CULA Programmer's Guide

www.culatools.com

Release R17 (CUDA 5.0)

EM Photonics, Inc.
www.emphotonics.com

May 07, 2013

CONTENTS

1 Introduction **1**
- 1.1 Versions . . . 1
- 1.2 CULA Dense Free Edition Functions . . . 2
- 1.3 CULA Dense Functions . . . 2
- 1.4 Supported Operating Systems . . . 5
- 1.5 Attributions . . . 5

2 License **6**

3 Getting Started **14**
- 3.1 Obtaining CULA . . . 14
- 3.2 System Requirements . . . 14
- 3.3 Installation . . . 14
- 3.4 Compiling with CULA . . . 15
- 3.5 Linking to CULA . . . 15
- 3.6 Uninstallation . . . 16

4 Differences Between CULA and LAPACK **17**
- 4.1 Naming Conventions . . . 17
- 4.2 Calling Conventions . . . 18
- 4.3 Data Type Support . . . 19
- 4.4 Error Handling . . . 19
- 4.5 Workspaces . . . 19

5 Programming Considerations **21**
- 5.1 Matrix Storage . . . 21
- 5.2 Optimizing for Performance . . . 23
- 5.3 Using the Device Interface . . . 24
- 5.4 Thread Safety/Multi-GPU Operation . . . 24
- 5.5 Developing in C++ . . . 25

6 Examples **26**
- 6.1 Initialization and Shutdown . . . 26
- 6.2 Argument Errors . . . 26
- 6.3 Data Errors . . . 27
- 6.4 Printing Errors to the Console . . . 27
- 6.5 Using the C++ Interface . . . 27
- 6.6 Checking That Libraries are Linked Correctly . . . 28

7 Programming in Fortran **29**

7.1 Introduction . . . 29
7.2 Routine Naming . . . 29
7.3 Module File Interfacing . . . 29
7.4 Using the CULA C-Style Device Memory Interface . . . 30
7.5 Using the CULA CUDA-Fortran Device Memory Interface . . . 30

8 Configuring Your Environment **31**
8.1 Microsoft Visual Studio . . . 31
8.2 Linux / Mac OS X - Command Line . . . 32

9 Troubleshooting **34**
9.1 Common Issues . . . 34
9.2 Support Options . . . 36

10 Changelog **37**
10.1 Release R17 CUDA 5.0 (May 8, 2013) . . . 37
10.2 Release R16a CUDA 5.0 (November 20, 2012) . . . 37
10.3 Release R16 CUDA 5.0 (October 16, 2012) . . . 37
10.4 Release R15 CUDA 4.2 (August 14, 2012) . . . 38
10.5 Release R14 CUDA 4.1 (January 30, 2012) . . . 38
10.6 Release R13 CUDA 4.0 (November 2, 2011) . . . 39
10.7 Release R12 CUDA 4.0 (May 26, 2011) . . . 39
10.8 Release R11 CUDA 3.2 (March 31, 2011) . . . 40
10.9 Release R10 CUDA 3.2 (December 10, 2010) . . . 40
10.10 Release 2.1 Final (August 31, 2010) . . . 41
10.11 Release 2.0 Final (June 28, 2010) . . . 41
10.12 Release 2.0 Preview (May 21, 2010) . . . 41
10.13 Release 1.3a Final (April 19, 2010) . . . 42
10.14 Release 1.3 Final (April 8, 2010) . . . 42
10.15 Release 1.2 Final (February 17, 2010) . . . 42
10.16 Release 1.1b Final (January 6, 2009) . . . 43
10.17 Release 1.1a Final (December 21, 2009) . . . 43
10.18 Release 1.1 Final (November 25, 2009) . . . 44
10.19 Release 1.1 Beta (November 13, 2009) . . . 44
10.20 Release 1.0 Final (September 30, 2009) . . . 45
10.21 Release 1.0 Beta 3 (September 15, 2009) . . . 45
10.22 Release 1.0 Beta 2 (August 27, 2009) . . . 45
10.23 Release 1.0 Beta 1 (August 13, 2009) . . . 46

CHAPTER

ONE

INTRODUCTION

This document describes CULA™, an implementation of the Linear Algebra PACKage (LAPACK) interface for *CUDA*™-enabled NVIDIA® graphics processing units (GPUs).

CULA is a next-generation linear algebra package that uses the GPU as a co-processor to achieve speedups over existing linear algebra packages. CULA provides the same functionality you receive with your existing package, only at a greater speed.

CULA provides easy access to the NVIDIA computing resources available in your computer system. The library is a self-contained package that enhances linear algebra programs with little to no knowledge of the GPU computing model. CULA presents four distinct interfaces:

- **Standard** - The functions in this interface work on data in main memory, as in the Netlib LAPACK interface, but with semantics more familiar to the C and C++ programmer.
- **Device** - This interface follows the standards set forth in the NVIDIA *CUBLAS* package. In this interface, the user allocates and populates GPU memory and then calls CULA functions to operate on that memory. This is useful for closed-loop GPU systems that do large sections of processing on the GPU.
- **Fortran** - This functions of this interface match the **Standard** and **Device** interfaces but use Fortran calling conventions.
- **Link** - This interface is targeted at users who are porting existing linear algebra codes to a GPU-accelerated environment. This interface provides a migration path by matching the function names and signatures of several popular linear algebra packages. It additionally provides a fallback to CPU execution when a user does not have a GPU or when problem size is too small to take advantage of GPU execution.

For maximum compatibility with the conventions of existing linear algebra packages, CULA uses column-major data storage and 1-based indexing, even on our C-oriented interfaces. More details are available in the *Programming Considerations* chapter.

1.1 Versions

The official release of CULA is available in several forms.

- **Dense Free Edition** - The basic package contains a basic set of functions and is free of cost. This version supports single-precision real and complex data types.
- **Dense** - The standard package is a low-cost solution that provides a greater set of functionality than the **Free Edition** package and includes all data precisions. This package is intended for institutional and academic use.

The following table summarizes the features of these three versions:

	Price	Precisions	Functions	Redistributable	Support
Dense Free Edition	Free	S, C	6 (see below)	Yes	Public forums
Dense	See website	S, D, C, Z	Many (see below)	For internal use	Ticket system
Dense for Redistribution	See website	S, D, C, Z	Many	Yes	Email & phone

Precisions in the table above are defined as follows (more details available later): *S* is single precision real, *D* is double precision real, *C* is single precision complex, and *Z* is double precision complex.

1.2 CULA Dense Free Edition Functions

CULA Dense Free Edition implements popular single precision real and complex functions from LAPACK. Included are the following routines:

Operations	S	C
Factorize and solve	SGESV	CGESV
LU factorization	SGETRF	CGETRF
QR factorization	SGEQRF	CGEQRF
General least squares solve	SGELS	CGELS
Equality-constrained least squares	SGGLSE	CGGLSE
General Singular Value Decomposition	SGESVD	CGESVD

All versions of CULA also include symbols to NVIDIA's *CUBLAS* functions for Level 3 BLAS routines (see the *cula_blas.h* and *cula_blas_device.h* headers). These symbols are present so that CULA may be used as a stand-alone linear algebra package without requiring several other packages to provide a capable development system. Additionally, in many cases, we have implemented performance tweaks to get even more performance out of these functions. Like our LAPACK functions, all of these BLAS Level 3 functions are supported in our **Standard**, **Device**, **Fortran**, and **Link** interfaces. Included are the following routines:

Matrix Type	Operation	S	C	D	Z
General	Matrix-matrix multiply	SGEMM	CGEMM	DGEMM	ZGEMM
	Matrix-vector multiply	SGEMV	CGEMV	DGEMV	ZGEMV
Triangular	Triangular matrix-matrix multiply	STRMM	CTRMM	DTRMM	ZTRMM
	Triangular matrix solve	STRSM	CTRSM	DTRSM	ZTRSM
Symmetric	Symmetric matrix-matrix multiply	SSYMM	CSYMM	DSYMM	ZSYMM
	Symmetric rank 2k update	SSYR2K	CSYR2K	DSYR2K	ZSYR2K
	Symmetric rank k update	SSYRK	CSYRK	DSYRK	ZSYRK
Hermitian	Hermitian matrix-matrix multiply		CHEMM		ZHEMM
	Hermitian rank 2k update		CHER2K		ZHER2K
	Hermitian rank k update		CHERK		ZHERK

1.3 CULA Dense Functions

CULA's standard Dense edition implements a much larger set of functions from LAPACK. As CULA evolves, more functions and function variants will be supported.

1.3.1 Linear Equations

CULA contains the following LAPACK function equivalents from the linear equations family of computational routines:

Matrix Type	Operation	S	C	D	Z
General	Factorize and solve	SGESV	CGESV	DGESV	ZGESV
	Factorize and solve with iterative refinement			DSGESV	ZCGESV
	LU factorization	SGETRF	CGETRF	DGETRF	ZGETRF
	Solve using LU factorization	SGETRS	CGETRS	DGETRS	ZGETRS
	Invert using LU factorization	SGETRI	CGETRI	DGETRI	ZGETRI
Positive Definite	Factorize and solve	SPOSV	CPOSV	DPOSV	ZPOSV
	Cholesky factorization	SPOTRF	CPOTRF	DPOTRF	ZPOTRF
Triangular	Invert triangular matrix	STRTRI	CTRTRI	DTRTRI	ZTRTRI
	Solve triangular system	STRTRS	CTRTRS	DTRTRS	ZTRTRS
Banded	LU factorization	SGBTRF	CGBTRF	DGBTRF	ZGBTRF
Pos Def Banded	Cholesky factorization	SPBTRF	CPBTRF	DPBTRF	ZPBTRF

1.3.2 Orthogonal Factorizations

CULA contains the following LAPACK function equivalents from the orthogonal factorization family of computational routines:

Matrix Type	Operation	S	C	D	Z
General	QR factorization	SGEQRF	CGEQRF	DGEQRF	ZGEQRF
	QR factorize and solve	SGEQRS	CGEQRS	DGEQRS	ZGEQRS
	Generate Q from QR factorization	SORGQR	CUNGQR	DORGQR	ZUNGQR
	Multiply matrix by Q from QR factorization	SORMQR	CUNMQR	DORMQR	ZUNMQR
General	LQ factorization	SGELQF	CGELQF	DGELQF	ZGELQF
	Generate Q from LQ factorization	SORGLQ	CUNGLQ	DORGLQ	ZUNGLQ
	Multiply matrix by Q from LQ factorization	SORMLQ	CUNMLQ	DORMLQ	ZUNMLQ
General	RQ factorization	SGERQF	CGERQF	DGERQF	ZGERQF
	Multiply matrix by Q from RQ factorization	SORMRQ	CUNMRQ	DORMRQ	ZUNMRQ
General	QL factorization	SGEQLF	CGEQLF	DGEQLF	ZGEQLF
	Generate Q from QL factorization	SORGQL	CUNGQL	DORGQL	ZUNGQL
	Multiply matrix by Q from QL factorization	SORMQL	CUNMQL	DORMQL	ZUNMQL

1.3.3 Least Squares Problems

CULA contains the following LAPACK function equivalents from the least squares solver family of computational routines:

Matrix Type	Operation	S	C	D	Z
General	General least squares solve	SGELS	CGELS	DGELS	ZGELS
	Equality-constrained least squares	SGGLSE	CGGLSE	DGGLSE	ZGGLSE

1.3.4 Symmetric Eigenproblems

CULA contains the following LAPACK function equivalents from the symmetric Eigenproblem family of computational routines.

