CULA Sparse Reference Manual

www.culatools.com

Release S3 (CUDA 4.2)

EM Photonics, Inc.
www.emphotonics.com

August 14, 2012

CONTENTS

1 Introduction **1**
- 1.1 Sparse Linear Systems . . . 1
- 1.2 Supported Operating Systems . . . 2
- 1.3 Attributions . . . 2

2 Getting Started **3**
- 2.1 System Requirements . . . 3
- 2.2 Installation . . . 3
- 2.3 Compiling with CULA Sparse . . . 4
- 2.4 Linking to CULA Sparse . . . 4
- 2.5 Uninstallation . . . 4

3 Using the API **6**
- 3.1 Initialization . . . 6
- 3.2 Memory Management . . . 6
- 3.3 Indexing Formats . . . 6
- 3.4 Sparse Matrix Storage Formats . . . 7
- 3.5 Numerical Types . . . 8
- 3.6 Common Solver Configuration . . . 8
- 3.7 Naming Conventions . . . 8
- 3.8 Choosing a Solver and Preconditioner . . . 9
- 3.9 Iterative Solver Results . . . 9
- 3.10 Data Errors . . . 10
- 3.11 Timing Results . . . 11
- 3.12 Residual Vector . . . 11

4 Data Types **12**
- 4.1 culaStatus . . . 12
- 4.2 culaVersion . . . 12
- 4.3 culaIterativeConfig . . . 13
- 4.4 culaIterativeResult . . . 13
- 4.5 culaIterativeFlag . . . 13
- 4.6 culaIterativeResidual . . . 14
- 4.7 culaIterativeTiming . . . 14
- 4.8 culaReordering . . . 14
- 4.9 Options Structures . . . 15

5 Framework Functions **16**
- 5.1 culaSparseInitialize . . . 16

5.2 culaSparseShutdown 16
5.3 culaIterativeConfigInit 16
5.4 culaIterativeConfigString 17
5.5 culaIterativeResultString 17
5.6 culaGetCusparseMinimumVersion 17
5.7 culaGetCusparseRuntimeVersion 17

6 Iterative Preconditioners **19**
6.1 No Preconditioner 19
6.2 Jacobi Preconditioner 19
6.3 Block Jacobi 20
6.4 ILU0 20

7 Iterative Solvers **21**
7.1 Conjugate Gradient (CG) 21
7.2 Biconjugate Gradient (BiCG) 22
7.3 Biconjugate Gradient Stabilized (BiCGSTAB) 23
7.4 Generalized Biconjugate Gradient Stabilized (L) (BiCGSTAB(L)) 24
7.5 Restarted General Minimum Residual (GMRES(m)) 25
7.6 Minimum residual method (MINRES) 27

8 Performance and Accuracy **29**
8.1 Performance Considerations 29
8.2 Accuracy Considerations 30

9 API Example **31**

10 Configuring Your Environment **33**
10.1 Microsoft Visual Studio 33
10.2 Linux / Mac OS X - Command Line 34
10.3 Checking That Libraries are Linked Correctly 35

11 Common Errors **37**
11.1 Argument Error 37
11.2 Malformed Matrix 37
11.3 Data Errors 38
11.4 Runtime Error 39
11.5 Initialization Error 40
11.6 No Hardware Error 40
11.7 Insufficient Runtime Error 40
11.8 Insufficient Compute Capability Error 41
11.9 Insufficient Memory Error 41

12 Support Options **42**
12.1 Matrix Submission Guidelines 42

13 Routine Selection Flowcharts **43**
13.1 Solver Selection Flowchart 44
13.2 Preconditioner Selection Flowchart 45

14 Changelog **46**
14.1 Release S3 CUDA 4.2 (August 14, 2012) 46
14.2 Release S2 CUDA 4.1 (January 30, 2012) 46
14.3 Release S1 CUDA 4.0 (November 2, 2011) 46
14.4 Release S1 Beta 2 CUDA 4.0 (September 27, 2011) 47

14.5 Release S1 Beta 1 CUDA 4.0 (August 24, 2011) 47

CHAPTER

ONE

INTRODUCTION

This guide documents *CULA Sparse's* programming interface. *CULA Sparse*™ is an implementation of sparse linear algebra routines for *CUDA*-enabled NVIDIA graphics processing units (GPUs). This guide is split into the following sections:

- *Using the API* - A high level overview of how to use configure, use, and interpret the results from the iterative solvers in the *CULA Sparse* library.
- *Data Types* - A description of the all datatypes used the library.
- *Framework Functions* - A description of the functions used to initialize and configure the library.
- *Iterative Preconditioners* - An overview of the preconditioners available within *CULA Sparse*.
- *Iterative Solvers* - A description of the iterative solvers functions available in the library.
- *Performance and Accuracy* - Information on how to maximize the performance of the library. Also includes a handful of performance charts.
- *Common Errors* - Solutions to errors commonly encountered when using the API.

1.1 Sparse Linear Systems

Many problems in science and engineering, particularly those related to partial differential equations (PDEs), can be represented by a linear system where only a few elements in the matrix are non-zero. For these systems, it would be wasteful, in both storage and computation, to represent all of the elements. To address these common problems, storage formats and methods have been developed to solve sparse matrices with minimal memory and computation requirements. These methods can be broken into two main categories: direct methods and iterative methods.

Direct methods, common for dense matrices, attempt to solve system in a two-step process. Typical algorithms include LU and QR factorization where the linear system is transformed into an equivalent system that can be solved using Gaussian elimination. Direct methods can also be applied to sparse systems but algorithms become increasingly complex in an attempt to minimize storage and computation.

The other class of sparse solvers, and those currently implemented in *CULA Sparse*, are iterative methods. These methods attempt to coverage on solution to the system $Ax = b$ by continuously iterating over new solutions until a solution's residual, typically defined as $||b - Ax||/||b||$, is under a given tolerance. At each step, a solution is calculated using a technique specific to the given algorithm. Because it is possible for iterative methods to fail to find a solution, they are commonly configured with a maximum number of iterations.

A common method to improve the speed at which a solution converges is called preconditioning. These methods attempt, at each iteration, to transform the original linear system into a new equivalent system that can more readily be solved. This adds overhead, in both memory and time per iteration, but will often result in a shorter end-to-end solution time.

1.2 Supported Operating Systems

CULA Sparse 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 Sparse* has been tested on the following systems:

- Windows XP / Vista / 7
- Ubuntu Linux 10.04 (and newer)
- Red Hat Enterprise Linux 5.3 (and newer)
- Fedora 11
- Mac OSX 10.6 Snow Leopard / 10.7 Lion

Please provide feedback on any other systems on which you attempt to use *CULA Sparse*. Although we are continually testing *CULA Sparse* 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.

1.3 Attributions

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

CULA Sparse is built on NVIDIA *CUDA* 4.0 and NVIDIA *CUSPARSE*.

CULA Sparse uses COLAMD, covered by the GNU LGPL license. The source code for COLAMD is used in an unmodified fashion; a copy of this code is distributed in the src/suitesparse directory of this package.

Many of the algorithms and methods from this library were developed based on Yousef Saad's textbook "Iterative Methods for Sparse Linear Systems".

CHAPTER

TWO

GETTING STARTED

2.1 System Requirements

CULA Sparse utilizes *CUDA* on an NVIDIA GPU to perform linear algebra operations. Therefore, an NVIDIA GPU with *CUDA* support is required to use *CULA Sparse*. 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 Sparse'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.

2.2 Installation

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

Windows

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

Linux

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

Mac OS X Leopard

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

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

2.3 Compiling with CULA Sparse

CULA Sparse presents one main *C* header, *cula_sparse.h*. You must include this header in your C source file to use *CULA Sparse*.

2.4 Linking to CULA Sparse

CULA Sparse 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_sparse.lib and cula_core.lib as a link-time option.

Linux / Mac OS X Leopard

Add `-l cula_core -lcula_sparse` to your program's link line.

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

Note: *CULA Sparse* is built against NVIDIA *CUDA 4.2* and ships with a copy of the *CUDA 4.2* redistributable files. If you have a different version of CUDA installed, you **must** ensure that the CUDA runtime libraries shipped with *CULA Sparse* are the first visible copies to your *CULA Sparse* program. This can be accomplished by placing the *CULA Sparse bin* path earlier in your system PATH than any CUDA *bin* path. If a non-*CUDA 4.2* runtime loads first, you will experience *CULA Sparse* errors. See the *Checking That Libraries are Linked Correctly* example for a description of how to programmatically check that the correct version is linked.

2.5 Uninstallation

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

Windows

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

Linux

Run the *CULA Sparse* 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 Sparse* for a complete uninstall.

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

CHAPTER

THREE

USING THE API

This chapter describes, at a high level, how to use the *CULA Sparse* API. Basic information about how to initialize and configure an iterative solver is discussed. Furthermore, we introduce how to collect and interpret the results from the iterative solvers as well any error condition that may occur.

Further specifics are found in the subsequent chapters.

