C function*), 64
rocsolver_dormtr (*C function*), 70
rocsolver_dposv (*C function*), 199
rocsolver_dposv_batched (*C function*), 200
rocsolver_dposv_strided_batched (*C function*), 201
rocsolver_dpotf2 (*C function*), 88
rocsolver_dpotf2_batched (*C function*), 89
rocsolver_dpotf2_strided_batched (*C function*), 90
rocsolver_dpotrf (*C function*), 91
rocsolver_dpotrf_batched (*C function*), 92
rocsolver_dpotrf_strided_batched (*C function*), 93
rocsolver_dpotri (*C function*), 193
rocsolver_dpotri_batched (*C function*), 194
rocsolver_dpotri_strided_batched (*C function*), 195
rocsolver_dpotrs (*C function*), 196
rocsolver_dpotrs_batched (*C function*), 197
rocsolver_dpotrs_strided_batched (*C function*), 198
rocsolver_dstedc (*C function*), 54
rocsolver_dsteqr (*C function*), 53
rocsolver_dsterf (*C function*), 53
rocsolver_dsyev (*C function*), 208
rocsolver_dsyev_batched (*C function*), 209
rocsolver_dsyev_strided_batched (*C function*), 210
rocsolver_dsyevd (*C function*), 214
rocsolver_dsyevd_batched (*C function*), 215
rocsolver_dsyevd_strided_batched (*C function*), 216
rocsolver_dsygs2 (*C function*), 165
rocsolver_dsygs2_batched (*C function*), 167
rocsolver_dsygs2_strided_batched (*C function*), 168
rocsolver_dsygst (*C function*), 173
rocsolver_dsygst_batched (*C function*), 174
rocsolver_dsygst_strided_batched (*C function*), 175
rocsolver_dsygv (*C function*), 220
rocsolver_dsygv_batched (*C function*), 221
rocsolver_dsygv_strided_batched (*C function*), 223
rocsolver_dsygvd (*C function*), 229
rocsolver_dsygvd_batched (*C function*), 230