Matrix Type	Operation	S	C	D	Z
Symmetric	Symmetric Eigenproblem solver	SSYEV	CHEEV	DSYEV	ZHEEV
	Symmetric Eigenproblem solver (expert)	SSYEVX	CHEEVX	DSYEVX	ZHEEVX
	Symmetric band reduction	SSYRDB	CHERDB	DSYRDB	ZHERDB
Tridiagonal	Find eigenvalues using bisection	SSTEBZ		DSTEBZ	
	Find eigenvalues using QR/QL iteration	SSTEQR	CSTEQR	DSYRDB	ZSTEQR

1.3.5 Non-Symmetric Eigenproblems

CULA contains the following LAPACK function equivalents from the non-symmetric Eigenproblem family of computational routines:

Matrix Type	Operation	S	C	D	Z
General	General Eigenproblem solver	SGEEV	CGEEV	DGEEV	ZGEEV
	Hessenberg reduction	SGEHRD	CGEHRD	DGEHRD	ZGEHRD
	Generate Q from Hessenberg reduction	SORGHR	CUNGHR	DORGHR	ZUNGHR

1.3.6 Generalized Eigenproblems

CULA contains the following LAPACK function equivalents from the generalized Eigenproblem family of computational routines.

Matrix Type	Operation	S	C	D	Z
Symmetric	Symmetric Generalized Eigenproblem solver	SSYGV	CHEGV	DSYGV	ZHEGV

1.3.7 Singular Value Decomposition

CULA contains the following LAPACK function equivalents from the Singular Value Decomposition family of computational routines:

Matrix Type	Operation	S	C	D	Z
General	General Singular Value Decomposition	SGESVD	CGESVD	DGESVD	ZGESVD
	Bidiagonal reduction	SGEBRD	CGEBRD	DGEBRD	ZGEBRD
	Generate Q from bidiagonal reduction	SORGBR	CUNGBR	DORGBR	ZUNGBR
Bidiagonal	Find singular values and vectors	SBDSQR	CBDSQR	DBDSQR	ZBDSQR

1.3.8 Auxiliary

CULA contains the following LAPACK function equivalents from the Auxiliary family of computational routines:

Matrix Type	Operation	S	C	D	Z
General	Copy from one Matrix into another	SLACPY	CLACPY	DLACPY	ZLACPY
	Convert a matrix's precision	SLAG2D	CLAG2Z	DLAG2S	ZLAG2D
	Apply a block reflector to a matrix	SLARFB	CLARFB	DLARFB	ZLARFB
	Generate an elementary reflector	SLARFG	CLARFG	DLARFG	ZLARFG
	Generate a vector of plane rotations	SLARGV	CLARGV	DLARGV	ZLARGV
	Apply a vector of plane rotations	SLARTV	CLARTV	DLARTV	ZLARTV
	Multiple a matrix by a scalar	SLASCL	CLASCL	DLASCL	ZLASCL
	Initialize a matrix	SLASET	CLASET	DLASET	ZLASET
	Apply a sequence of plane rotations	SLASR	CLASR	DLASR	ZLASR
Symmetric	Apply a vector of plane rotations	SLAR2V	CLAR2V	DLAR2V	ZLAR2V
Triangular	Triangular precision conversion	SLAT2D	CLAT2Z	DLAT2S	DLAT2Z

1.4 Supported Operating Systems

CULA intends to support the full range of operating systems that are supported by *CUDA*. Installers are currently available for Windows, Linux, and MAC OS X in 32-bit and 64-bit versions. *CULA* has been tested on the following systems:

- Windows XP / Vista / 7
- Ubuntu Linux 10.04 (and newer)
- Red Hat Enterprise Linux 5.7 (and newer)
- Fedora 16
- Mac OSX 10.6 Snow Leopard / 10.7 Lion

Please provide feedback on any other systems on which you attempt to use *CULA*. Although we are continually testing *CULA* on other systems, at present we officially support the above list. If your system is not listed, please let us know through the provided feedback channels.

1.5 Attributions

This work has been made possible by the NASA Small Business Innovation Research (SBIR) program. We recognize NVIDIA for their support.

CULA is built on NVIDIA *CUDA* and NVIDIA *CUBLAS*. For more information, please see the *CUDA* product page at http://www.nvidia.com/object/cuda_home.html.

CULA uses the Intel® Math Kernel Library (MKL) internally. For more information, please see the MKL product page at http://www.intel.com/software/products/mkl.

The original version of LAPACK from which CULA implements a similar interface can be obtained at http://www.netlib.org/lapack.

CHAPTER

TWO

LICENSE

EM Photonics, Inc.

Software License Agreement

CULA(TM) DENSE FREE EDITION or CULA DENSE or CULA SPARSE

IMPORTANT - READ BEFORE COPYING, INSTALLING, USING OR DISTRIBUTING

This is a legal agreement (“Agreement”) between either a Subscriber or a Licensee, the party licensing either CULA Dense Free Edition or CULA Dense or CULA Sparse, and EM PHOTONICS Inc., a Delaware corporation with its principal place of business at 51 East Main Street, Newark, Delaware, 19711 (“Licensor”). BY CLICKING ON THE “AGREE” BUTTON BELOW AND PRESSING THE ENTER KEY, YOU ACKNOWLEDGE THAT YOU HAVE READ ALL OF THE TERMS AND CONDITIONS OF THIS AGREEMENT, UNDERSTAND THEM, AND AGREE TO BE LEGALLY BOUND BY THEM. If you do not agree with the terms of this Agreement, you may not download, install, use or distribute either CULA Dense Free Edition or CULA Dense or CULA Sparse, as applicable.

1 SCOPE; DEFINITIONS. This Agreement governs your rights and obligations surrounding the permitted download, installation, use and, if applicable, distribution of either CULA Dense Free Edition or CULA Dense or CULA Sparse. Unless otherwise defined in this Section 1, the capitalized terms used in this Agreement shall be defined in the context in which they are used. The following terms shall have the following meanings:

1.1 “Commercial Purpose” means the use, reproduction or distribution, directly or indirectly, of the Software, or any portion of the foregoing, that is intended to result in a direct or indirect pecuniary gain or any other consideration or economic benefit to any person or entity involved in such use, reproduction or distribution. Examples of a Commercial Purpose, include without limitation, (v) integrating the Software with other software or hardware for sale, (w) licensing the Software for a fee, (x) using the Software to provide a service to a third party, (y) selling the Software, or (z) distributing the Software for use with other products or other services.

1.2 “CULA Dense Free Edition” means Licensor’s limited, pre-complied implementation of linear algebra routines for certain third party graphics processing units.

1.3 “CULA Dense” means Licensor’s more expanded, pre-compiled implementation of linear algebra routines for certain third party graphic processing units.

1.4 “CULA Sparse” means Licensor’s pre-compiled implementation of sparse linear algebra routines for certain third party graphic processing units.

1.5 “Licensee” shall mean an individual or entity who has registered as a Licensee on www.culatools.com/register to use CULA Dense or CULA Sparse and who has paid the applicable license fees for such use.

1.6 “Intellectual Property Rights” shall mean all proprietary rights, including all patents, trademarks, copyrights, know-how, trade secrets, mask works, including all applications and registrations thereto, and any other similar protected rights in any country.

1.7 “Software” means collectively the CULA Dense Free Edition and the CULA Dense and the CULA Sparse.

1.8 “Subscriber” shall mean an individual or entity who has registered as a subscriber on www.culatools.com/register to use CULA Dense Free Edition.

2 GRANT OF LICENSE.

2.1 CULA Dense Free Edition License Grant. Only a Subscriber may exercise the rights under this Section 2.1. Subject to the terms and conditions of this Agreement, Licensor hereby grants to such Subscriber a world-wide, non-transferable, non sub-licensable, non-exclusive, perpetual license to do any of the following with respect to CULA Dense Free Edition in each case, in accordance with the Documentation and the system minimum user requirements:

1. download, install and use CULA Dense Free Edition for any purpose;
2. distribute certain unmodified CULA Dense Free Edition files identified on Exhibit A (the “Distributable Files”) on a limited stand alone basis to any third party;
3. download and use the Documentation and
4. reproduce CULA Dense Free Edition and/or the Documentation as strictly necessary in exercising its rights under this Section 2.1.

2.2 CULA Dense License Grant. Subject to the terms and conditions of this Agreement, Licensor hereby grants each Licensee a world-wide, non-transferable, non sub-licensable, non-exclusive, perpetual license to do any of the following with respect to CULA Dense:

1. download, install and use CULA Dense only for internal use;
2. reproduce CULA Dense and/or the Documentation as strictly necessary in exercising its rights under this Section 2.2.
3. You as an individual may install and use the CULA Dense on an unlimited number of computers provided that only one copy of the CULA Dense is in use at any one time. A separate license is required for each additional use in all other cases. If you are an entity, you may designate one individual within your organization to have the sole right to use the CULA Dense in the manner provided above.

2.3 CULA Sparse License Grant. Subject to the terms and conditions of this Agreement, Licensor hereby grants each Licensee a world-wide, non-transferable, non sub-licensable, non-exclusive, perpetual license to do any of the following with respect to CULA Sparse:

1. download, install and use CULA Sparse only for internal use;
2. reproduce CULA Sparse and/or the Documentation as strictly necessary in exercising its rights under this Section 2.3.
3. You as an individual may install and use the CULA Sparse on an unlimited number of computers provided that only one copy of the CULA Sparse is in use at any one time. A separate license is required for each additional use in all other cases. If you are an entity, you may designate one individual within your organization to have the sole right to use the CULA Sparse in the manner provided above.

2.4 Limitations on License; Restricted Activities. Each of the Subscribers and/or Licensees recognize and agree that the Software is the property of Licensor, contain valuable assets and proprietary information and property of Licensor, and are provided to such Subscriber or Licensee, as the case may be, under the terms and conditions of this Agreement. Notwithstanding anything to the contrary in this Agreement, each

Subscriber and/or Licensee agrees that he, she or it shall not do any of the following without Licensor's prior written consent:

1. Download, use, install, deploy, perform, modify, license, display, reproduce, distribute or disclose the Software other than as allowed under this Agreement;
2. sell, license, transfer, rent, loan, perform, modify, reproduce, distribute or disclose the Software (in whole or in part and whether done independently or as part of a compilation) for a Commercial Purpose;
3. post or make generally available the Software (in whole or in part) to individuals or a group of individuals who have not agreed to the terms and conditions of this Agreement;
4. share any license key or authentication information provided to a Licensee by Licensor with any third party to allow such party to access the Software; and
5. alter or remove any copyright notice or proprietary legend contained in or on the Software.

Paragraphs (a) though (e) of this Section 2.4 are collectively referred to as the "Restricted Activities").

2.5 Reproduction Obligations. Each Subscriber and Licensee agrees that any copy or distribution of the Software permitted under this Agreement shall contain the notices set forth in Exhibit A. In addition, to the extent a Licensee makes any copies of the Software or Documentation under this Agreement, each Subscriber and/or Licensee agrees to ensure that any and all such copies shall contain:

1. a copy of an appropriate copyright notice and all other applicable proprietary legends;
2. a disclaimer of any warranty consistent with this Agreement; and
3. any and all notices referencing this Agreement and absence of warranties.

2.6 Distribution Obligations. The Distributable Files may be distributed pursuant to Section 2.1(b), and any use of such Distributable Files by a recipient must be governed by the terms and conditions of this Agreement. Each Subscriber must include a copy of this Agreement with every copy of the Distributable Files it distributes. Under no circumstance may a Subscriber or Licensee distribute CULA Dense or CULA Sparse or any files comprising CULA Dense Free Edition not identified on Exhibit A. Each Subscriber must duplicate the notice in Exhibit A with the Distributable Files in a location (such as a relevant directory) where a user would be likely to look for such a notice.