3.1 Initialization

The *CULA Sparse* library is initialized by calling the `culaSparseInitialize()` function. This function must be called before any of the iterative solvers are invoked. It is possible to call other *CULA* framework functions such as `culaSelectDevice()` prior to initialization. For interoperability with *CULA*, this routine will also perform the `culaInitialize()` operation.

```
// initialize cula sparse library
culaSparseInitialize();
```

3.2 Memory Management

In *CULA Sparse*, all functions accept pointers to matrices defined on the host; that is, matrices allocated with `malloc()`, `new`, or `std::vector`. *CULA Sparse* will automatically transfer the data to-and-from the accelerated device. This step is performed concurrently to other operations and yields almost no performance impact for this transfer.

Note: Future versions may relax this requirement and allow direct device memory access.

```
// allocate solver data on host using malloc (C)
double* data = (double*) malloc( nnz * sizeof(double) );
// allocate solver data on host using new[]
double* data = new double[nnz];
// allocate solver data on host using a std vector (C++)
std::vector <double> data( nnz );
```

3.3 Indexing Formats

CULA Sparse supports both 0 (C/C++) and 1 (FORTRAN) based indexing through the indexing field of the `culaIterativeConfig` configuration structure. The default is zero-based indexing.

3.4 Sparse Matrix Storage Formats

CULA Sparse currently supports three major matrix storage formats: coordinate, compressed row, and compressed column. It is recommended to use the compressed row format when possible.

3.4.1 Coordinate Format (COO)

In the coordinate format, a sparse matrix is represented by three vectors sized by the number of non-zero elements of the system. The common abbreviation for this length, nnz is used throughout the API and this document.

- values - the non-zero data values within the matrix; length nnz
- row index - the associated row index of each non-zero element; length nnz
- column index - the associated column index of each non-zero element; length nnz

Consider the following 3x3 matrix with 6 non-zero elements:

$$A = \begin{pmatrix} 1.0 & 4.0 & 0.0 \\ 2.0 & 5.0 & 0.0 \\ 3.0 & 0.0 & 6.0 \end{pmatrix}$$

In a zero-indexed coordinate format, this matrix can be represented by the three vectors:

$$\text{values} = \begin{bmatrix} 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 \end{bmatrix}$$
$$\text{column index} = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 2 \end{bmatrix}$$
$$\text{row index} = \begin{bmatrix} 0 & 1 & 2 & 0 & 1 & 2 \end{bmatrix}$$

In the *CULA Sparse* interface, the values are denoted as `a`, the column index as colInd, and the row index as rowInd.

3.4.2 Compressed Sparse Row (CSR) Format

In the compressed sparse row format, the row index vector is replaced by a row pointer vector of size $m + 1$. This vector now stores the locations of values that start a row; the last entry of this vector points to one past the final data element. The column index is as in COO format.

Consider the same 3x3 matrix, a zero-indexed CSR format can be represented by three vectors:

$$\text{values} = \begin{bmatrix} 1.0 & 4.0 & 2.0 & 5.0 & 3.0 & 6.0 \end{bmatrix}$$
$$\text{column index} = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 2 \end{bmatrix}$$
$$\text{row pointer} = \begin{bmatrix} 0 & 2 & 4 & 6 \end{bmatrix}$$

In the *CULA Sparse* interface, the values are denoted as `a`, the column index as colInd, and the row pointer as rowPtr.

3.4.3 Compressed Sparse Column (CSC) Format

In the compressed sparse column format, the column index vector is replaced by a column pointer vector of size $n+1$. This vector now stores the locations of values that start a column; the last entry of this vector points to one past the final data element. The row index is as in COO format.

Consider the same 3x3 matrix, a zero-indexed CSC format can be represented by three vectors:

$$\text{values} = \begin{bmatrix} 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 \end{bmatrix}$$
$$\text{row index} = \begin{bmatrix} 0 & 1 & 2 & 0 & 1 & 2 \end{bmatrix}$$
$$\text{column pointer} = \begin{bmatrix} 0 & 3 & 5 & 6 \end{bmatrix}$$

In the *CULA Sparse* interface, the values are denoted as a, the column pointer as colPtr, and the row index as rowInd.

3.4.4 More Information

For more information regarding these storage formats, we recommend reading Section 3.4, Storage Schemes, Yousef Saad's textbook "Iterative Methods for Sparse Linear Systems".

3.5 Numerical Types

CULA Sparse provides two data types with which you can perform computations.

Symbol	Interface Type	Meaning
D	culaDouble	double precision floating point
Z	culaDoubleComplex	double precision complex floating point

The `culaDoubleComplex` type can be made to be identical to the `cuComplex` type provided by *CUDA*, by `#define CULA_USE_CUDA_COMPLEX` prior to including any *CULA* headers.

3.6 Common Solver Configuration

All iterative solvers within *CULA Sparse* utilize a common configuration parameter to steer the execution of the solver. The configuration parameter is represented by the `culaIterativeConfig` structure and is the first parameter to any solver function within the library. The configuration parameter informs the API of the desired solve by specifying:

- The tolerance at which a solve is marked as converged
- The maximum number of iterations to run
- Whether the input matrices use zero- or one-based indexing
- Whether a debugging mode should be enabled

More parameters may be added in the future.

Configuration parameters must be set up before a solver can be called. The configuration parameter is initialized by the `culaIterativeConfigInit()` function. This function ensures that all parameters within this structure are set to reasonable defaults. After calling this function, you may set specific parameters to your needs.

Example configuration:

```
// create configuration structure
culaIterativeConfig config;

// initialize values
culaIterativeConfigInit(&config);

// configure specific parameters
config.tolerance = 1.e-6;
config.maxIterations = 300;
config.indexing = 0;
```

3.7 Naming Conventions

Four major concepts are conveyed by the function names of the iterative system solvers within the *CULA Sparse* library:

```
cula{type}{storage}{solver}{precond}(...)
```

Here, each {...} segment represents a different major component of the routine. The following table explains each of these components:

Name	Meaning
type	data type used
storage	sparse matrix storage format used
solvers	iterative method used
precond	preconditioner method used

For example, the routine `culaDcsrCgJacobi()` will attempt to solve a double precision sparse matrix stored in the compressed sparse row (CSR) storage format using the conjugate gradient (CG) method with Jacobi preconditioning.

3.8 Choosing a Solver and Preconditioner

Choosing a proper solver is typically determined by the class of the input data. For example, the CG method is only appropriate for symmetric positive definite matrices.

Preconditioner selection is more of an art - the method chosen is tightly coupled to the specifics of the linear system. That is, the computational tradeoffs of generation and application of the preconditioner will be different for different systems.

```
// call CG solver with Jacobi preconditioner
culaStatus status = culaDcsrCgJacobi( &config, n, nz, val, colInd, rowPtr,
                                      x, rhs, &result );
```

3.9 Iterative Solver Results

To convey the status of a given routine, the iterative solver routines return a `culaStatus` code and also populates a `culaIterativeResult` output structure.

The `culaStatus` return code is common between the *CULA* and *CULA Sparse* libraries. It is a high level status code used to indicate if the associated call has completed successfully. Systemic errors that prevented execution are presented in this code, such as not initializing *CULA*, GPU of insufficient capability, or out-of-memory conditions. Most details of mathematical progress of a solver will be presented in `culaIterativeResult`.

```
if ( culaStatus == culaNoEror )
{
    // solution within given runtime parameters was found
}
```

The `culaIterativeResult` structure provides additional information regarding the computation such as:

- A flag that denotes the solvers converged or a reason why convergence failed
- The number of iterations performed, regardless of whether this led to convergence
- The solution residual when the solve ended
- Timing information for the overhead, preconditioner generation, and solving

This structure is returned from all iterative solves.

CULA Sparse also provides a utility function for `culaIterativeResult`, which is called `culaIterativeResultString()`, that constructs a readable string of information that is suitable for printing. In many cases, this function is able to provide information beyond that which is available by inspection of the structure, and so it is recommended that it be used whenever attempting to debug a solver problem.

```
// allocate result string buffer
const int bufferSize = 256;
char buffer[bufferSize];

// fill buffer with result string
culaIterativeResultString( &result, buffer, bufferSize );

// print result string to screen
printf( "%s\n", buffer );
```

Example output:

```
Solver:      CG (Jacobi preconditioner)
Flag:        Converged successfully in 213 iterations
Residual:    9.675273e-007
Total Time:  0.01363s (overhead + precond + solve)
   Overhead: 3.845e-005s
   Precond:  0.001944s
   Solve:    0.01165s
```

3.10 Data Errors

In *CULA Sparse*, the `culaDataError` return status is used to describe any condition for which the solver failed to converge on solution within the given configuration parameters.

Possible reasons for returning a `culaDataError` include:

- Preconditioner failed to generate; no iterations attempted
- Maximum number of iterations exceeded without reaching desired tolerance
- An internal scalar quantity became too large or small to continue
- The method stagnated
- The input data contained values of nan or inf

These possible reasons are enumerated by the flag field of the `culaIterativeResult` structure.