rocsolver_dsygvd_strided_batched (*C function*), 231
rocsolver_dsytd2 (*C function*), 150
rocsolver_dsytd2_batched (*C function*), 151
rocsolver_dsytd2_strided_batched (*C function*), 153
rocsolver_dsytf2 (*C function*), 100
rocsolver_dsytf2_batched (*C function*), 102
rocsolver_dsytf2_strided_batched (*C function*), 104
rocsolver_dsytrd (*C function*), 158
rocsolver_dsytrd_batched (*C function*), 159
rocsolver_dsytrd_strided_batched (*C function*), 160
rocsolver_dsytrf (*C function*), 105
rocsolver_dsytrf_batched (*C function*), 107
rocsolver_dsytrf_strided_batched (*C function*), 109
rocsolver_dtrtri (*C function*), 180
rocsolver_dtrtri_batched (*C function*), 181
rocsolver_dtrtri_strided_batched (*C function*), 182
rocsolver_fill (*C type*), 266
rocsolver_get_matrix (*C function*), 269
rocsolver_get_stream (*C function*), 268
rocsolver_get_vector (*C function*), 268
rocsolver_get_version_string (*C function*), 264
rocsolver_get_version_string_size (*C function*), 265
rocsolver_handle (*C type*), 266
rocsolver_int (*C type*), 265
rocsolver_log_begin (*C function*), 263
rocsolver_log_end (*C function*), 263
rocsolver_log_flush_profile (*C function*), 264
rocsolver_log_restore_defaults (*C function*), 264
rocsolver_log_set_layer_mode (*C function*), 263
rocsolver_log_set_max_levels (*C function*), 264
rocsolver_log_write_profile (*C function*), 264
rocsolver_operation (*C type*), 266
rocsolver_sbdsqr (*C function*), 50
rocsolver_set_matrix (*C function*), 269
rocsolver_set_stream (*C function*), 268
rocsolver_set_vector (*C function*), 268
rocsolver_sgebd2 (*C function*), 140
rocsolver_sgebd2_batched (*C function*), 141
rocsolver_sgebd2_strided_batched (*C function*), 143
rocsolver_sgebrd (*C function*), 145
rocsolver_sgebrd_batched (*C function*), 147
rocsolver_sgebrd_strided_batched (*C function*), 149
rocsolver_sgelq2 (*C function*), 132
rocsolver_sgelq2_batched (*C function*), 133
rocsolver_sgelq2_strided_batched (*C function*), 134
rocsolver_sgelqf (*C function*), 136
rocsolver_sgelqf_batched (*C function*), 137
rocsolver_sgelqf_strided_batched (*C function*), 138
rocsolver_sgels (*C function*), 203
rocsolver_sgels_batched (*C function*), 204
rocsolver_sgels_strided_batched (*C function*), 206
rocsolver_sgeql2 (*C function*), 125
rocsolver_sgeql2_batched (*C function*), 126
rocsolver_sgeql2_strided_batched (*C function*), 127
rocsolver_sgeqlf (*C function*), 129
rocsolver_sgeqlf_batched (*C function*), 130
rocsolver_sgeqlf_strided_batched (*C function*), 131
rocsolver_sgeqr2 (*C function*), 111
rocsolver_sgeqr2_batched (*C function*), 112
rocsolver_sgeqr2_strided_batched (*C function*), 114
rocsolver_sgeqrf (*C function*), 115
rocsolver_sgeqrf_batched (*C function*), 116
rocsolver_sgeqrf_strided_batched (*C function*), 117
rocsolver_sgerq2 (*C function*), 118
rocsolver_sgerq2_batched (*C function*), 119
rocsolver_sgerq2_strided_batched (*C function*), 120
rocsolver_sgerqf (*C function*), 122
rocsolver_sgerqf_batched (*C function*), 123
rocsolver_sgerqf_strided_batched (*C function*), 124
rocsolver_sgesv (*C function*), 189
rocsolver_sgesv_batched (*C function*), 191
rocsolver_sgesv_strided_batched (*C function*), 192
rocsolver_sgesvd (*C function*), 238
rocsolver_sgesvd_batched (*C function*), 240
rocsolver_sgesvd_strided_batched (*C function*), 243
rocsolver_sgetf2 (*C function*), 94
rocsolver_sgetf2_batched (*C function*), 95
rocsolver_sgetf2_npvt (*C function*), 246
rocsolver_sgetf2_npvt_batched (*C function*), 247
rocsolver_sgetf2_npvt_strided_batched (*C function*), 248