3 Ownership of Software and Intellectual Property Rights. All right, title and interest to the Software and the Documentation, and all copies thereof, are and shall remain the exclusive property of Licensor and/or its licensors or suppliers. The Software is copyrighted and protected by the laws of the United States and other countries, and international treaty provisions. Licensor may make changes to the Software, or to items referenced therein, at any time and without notice, and Licensor is not obligated to support and/or update the Software or the Documentation unless otherwise agreed to herein. Except as otherwise expressly provided, Licensor grants no express or implied right or license (whether by implication, inducement, estoppel or otherwise) under any Licensor patents, copyrights, trademarks, or other intellectual property rights.

3.1 Feedback. While neither Party is required to provide the other party any suggestions, comments or other feedback regarding the Software, the Documentation or a Subscriber's and/or Licensee's use or implementation of the Software and/or Documentation ("Feedback"), to the extent a Subscriber or a Licensee provides Feedback to the Licensor, Licensor may use and include any Feedback so provided to improve the Software or other Licensor technologies and any new features, functionality, or performance based upon the Feedback that Licensor subsequently incorporates into its products shall be the sole and exclusive property of Licensor. Accordingly, if a Subscriber and/or Licensee provide Feedback, such Subscriber and/or Licensee hereby agrees that Licensor may freely use, reproduce, license, distribute, and otherwise commercialize the Feedback in the Software or other related technologies without the payment of any royalties or fees.

3.2 No Reverse Engineering and other Restrictions. In addition to agreeing to restrictions in Section 2.4 above, each Subscriber and/or Licensee shall not directly or indirectly: (i) sell, lease, redistribute or transfer any of the Software or Documentation; (ii) modify, translate, reverse engineer (except to the limited extent permitted by law), decompile, disassemble, create derivative works based on, sublicense, or distribute any of the Software; (iii) rent or lease any rights in any of the Software or Documentation in any form to any person; (iv) use any Software for the benefit of any third parties (e.g., in an ASP, outsourcing or service bureau relationship) or in any way other than in its intended manner; (v) remove, alter or obscure any proprietary or copyright notice, labels, or marks on the hardware components of the Software or Documentation; or (vi) disable or circumvent any access control or related security measure, process or procedure established with respect to the Software or any other part thereof. Each Subscriber and/or Licensee is responsible for all use of the Software and the Documentation and any downloading, installing and using the Software and for compliance with this Agreement; any breach by any user shall be deemed to have been made by the applicable Subscriber and/or Licensee.

4 Third Party Licenses. This Software includes third-party software, listed in Exhibit B, that is governed by a separate license agreement. By using the CULA software and accepting this Agreement, you additionally agree to adhere to the license agreements for each of the software products. Where required by a third-party license, source code for these products are made available within this Software package.

5 Support.

1. A Licensee may subscribe to Licensor's CULA Dense maintenance and support program by paying Licensor the then-applicable annual maintenance and support fee (the "Support Fee"). Upon payment of the Support Fee, Licensor shall provide Licensee with the applicable level of maintenance and support services set forth in the support program. Any CULA Dense updates provided to Licensee pursuant to the support program shall be deemed part of the CULA Dense and shall be licensed under the terms and conditions of the CULA Dense.
2. A Licensee may subscribe to Licensor's CULA Sparse maintenance and support program by paying Licensor the then-applicable annual maintenance and support fee (the "Support Fee"). Upon payment of the Support Fee, Licensor shall provide Licensee with the applicable level of maintenance and support services set forth in the support program. Any CULA Sparse updates provided to Licensee pursuant to the support program shall be deemed part of the CULA Sparse and shall be licensed under the terms and conditions of the CULA Sparse.

6 Payments. Licensee agrees to pay amounts invoiced by Licensor for any CULA Dense and/or CULA Sparse made available pursuant to this Agreement. If any authority imposes a duty, tax or similar levy (other than taxes based on Licensor's income), Licensee agrees to pay, or to promptly reimburse Licensor for, all such amounts. Unless otherwise indicated in an invoice, all Licensor invoices are payable thirty (30) days from the date of the invoice. Licensor reserves the right to charge a late payment in the event Licensee fails to remit payments when due.

7 Confidentiality. "Confidential Information" means any non-public technical or business information of a party, including without limitation any information relating to a party's techniques, algorithms, software, know-how, current and future products and services, research, engineering, vulnerabilities, designs, financial information, procurement requirements, manufacturing, customer lists, business forecasts, marketing plans and information. Each Party shall maintain in confidence all Confidential Information of the disclosing Party that is delivered to the receiving Party and will not use such Confidential Information except as expressly permitted herein. Each Party will take all reasonable measures to maintain the confidentiality of such Confidential Information, but in no event less than the measures it uses to protect its own Confidential Information.

7.1 Each Subscriber and/or Licensee hereby agrees that the Licensor shall be free to use any general knowledge, skills and experience, (including, but not limited to, ideas, concepts, know-how, or techniques) ("Residuals"), contained in any (i)_ Subscriber and/or Licensee Confidential Information, (ii) Feedback provided by a Subscriber and/or Licensee; (iii) Subscriber's and/or Licensee's products shared or disclosed to Licensor in connection with the Feedback, in each case, which are retained in the mem-

ories of Licensor's employees, agents, or contractors who have had access to such materials. Licensor shall have no obligation to limit or restrict the assignment of its employees, agents or contractors or to pay royalties for any work resulting from the use of Residuals.

8 Limited Warranty and Disclaimer.

8.1 NO WARRANTIES. THE SOFTWARE IS PROVIDED "AS IS" WITHOUT ANY EXPRESS OR IMPLIED WARRANTY OF ANY KIND, INCLUDING WARRANTIES OF MERCHANTABILITY, NONINFRINGEMENT, OR FITNESS FOR A PARTICULAR PURPOSE. Licensor does not warrant or assume responsibility for the accuracy or completeness of any information, text, graphics, links or other items contained within the Software. Licensor does not represent that errors or other defects will be identified or corrected.

8.2 LIMITATION OF LIABILITY. EXCEPT WITH RESPECT TO THE MISUSE OF THE OTHER PARTY'S INTELLECTUAL PROPERTY OR DISCLOSURE OF THE OTHER PARTY'S CONFIDENTIAL INFORMATION IN BREACH OF THIS AGREEMENT, IN NO EVENT SHALL LICENSOR, SUBSIDIARIES, LICENSORS, OR ITS SUPPLIERS BE LIABLE FOR ANY DAMAGES WHATSOEVER (INCLUDING, WITHOUT LIMITATION, INDIRECT, LOST PROFITS, CONSEQUENTIAL, BUSINESS INTERRUPTION OR LOST INFORMATION) ARISING OUT OF THE USE OF OR INABILITY TO USE THE SOFTWARE, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. SOME JURISDICTIONS PROHIBIT EXCLUSION OR LIMITATION OF LIABILITY FOR IMPLIED WARRANTIES OR CONSEQUENTIAL OR INCIDENTAL DAMAGES, SO THE ABOVE LIMITATION MAY NOT APPLY TO SCRIBER AND/OR LICENSEE. SUBSCRIBER AND/OR LICENSEE MAY ALSO HAVE OTHER LEGAL RIGHTS THAT VARY FROM JURISDICTION TO JURISDICTION. NOTWITHSTANDING THE FOREGOING, LICENSOR'S AGGREGATE LIABILITY ARISING OUT OF THIS AGREEMENT SHALL NOT EXCEED ONE HUNDRED UNITED STATES DOLLARS (USD$100).

9 Term; Termination.

9.1 Term. The term of this Agreement shall commence on the Effective Date and shall expire on the first (1st) anniversary of the Effective Date (the "Initial Term"). This Agreement shall automatically renew for successive one (1) year periods (the "Renewal Term," and together with the Initial Term, the "Term") unless Licensee provides Licensor with notice of non-renewal at least thirty (30) calendar days prior to its expiration.

9.2 Termination.

1. Breach. Either Party may terminate this Agreement upon thirty (30) days' prior written notice if the other Party materially breaches this Agreement and does not cure such breach within thirty (30) days following receipt of notice specifying the breach.
2. Insolvency. Either Party may also have the right to terminate this Agreement in the event the other party (i) becomes insolvent, (ii) becomes subject to a petition in bankruptcy filed by or against it that is not dismissed within thirty days of the filing of such petition, (iii) is placed under the control of a receiver, liquidator or committee of creditors, or (iv) dissolves, ceases to function as a going concern or to conduct its business in the normal course.

9.3 Effect of Termination. Upon the expiration or termination of this Agreement, Customer agrees to pay all amounts accrued or otherwise owing to Licensor on the date of termination, and each Party shall return, or certify the destruction of, the Confidential Information of the other Party. In the event a support program is in place at the effective date of termination, Licensor agrees to continue providing maintenance and support under the terms of this Agreement and the applicable Support Plan through the expiration date of such Support Plan. Termination in accordance with this Agreement shall be without prejudice to any other rights or remedies of the Parties.

10 Miscellaneous.

10.1 Legal Compliance; Restricted Rights. Each Party agrees to comply with all applicable Laws. With-

out limiting the foregoing, Customer agrees to comply with all U.S. export Laws and applicable export Laws of its locality (if Customer is not located in the United States), and Customer agrees not to export any Network Appliances, Software or other materials provided by Licensor without first obtaining all required authorizations or licenses. The Network Appliances and Software provided to the United States government are provided with only "LIMITED RIGHTS" and "RESTRICTED RIGHTS" as defined in FAR 52.227-14 if the commercial terms are deemed not to apply.

10.2 Governing Law; Severability. This Agreement shall be governed by the laws of the State of New Jersey, USA, without regard to choice-of-law provisions. If any provision of this Agreement is held to be illegal or unenforceable for any reason, then such provision shall be deemed to be restated so as to be enforceable to the maximum extent permissible under law, and the remainder of this Agreement shall remain in full force and effect. Each Subscriber and/Licensee and Licensor agree that this Agreement shall not be governed by the U.N. Convention on Contracts for the International Sale of Goods. Any and all proceedings relating to the subject matter of this Agreement shall be maintained in the courts of the State of Delaware or Federal District Courts sitting in the District of Delaware, which courts shall have exclusive jurisdiction for such purpose, and Subscriber and/or Licensee hereby consents to the personal jurisdiction of such courts.

10.3 Notices. Any notices under this Agreement will be personally delivered or sent by certified or registered mail, return receipt requested, or by nationally recognized overnight express courier, to the address specified herein or such other address as a Party may specify in writing. Such notices will be effective upon receipt, which may be shown by confirmation of delivery. All notices to Licensor shall be sent to the attention of General Counsel (unless otherwise specified by Licensor).

10.4 Assignment. Neither Party may assign or otherwise transfer this Agreement without the other Party's prior written consent, which consent shall not be unreasonably withheld, conditioned or delayed. Notwithstanding the foregoing, either Party may assign this Agreement without the consent of the other Party if a majority of its outstanding voting capital stock is sold to a third party, or if it sells all or substantially all of its assets or if there is otherwise a change of control. This Agreement shall be binding upon and inure to the benefit of the Parties' successors and permitted assigns.

10.5 Force Majeure. Neither Party shall be liable for any delay or failure due to a force majeure event and other causes beyond its reasonable control. This provision shall not apply to any of Customer's payment obligations.