In some cases, a user may wish to get the best possible result in a fixed number of iterations. In such a case, a data error can be interpreted as success if and only if the flag is `culaMaxItReached`. The residual should then be consulted to judge the quality of solution obtained in the budgeted number of iterations.

```
if ( status == culaDataError && result.flag == culaMaxItReached )
{
    // solver executed but the method did not converge to tolerance
    // within the given iteration limit; it is up to programmer
    // judgement whether the result is now acceptable
}
```

3.11 Timing Results

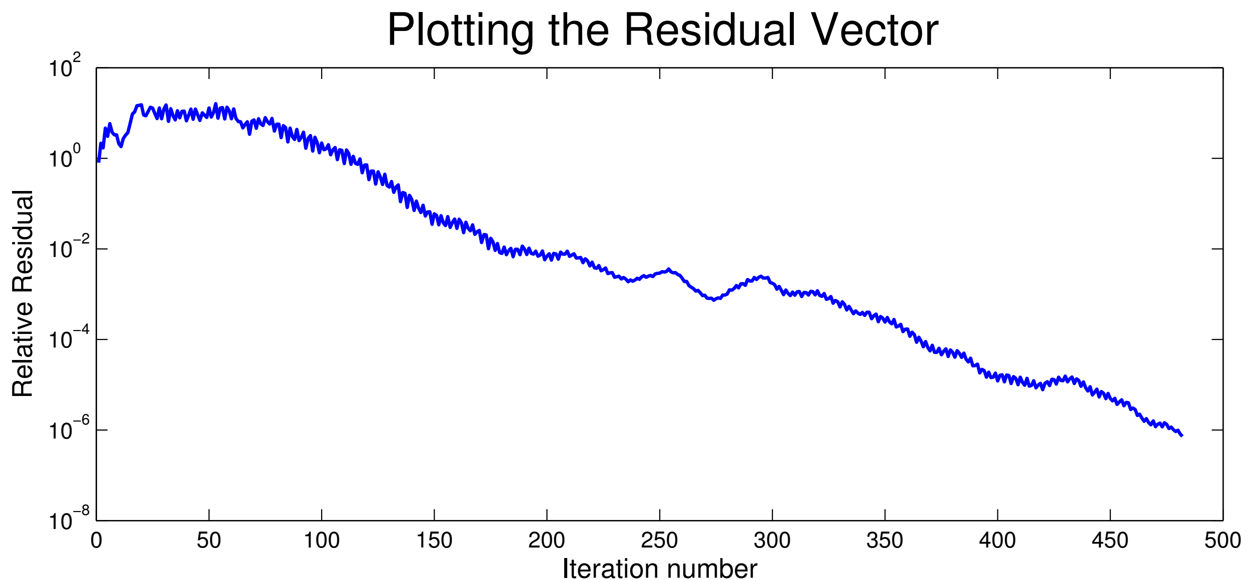
For convenience, the `culaIterativeResult` result structure contains high precision timing information regarding the runtime of the iterative solver. This timing is broken down into three major components:

- Overhead - This includes memory allocations, transfers to-and-from the GPU, and internal operations such as storage type conversion.
- Preconditioner - The time taken to generate the requested preconditioner. This does not include the per-iteration time to apply the preconditioner; the per-iteration time of a preconditioner is included in the solver time.
- Solver - This represents the time spent in the actual iteration loop

Additionally, the total time is returned which is a sum of these three values. All values are in seconds.

3.12 Residual Vector

For some users, it may be desirable to capture the relative residual of the solver at each iteration. *CULA Sparse* provides a mechanism to obtain this information, via a parameter in the configuration structure. This parameter, `result.byIteration` is normally set to NULL, but may be assigned by the user to specify an array into which the residual by iteration should be stored. It is up to the user to ensure that this location has enough memory to store the residual for each iteration; in practice this is achieved by ensuring that the specified array is long enough to store one double precision value for each requested iteration.



CHAPTER

FOUR

DATA TYPES

This function describes the various data types used throughout the *CULA Sparse* library. These types are accepted as arguments by or returned by various interface functions in the library, which will be described later.

Unless otherwise specified, data types are declared in *cula_sparse.h*.

4.1 culaStatus

This type is declared in *cula_status.h*.

The `culaStatus` type is used for all status returns. All *CULA* and *CULA Sparse* functions return their statuses with the following values being defined:

Status Code	Meaning
culaNoError	No error
culaNotInitialized	*CULA* has not been initialized
culaNoHardware	No hardware is available to run
culaInsufficientRuntime	*CUDA* runtime or driver is not supported
culaInsufficientComputeCapability	Available GPUs do not support the requested operation
culaInsufficientMemory	There is insufficient memory to continue
culaFeatureNotImplemented	The requested feature has not been implemented
culaArgumentError	An invalid argument was passed to a function
culaDataError	An operation could not complete because of singular data
culaBlasError	A blas error was encountered
culaRuntimeError	A runtime error has occurred

4.2 culaVersion

This type is declared in *cula_types.h*.

The `culaVersion` data type denotes the version of a library in the format XXXYY where XXX is the major version number and YY is the minor version number.

4.3 culaIterativeConfig

The `culaIterativeConfig` data type is an input structure that contains information that steers execution of iterative functions with the following fields:

Name	Type	Description
indexing	int	Indicates whether the sparse indexing arrays are represented using 0 (C/C++) or 1 (FORTRAN) based indexing.
tolerance	double	The tolerance is the point at which a lower residual will cause the solver to determine that the solution has converged.
maxIterations	int	The maximum number of iterations that the solver will attempt
residualVector	double*	This parameter provides the means for a user to capture the residual at each iteration. The specified array must be at least maxIter in length. This parameter may be NULL if these quantities are not desired.
useInitialResultVector	int	Indicates whether the 'x' vector in iterative solves should be used as given or ignored. When ignored, the 'x' vector is considered a zero.
useBestAnswer	int	Indicates whether the 'x' vector in iterative solves should return the final answer or the best answer and its associated iteration number in the case of non-convergence.
useStagnationCheck	int	Indicates whether to check whether the iterative solve stagnated. This option defaults to on; turning this option off will increase performance if a problem is certain not to stagnate.
debug	int	Specifies whether to perform extra checks to aid in debugging

4.4 culaIterativeResult

The `culaIterativeResult` data type is an output structure containing information regarding the execution of the iterative solver and associated preconditioner. Fields in this data type include:

Name	Type	Description
flag	culaIterativeFlag	Enumeration containing information about the success or failure of the iterative solver
code	unsigned long long	Internal information code
iterations	int	Number of iterations taken by the iterative solver
residual	culaIterativeResidual	Structure containing information about the residual
timing	culaIterativeTiming	Structure containing timing information about the iterative solver and associated preconditioners

4.5 culaIterativeFlag

The `culaIterativeFlag` data type is an output enumeration containing a set of all possible success and mathematical error conditions that can be returned by the iterative solver routines. Possible elements include:

Flag Value	Meaning
culaConverged	The solve converged successfully
culaMaxItReached	Maximum iterations reached without convergence
culaPreconditionerFailed	The specified preconditioner failed
culaStagnation	The iterative solve stagnated
culaScalarOutOfRange	A scalar value was out of range
culaUnknownIterationError	An unknown iteration error was encountered

For more information about various failure conditions, see the *Common Errors* chapter.

4.6 culaIterativeResidual

The `culaIterativeResidual` data type is an output structure that contains information about the residual of an iterative function with the following fields:

Member	Type	Description
relative	double	The relative residual obtained by the iterative solver when computation has completed or halted
byIteration	double*	If requested, the residual at every step of iteration

For more information, see *Residual Vector*.

4.7 culaIterativeTiming

The `culaIterativeTiming` data type is an output structure containing timing information for execution of an iterative function with the following fields:

Member	Type	Description
solve	double	Time, in seconds, the solve portion of the iterative solver took to complete
preconditioner	double	Time, in seconds, the preconditioner generative portion of the iterative solver took to complete
overhead	double	Time, in seconds, of overhead needed by the iterative solver; includes memory transfers to-and-from the GPU
total	double	Time, in seconds, the entire iterative solver took to complete

For more information see *Timing Results*.

4.8 culaReordering

The `culaReordering` data type is an enum that specifies a reordering strategy for certain preconditioners. For some matrices, reordering can introduce additional parallelism that can allow the solver to proceed more efficiently on a parallel device.

Flag Value	Meaning
culaNoReordering	Do not do any reordering
culaAmdReordering	Reorder using the approximate minimum degree ordering method
culaSymamdReordering	Reorder using the symmetric minimum degree ordering method (SYMAMD)

Reordering can be expensive in terms of additional memory required. COLAMD requires approximately $2.2*NNZ+7*N+4*M$ extra elements of storage.

4.9 Options Structures

Solver and preconditioner options structures allow you to steer the execution of a given solver and preconditioner. They are the second and third parameters for all solver functions, respectively. For documentation on individual options structures, see the corresponding solver or preconditioner section.

Initializing these structures is done with a method that matches the name of the associated solver or preconditioner with an `Init` appended.

```
// create options structure
culaBicgOptions solverOpts;

// initialize values
culaBicgOptionsInit(&solverOpts);

// configure specific parameters (if applicable)
// . . .
```

Several options structures have reserved parameters. These structures are implemented in this way so as to maintain uniformity in the solver parameter list and to provide compatibility for possible future code changes. We recommend that you make sure to call the options initialization function (as shown above) for all options in the case that any parameters are added to it in the future. A NULL pointer may be passed, in which case reasonable defaults will be assigned. As this may change in future versions, it is recommended to explicitly construct the structures.

CHAPTER

FIVE

FRAMEWORK FUNCTIONS

This section describes the helper functions associated *CULA Sparse* library. These functions include initializing, configuration, and result analysis routines found in *cula_sparse.h*.