rocsolver_sgetf2_strided_batched (*C function*), 96
rocsolver_sgetrf (*C function*), 97
rocsolver_sgetrf_batched (*C function*), 98
rocsolver_sgetrf_npvt (*C function*), 249
rocsolver_sgetrf_npvt_batched (*C function*), 250
rocsolver_sgetrf_npvt_strided_batched (*C function*), 251
rocsolver_sgetrf_strided_batched (*C function*), 99
rocsolver_sgetri (*C function*), 183
rocsolver_sgetri_batched (*C function*), 184
rocsolver_sgetri_npvt (*C function*), 252
rocsolver_sgetri_npvt_batched (*C function*), 253
rocsolver_sgetri_npvt_outofplace (*C function*), 259
rocsolver_sgetri_npvt_outofplace_batched (*C function*), 260
rocsolver_sgetri_npvt_outofplace_strided_batched (*C function*), 261
rocsolver_sgetri_npvt_strided_batched (*C function*), 254
rocsolver_sgetri_outofplace (*C function*), 255
rocsolver_sgetri_outofplace_batched (*C function*), 256
rocsolver_sgetri_outofplace_strided_batched (*C function*), 257
rocsolver_sgetri_strided_batched (*C function*), 185
rocsolver_sgetrs (*C function*), 186
rocsolver_sgetrs_batched (*C function*), 187
rocsolver_sgetrs_strided_batched (*C function*), 188
rocsolver_side (*C type*), 267
rocsolver_slabrd (*C function*), 48
rocsolver_slarf (*C function*), 46
rocsolver_slarfb (*C function*), 47
rocsolver_slarfg (*C function*), 43
rocsolver_slarft (*C function*), 44
rocsolver_slaswp (*C function*), 42
rocsolver_slasyf (*C function*), 56
rocsolver_slatrd (*C function*), 52
rocsolver_sorg2l (*C function*), 60
rocsolver_sorg2r (*C function*), 57
rocsolver_sorgbr (*C function*), 61
rocsolver_sorgl2 (*C function*), 59
rocsolver_sorglq (*C function*), 59
rocsolver_sorgql (*C function*), 61
rocsolver_sorgqr (*C function*), 58
rocsolver_sorgtr (*C function*), 62
rocsolver_sorm2l (*C function*), 67
rocsolver_sorm2r (*C function*), 63
rocsolver_sormbr (*C function*), 69
rocsolver_sorml2 (*C function*), 65
rocsolver_sormlq (*C function*), 66
rocsolver_sormql (*C function*), 68
rocsolver_sormqr (*C function*), 64
rocsolver_sormtr (*C function*), 70
rocsolver_sposv (*C function*), 199
rocsolver_sposv_batched (*C function*), 200
rocsolver_sposv_strided_batched (*C function*), 201
rocsolver_spotf2 (*C function*), 88
rocsolver_spotf2_batched (*C function*), 89
rocsolver_spotf2_strided_batched (*C function*), 90
rocsolver_spotrf (*C function*), 91
rocsolver_spotrf_batched (*C function*), 92
rocsolver_spotrf_strided_batched (*C function*), 93
rocsolver_spotri (*C function*), 193
rocsolver_spotri_batched (*C function*), 194
rocsolver_spotri_strided_batched (*C function*), 195
rocsolver_spotrs (*C function*), 196
rocsolver_spotrs_batched (*C function*), 197
rocsolver_spotrs_strided_batched (*C function*), 198
rocsolver_sstedc (*C function*), 54
rocsolver_ssteqr (*C function*), 54
rocsolver_ssterf (*C function*), 53
rocsolver_ssyev (*C function*), 208
rocsolver_ssyev_batched (*C function*), 209
rocsolver_ssyev_strided_batched (*C function*), 210
rocsolver_ssyevd (*C function*), 214
rocsolver_ssyevd_batched (*C function*), 215
rocsolver_ssyevd_strided_batched (*C function*), 216
rocsolver_ssygs2 (*C function*), 165
rocsolver_ssygs2_batched (*C function*), 167
rocsolver_ssygs2_strided_batched (*C function*), 168
rocsolver_ssygst (*C function*), 173
rocsolver_ssygst_batched (*C function*), 174
rocsolver_ssygst_strided_batched (*C function*), 175
rocsolver_ssygv (*C function*), 220
rocsolver_ssygv_batched (*C function*), 221
rocsolver_ssygv_strided_batched (*C function*), 223
rocsolver_ssygvd (*C function*), 229
rocsolver_ssygvd_batched (*C function*), 230
rocsolver_ssygvd_strided_batched (*C function*), 231