10.6 Counterparts. This Agreement may be executed in counterparts, each of which will be deemed an original, but all of which together will constitute one and the same instrument. This Agreement may be executed by facsimile or scanned signatures.

10.7 General. This Agreement, including its exhibits (all of which are incorporated herein), are collectively the Parties' complete agreement regarding its subject matter, superseding any prior oral or written communications. Amendments or changes to this Agreement must be in mutually executed writings to be effective. The individual executing this Agreement on behalf of the Customer has the requisite power and authority to sign this Agreement on behalf of Customer. The Parties agree that, to the extent any Customer purchase or sales order contains terms or conditions that conflict with, or supplement, this Agreement, such terms and conditions shall be void and have no effect, and the provisions of this Agreement shall control. Unless otherwise expressly set forth in an exhibit that is executed by the Parties, this Agreement shall control in the event of any conflict with an exhibit. Sections 2, 3, 5, 6, 7, 8, 9 and 10 and all warranty disclaimers, use restrictions and provisions relating to Licensor's intellectual property ownership, shall survive the termination or expiration of this Agreement. The Parties are independent contractors for all purposes under this Agreement.

EXHIBIT A

CULA Dense Free Edition, CULA Dense, and CULA Sparse Software License Agreement

The contents of this file are subject to the Software License Agreement (the "Agreement"). You may not use this file except in compliance with the Agreement.

Software distributed under the Agreement is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the Agreement for the specific language governing rights and limitations under the Agreement.

The developer of the CULA Dense Free Edition, CULA Dense, and CULA Sparse is EM Photonics, Inc., a Delaware corporation.

2009 - 2012 Copyright EM Photonics, Inc. All Rights Reserved.

CULATOOLS(TM), CULA(TM), the EM Photonics logo, and certain other trademarks and logos are trademarks or registered trademarks of EM Photonics, Inc. in the United States and other countries.

Note: A printer friendly version of this Agreement is available in RTF format.

CULA Dense Free Edition Files Re-distributable Pursuant to Section 2.1 (b).

cula_core.dll

cula_lapack.dll

cula_lapack_fortran.dll

libcula_lapack_pgfortran.dll

cula_lapack_link.dll

libcula_core.so.*

libcula_lapack.so.*

libcula_lapack_fortran.so.*

libcula_lapack_pgfortran.so.*

libcula_lapack_link.so.*

libcula_core.dylib

libcula_lapack.dylib

libcula_lapack_fortran.dylib

libcula_lapack_pgfortran.dylib

libcula_lapack_link.dylib

CULA Dense Files Files Re-distributable Pursuant to Section 2.2 (b).

None. To Redistribute CULA Dense please contact EM Photonics, Inc. at info@culatools.com

CULA Sparse Files Re-distributable Pursuant to Section 2.3 (b).

None, except for Third Party content as listed in Exhibit B. To Redistribute CULA Sparse please contact EM Photonics, Inc. at info@culatools.com.

EXHIBIT B

Third Party Files Re-distributable Pursuant to Section 4.

UFconfig - LGPL

src/suitesparse/UFconfig.tar.gz

COLAMD - LGPL

src/suitesparse/COLAMD.tar.gz

colamd.dll

libcolamd.so

libcolamd.dylib

..Getting Started {{{

CHAPTER

THREE

GETTING STARTED

3.1 Obtaining CULA

CULA can be downloaded from www.culatools.com. The Dense Free Edition release is available to those who register on the website. CULA Dense, which add dozens of extra LAPACK routines as well as double-precision and double-precision complex data formats, can be purchased from the CULAtools website at www.culatools.com/purchase.

For vendors interested in CULA Commercial, please contact EM Photonics directly www.culatools.com/contact.

3.2 System Requirements

CULA utilizes *CUDA* on an NVIDIA GPU to perform linear algebra operations. Therefore, an NVIDIA GPU with *CUDA* support is required to use CULA. A list of supported GPUs can be found on NVIDIA's CUDA Enabled webpage.

Support for double-precision operations requires a GPU that supports *CUDA* Compute Model 1.3. To find out what Compute Model your GPU supports, please refer to the NVIDIA CUDA Programming Guide.

Note: CULA's performance is primarily influenced by the processing power of your system's GPU, and as such a more powerful graphics card will yield better performance.

3.3 Installation

Installation is completed via the downloadable installation packages. To install CULA, refer to the section below that applies to your system.

Windows

Run the CULA installer and when prompted select the location to which to install. The default install location is *c:\Program Files\CULAR#*, where R# represents the release number of CULA.

Linux

It is recommended that you run the CULA installer as an administrator in order to install to a system-level directory. The default install location is */usr/local/cula*.

Mac OS X Leopard

Open the CULA .dmg file and run the installer located inside. The default install location is */usr/local/cula*.

Note: You may wish to set up environment variables to common CULA locations. More details are available in the *Configuring Your Environment* chapter.

3.4 Compiling with CULA

As described in the *Introduction*, CULA presents four interfaces. This section describes the differences between the Standard and Device interfaces. The third interface (Fortran) is discussed in a later section.

CULA presents two main *C* headers, *cula_lapack.h* for the standard interface and *cula_lapack_device.h* for the device interface. These files differ in the type of memory they accept. For the Standard interface, the pointers that these functions accept are to main memory, while in Device interface the pointers are to GPU memory. A convenience header *cula.h* is provided and includes all C functionality.

CULA's Standard interface automatically transfers the contents of the main memory to the GPU before beginning processing, and transfers the results back to the main memory upon completion of the operation. Due to internal memory mechanisms, this can be many times more efficient than allowing the user to pre-load the data to their GPU. However, if the programmer is building a closed-loop GPU system, the data may already be on-card and the CULA Device interface may be more convenient.

CULA's Device interface operates on pre-allocated GPU memory. This interface is intended for those who have pre-existing systems or are trying to build a closed-loop GPU processing system. These inputs are located in GPU memory and the results are placed in GPU memory. Allocation and transfers of data are the responsibility of the programmer and are performed using the *CUDA* API.

3.5 Linking to CULA

CULA provides a link-time stub library, but is otherwise built as a shared library. Applications should link against the following libraries:

Windows

Choose to link against cula_lapack.lib or cula_lapack_basic.lib as a link-time option.

Linux / Mac OS X Leopard

Add `-lcula_lapack` or `-lcula_lapack_basic` to your program's link line.

CULA is built as a shared library, and as such it must be visible to your runtime system. This requires that the shared library is located in a directory that is a member of your system's runtime library path . For more detailed information regarding operating-system-specific linking procedures, please refer to the *Configuring Your Environment* chapter.

CULA's example projects are a good resource for learning how to set up CULA for your own project.

Note: CULA is built against NVIDIA *CUDA 5.0* and ships with a copy of the *CUDA 5.0* redistributable files. If you have a different version of *CUDA* installed, you **must** ensure that the *CUDA* runtime libraries shipped with CULA are the first visible copies to your CULA program. This can be accomplished by placing the CULA *bin* path earlier in your system PATH than any *CUDA bin* path. If a non-*CUDA 5.0* runtime loads first, you will experience CULA errors. See the *Checking That Libraries are Linked Correctly* example for a description of how to programmatically check that the correct version is linked.

3.6 Uninstallation

After installation, CULA leaves a record to uninstall itself easily. To uninstall CULA, refer to the section below that applies to your system.

Windows

From the Start Menu, navigate to the *CULA* menu entry under *Programs*, and select the Uninstall option. The CULA uninstaller will remove CULA from your system.

Linux

Run the CULA installer, providing an 'uninstall' argument.
e.g. ./cula.run uninstall

Mac OS X Leopard

There is no uninstallation on OS X, but you can remove the folder to which you installed CULA for a complete uninstall.

Note: If you have created environment variables with references to CULA, you may wish to remove them after uninstallation.

CHAPTER

FOUR

DIFFERENCES BETWEEN CULA AND LAPACK

Unlike LAPACK, CULA uses both your system's CPU and GPU resources. Because of this difference, CULA diverges slightly from the original LAPACK interface. CULA reduces some of the complexities of traditional LAPACK interfaces, while providing the flexibility you need to manage the GPU as a compute device. This section describes the differences between CULA and LAPACK and how they affect your program.

For a discussion on individual routines, see the *CULA Reference Manual*.

4.1 Naming Conventions

CULA follows a function naming conventions that is similar but different to that of LAPACK. The primary reason for this different is so that the accelerated routines in CULA will be able to link cleanly against other linear algebra packages that supply routines that are not present in CULA. This restriction comes from linkage rules in C/C++ that dictate that there can only be one function of a given name within a program. If CULA used names that are equivalent to those of other packages, this would prevent users from compiling CULA and another linear algebra package in the same program.

CULA's routines are named using a system where:

- The function begins with `cula` (in lowercase). Device interface functions use an extended prefix of `culaDevice`.
- The next character (in uppercase) represents the data type.
- The following two characters (in lowercase) represent the matrix type.
- The final two to three characters (in lowercase) represent the computation performed by the routine.

For example, the following call will perform CULA's single-precision QR factorization on a general matrix.

culaSgeqrf

Culapack Prefix | Data Type | Matrix Type | Computation Routine

The following table outlines the matrix types specific to CULA.

Abbreviation	Matrix Type
bd	Bidiagonal
ge	General
gg	General matrices, generalized problem
he	Hermitian Symmetric
or	(Real) Orthogonal
sb	Symmetric Band
sy	Symmetric
tr	Triangular
un	(Complex) Unitary

The following table lists the suffix for some common routines CULA provides.

Abbreviation	Computation Routine
trf	Compute a triangular factorization
sv	Factor the matrix and solve a system of equations
qrf	Compute a QR factorization without pivoting
svd	Compute the singular value decomposition
ls	Solve over- or under-determined linear system

The full list of abbreviations can be found at the LAPACK website.

4.2 Calling Conventions

In addition to using different function names, CULA has moved away from the LAPACK interface and takes arguments by value rather than by reference, which is a convention more familiar to C/C++ programmers. Additionally, this simplifies calling code by reducing the need for some temporary variables.

For example, compare a call to a CULA function with one that uses traditional LAPACK conventions:

```
// Common variables
...

// CULA
culaStatus s;
s = culaSgesvd('O', 'N', m, m, a, lda, s, u, ldu, vt, ldvt);

// Traditional
char jobu = 'O';
char jobvt = 'N';
int info;
sgesvd(&jobu, &jobvt, &m, &n, a, &lda, s, u, &ldu, vt, &ldvt, &info);
```

Compared the to the traditional code, CULA uses fewer lines, has a shorter call line, and is no less clear than the traditional usage – one might argue that it is even more clear because there is less code to inspect.

In addition to the Standard and Device interfaces, CULA has a third interface, **Fortran**, is intended to be used in Fortran programs. This interface follows the conventions of Fortran programming by taking all parameters by reference rather than by value. For more information on this interface, see *Programming in Fortran*.

With version R12, CULA introduced the **Link** interface. This interface is targeted at users who are porting existing linear algebra codes to a GPU-accelerated environment. The **Link** interface provides a migration path by matching the

function names and signatures of several popular linear algebra packages. It additionally provides a fallback to CPU execution when a user does not have a GPU or when problem size is too small to take advantage of GPU execution. For more information on this interface, see the 'link_interface.txt' document in the 'doc' directory.

4.3 Data Type Support

CULA provides support for float and float-complex data types by using C's built-in `float` and the CULA type `culaFloatComplex`. Similarly, CULA provides support for double and double-complex data types by using C's built-in `double` and the CULA type `culaDoubleComplex`, though these data types require support from the GPU (see *System Requirements*).