5.1 culaSparseInitialize

Description

Initializes CULA Sparse Must be called before using any other function. Some functions have an exception to this rule: culaGetDeviceCount, culaSelectDevice, and version query functions

Returns

culaNoError on a successful initialization or a culaStatus enum that specifies an error

5.2 culaSparseShutdown

Description

Shuts down CULA Sparse

5.3 culaIterativeConfigInit

Description

Constructs a config structure, initializing it to default values.

Parameters

Name	Description
config	Pointer to the culaIterativeConfig struct that will be initialized by this routine.

Returns

culaNoError on success or culaArgumentError if a parameter is invalid

5.4 culaIterativeConfigString

Description

Associates an iterative config structure with a readable config report

Parameters

Name	Description
config	Pointer to the culaIterativeConfig struct that will be analyzed by this routine
buf	Pointer to a buffer into which information will be printed (recommend size 256 or greater)
bufsize	The size of buf, printed information will not exceed bufsize

Returns

culaNoError on a successful config report or culaArgumentError on an invalid argument to this function

5.5 culaIterativeResultString

Description

Associates an iterative result structure with a readable result result

Parameters

Name	Description
e	A culaStatus error code
i	A pointer to a culaIterativeResult structure
buf	Pointer to a buffer into which information will be printed (recommend size 256 or greater)
bufsize	The size of buf, printed information will not exceed bufsize

Returns

culaNoError on a successful result report or culaArgumentError on an invalid argument to this function

5.6 culaGetCusparseMinimumVersion

Description

Reports the CUSPARSE_VERSION that the running version of CULA was compiled against, which indicates the minimum version of CUSPARSE that is required to use this library

Returns

An integer in the format XXXYY where XXX is the major version number and YY is the minor version number of CUSPARSE that this version of CULA was compiled against. On error, a 0 is returned.

5.7 culaGetCusparseRuntimeVersion

Description

Reports the version of the CUSPARSE runtime that operating system linked against when the program was loaded

Returns

An integer in the format XXXYY where XXX is the major version number and YY is the minor version number of the CUSPARSE runtime. On error, a 0 is returned.

CHAPTER

SIX

ITERATIVE PRECONDITIONERS

Preconditioning is an additional step to aid in convergence of an iterative solver. This step is simply a means of transforming the original linear system into one which has the same solution, but which is most likely easier to solve with an iterative solver. Generally speaking, the inclusion of a preconditioner will decrease the number of iterations needed to converge upon a solution. Proper preconditioner selection is necessary for minimizing the number of iterations required to reach a solution. However, the preconditioning step does add additional work to every iteration as well as up-front processing time and additional memory. In some cases this additional work may end up being a new bottleneck and improved overall performance may be obtained using a preconditioner that takes more steps to converge, but the overall runtime is actually lower. As such, we recommend analyzing multiple preconditioner methods and looking at the total runtime as well the number of iterations required.

This chapter describes several preconditioners, which have different parameters in terms of effectiveness, memory usage, and setup/apply time.

For additional algorithmic details regarding the methods detailed in this chapter, we recommend reading the "Preconditioning Techniques" chapter from Yousef Saad's textbook "Iterative Methods for Sparse Linear Systems".

6.1 No Preconditioner

To solve a system without a preconditioner, simply call the solving routine without a preconditioner suffix, such as `culaDcsrCg()`. A NULL pointer may be passed for the precondOpts parameter.

6.2 Jacobi Preconditioner

The Jacobi precondition is a simple preconditioner that is a replication of the diagonal:

$$M_{i,k} = \begin{cases} A_{i,j} & \text{if } i = j \\ 0 & otherwise \end{cases}$$

The Jacobi preconditioner is very lightweight in generation, memory usage, and application. As such, it is often a strong choice for GPU accelerated solvers.

As detailed in *Iterative Solvers*, the Jacobi preconditioner is invoked by calling an iterative solver with the Jacobi suffix along with the `culaJacobiOptions` input struct parameter (see *Options Structures*).

6.3 Block Jacobi

The Block Jacobi preconditioner is an extension of the Jacobi preconditioner where the matrix is now represented as a block diagonal of size b:

$$M_{i,k} = \begin{cases} A_{i,j} & \text{if } i \text{ and } j \text{ are within the block subset, } b \\ 0 & otherwise \end{cases}$$

$$M = \begin{bmatrix} B_0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & B_n \end{bmatrix}$$

This preconditioner is a natural fit for systems with multiple physical variables that have been grouped into blocks.

The Block Jacobi preconditioner requires more computation in both generation and application than the simpler Jacobi preconditioner. However, both generation and application are parallel operations that map well to the GPU.

As detailed in *Iterative Solvers*, the block Jacobi preconditioner is invoked by calling an iterative solver with the Blockjacobi suffix along with the `culaBlockjacobiOptions` input struct parameter.

6.3.1 culaBlockjacobiOptions

Name	Type	Description
blockSize	int	Block size for the Jacobi Preconditioner

6.4 ILU0

The ILU0 preconditioner is an incomplete LU factorization with zero fill-in, where $L * U \approx A$:

$$M = \begin{cases} L, & \text{lower triangular} \\ U, & \text{upper triangular} \end{cases}$$

The ILU0 preconditioner is lightweight in generation, and due to the zero-fill component, requires roughly the same memory as the linear system trying to be solved. In application, the ILU0 preconditioner requires two triangular solve routines - a method not well suited for parallel processing platforms such as the GPU or multicore processors. As such, using the ILU0 preconditioner may result in a reduced number of iterations at the cost of a longer runtime.

In comparison to Jacobi, the construction and application time and the memory requirements are higher for ILU0. For some matrices, the ILU0 might result in significantly improved convergence, which can offset the costs.

In order to successfully complete the factorization, the input matrix must have a diagonal entry in every row, and it must be nonzero. Failure to meet this criteria will result in a `culaPreconditionerFailed` code.

As detailed in *Iterative Solvers*, the ILU0 preconditioner is invoked by calling an iterative solver with the “Ilu0” suffix along with the `culaIlu0Options` input struct parameter.

6.4.1 culaIlu0Options

Name	Type	Description
reordering	culaReordering	Specifies a reordering strategy for the input matrix. This option defaults to `culaNoReordering`. For more information, see the *culaReordering* section.

CHAPTER

SEVEN

ITERATIVE SOLVERS

This section describes the iterative solvers routines available in the *CULA Sparse* library.

For algorithmic details regarding the methods detailed in this chapter, we recommend reading the "Krylov Subspace Methods: Part 1 & 2" chapters from Yousef Saad's textbook "Iterative Methods for Sparse Linear Systems".

7.1 Conjugate Gradient (CG)

```
culaStatus cula{storage}Cg{preconditioner}(
     const culaIterativeConfig* config,
     const culaCgOptions* solverOpts,
     {preconditioner options},
     int n, int nnz,
     {storage parameters},
     double* x, const double* b,
     culaIterativeResult* result
);
```

culaDcsrCg	culaDcscCg	culaDcooCg
culaDcsrCgIlu0	culaDcscCgIlu0	culaDcooCgIlu0
culaDcsrCgJacobi	culaDcscCgJacobi	culaDcooCgJacobi
culaDcsrCgBlockjacobi	culaDcscCgBlockjacobi	culaDcooCgBlockjacobi

culaZcsrCg	culaZcscCg	culaZcooCg
culaZcsrCgIlu0	culaZcscCgIlu0	culaZcooCgIlu0
culaZcsrCgJacobi	culaZcscCgJacobi	culaZcooCgJacobi
culaZcsrCgBlockjacobi	culaZcscCgBlockjacobi	culaZcooCgBlockjacobi

This family of functions attempt to solve $Ax = b$ using the conjugate gradient (CG) method where A is a symmetric positive definite matrix stored in a sparse matrix format and x and b are dense vectors. The matrix must be a fully populated symmetric; i.e. for each populated entry A_{ij} there must be an identical entry A_{ji}.

Solver Trait	Value
matrix class	Symmetric Positive Definite
memory overhead	$6n$

The associated preconditioner is indicated by the function suffix. See *Iterative Preconditioners* for more information.

7.1.1 Parameters

Param.	Memory	In/out	Meaning
config	host	in	configuration structure
solverOpts	host	in	culaCgOptions structure
precondOpts	host	in	options for specified preconditioner
n	host	in	number of rows and columns in the matrix; must be ≥ 0
nnz	host	in	number of non-zero elements in the matrix; must be ≥ 0
{storage}	host	in	sparse input matrix in corresponding storage format
x	host	out	array of n data elements
b	host	in	array of n data elements
result	host	out	result structure

7.1.2 culaCgOptions

Name	Memory	In/out	Meaning
reserved	host	in	reserved for future compatibility

7.2 Biconjugate Gradient (BiCG)

```
culaStatus cula{storage}Bicg{preconditioner}(
     const culaIterativeConfig* config,
     const culaBicgOptions* solverOpts,
     {preconditioner options},
     int n, int nnz,
     {storage parameters},
     double* x, const double* b,
     culaIterativeResult* result
);
```

culaDcsrBicg	culaDcscBicg	culaDcooBicg
culaDcsrBicgIlu0	culaDcscBicgIlu0	culaDcooBicgIlu0
culaDcsrBicgJacobi	culaDcscBicgBlockjacobi	culaDcooBicgJacobi
culaDcsrBicgBlockjacobi	culaDcscBicgJacobi	culaDcooBicgBlockjacobi

culaZcsrBicg	culaZcscBicg	culaZcooBicg
culaZcsrBicgIlu0	culaZcscBicgIlu0	culaZcooBicgIlu0
culaZcsrBicgJacobi	culaZcscBicgBlockjacobi	culaZcooBicgJacobi
culaZcsrBicgBlockjacobi	culaZcscBicgJacobi	culaZcooBicgBlockjacobi

This family of functions attempt to solve $Ax = b$ using the conjugate gradient (BiCG) method where A is a square matrix stored in a sparse matrix format format and x and b are dense vectors. While BiCG may converge for general matrices, it is mathematically most suitable for symmetric systems that are not positive definite. For symmetric positive definite systems, this method is identical to but considerably more expensive than CG.