rocsolver_ssytd2 (*C function*), 150
rocsolver_ssytd2_batched (*C function*), 151
rocsolver_ssytd2_strided_batched (*C function*), 153
rocsolver_ssytf2 (*C function*), 100
rocsolver_ssytf2_batched (*C function*), 102
rocsolver_ssytf2_strided_batched (*C function*), 104
rocsolver_ssytrd (*C function*), 158
rocsolver_ssytrd_batched (*C function*), 159
rocsolver_ssytrd_strided_batched (*C function*), 160
rocsolver_ssytrf (*C function*), 106
rocsolver_ssytrf_batched (*C function*), 107
rocsolver_ssytrf_strided_batched (*C function*), 109
rocsolver_status (*C type*), 267
rocsolver_storev (*C type*), 266
rocsolver_strtri (*C function*), 180
rocsolver_strtri_batched (*C function*), 181
rocsolver_strtri_strided_batched (*C function*), 182
rocsolver_zbdsqr (*C function*), 50
rocsolver_zgebd2 (*C function*), 140
rocsolver_zgebd2_batched (*C function*), 141
rocsolver_zgebd2_strided_batched (*C function*), 143
rocsolver_zgebrd (*C function*), 145
rocsolver_zgebrd_batched (*C function*), 146
rocsolver_zgebrd_strided_batched (*C function*), 148
rocsolver_zgelq2 (*C function*), 132
rocsolver_zgelq2_batched (*C function*), 133
rocsolver_zgelq2_strided_batched (*C function*), 134
rocsolver_zgelqf (*C function*), 135
rocsolver_zgelqf_batched (*C function*), 136
rocsolver_zgelqf_strided_batched (*C function*), 138
rocsolver_zgels (*C function*), 203
rocsolver_zgels_batched (*C function*), 204
rocsolver_zgels_strided_batched (*C function*), 205
rocsolver_zgeql2 (*C function*), 125
rocsolver_zgeql2_batched (*C function*), 126
rocsolver_zgeql2_strided_batched (*C function*), 127
rocsolver_zgeqlf (*C function*), 129
rocsolver_zgeqlf_batched (*C function*), 130
rocsolver_zgeqlf_strided_batched (*C function*), 131
rocsolver_zgeqr2 (*C function*), 111
rocsolver_zgeqr2_batched (*C function*), 112
rocsolver_zgeqr2_strided_batched (*C function*), 113
rocsolver_zgeqrf (*C function*), 115
rocsolver_zgeqrf_batched (*C function*), 116
rocsolver_zgeqrf_strided_batched (*C function*), 117
rocsolver_zgerq2 (*C function*), 118
rocsolver_zgerq2_batched (*C function*), 119
rocsolver_zgerq2_strided_batched (*C function*), 120
rocsolver_zgerqf (*C function*), 122
rocsolver_zgerqf_batched (*C function*), 123
rocsolver_zgerqf_strided_batched (*C function*), 124
rocsolver_zgesv (*C function*), 189
rocsolver_zgesv_batched (*C function*), 190
rocsolver_zgesv_strided_batched (*C function*), 192
rocsolver_zgesvd (*C function*), 238
rocsolver_zgesvd_batched (*C function*), 240
rocsolver_zgesvd_strided_batched (*C function*), 242
rocsolver_zgetf2 (*C function*), 94
rocsolver_zgetf2_batched (*C function*), 95
rocsolver_zgetf2_npvt (*C function*), 246
rocsolver_zgetf2_npvt_batched (*C function*), 247
rocsolver_zgetf2_npvt_strided_batched (*C function*), 248
rocsolver_zgetf2_strided_batched (*C function*), 96
rocsolver_zgetrf (*C function*), 97
rocsolver_zgetrf_batched (*C function*), 98
rocsolver_zgetrf_npvt (*C function*), 249
rocsolver_zgetrf_npvt_batched (*C function*), 250
rocsolver_zgetrf_npvt_strided_batched (*C function*), 251
rocsolver_zgetrf_strided_batched (*C function*), 99
rocsolver_zgetri (*C function*), 183
rocsolver_zgetri_batched (*C function*), 184
rocsolver_zgetri_npvt (*C function*), 252
rocsolver_zgetri_npvt_batched (*C function*), 253
rocsolver_zgetri_npvt_outofplace (*C function*), 259
rocsolver_zgetri_npvt_outofplace_batched (*C function*), 259
rocsolver_zgetri_npvt_outofplace_strided_batched (*C function*), 261
rocsolver_zgetri_npvt_strided_batched (*C function*), 254
rocsolver_zgetri_outofplace (*C function*),