Prefix	Data Type	CULA Standard Interface Type	CULA Device Interface Type
S	Single Precision Real	culaFloat	culaDeviceFloat
C	Single Precision Complex	culaFloatComplex	culaDeviceFloatComplex
D	Double Precision Real	culaDouble	culaDeviceDouble
Z	Double Precision Complex	culaDoubleComplex	culaDeviceDoubleComplex

Note: *CUDA* complex types may be used in place of CULA types by defining the preprocessor macro CULA_USE_CUDA_COMPLEX before including a CULA header.

4.4 Error Handling

LAPACK's error handling is accomplished through an *info* parameter. Upon return, this parameter specifies whether or not the function succeeded (return value of *0*) and if it did not, indicates the particular parameter that is in error by stating the parameter number. Additionally, LAPACK may print an error to *stdout*.

CULA uses a system similar to LAPACK's error handling but adds additional functionality. Instead of requiring an *info* parameter, CULA returns the success or failure of a function using a *culaStatus* enumeration. This return code specifies conditions that the *info* parameter does not cover, including:

- a valid co-processing device is not available;
- insufficient memory is available to continue;
- a requested feature is not available;
- the co-processing device is missing a required feature to complete the requested operation.

Two classes of errors that are typically reported by LAPACK are those in arguments passed to a function and those resulting from malformed data. CULA reports these errors using *culaArgumentError* and *culaDataError* return values. When this value is returned, further information about this error can be requested by calling *culaGetErrorInfo*, which returns an integer equivalent to that which would be returned by the analogous LAPACK function. Unlike LAPACK, CULA will not print information to *stdout*. Once the info code is received, a readable string may be generated by calling *culaGetErrorInfoString*, which accepts the CULA status code and info code to construct a human-readable string that provides an anaylsis of the error condition.

4.5 Workspaces

Many LAPACK functions require a workspace for internal operation. For those LAPACK functions that utilize a workspace, workspace sizes are queried by providing a -1 argument to what is typically an *LWORK* parameter. Upon

inspecting this parameter, the LAPACK function will determine the workspace required for this particular problem size and will return the value in the *WORK* parameter. LAPACK (and other similar packages) then require the programmer to provide a pointer to memory of sufficient size, which often requires that the programmer allocate new memory.

CULA uses both main and GPU workspace memories, and as such, LAPACK's workspace query is not appropriate, as the LAPACK interface allows for the specification of only one workspace. Instead of providing a more complicated interface that adds parameters for both main and GPU workspace memories, CULA requires neither. Instead, any workspaces that are required are allocated and tracked internally. This organization yields no significant performance loss, and furthermore reduces the number of function calls by removing the need for a workspace query.

Note: Any workspaces that have been allocated internally may be cleared by calling `culaFreeBuffers()`.

CHAPTER

FIVE

PROGRAMMING CONSIDERATIONS

This chapter contains a variety of subjects related to programming using CULA.

5.1 Matrix Storage

When providing data to CULA routines, it is important to consider the manner in which the data is stored in memory. In particular, there are two important requirements to ensure correct results:

- Data must be in "column-major" order.
- Complex data must be interleaved.

The following sections will briefly explain these requirements.

5.1.1 Column-Major Ordering

CULA routines expect that any data provided will be stored in column-major order. While this storage scheme is also expected for LAPACK routines, column-major order may be uncommon to some programmers, especially those unfamiliar with Fortran conventions. Consider the following *M-by-N* matrix.

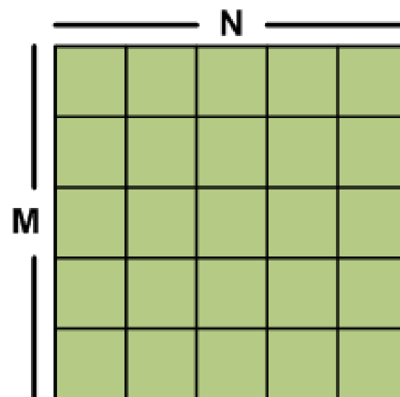

Figure 5.1: A basic *M x N* matrix.

When storing the matrix in memory, two methods are frequently used: row-major and column-major.

For many C programmers, a familiar storage pattern would involve storing the elements of a particular row first, followed by the elements of successive rows. In this storage scheme, elements of a row are contiguous in memory, while elements of a column are not. C, for example, is row-major storage by default.

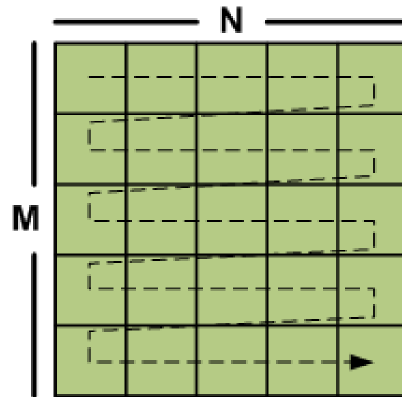

Figure 5.2: A row-major ordered matrix. Elements are stored in memory in the order shown by the arrow.

Column-major ordering is the opposite of this because elements of the matrix are instead stored by column, rather than by row. In this storage scheme, elements of a column are contiguous in memory, while elements of a row are not.

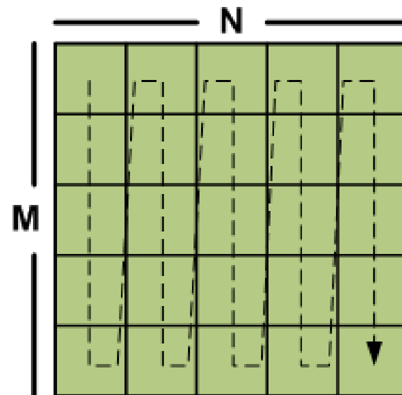

Figure 5.3: A column-major ordered matrix. Elements are stored in memory in the order shown by the arrow.

As mentioned previously, CULA expects data to be represented in column-major order. Performing a transpose on the row-major data will convert it to column-major and vice-versa.

Note: Many CULA functions take a "leading dimension" argument after matrix arguments. Typically these are named LD*. For column-major data, the leading dimension the leading dimension is equal to the height of a column, or equivalently, the number of rows in the matrix. This is the height of the matrix *as allocated* and may be larger than the matrix used in the computation. For submatrix operations or pitched allocations, remember to report the leading dimension parameter as the number of *allocated* rows in the matrix.

Note: Note the differences in addressing row-major and column-major order.

Given an element of matrix A (with leading dimension LDA) located at row "i" and column "j," this element would be accessed as follows:

- `A[i*LDA + j]` for row-major data.
- `A[j*LDA + i]` for column-major data in CULA.

5.1.2 Complex Data Types

When working with complex data, CULA expects real and complex portions of the matrix to be interleaved. In other words, for any particular element, the real and complex parts are adjacent in memory, with the complex part being stored after the real part.

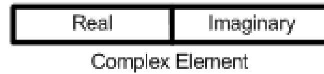

Figure 5.4: This figure shows the packing of a complex data element. In memory, the real part of an element is directly followed by the complex component of that element.

5.2 Optimizing for Performance

CULA is specifically designed to leverage the massively parallel computational resources of the GPU, with a particular focus on large problems whose execution on a standard CPU is too time consuming. As such, several considerations should be taken into account when applying CULA routines to maximize overall performance gains.

5.2.1 Problem Size

Prior to the application of CULA routines it is important to consider problem size. As a general rule, applying CULA for larger problems will maximize performance gains with respect to other computational linear algebra packages. For example, consider the *SGEQRF Speedups* Figure.

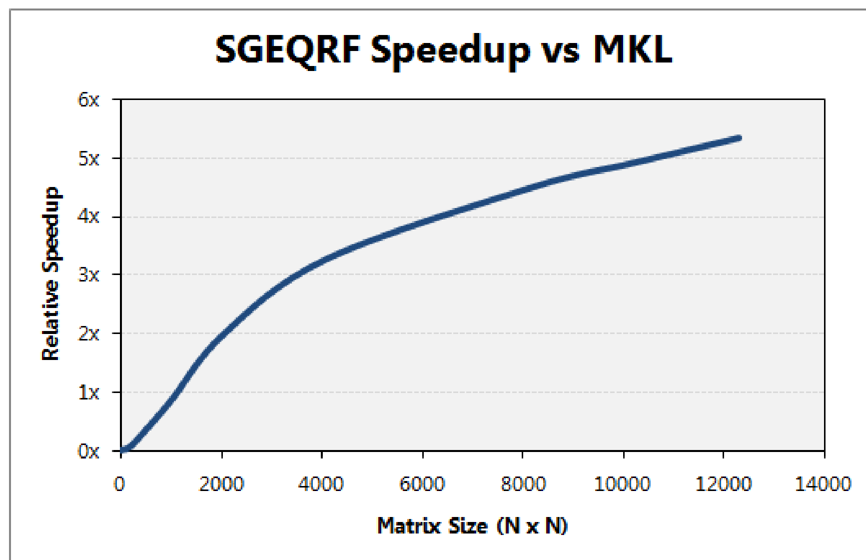

Figure 5.5: This figure shows the speedup for a selected routine versus problem size. High speedups are attained for larger problem sizes.

The speedup chart illustrates the performance of CULA's QR decomposition with respect to Intel's Math Kernel Library (MKL). Note that for the smallest problem sizes, no performance gains are seen; however, problem complexity quickly scales in such a manner that the GPU is able to outperform the CPU, for even modest problem sizes.

Note: The maximum problem size is constrained by the data type in use and the maximum GPU memory. For example, the maximum size for a problem that uses double-precision complex data is roughly one fourth of the maximum problem size of a single-precision problem for the same matrix dimensions, since the size of these data types differ by a factor of four.

5.2.2 Accuracy Requirements

CULA offers both single and double-precision floating point support for its included routines. While the latest NVIDIA GPU hardware offers support for both of these data types, it should be noted that current NVIDIA GPU hardware performs best when operating on single-precision data.

When applying CULA routines, the accuracy requirements of the program should be evaluated. If extended precision is required, then the use of double-precision routines is advised. For programs with more relaxed accuracy requirements, however, additional performance can be achieved at the cost of accuracy through the use of single-precision routines.

5.3 Using the Device Interface

CULA's device interface allows a user to directly control GPU memory. This is useful for closed-loop GPU systems that do large sections of processing on the GPU. The device interface is not recommended to those without *CUDA* experience.

When using the device interface, include *cula_lapack_device.h* instead of *cula_lapack.h*; see the *Matrix Storage* section for differences in naming conventions between the Standard and Device interfaces. Memory allocation is handled via cudaMalloc and cudaFree, available in the *CUDA* toolkit. If the programmer uses pitched memory, it is up to that programmer to ensure that their allocations are appropriate.

Note: CULA's standard Dense version provides specialized allocation functions that pitch data to the optimal size for CULA. These functions are `culaDeviceMalloc()` and `culaDeviceFree()`, found in the *cula_device.h* header of CULA Dense.

5.4 Thread Safety/Multi-GPU Operation

Currently, the default operating mode of CULA is to operate on only one GPU. Aside from this default mode, a programmer can manually control the way that CULA is used across the various GPU devices in their system. For a given function call, CULA currently supports single-GPU operation only. However, function calls may be made from several threads because all API functions are designed to be thread safe. Thus, you may write a program that is multithreaded, with various threads making accesses to CULA to perform their individual linear algebra operations.