Solver Trait	Value
matrix class	General (Symmetric Preferred)
memory overhead	$10n$

The associated preconditioner is indicated by the function suffix. See *Iterative Preconditioners* for more information.

7.2.1 Parameters

Param.	Memory	In/out	Meaning
config	host	in	configuration structure
solverOpts	host	in	culaBicgOptions structure
precondOpts	host	in	options for specified preconditioner
n	host	in	number of rows and columns in the matrix; must be ≥ 0
nnz	host	in	number of non-zero elements in the matrix; must be ≥ 0
{storage}	host	in	sparse input matrix in corresponding storage format
x	host	out	array of n data elements
b	host	in	array of n data elements
result	host	out	result structure

7.2.2 culaBicgOptions

Name	Memory	In/out	Meaning
avoidTranspose	host	in	Avoids repeated transpose operations by creating a transposed copy of the input matrix. May lead to improved speeds and accuracy at the expense of memory and computational overheads.

7.3 Biconjugate Gradient Stabilized (BiCGSTAB)

```
culaStatus cula{storage}Bicgstab{preconditioner}(
     const culaIterativeConfig* config,
     const culaBicgstabOptions* solverOpts,
     {preconditioner options},
     int n, int nnz,
     {storage parameters},
     double* x, const double* b,
     culaIterativeResult* result
);
```

culaDcsrBicgstab	culaDcscBicgstab	culaDcooBicgstab
culaDcsrBicgstabIlu0	culaDcscBicgstabIlu0	culaDcooBicgstabIlu0
culaDcsrBicgstabJacobi	culaDcscBicgstabBlockjacobi	culaDcooBicgstabJacobi
culaDcsrBicgstabBlockjacobi	culaDcscBicgstabJacobi	culaDcooBicgstabBlockjacobi

culaZcsrBicgstab	culaZcscBicgstab	culaZcooBicgstab
culaZcsrBicgstabIlu0	culaZcscBicgstabIlu0	culaZcooBicgstabIlu0
culaZcsrBicgstabJacobi	culaZcscBicgstabBlockjacobi	culaZcooBicgstabJacobi
culaZcsrBicgstabBlockjacobi	culaZcscBicgstabJacobi	culaZcooBicgstabBlockjacobi

This family of functions attempt to solve $Ax = b$ using the conjugate gradient stabilized (BiCGSTAB) method where A is a square matrix stored in a sparse matrix format format and x and b are dense vectors. This method was developed to solve non-symmetric linear systems while avoiding the irregular convergence patterns of the Conjugate Gradient Squared (CGS) method.

Solver Trait	Value
matrix class	General
memory overhead	$10n$

The associated preconditioner is indicated by the function suffix. See *Iterative Preconditioners* for more information.

7.3.1 Parameters

Param.	Memory	In/out	Meaning
config	host	in	configuration structure
solverOpts	host	in	culaBicgstabOptions structure
precondOpts	host	in	options for specified preconditioner
n	host	in	number of rows and columns in the matrix; must be ≥ 0
nnz	host	in	number of non-zero elements in the matrix; must be ≥ 0
{storage}	host	in	sparse input matrix in corresponding storage format
x	host	out	array of n data elements
b	host	in	array of n data elements
result	host	out	result structure

7.3.2 culaBicgstabOptions

Name	Memory	In/out	Meaning
reserved	host	in	reserved for future compatibility

7.4 Generalized Biconjugate Gradient Stabilized (*L*) (BiCGSTAB(*L*))

```
culaStatus cula{storage}Bicgstabl{preconditioner}(
     const culaIterativeConfig* config,
     const culaBicgstablOptions* solverOpts,
     {preconditioner options},
     int n, int nnz,
     {storage parameters},
     double* x, const double* b,
     culaIterativeResult* result
);
```

culaDcsrBicgstabl	culaDcscBicgstabl	culaDcooBicgstabl
culaDcsrBicgstablIlu0	culaDcscBicgstablIlu0	culaDcooBicgstablIlu0
culaDcsrBicgstablJacobi	culaDcscBicgstablBlockjacobi	culaDcooBicgstablJacobi
culaDcsrBicgstablBlockjacobi	culaDcscBicgstablJacobi	culaDcooBicgstablBlockjacobi

culaZcsrBicgstabl	culaZcscBicgstabl	culaZcooBicgstabl
culaZcsrBicgstablIlu0	culaZcscBicgstablIlu0	culaZcooBicgstablIlu0
culaZcsrBicgstablJacobi	culaZcscBicgstablBlockjacobi	culaZcooBicgstablJacobi
culaZcsrBicgstablBlockjacobi	culaZcscBicgstablJacobi	culaZcooBicgstablBlockjacobi

This family of functions attempt to solve $Ax = b$ using the conjugate gradient stabilized (BiCGSTAB(L)) method where A is a square matrix stored in a sparse matrix format format and x and b are dense vectors. This method extendeds the BiCG algorithm by adding an additional GMRES step with a restart value of L after each BiCGSTAB iteration. In practice, this may help to smooth convergence - especially in cases where A has large complex eigenpairs.

Solver Trait	Value
matrix class	General
memory overhead	$n * L + 8n$

The associated preconditioner is indicated by the function suffix. See *Iterative Preconditioners* for more information.

7.4.1 Parameters

Param.	Memory	In/out	Meaning
config	host	in	configuration structure
solverOpts	host	in	culaBicgstablOptions structure
precondOpts	host	in	options for specified preconditioner
n	host	in	number of rows and columns in the matrix; must be ≥ 0
nnz	host	in	number of non-zero elements in the matrix; must be ≥ 0
{storage}	host	in	sparse input matrix in corresponding storage format
x	host	out	array of n data elements
b	host	in	array of n data elements
result	host	out	result structure

7.4.2 culaBicgstablOptions

Name	Memory	In/out	Meaning
l	host	in	restart value of the GMRES portion of the algorithm; directly related to memory useage

7.5 Restarted General Minimum Residual (GMRES(m))

```
culaStatus cula{storage}Gmres{preconditioner}(
     const culaIterativeConfig* config,
     const culaGmresOptions* solverOpts,
```

```
    {preconditioner options},
    int n, int nnz,
    {storage parameters},
    double* x, const double* b,
    culaIterativeResult* result
);
```

culaDcsrGmres	culaDcscGmres	culaDcooGmres
culaDcsrGmresIlu0	culaDcscGmresIlu0	culaDcooGmresIlu0
culaDcsrGmresJacobi	culaDcscGmresJacobi	culaDcooGmresJacobi
culaDcsrGmresBlockjacobi	culaDcscGmresBlockjacobi	culaDcooGmresBlockjacobi

culaZcsrGmres	culaZcscGmres	culaZcooGmres
culaZcsrGmresIlu0	culaZcscGmresIlu0	culaZcooGmresIlu0
culaZcsrGmresJacobi	culaZcscGmresJacobi	culaZcooGmresJacobi
culaZcsrGmresBlockjacobi	culaZcscGmresBlockjacobi	culaZcooGmresBlockjacobi

This family of functions attempt to solve $Ax = b$ using the restarted general minimal residual GMRES(m) method where A is a square matrix stored in a sparse matrix format and x and b are dense vectors. This method is implemented using the modified Gram-Schmidt method for orthogonalization. When a preconditioner is specified, GMRES attempts to minimize $||Mb - MAx||/||b||$ opposed to $||b - Ax||/||b||$ in the absence of a preconditioner.

The maximum iterations, specified by `culaIterativeConfig`, are in reference to the outer iteration count. The maximum inner iteration count is specified by the `restart` value contained in the `culaGmresOptions` parameter.

Solver Trait	Value
matrix class	General
memory overhead	$n * m + 5n$

Note that the total memory overhead is directly proportional to the restart value, and so care should be taken with this parameter.

The associated preconditioner is indicated by the function suffix. See *Iterative Preconditioners* for more information.