255
rocsolver_zgetri_outofplace_batched (C function), 256
rocsolver_zgetri_outofplace_strided_batched (C function), 257
rocsolver_zgetri_strided_batched (C function), 185
rocsolver_zgetrs (C function), 186
rocsolver_zgetrs_batched (C function), 187
rocsolver_zgetrs_strided_batched (C function), 188
rocsolver_zheev (C function), 211
rocsolver_zheev_batched (C function), 212
rocsolver_zheev_strided_batched (C function), 213
rocsolver_zheevd (C function), 217
rocsolver_zheevd_batched (C function), 218
rocsolver_zheevd_strided_batched (C function), 219
rocsolver_zhegs2 (C function), 169
rocsolver_zhegs2_batched (C function), 170
rocsolver_zhegs2_strided_batched (C function), 171
rocsolver_zhegst (C function), 176
rocsolver_zhegst_batched (C function), 177
rocsolver_zhegst_strided_batched (C function), 178
rocsolver_zhegv (C function), 224
rocsolver_zhegv_batched (C function), 225
rocsolver_zhegv_strided_batched (C function), 227
rocsolver_zhegvd (C function), 233
rocsolver_zhegvd_batched (C function), 234
rocsolver_zhegvd_strided_batched (C function), 236
rocsolver_zhetd2 (C function), 154
rocsolver_zhetd2_batched (C function), 155
rocsolver_zhetd2_strided_batched (C function), 156
rocsolver_zhetrd (C function), 162
rocsolver_zhetrd_batched (C function), 163
rocsolver_zhetrd_strided_batched (C function), 164
rocsolver_zlabrd (C function), 48
rocsolver_zlacgv (C function), 42
rocsolver_zlarf (C function), 45
rocsolver_zlarfb (C function), 46
rocsolver_zlarfg (C function), 43
rocsolver_zlarft (C function), 44
rocsolver_zlaswp (C function), 42
rocsolver_zlasyf (C function), 56
rocsolver_zlatrd (C function), 51
rocsolver_zposv (C function), 199
rocsolver_zposv_batched (C function), 200
rocsolver_zposv_strided_batched (C function), 201
rocsolver_zpotf2 (C function), 88
rocsolver_zpotf2_batched (C function), 89
rocsolver_zpotf2_strided_batched (C function), 90
rocsolver_zpotrf (C function), 91
rocsolver_zpotrf_batched (C function), 92
rocsolver_zpotrf_strided_batched (C function), 93
rocsolver_zpotri (C function), 193
rocsolver_zpotri_batched (C function), 194
rocsolver_zpotri_strided_batched (C function), 195
rocsolver_zpotrs (C function), 196
rocsolver_zpotrs_batched (C function), 197
rocsolver_zpotrs_strided_batched (C function), 198
rocsolver_zstedc (C function), 54
rocsolver_zsteqr (C function), 53
rocsolver_zsytf2 (C function), 100
rocsolver_zsytf2_batched (C function), 102
rocsolver_zsytf2_strided_batched (C function), 104
rocsolver_zsytrf (C function), 105
rocsolver_zsytrf_batched (C function), 107
rocsolver_zsytrf_strided_batched (C function), 109
rocsolver_ztrtri (C function), 180
rocsolver_ztrtri_batched (C function), 181
rocsolver_ztrtri_strided_batched (C function), 182
rocsolver_zung2l (C function), 75
rocsolver_zung2r (C function), 72
rocsolver_zungbr (C function), 76
rocsolver_zungl2 (C function), 74
rocsolver_zunglq (C function), 74
rocsolver_zungql (C function), 76
rocsolver_zungqr (C function), 73
rocsolver_zungtr (C function), 77
rocsolver_zunm2l (C function), 83
rocsolver_zunm2r (C function), 78
rocsolver_zunmbr (C function), 85
rocsolver_zunml2 (C function), 80
rocsolver_zunmlq (C function), 82
rocsolver_zunmql (C function), 84
rocsolver_zunmqr (C function), 79
rocsolver_zunmtr (C function), 86

S

STEDC_MIN_DC_SIZE (C macro), 35
SYTRF_BLOCKSIZE (C macro), 36
SYTRF_SYTF2_SWITCHSIZE (C macro), 36

T

THIN_SVD_SWITCH (*C macro*), 34

X

xxGQx_BLOCKSIZE (*C macro*), 31
xxGQx_xxGQx2_SWITCHSIZE (*C macro*), 31
xxGST_BLOCKSIZE (*C macro*), 35
xxGxQ_BLOCKSIZE (*C macro*), 32
xxGxQ_xxGxQ2_SWITCHSIZE (*C macro*), 32
xxMQx_BLOCKSIZE (*C macro*), 32
xxMxQ_BLOCKSIZE (*C macro*), 33
xxTRD_BLOCKSIZE (*C macro*), 34
xxTRD_xxTD2_SWITCHSIZE (*C macro*), 34