By default, CULA binds to the best GPU in a system. However, this binding may be manually controlled by using `culaSelectDevice` (found in *cula_device.h*). It is important to note that if a given thread is already bound to a particular GPU device, CULA cannot reset this binding, as this is a restriction of *CUDA*.

To use multiple GPUs, multiple threads must be created. Each thread must be bound to a different GPU device by calling `culaSelectDevice`, with a different GPU id for each thread. After doing so, call `culaInitialize` in each thread, and proceed to call CULA functions as you need. Before each thread exits, ensure that `culaShutdown` is called in each.

When using CULA, normal parallel programming rules apply. The section below outlines a few of these considerations for each of CULA's interfaces.

Standard Interface

The Standard interface uses (traditional) main memory pointers. It is up to the programmer to ensure that CULA does not concurrently write to overlapping sections of memory; doing so will yield undefined results.

Device Interface

The *CUDA* programming model specifies that a given thread may only communicate with a single GPU in a system. To follow this requirement, any call to a *CUDA* API function or a CULA function will "bind" that thread to that GPU (except where specified). These calls include memory allocations functions. A device pointer is only valid on a particular GPU, so care must be taken when calling CULA in a multithreaded environment. The important point is to only pass device pointers that were allocated in that particular thread context. To learn which device a thread is bound to, you may call `culaGetExecutingDevice()` routine found in *cula_device.h*.

5.5 Developing in C++

CULA provides a set of generic functions for use in C++ applications. These functions are overloaded to select a function according to the data type of its operand, allowing generic code. For every function in the *cula_lapack.h* and *cula_lapack_device.h* headers, there is a generic version of that function in the *cula_lapack.hpp* and *cula_lapack_device.hpp* headers. The generic functions will determine the type of the calling data and dispatch the operation to the appropriate typed function. Similarly to the C interfaces, a *cula.hpp* file is provided as a convenience header to include both the Standard and Device interfaces simultaneously if desired.

Please see the *Examples* chapter for an example of this interface.

CHAPTER

SIX

EXAMPLES

The CULA installation contains several examples that show how to use CULA. These examples are contained in the 'examples' directory.

This section lists several of the common use cases for CULA functions.

6.1 Initialization and Shutdown

```
#include <cula.h>

culaStatus s;

s = culaInitialize();
if(s != culaNoError)
{
    printf("%s\n", culaGetStatusString(s));
    /* ... Error Handling ... */
}

/* ... Your code ... */

culaShutdown();
```

6.2 Argument Errors

```
#include <cula.h>

culaStatus s;

s = culaSgeqrf(-1, -1, NULL, -1, NULL); /* obviously wrong */
if(s != culaNoError)
{
    if(s == culaArgumentError)
        printf("Argument %d has an illegal value\n", culaGetErrorInfo());
    else
        printf("%s\n", culaGetStatusString(s));
}
```

6.3 Data Errors

```
#include <cula.h>

float* A = malloc(20*20*sizeof(float));
memset(A, 0, 20*20*sizeof(float)); /* singular matrix, illegal for LU (getrf) */
int ipiv[20];
s = culaSgetrf(20, 20, A, 20, ipiv);
if( s != culaNoError )
{
    if( s == culaDataError )
        printf("Data error with code %d, please see LAPACK documentation\n",
            culaGetErrorInfo());
    else
        printf("%s\n", culaGetStatusString(s));
}
```

6.4 Printing Errors to the Console

```
#include <cula.h>

culaStatus s;
int info;
char buf[256];

s = <cula function>;

if( s != culaNoError )
{
    info = culaGetErrorInfo();
    culaGetErrorInfoString(s, info, buf, sizeof(buf));

    printf("%s", buf);
}
```

6.5 Using the C++ Interface

```
#include <cula.hpp>

template<class T>
void GenericLU(T* A, int N) // type of T will be determined by the compiler
{
    std::vector<int> piv(N);

    // no need for type specifier - determined automatically on overloads
    culaStatus s = culaGetrf(N, N, A, N, &ipiv[0]);

    // check errors
}
```

6.6 Checking That Libraries are Linked Correctly

```
#include <cula.h>

int MeetsMinimumCulaRequirements()
{
    int cudaMinimumVersion = culaGetCudaMinimumVersion();
    int cudaRuntimeVersion = culaGetCudaRuntimeVersion();
    int cudaDriverVersion = culaGetCudaDriverVersion();
    int cublasMinimumVersion = culaGetCublasMinimumVersion();
    int cublasRuntimeVersion = culaGetCublasRuntimeVersion();

    if(cudaRuntimeVersion < cudaMinimumVersion)
    {
        printf("CUDA runtime version is insufficient; "
               "version %d or greater is required\n", cudaMinimumVersion);
        return 0;
    }

    if(cudaDriverVersion < cudaMinimumVersion)
    {
        printf("CUDA driver version is insufficient; "
               "version %d or greater is required\n", cudaMinimumVersion);
        return 0;
    }

    if(cublasRuntimeVersion < cublasMinimumVersion)
    {
        printf("CUBLAS runtime version is insufficient; "
               "version %d or greater is required\n", cublasMinimumVersion);
        return 0;
    }

    return 1;
}
```

CHAPTER

SEVEN

PROGRAMMING IN FORTRAN

This section describes the usage of CULA with the Fortran programming language.

7.1 Introduction

There are two ways to interface most *CUDA* programs with the Fortran language. CULA supports both of these methods. The first, or traditional, method is to mimic the C-style of *CUDA* programming. In this model, the programmer allocates data through the *CUDA* C runtime environment, and care must be taken to properly interface with this model. The other, more recent model, is the Portland Group's CUDA-Fortran programming environment, in which *CUDA* memory is tagged with the "device" attribute but otherwise treated very similarly to other Fortran data. CULA supports both of these methods in Fortran, and provides the host memory and device memory interfaces, much as with the C-style programming mentioned throughout the rest of this guide.

7.2 Routine Naming

Routines in the CULA-Fortran interface are named identically to the C-style functions mentioned throughout this guide, with one exception: the `dropCaps` routine naming is replaced by a more Fortran-like style with underscores as separators, i.e., `DROP_CAPS`. Fortran is case insensitive and as such, `drop_caps` is equally applicable. Therefore the examples in this guide are largely applicable, except that a routine like `culaInitialize()` should be called in Fortran as `CULA_INITIALIZE()`.

Some example routine names are shown below:

Standard	Fortran
culaInitialize	CULA_INTIALIZE
culaSgeqrf	CULA_SGEQRF
culaDeviceSgeqrf	CULA_DEVICE_SGEQRF

7.3 Module File Interfacing

CULA provides a range of module files in the *include* folder. These are roughly analogous to the C-interface header files located in the same folder of the installation. A program will typically invoke one or more of these (depending on which routines are needed) via the "use" statement.

The module files are preferred because array dimensionality, data types, and number of arguments will be checked properly by the compiler. Please note that array dimensions are not checked (only dimensionality is checked). It is

recommended to couple the use of the module file with the "implicit none" statement in order to enforce the fullest available checking.

Module files must be first be compiled because they are source code. These can be passed to your compiler like any other Fortran source code file. Only the modules that are referenced via the "use" statement in a given program should be supplied to the compiler.

Note: GFORTRAN 4.3 is the oldest compiler from the GNU Compiler Collection that supports the ISO_C_BINDING semantics required to use the CULA Fortran modules.

7.4 Using the CULA C-Style Device Memory Interface

Using a C-style *CUDA* memory interface in Fortran can be a difficult proposition because it requires interacting with libraries intended for a C language programming environment. The principal method for memory allocation is via a module declared with ISO_C_BINDING language extension. Modules which declare the majority of the *CUDA* API can be found on the Internet, but are not provided in the default *CUDA* installation. It is the programmer's responsibility to properly interact with the *CUDA* library.

Note: An example of this integration can be found in *examples/fortranDeviceInterface*.

With the *CUDA* memory allocation issues addressed, using CULA is a straightforward exercise, as the referenced example will illustrate. The experience of using the CULA library's Device interface is exactly the same as using the CULA library's Host interface once *CUDA* memory is allocated and transfered.

This is the only compatible method for the Intel Fortran and GNU Fortran compilers for use of the CULA Device interface.

7.5 Using the CULA CUDA-Fortran Device Memory Interface

The PGI CUDA-Fortran programming style for *CUDA* allows memory to be allocated and transfered as follows:

```
real, allocatable, dimension(:) :: my_host_vector
real, allocatable, dimension(:), device :: my_device_vector

allocate(my_host_vector(1000))
random_number(my_host_vector)

! allocate device memory
allocate(my_device_vector(1000))

! transfer host data to device memory
my_device_vector = my_host_vector
```

Using CULA with this memory model is simple. The user should "use cula_lapack_device_pgfortran" and then can invoke CULA Device functions on the arrays tagged with the "device" attribute. See the module file or the API Reference to learn which parameters to each function are expected to be "device" data and which are host data (host data arrays are those not tagged "device.")

CHAPTER

EIGHT

CONFIGURING YOUR ENVIRONMENT

This section describes how to set up CULA using common tools, such as Microsoft® Visual Studio®, as well as command line tools for Linux and Mac OS X.

8.1 Microsoft Visual Studio

This section describes how to configure Microsoft Visual Studio to use CULA. Before following the steps within this section, take note of where you installed CULA (the default is *C:\Program Files\CULA*). To set up Visual Studio, you will need to set both Global- and Project-level settings. Each of these steps is described in the sections below.

8.1.1 Global Settings

When inside Visual Studio, navigate to the menu bar and select *Tools* > *Options*. A window will open that offers several options; in this window, navigate to *Projects and Solutions* > *VC++ Directories*. From this dialog you will be able to configure global executable, include, and library paths, which will allow any project that you create to use CULA.

The table below specifies the recommended settings for the various directories that the *VC++ Directories* dialog makes available. When setting up your environment, prepend the path of your CULA installation to each of the entries in the table below. For example, to set the include path for a typical installation, enter *C:\Program Files\CULA\include* for the *Include Files* field.

Option	Win32	x64
Executable Files	bin	bin64
Include Files	include	include
Library Files	lib	lib64

With these global settings complete, Visual Studio will be able to include CULA files in your application. Before you can compile and link an application that uses CULA, however, you will need to set up your project to link CULA.

8.1.2 Project Settings

To use CULA, you must instruct Visual Studio to link CULA to your application. To do this, right-click on your project and select *Properties*. From here, navigate to *Configuration Properties* > *Linker* > *Input*. In the *Additional Dependencies* field, enter “cula_lapack.lib” or “cula_lapack_basic.lib” as appropriate for your version.

On the Windows platform, CULA’s libraries are distributed as a dynamic link library (DLL) (*cula.dll*) and an import library (*cula_lapack.lib*), located in the *bin* and *lib* directories of the CULA installation, respectively. By Linking

cula_lapack.lib, you are instructing Visual Studio to make an association between your application and the CULA DLL, which will allow your application to use the code that is contained within the CULA DLL.

8.1.3 Runtime Path

CULA is built as a dynamically linked library, and as such it must be visible to your runtime system. This requires that cula.dll and its supporting dll's are located in a directory that is a member of your system's runtime path. On Windows, you may do one of several things:

1. Add CULA_BIN_PATH_32 or CULA_BIN_PATH_64 to your PATH environment variable.
2. Copy cula.dll and its supporting dll's to the working directory or your project's executable.

8.2 Linux / Mac OS X - Command Line

On a Linux system, a common way of building software is by using command line tools. This section describes how a project that is command line driven can be configured to use CULA.

8.2.1 Configure Environment Variables

The first step in this process is to set up environment variables so that your build scripts can infer the location of CULA.