7.5.1 Parameters

Param.	Memory	In/out	Meaning
config	host	in	configuration structure
solverOpts	host	in	culaGmresOptions structure
precondOpts	host	in	options for specified preconditioner
n	host	in	number of rows and columns in the matrix; must be ≥ 0
nnz	host	in	number of non-zero elements in the matrix; must be ≥ 0
{storage}	host	in	sparse input matrix in corresponding storage format
x	host	out	array of n data elements
b	host	in	array of n data elements
result	host	out	result structure

7.5.2 culaMinresOptions

Name	Memory	In/out	Meaning
reserved	host	in	reserved for future compatibility

7.6 Minimum residual method (MINRES)

```
culaStatus cula{storage}Minres{preconditioner}(
     const culaIterativeConfig* config,
     const culaMinresOptions* solverOpts,
     {preconditioner options},
     int n, int nnz,
     {storage parameters},
     double* x, const double* b,
     culaIterativeResult* result
);
```

culaDcsrMinres	culaDcscMinres	culaDcooMinres
culaDcsrMinresIlu0	culaDcscMinresIlu0	culaDcooMinresIlu0
culaDcsrMinresJacobi	culaDcscMinresJacobi	culaDcooMinresJacobi
culaDcsrMinresBlockjacobi	culaDcscMinresBlockjacobi	culaDcooMinresBlockjacobi

culaZcsrMinres	culaZcscMinres	culaZcooMinres
culaZcsrMinresIlu0	culaZcscMinresIlu0	culaZcooMinresIlu0
culaZcsrMinresJacobi	culaZcscMinresJacobi	culaZcooMinresJacobi
culaZcsrMinresBlockjacobi	culaZcscMinresBlockjacobi	culaZcooMinresBlockjacobi

This family of functions attempt to solve $Ax = b$ using the minimum residual method MINRES method where A is a square matrix stored in a sparse matrix format and x and b are dense vectors. When a preconditioner is specified, MINRES attempts to minimize $||Mb - MAx||/||b||$ opposed to $||b - Ax||/||b||$ in the absence of a preconditioner.

Solver Trait	Value
matrix class	General
memory overhead	$11n$

The associated preconditioner is indicated by the function suffix. See *Iterative Preconditioners* for more information.

7.6.1 Parameters

Param.	Memory	In/out	Meaning
config	host	in	configuration structure
solverOpts	host	in	culaMinresOptions structure
precondOpts	host	in	options for specified preconditioner
n	host	in	number of rows and columns in the matrix; must be ≥ 0
nnz	host	in	number of non-zero elements in the matrix; must be ≥ 0
{storage}	host	in	sparse input matrix in corresponding storage format
x	host	out	array of n data elements
b	host	in	array of n data elements
result	host	out	result structure

7.6.2 culaMinresOptions

Name	Memory	In/out	Meaning
restart	host	in	number of inner iterations at which point the algorithm restarts; directly related to memory usage

CHAPTER

EIGHT

PERFORMANCE AND ACCURACY

This chapter outlines many of the performance and accuracy considerations pertaining to the *CULA Sparse* library. There are details regarding how to get the most performance out of your solvers and provide possible reasons why a particular solver may be under performing.

8.1 Performance Considerations

8.1.1 Double Precision

All of the solvers in *CULA Sparse* perform calculations in double precision. While users of the NVIDIA GeForce line may still see an appreciable speedup, we recommend using the NVIDIA Tesla line of compute cards with greatly improved double precision performance.

Since double precision is required, a *CUDA* device supporting compute capability 1.3 is needed to use the *CULA Sparse* library. At this time, single precision solvers are excluded.

8.1.2 Problem Size

The modern GPU is optimized to handle large, massively parallel problems with a high computation to memory access ratio. As such, small problems with minimal parallelism will perform poorly on the GPU and are much better suited for the CPU where they can reside in cache memory. Typical problem sizes worth GPU-acceleration are systems with at least 10,000 unknowns and at least 30,000 non-zero elements.

8.1.3 Storage Formats

For storage and performance reasons, the compressed sparse row (CSR) format is preferred as many internal operations have been optimized for this format. For other storage formats, *CULA Sparse* will invoke accelerated conversions routines to convert to CSR internally. To measure this conversion overhead, inspect the overhead field of the timing structure in the `culaIterativeResult` return structure. These conversion routines also require an internal buffer of size nnz which for large problems may be more memory than is available on the GPU.

8.1.4 Preconditioner Selection

Proper preconditioner selection is necessary for minimizing the number of iterations required to solve a solution. However, as mentioned in previous chapters, the preconditioning step does add additional work to every iteration. In some cases this additional work may end up being a new bottleneck and improved overall performance may be obtained using a preconditioner that takes more steps to converge, but the overall runtime is actually lower. As such,

we recommend analyzing multiple preconditioner methods and looking at the total runtime as well the number of iterations required.

8.2 Accuracy Considerations

8.2.1 Numerical Precision

Iterative methods are typically very sensitive to numerical precision. Therefore, different implementations of the same algorithm may take a different number of iterations to converge to the same solution. This is as expected when dealing with the non-associative nature of floating point computations.

8.2.2 Relative Residual

Unless otherwise specified, all of *CULA Sparse's* iteration calculations are done with regards to a residual relative to the norm of the right hand side of the linear system. The defining equation for this is $||b - Ax||/||b||$.

CHAPTER

NINE

API EXAMPLE

This section shows a very simple example of how to use the *CULA Sparse* API.

```
#include "culadist/cula_sparse.h"

int main()
{
    // test data
    const int n = 8;
    const int nnz = 8;
    double a[nnz] =  { 1., 2., 3., 4., 5., 6., 7., 8. };
    double x[n] = { 1., 1., 1., 1., 1., 1., 1., 1. };
    double b[n];
    int colInd[nnz] = { 0, 1, 2, 3, 4, 5, 6, 7 };
    int rowInd[nnz] = { 0, 1, 2, 3, 4, 5, 6, 7 };

    // character buffer used for results and error messages
    char buf[256];

    // status returned by each and every cula routine
    culaStatus status;

    // intialize cula sparse library
    status = culaSparseInitialize();

    // check for initialization error
    if ( status != culaNoError )
    {
        culaGetErrorInfoString( status, culaGetErrorInfo(), buf, sizeof(buf) );
        printf("%s\n", buf);
        return;
    }

    // configuration structures
    culaIterativeConfig config;
    culaBicgOptions solverOpts;
    culaJacobiOptions precondOpts;

    // initialize values
    status = culaIterativeConfigInit( &config );
    status = culaBicgOptionsInit( &solverOpts );
    status = culaJacobiOptionsInit( &precondOpts );

    // configure specific parameters
```

```
    config.tolerance = 1.e-6;
    config.maxIterations = 20;
    config.indexing = 0;
    config.debug = 1;

    // result structure
    culaIterativeResult result;

    // call bicg with jacobi preconditioner
    status = culaDcooBicgJacobi( &config, &solverOpts, &precondOpts, n, nnz, a, colInd, rowInd, x, b,

    // see if solver failed for a non-data related reason
    if ( status != culaNoError && status != culaDataError )
    {
        culaGetErrorInfoString( status, culaGetErrorInfo(), buf, sizeof(buf) );
        printf("%s\n", buf);
        return;
    }

    // print result string
    culaIterativeResultString( &result, buf, sizeof(buf) );
    printf("%s\n", buf);

    return;
}
```

CHAPTER

TEN

CONFIGURING YOUR ENVIRONMENT

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

10.1 Microsoft Visual Studio

This section describes how to configure Microsoft Visual Studio to use *CULA Sparse*. Before following the steps within this section, take note of where you installed *CULA Sparse* (the default is *C:\Program Files\CULA\S#*). 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.

10.1.1 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 Sparse*.

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 Sparse* 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 Sparse* files in your application. Before you can compile and link an application that uses *CULA Sparse*, however, you will need to set up your project to link *CULA Sparse*.

10.1.2 Project Settings

To use *CULA Sparse*, you must instruct Visual Studio to link *CULA Sparse* 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_core.lib cula_sparse.lib".

On the Windows platform, *CULA Sparse's* libraries are distributed as a dynamic link library (DLL) (*cula_sparse.dll*) and an import library (*cula_sparse.lib*), located in the *bin* and *lib* directories of the *CULA Sparse* installation, respectively. By Linking *cula_sparse.lib*, you are instructing Visual Studio to make an association between your application and the *CULA Sparse* DLL, which will allow your application to use the code that is contained within the *CULA Sparse* DLL.

10.1.3 Runtime Path

CULA Sparse is built as a dynamically linked library, and as such it must be visible to your runtime system. This requires that cula_sparse.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 CULASPARSE_BIN_PATH_32 or CULASPARSE_BIN_PATH_64 to your PATH environment variable.
2. Copy cula_sparse.dll and its supporting dll's to the working directory or your project's executable.

10.2 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 Sparse*.

10.2.1 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 Sparse*.