On a Linux or Mac OS X system, a simple way to set up CULA to use environment variables. For example, on a system that uses the *bourne* (*sh*) or *bash* shells, add the following lines to an appropriate shell configuration file (e.g. *.bashrc*).

```
export CULA_ROOT=/usr/local/cula
export CULA_INC_PATH=$CULA_ROOT/include
export CULA_LIB_PATH_32=$CULA_ROOT/lib
export CULA_LIB_PATH_64=$CULA_ROOT/lib64
```

(where `CULA_ROOT` is customized to the location you chose to install CULA)

After setting environment variables, you can now configure your build scripts to use CULA.

Note: You may need to reload your shell before you can use these variables.

8.2.2 Configure Project Paths

This section describes how to set up the *gcc* compiler to include CULA in your application. When compiling an application, you will typically need to add the following arguments to your compiler's argument list:

Item	Command
Include Path	`-I$CULA_INC_PATH`
Library Path (32-bit arch)	`-L$CULA_LIB_PATH_32`
Library Path (64-bit arch)	`-L$CULA_LIB_PATH_64`
Libraries to Link against	`-lcula_lapack` or `-lcula_lapack_basic`

For a 32-bit compile:

```
gcc ... -I$CULA_INC_PATH -L$CULA_LIB_PATH_32 . . .
        -lcula_lapack -lcublas -lcudart ...
```

For a 64-bit compile:

```
gcc ... -I$CULA_INC_PATH -L$CULA_LIB_PATH_64 . . .
        -lcula_lapack -lcublas -lcudart ...
```

Substitute “cula_lapack_basic” in the above commands if using that version.

8.2.3 Runtime Path

CULA is built as a shared library, and as such it must be visible to your runtime system. This requires that CULA’s shared libraries are located in a directory that is a member of your system’s runtime library path. On Linux, you may do one of several things:

1. Add CULA_LIB_PATH_32 or CULA_LIB_PATH_64 to your LD_LIBRARY_PATH environment variable.
2. Edit your system’s ld.so.conf (found in /etc) to include either CULA_LIB_PATH_32 or CULA_LIB_PATH_64.

On the Mac OS X platform, you must edit the DYLD_LIBRARY_PATH environment variable for your shell, as above.

CHAPTER

NINE

TROUBLESHOOTING

This section lists solutions for common problems encountered when using CULA and describes your support options.

9.1 Common Issues

How do I report information about my system?

Run one of the *sysinfo.bat* or *sysinfo.sh* scripts, for Windows or Linux/Mac OS X systems, respectively. The information these scripts report can be uploaded as an attachment to your support requests or forum posts and will aid us greatly in diagnosing your problems.

An LAPACK routine I need is missing. What should I do?

LAPACK is a very large library and we have only written a subset of its capabilities so far. CULA Dense is constantly growing as new functions are added. If there is function you would like to see added, contact us on our forums and voice your opinion on which functions you use.

I'm having problems with my GPU and/or video drivers. Can you help?

Problems specific to GPU devices and their drivers should be handled by NVIDIA. The latest drivers can always be obtained directly from NVIDIA on their Download page. CULA requires *CUDA 5.0* compatible drivers (or newer) to be installed on your system. On Linux, this requires driver version 5.0.35 or newer, and for Windows it is version 5.0.35 or newer. On Mac OS X, please use 5.0.36 or newer.

I think I found a bug. What should I do?

First, be sure that you are experiencing a software bug and not an issue related to faulty hardware or drivers. If you are still reasonably sure you have encountered a bug, see the Support Options Listed below.

cuda.dll or libcula.so could not be found

Your runtime system has not been set up to locate CULA. See the *Linking to CULA* section.

CULA returns culaNotInitialized.

`culaInitialize` must be called before any CULA functions may be used.

Note: Some functions have an exception to this rule, see *cula_status.h* for a list of these functions.

CULA returns culaNoHardware.

Your system doesn't have the proper hardware required to support CULA. Your system may have a non-NVIDIA GPU, may have an NVIDIA GPU that is too old, or your system may be missing the appropriate drivers for your graphics card.

Note: This error will also occur when running code that uses CULA functions over remote desktop in Windows, as the *CUDA* runtime currently doesn't support GPU execution over remote desktop.

CULA returns culaInsufficientRuntime.

This error indicates that the GPU driver or library dependencies are too old or is too limited to run this code. See the *Checking That Libraries are Linked Correctly* example for how to programmatically check these dependencies. If the driver is insufficient, update your drivers a newer package, available at NVIDIA's website.

Note that CULA ships with the libraries it requires, and therefore this error may indicate that a user's path is incorrectly configured to link against these dependencies. See the *Linking to CULA* section for a discussion of how this problem can be fixed.

CULA returns culaInsufficientComputeCapability.

Your GPU doesn't support a required feature to complete this operation. Most commonly, this is returned when trying to run a double-precision operation on a GPU that doesn't have double-precision hardware.

CULA returns culaInsufficientMemory.

Your GPU doesn't have enough memory to complete the operation. Consider purchasing a GPU with more memory.

CULA returns culaFeatureNotImplemented.

The requested variant of a given routine hasn't been implemented by the CULA developers. While most functions have feature-parity with respect to Netlib, this may be the case in an uncommon variant of a function.

CULA returns culaArgumentError.

Your input to one of CULA's routines is incorrect. The faulty parameter is reported by culaGetErrorInfo; ensure that all of your parameters are correct. See the *Error Handling* section.

CULA returns culaDataError.

An input to one of CULA's routines is malformed or singular. More information about the error is reported by culaGetErrorInfo. See the *Error Handling* section.

CULA returns culaBlasError.

CUBLAS has returned an error. Please file a bug report.

CULA returns culaRuntimeError.

The most common cause of this error is related to improperly configured or old GPU drivers. Ensure that you system is up to date by installing the latest drivers from NVIDIA.

If you have made sure that your drivers are up to date, you can learn more about this particular error by calling culaGetErrorInfo. The value returned here will correspond to a *CUDA* runtime API error code; see driver_types.h in the *CUDA* toolkit for these codes.

See the *Checking That Libraries are Linked Correctly* example for how to programmatically check if the GPU driver is sufficient.

If this does not solve your problem, please file a bug report.

9.2 Support Options

If none of the entries above solve your issue, you can seek technical support. See your license level below to learn about your support options.

- **Free Edition** - Free Edition users may post problems to the CULA forums for community support.
- **CULA Dense** - Paid users can submit support tickets through the CULA website to receive personalized help.
- **Integrators** - Commercial users have direct support. Please contact your support representative directly for help with your issue.

When reporting a problem, make sure to include the following information:

- System Information (see *Common Issues*)
- CULA Version
- Version of NVIDIA® *CUDA* Toolkit installed, if any
- Problem Description (with code if applicable)

CHAPTER

TEN

CHANGELOG

10.1 Release R17 CUDA 5.0 (May 8, 2013)

All Versions

- Changed: cula_core.dll/so is removed - functionality is in cula_lapack.dll/so
- Changed: culaGetLastStatus and culaSetLastStatus are removed
- Changed: Runtime dependency on Intel OpenMP 5 redistributable (libiomp5) added
- Fixed: Several issues in the Link Interface

CULA Dense Free Version

- Changed: cula_lapack.dll/so is renamed to cula_lapack_basic.dll/so

10.2 Release R16a CUDA 5.0 (November 20, 2012)

All Versions

- Improved: All Fortran modules have been converted to ISO_C_BINDING
- Removed: Implicit Fortran calling method is no longer supported
- Fixed: Inadvertantly missing Fortran examples have been replaced
- Fixed: Updated CUBLAS minor version to remove false dependency on the CUDA driver

10.3 Release R16 CUDA 5.0 (October 16, 2012)

All Versions

- Feature: CUDA runtime upgraded to 5.0
- Feature: K20 support
- Fixed: Incompatibility between Fortran module files and Cray Compiler
- Fixed: Resource leak caused by culaShutdown

CULA Dense

- Feature: Implemented symmetric generalized Eigensolvers (sygv)

- Alpha Feature: pgesv (multi-GPU LU-based solve)
- Alpha Feature: pgetrs (multi-GPU LU backsolve)

10.4 Release R15 CUDA 4.2 (August 14, 2012)

Announcement

- All packages are now “universal” and contain both 32-bit and 64-bit binaries
- Multi-GPU routines (pCULA) remain in alpha

All Versions

- Feature: CUDA runtime upgraded to 4.2
- Feature: Kepler support
- Feature: Link interface LAPACK compatibility version upgraded to 3.3.1
- Feature: New Fortran module files for CULA Core and LAPACK subsets, with Device interface
- Feature: New PGI Fortran example using CUDA-Fortran semantics
- Feature: New Fortran Device Interface example
- Improved: Performance of geqrf improved by up to 10%
- Improved: Fortran documentation in Programmer’s Guide
- Improved: Link interface compatible with Matlab 2012
- Fixed: Link interface properly functions on GEMM for sizes > 1k
- Fixed: Resource overflow possibility when certain dimensions to gesv are very large
- Changed: Fortran modules are now located in “include”

CULA Dense

- Improved: improved speed for multi-GPU routines
- Improved: improved scalability for multi-GPU routines
- Improved: reduced memory overhead

10.5 Release R14 CUDA 4.1 (January 30, 2012)

Announcement

- Alpha Feature: Multi-GPU routines are included in the full CULA Dense version as a preview to be finalized in R15
- Alpha Feature: pgetrf (multi-GPU LU factorization)
- Alpha Feature: ppotrf (multi-GPU Cholesky decomposition)
- Alpha Feature: ppotrs (multi-GPU Cholesky backsolve)
- Alpha Feature: pposv (multi-GPU symmetric/hermitian positive-definite factorize and solve)
- Alpha Feature: pgemm (multi-GPU matrix-matrix multiply)
- Alpha Feature: ptrsm (multi-GPU triangular solve)

- Alpha Advisory: Performance, accuracy, routine list, and interface are all subject to change

All Versions

- Feature: CUDA runtime upgraded to 4.1
- Changed: Transitional headers have been removed
- Fixed: Now shipping all dependencies required by OSX systems

CULA Dense

- Improved: Up to 3x performance improvement to trtri
- Improved: 10% performance improvement to potrf

10.6 Release R13 CUDA 4.0 (November 2, 2011)

All Versions

- Feature: Compatibility with CULA Sparse S1
- Improved: Significantly improved thread safety
- Fixed: Host transpose function
- Fixed: Workaround for a resource leak in the CUDA toolkit
- Changed: Headers renamed to cula_lapack.h (etc); transitional header available

CULA Dense

- Feature: Implemented potri (inverse of symmetric positive definite matrix)
- Feature: Implemented gesdd (singular value decomposition variant)
- Feature: Implemented geqrfp (qr decomposition variant)

10.7 Release R12 CUDA 4.0 (May 26, 2011)

All Versions

- Feature: CUDA runtime upgraded to 4.0
- Feature: New link-compatible interface for compatibility with existing programs
- Improved: Now reserving less memory in multithreaded programs
- Improved: More closely matching cuComplex type for better compatibility
- Improved: Renamed all examples to clarity the purpose of each
- Improved: Compatibility with future GPUs

Premium

- Fixed: gebrd accuracy for M<N case
- Fixed: Rarely occuring illegal memory access in symv