On a Linux or Mac OS X system, a simple way to set up *CULA Sparse* 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 CULASPARSE_ROOT=/usr/local/culasparse
export CULASPARSE_INC_PATH=$CULASPARSE_ROOT/include
export CULASPARSE_BIN_PATH_32=$CULASPARSE_ROOT/bin
export CULASPARSE_BIN_PATH_64=$CULASPARSE_ROOT/bin64
export CULASPARSE_LIB_PATH_32=$CULASPARSE_ROOT/lib
export CULASPARSE_LIB_PATH_64=$CULASPARSE_ROOT/lib64
```

(where `CULASPARSE_ROOT` is customized to the location you chose to install *CULA Sparse*)

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

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

10.2.2 Configure Project Paths

This section describes how to set up the *gcc* compiler to include *CULA Sparse* 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$CULASPARSE_INC_PATH`
Library Path (32-bit arch)	`-L$CULASPARSE_LIB_PATH_32`
Library Path (64-bit arch)	`-L$CULASPARSE_LIB_PATH_64`
Libraries to Link against	`-lcula_core -lcula_sparse`

For a 32-bit compile:

```
gcc ... -I$CULASPARSE_INC_PATH -L$CULASPARSE_LIB_PATH_32 ...
        -lcula_core -lcula_sparse -lcublas -lcudart -lcusparse ...
```

For a 64-bit compile (not applicable to Mac OS X):

```
gcc ... -I$CULASPARSE_INC_PATH -L$CULASPARSE_LIB_PATH_64 ...
        -lcula_core -lcula_sparse -lcublas -lcudart -lcusparse ...
```

10.2.3 Runtime Path

CULA Sparse is built as a shared library, and as such it must be visible to your runtime system. This requires that *CULA Sparse'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 CULASPARSE_LIB_PATH_32 or CULASPARSE_LIB_PATH_64 to your LD_LIBRARY_PATH environment variable.
2. Edit your system's ld.so.conf (found in /etc) to include either CULASPARSE_LIB_PATH_32 or CULASPARSE_LIB_PATH_64.

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

10.3 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();
    int cusparseMinimumVersion = culaGetCusparseMinimumVersion();
    int cusparseRuntimeVersion = culaGetCusparseRuntimeVersion();

    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;
    }

    if(cusparseRuntimeVersion < cusparseMinimumVersion)
    {
        printf("CUSPARSE runtime version is insufficient; "
               "version %d or greater is required\n", cusparseMinimumVersion);
        return 0;
    }

    return 1;
}
```

CHAPTER

ELEVEN

COMMON ERRORS

This chapter provides solutions to errors commonly encountered when using the *CULA Sparse* library.

As a general note, whenever an error is encountered, consider enabling debugging mode. The configuration parameter offers a debugging flag that, when set, causes *CULA Sparse* to perform many more checks than it would normally. These checks can be computationally expensive and so are not enabled on a default run. These checks may highlight the issue for you directly, saving you from having to do more time consuming debugging. Debugging output will occur as either a `culaStatus` return code (i.e., a malformed matrix may be printed to the console, or returned via the result `culaBadStorageFormat` or via the result structure.

11.1 Argument Error

Problem

A function's `culaStatus` return code is equal to `culaArgumentError`.

Description

This error indicates that one of your parameters to your function is in error. The `culaGetErrorInfo` function will report which particular parameter is in error. Typical errors include invalid sizes or null pointers.

For a readable string that reports this information, use the `culaGetErrorInfoString()` function. Whereas normal mode will not indicate why the argument is in error, debugging mode may report more information.

Solution

Check the noted parameter against the routine's documentation to make sure the input is valid.

11.2 Malformed Matrix

Problem

A function's `culaStatus` return code is equal to `culaBadStorageFormat`.

Description

This error indicates that the set of inputs describing a sparse matrix is somehow malformed. This error code is principally encountered with the configuration structure has activated the "debug" field by setting it to 1. For a readable string that reports this information, use the `culaGetErrorInfoString()` function. Whereas normal mode will not indicate why the argument is in error, debugging mode may report more information.

There are many conditions which can trigger this, of which a few common examples are listed below.

- For 0-based indexing, any entry in the row or column values is less than zero or greater than $n - 1$

- For 1-based indexing, any entry in the row or column values is less than one or greater than n
- Duplicated indices
- Entries in an Index array are not ascending
- The $n+1$ element of an Index array is not set properly; ie it does not account for all nnz elements

There are many others, and the above may not be true for all matrix types.

Solution

Check the matrix data against the documentation for matrix storage types to ensure that it meets any necessary criteria.

11.3 Data Errors

Upon a `culaDataError` return code, it is possible to obtain more information by examining the `culaIterativeFlag` within the `culaIterativeResult` structure. This will indicate a problem with of the following errors:

11.3.1 Maximum Iterations Reached

Problem

A function's `culaStatus` return code is equal to `culaDataError` and the `culaIterativeFlag` is indicating `culaMaxItReached`.

Description

This error indicates that the solver has reached a maximum number of iterations before the an answer within the given tolerance was reached.

Solution

Increase the iteration count or lower the desired tolerance. Also, the given solver and/or preconditioner might not be appropriate for your data. If this is the case, try a different solver and/or preconditioner. It is also possible that the input matrix may not be solvable with any of the methods available in *CULA Sparse*.

This might also be a desirable outcome in the case that the user is seeking the best possible answer within a budgeted number of iterations. In this case, the "error" can be safely ignored.

11.3.2 Preconditioner Failure

Problem

A function's `culaStatus` return code is equal to `culaDataError` and the `culaIterativeFlag` is indicating `culaPreconditionerFailed`.

Description

The preconditioner failed to generate and no iterations were attempted. This error is usually specific to different preconditioner methods but typically indicates that there is either bad data (i.e., malformed matrix) or a singular matrix. See the documentation for the preconditioner used, as it may specify certain conditions which must be met.

Solution

More information can be obtained through the `culaIterativeResultString()` function. In many cases the input matrix is singular and factorization methods such as ILU0 are not appropriate. In this case, try a different preconditioner and check that the structure of your matrix is correct.

11.3.3 Stagnation

Problem

A function's `culaStatus` return code is equal to `culaDataError` and the `culaIterativeFlag` is indicating `culaStagnation`.

Description

The selected iterative solver has stagnated by calculating the same residual for multiple iterations in a row. The solver has exited early because a better solution cannot be calculated.

Solution

A different iterative solver and/or preconditioner may be necessary. It is also possible that the input matrix may not be solvable with any of the methods available in *CULA Sparse*.

It is implicit when this error is issued that the current residual still exceeds the specified tolerance, but the result may still be usable if the user is looking only for a "best effort" solution. In that case, this "error" can be disregarded.

11.3.4 Scalar Out of Range

Problem

A function's `culaStatus` return code is equal to `culaDataError` and the `culaIterativeFlag` is indicating `culaScalarOutOfRange`.

Description

The selected iterative solver has encountered an invalid floating point value during calculations. It is possible the method has broken down and isn't able to solve the provided linear system.

Solution

Therefore, a different iterative solver and/or preconditioner may be necessary. It is also possible that the input matrix may not be solvable with any of the methods available in *CULA Sparse*. Also, it is possible the input matrix has improperly formed data.

11.3.5 Unknown Iteration Error

Problem

A function's `culaStatus` return code is equal to `culaDataError` and the `culaIterativeFlag` is indicating `culaUnknownIterationError`.

Description

The selected iterative solver has encountered an unknown error.

Solution

This error is unexpected and should be reported the *CULA Sparse* development team. Please provide the full output of `culaIterativeResultString()` and `culaIterativeConfigString()`.

11.4 Runtime Error

Problem

A function's `culaStatus` return code is equal to `culaRuntimeError`.

Description

An error associated with the *CUDA* runtime library has occurred. This error is commonly seen when trying to pass a device pointer into a function that is expected a host pointer or vise-versa.

Solution

Make sure device pointers aren't being used in a host function or vise-versa.

11.5 Initialization Error

Problem

A function's `culaStatus` return code is equal to `culaNotInitialized`.

Description

This error occurs when the *CULA Sparse* library has not yet been initialized.

Solution

Call `culaSparseInitialize()` prior to any other API calls. Exceptions include device functions and helper routines.

11.6 No Hardware Error

Problem

A function's `culaStatus` return code is equal to `culaNoHardware`.

Description

No *CUDA* capable device was detected in the system.

Solution

Be sure that your system has a *CUDA* capable NVIDIA GPU device; a full list of *CUDA* GPUs can be obtained through NVIDIA's webpage: http://developer.nvidia.com/cuda-gpus

11.7 Insufficient Runtime Error

Problem

A function's `culaStatus` return code is equal to `culaInsufficientRuntime`.

Description

When explicitly mixing *CUDA* and *CULA* functionality, the *CUDA* runtime must be at least equal to the *CUDA* runtime used to build *CULA*.

Solution

Upgrade your *CUDA* install to the version that *CULA* has been built against. This is typically indicated immediately after the *CULA* version. If upgrading *CUDA* is not possible, select an older *CULA* install with an appropriate *CUDA* runtime.

11.8 Insufficient Compute Capability Error

Problem

A function's `culaStatus` return code is equal to `culaInsufficientComputeCapability`.

Description

The *CULA Sparse* library requires a minimum *CUDA* Compute Capability of 1.3 for double precision.

Solution

Be sure that your system has a *CUDA* device with at least Compute Capability 1.1; a full list of *CUDA* GPUs can be obtained through NVIDIA's webpage: http://developer.nvidia.com/cuda-gpus

11.9 Insufficient Memory Error

Problem

A function's `culaStatus` return code is equal to `culaInsufficientMemory`.

Description

Insufficient GPU memory was available to complete the requested operation. This includes storage for the input data, output data, and intermediates required by the solver.

Solution

Try another solver and/or preconditioner with a lower memory requirement. See each routine for details on how much memory is required to store the intermediate values.