10.8 Release R11 CUDA 3.2 (March 31, 2011)

Premium

- Feature: Implemented symv (symmetric matrix vector product)
- Feature: Implemented hemv (hermitian matrix vector product)
- Feature: Implemented geConjugate (conjugate general matrix)
- Feature: Implemented trConjugate (conjugate triangular matrix)
- Feature: Implemented geNancheck (check for NaNs in matrix)
- Feature: Implemented geTranspose (transpose general matrix out of place)
- Feature: Implemented geTransposeConjugate (transpose and conjugate out of place)
- Feature: Implemented geTransposeInplace (transpose square matrix inplace)
- Feature: Implemented geTransposeConjugateInplace (transpose and conjugate square matrix)
- Feature: Implemented lacpy (copy matrix)
- Feature: Implemented lag2 (convert precision)
- Feature: Implemented lar2v (apply rotations)
- Feature: Implemented larfb (apply reflector)
- Feature: Implemented larfg (generate reflector)
- Feature: Implemented largv (generate rotations)
- Feature: Implemented lartv (apply rotations)
- Feature: Implemented lascl (scale matrix)
- Feature: Implemented laset (set matrix)
- Feature: Implemented lasr (apply rotation)
- Feature: Implemented lat2z (convert precision, triangular matrix)
- Improved: Increased speed of syev/syevx significantly when finding eigenvectors
- Improved: Increased speed of gebrd
- Improved: Increased speed of gesvd when calculating singular values

All Versions

- Fixed: Failure to initialize for Quadro X000 cards

10.9 Release R10 CUDA 3.2 (December 10, 2010)

Premium

- Feature: Implemented pbtrf (positive definite banded matrix factorization)
- Feature: Implemented gbtrf (banded matrix factorization)

All Versions

- Feature: CUDA runtime upgraded to 3.2
- Feature: Explicit support and tuning added for 500-series GPUs

- Feature: Added BLAS interfaces
- Feature: Implemented culaGetErrorInfoString to aid in error diagnosis
- Feature: culatypes.h defines either CULA_BASIC or CULA_PREMIUM
- Improved: All routines retuned for Fermi. Gains of up to 100% are available.
- Improved: Multi-thread performance and stability
- Improved: All examples have more descriptive error output

10.10 Release 2.1 Final (August 31, 2010)

Premium

- Feature: Implemented orgrq

All Versions

- Feature: Support for PGI CUDA Fortran Compiler (link -lcula_pgfortran)
- Feature: OS X supports 64-bit
- Fixed: More reliably detect and produce culaInsufficientRuntime condition
- Fixed: culaShutdown is now safe in a multithreaded context

10.11 Release 2.0 Final (June 28, 2010)

Premium

- Fixed: Improved accuracy of geev for some specific matrices

All Versions

- Feature: CUDA Runtime upgraded to CUDA 3.1
- Fixed: Fortran interface properly accepts floating point constant arguments
- Fixed: Properly detect if an insufficient runtime or driver is installed

10.12 Release 2.0 Preview (May 21, 2010)

Premium

- Feature: Implemented dsposv and zcposv (iteratively refined solvers)
- Feature: Implemented complex versions of syev (symmetric eigenvalues)
- Feature: Implemented complex versions of syevx (expert version of syev)
- Feature: Implemented complex versions of syrdb (symmetric reduction to tridiagonal form)
- Feature: Implemented complex versions of steqr (eigenvalues and vectors of a symmetric tridiagonal matrix)
- Improved: Improved performance of syrdb by up to 30%
- Improved: Improved performance of syev by up to 20%
- Improved: Improved performance of potrf by up to 55%

- Improved: Improved performance of most routines in D/Z precision by up to 30%

All Versions

- Feature: CUDA runtime upgraded to 3.1 Beta
- Feature: Support for Fermi-class GPUs
- Feature: Mac OS X version supports all complex and double precision functionality

10.13 Release 1.3a Final (April 19, 2010)

Premium

- Fixed: Increased stability of syev, syevx, and stebz for certain types of matrices
- Improved: Improved accuracy of stebz

10.14 Release 1.3 Final (April 8, 2010)

Basic

- Improved: Removed performance degradation for large NRHS in gesv
- Improved: Increased performance of gesv by up to 45%
- Fixed: gglse properly handles P=0 case

Premium

- Feature: Benchmark example now supports double precision
- Improved: Increased performance of getrs by up to 50%
- Improved: Increased performance of posv by up to 45%
- Improved: Increased performance of potrf by up to 10%
- Improved: Increased performance of trtrs by up to 60%

All Versions

- Improved: Mac OSX builds have install_name rpath set

10.15 Release 1.2 Final (February 17, 2010)

Premium

- Feature: Implemented syev (symmetric eigenvalues)
- Feature: Implemented syevx (expert version of syev)
- Feature: Implemented syrdb (symmetric reduction to tridiagonal form)
- Feature: Implemented stebz (calculate eigenvalues of a symmetric matrix)
- Feature: Implemented steqr (eigenvalues and vectors of a symmetric tridiagonal matrix)
- Feature: Implemented geqrs (system solve from QR data)
- Feature: Implemented geqlf (QL factorize)

- Feature: Implemented orgql/ungql (Q matrix generate from LQ data)
- Feature: Implemented ormql/unmql (multiply by Q matrix from LQ data)
- Feature: Implemented ds/zc gesv routine (iteratively refined gesv)
- Feature: Implemented ggrqf (generalized RQ factorization of two matrices)
- Improved: Increased performance of bdsqr by 10%
- Improved: Increased performance of gebrd by up to 100%

All Versions

- Improved: Increased performance of getrf by up to 50% for square matrices
- Improved: Increased performance of getrf by up to 30% for non-square matrices
- Improved: Increased performance of geqrf by up to 10%
- Improved: gesvd produces significantly more accurate unitary matrices
- Improved: gesvd/bdsqr memory requirements reduced significantly
- Fixed: gesvd produces correct unitary matrices for all data inputs
- Fixed: Fortran device interface is now functional
- Fixed: getrf now continues to factorize after encountering a singularity

10.16 Release 1.1b Final (January 6, 2009)

Premium

- Fixed: Interface of getrs properly interprets the ‘N’ job code

10.17 Release 1.1a Final (December 21, 2009)

All Versions

- Improved: Benchmark example easier to use and provides more user control
- Improved: System info script (sysinfo.sh) now properly reports GPU on 195-series driver
- Improved: All error codes are thorougly described in the Programmer’s Guide
- Fixed: OS X builds now no longer reference unnecessary external libraries
- Fixed: All routines properly accept job codes in both upper- and lower-case
- Fixed: Potential infinite loop when allocating mixed-precision data
- Fixed: Now reporting host out-of-memory condition as culaInsufficientMemory
- Fixed: RHEL 4.7 builds include proper dependent libraries

Premium

- Fixed: cudaDeviceMalloc underallocates for non-float data types

10.18 Release 1.1 Final (November 25, 2009)

All Versions

- Improved: Removed culaInitialize() delay
- Improved: GEQRF performance up to 20% improvement for users with older CPUs
- Fixed: Correction to Fortran example Makefile to specify “arch” parameter

Basic

- Improved: SVD significant memory usage reduction
- Improved: GELS stability for N > M case

Premium

- Improved: GELQF performance increase 2-3x
- Improved: GEHRD/GERQF/ORGLQ/ORGQR performance increased by 10-20%
- Improved: GEHRD routine accurate for size N==1

10.19 Release 1.1 Beta (November 13, 2009)

All Versions

- Feature: Mac OS X 10.5 Leopard “preview” release - single precision only
- Feature: New “Bridge” interface provides for easy and seamless porting of existing LAPACK/MKL/ACML applications (see doc/bridge_interface.txt)
- Feature: New document describing full CULA API
- Feature: New function culaSelectDevice to set executing device
- Feature: New “gesv” example shows operation of all S/C/D/Z data types
- Feature: New “multigpu” example showing multi-GPU CULA operation
- Feature: New “bridge” example showing usage of the Bridge interface
- Improved: SVD optimized for non-square cases
- Improved: Documentation clarified on error conditions and codes
- Improved: Stronger error reporting from example projects
- Improved: culaInitialize detects and reports if driver/runtime version are inadequate
- Improved: Documentation clearer on thread safety issues
- Fixed: CULA can now handle extremely non-square matrices (eg 500000x16)
- Fixed: An error in the “benchmark” example causing it to ignore user arguments
- Fixed: Properly reporting cudaErrorMemoryValueTooLarge as culaInsufficientMemory

Basic

- Improved: GESV performance increased by up to 30%
- Improved: Stability of GELS in certain cases
- Improved: Stability of SVD in certain cases

Premium

- Feature: Implemented geev (general Eigensolver) in S/D/C/Z precisions
- Feature: Implemented gehrd (general Hessenberg reduction) in S/D/C/Z precisions
- Feature: Implemented orghr
- Feature: .hpp headers have name overloads of ORG/UNG functions
- Fixed: Host interface "ORG" functions different results from device interface

10.20 Release 1.0 Final (September 30, 2009)

Basic

- Feature: All functions feature complex variants
- Fixed: Crash related to getrs pivot array

Premium

- Feature: All functions implemented in all supported data types

10.21 Release 1.0 Beta 3 (September 15, 2009)

All Versions

- Feature: New documentation section on specific routine conventions
- Improved: Updated sysinfo script with more descriptive output
- Improved: Added example that demonstrates the device interface
- Fixed: Various corrections for small-matrix inputs, especially M=N=1
- Fixed: culaInitialize now sets environment variable KMP_DUPLICATE_LIB_OK

Basic

- Feature: Complex geqrf included
- Feature: Added culaGetDeviceCount to report the number of available devices
- Feature: Added culaGetDeviceInfo to report information about a device
- Feature: Added culaGetExecutingDevice to report the executing device
- Fixed: Further corrections for unitary output in gesvd for all job codes

Premium

- Feature: New functions culaDeviceMalloc/culaDeviceFree in culadevice.h
- Fixed: Orglq and orgqr should behave more reliably

10.22 Release 1.0 Beta 2 (August 27, 2009)

All Versions

- Feature: Including both 32- and 64-bit libraries on 64-bit Linux release

- Feature: Now shipping precompiled Benchmark example on Linux builds
- Feature: Troubleshooting section added to Programmer's Guide
- Feature: Added scripts that report system information to *examples* folder
- Improved: Error output for examples is now more descriptive
- Improved: Documentation is more specific about configuring system runtime
- Fixed: Incompatibilities with gcc 4.2 and earlier; gcc 4.1 is now compatible

Basic

- Improved: gesvd was optimized for up to a 60% speedup over Beta 1
- Fixed: Error in geqrf for matrices of M << N
- Fixed: Error in gesvd where some matrices would yield non-unitary U and Vt

Premium

- Feature: Implemented getri
- Feature: Implemented potrf
- Feature: Implemented potrs
- Feature: Implemented posv
- Feature: Implemented trtrs
- Improved: orglq was optimized for up to a 700% speedup

10.23 Release 1.0 Beta 1 (August 13, 2009)

All Versions

- Feature: Support Windows XP 32/64
- Feature: Support Linux 32/64

Basic

- Feature: Implemented gels
- Feature: Implemented geqrf
- Feature: Implemented gesv
- Feature: Implemented gesvd
- Feature: Implemented getrf
- Feature: Implemented gglse

Premium

- Feature: Implemented gebrd
- Feature: Implemented getrs
- Feature: Implemented trtrs
- Feature: Implemented gelqf
- Feature: Implemented gerqf

- Feature: Implemented orgqr
- Feature: Implemented orglq
- Feature: Implemented orgbr
- Feature: Implemented ormqr
- Feature: Implemented ormlq
- Feature: Implemented ormrq
- Feature: Implemented bdsqr