CHAPTER

TWELVE

SUPPORT OPTIONS

If none of the entries in the *Common Errors* chapter solve your issue, you can seek technical support.

EM Photonics provides a user forum at http://www.culatools.com/forums at which you can seek help. Additionally, if your license level provides direct support, you may contact us directly.

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

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

12.1 Matrix Submission Guidelines

Occasionally you may need to send us your sparse matrix so that we can work with it directly. EM Photonics accepts two different sparse matrix formats:

- Matlab sparse matrices (.mat)
- Matrix-market format (.mtx)

Matlab's sparse matrices are stored in .mat files. Matlab matrices can be saved with the 'save' command or by selecting a workspace variable and selecting 'Save As'.

Matrix-market formats are discussed here: http://math.nist.gov/MatrixMarket/formats.html . This site contains routines for several languages for reading and writing to these file types.

For easy transfer to EM Photonics, please compress the matrix using one of .zip or .tar.gz compression methods.

CHAPTER

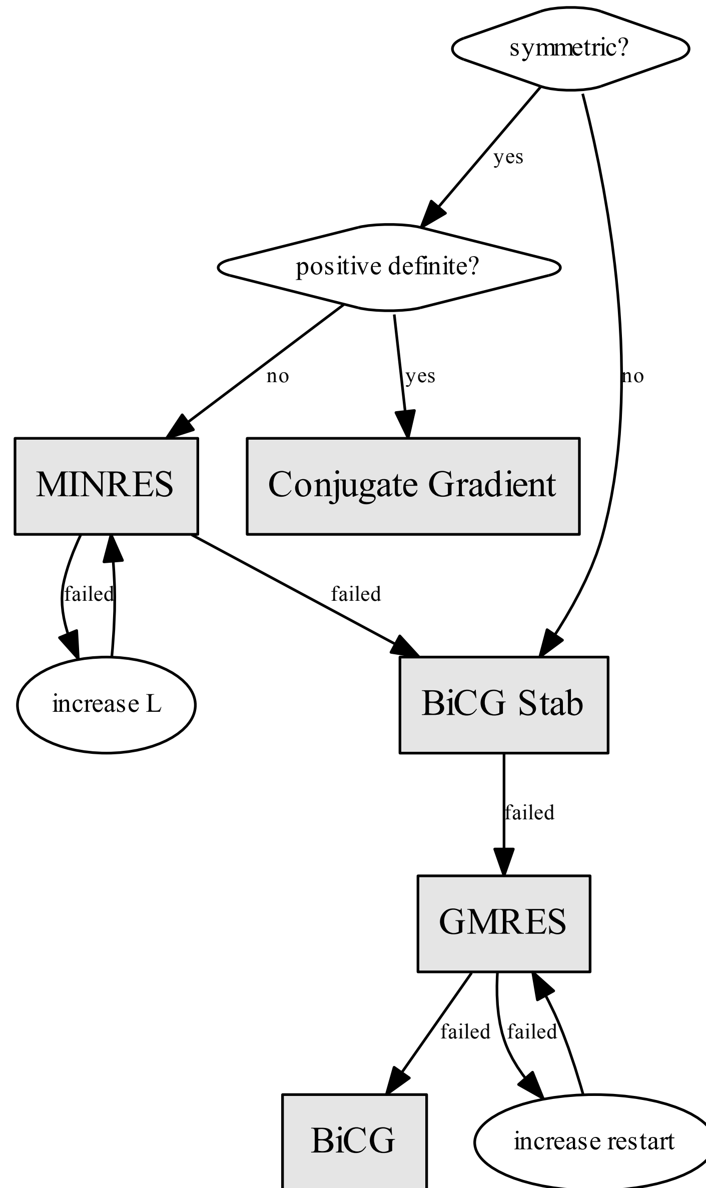
THIRTEEN

ROUTINE SELECTION FLOWCHARTS

Selecting the best sparse iterative solver and preconditioner is often a difficult decision. Very rarely can one simply know which combination will converge quickest to find a solution within the given constraints. Often the best answer requires knowledge pertaining to the structure of the matrix and the properties it exhibits. To help aid in the selection of a solver and preconditioner, we have constructed two flow charts to help gauge which solver and preconditioner might work best. Again, since there is no correct answer for any given system, we encourage users to experiment with different solvers, preconditioners, and options. These flowcharts are simply designed to give suggestions, and not absolute answers.

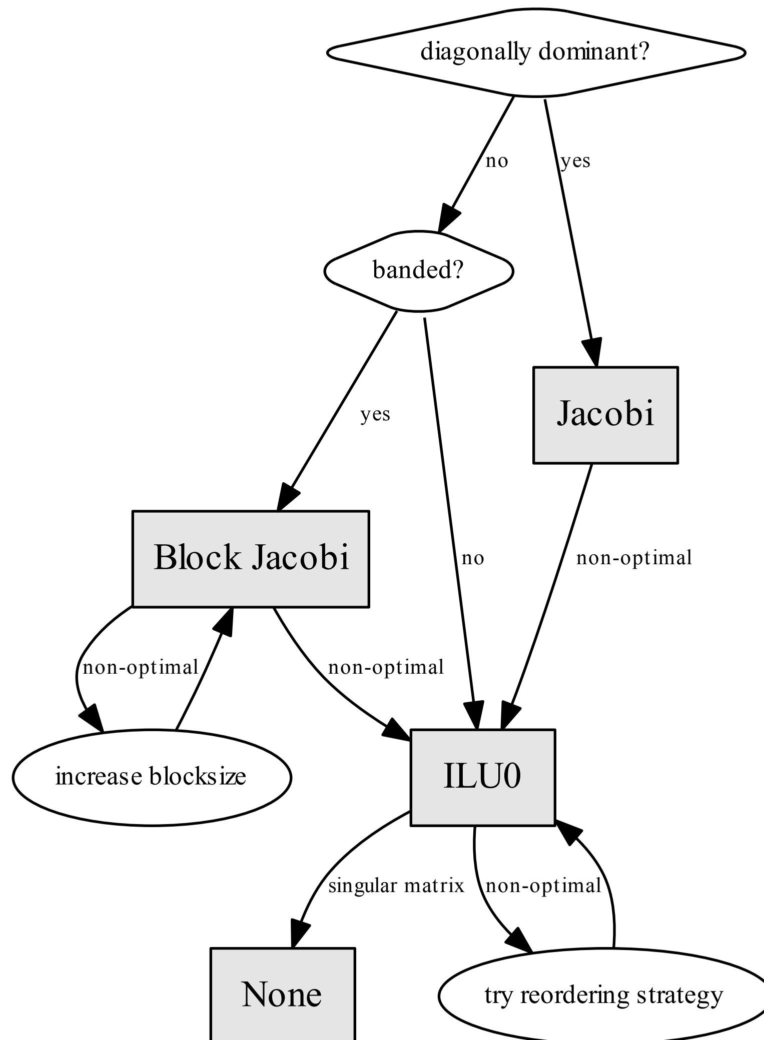
13.1 Solver Selection Flowchart

Figure 13.1: This flowchart assists in solver selection for your application.



13.2 Preconditioner Selection Flowchart

Figure 13.2: This flowchart assists in preconditioner selection for your application.



CHAPTER

FOURTEEN

CHANGELOG

14.1 Release S3 CUDA 4.2 (August 14, 2012)

- Announcement: All packages are now "universal" and contain both 32-bit and 64-bit binaries
- Feature: CUDA runtime upgraded to 4.2
- Feature: Kepler support
- Changed: Fortran module is now located in "include"

14.2 Release S2 CUDA 4.1 (January 30, 2012)

- Feature: CUDA runtime upgraded to version 4.1
- Improved: Stability of COO and CSC interfaces
- Fixed: Now shipping all dependencies required by OSX systems

14.3 Release S1 CUDA 4.0 (November 2, 2011)

- Feature: Improved speeds for all solvers
- Feature: Matrix reordering option; can lead to large perf gains for ILU
- Feature: MINRES solver
- Feature: Fully compatible with CULA R13 and above
- Feature: Option to disable stagnation checking for more speed
- Feature: Added iterativeBenchmark example for evaluating the performance of different solvers and options
- Improved: Result printout will show if useBestAnswer was invoked
- Changed: Header renamed to cula_sparse.h; transitional header available
- Notice: Integrated LGPL COLAMD package; see src folder and license

14.4 Release S1 Beta 2 CUDA 4.0 (September 27, 2011)

- Feature: BiCGSTAB solver
- Feature: BiCGSTAB(L) solver
- Feature: Complex (Z) data types available for all solvers
- Feature: Fortran module added
- Feature: Configuration parameter to return best experienced solution
- Feature: Maximum runtime configuration parameter
- Feature: New example for Fortran interface
- Feature: New example for MatrixMarket data
- Changed: Must link two libraries now (cula_sparse and cula_core)

14.5 Release S1 Beta 1 CUDA 4.0 (August 24, 2011)

- Feature: Cg, BiCg, and GMRES solvers
- Feature: CSC, CSR, COO storage formats
- Feature: Jacobi, Block Jacobi, ILU0 preconditioners
- Feature: Double precision only
- Feature: Support for all standard CUDA platforms; Linux 32/64, Win 32/64, OSX