C++ API for BLAS and LAPACK

Mark Gates	ICL[1]
Piotr Luszczek	ICL
Ahmad Abdelfattah	ICL
Jakub Kurzak	ICL
Jack Dongarra	ICL
Konstantin Arturov	Intel[2]
Cris Cecka	NVIDIA[3]
Chip Freitag	AMD[4]

[1]Innovative Computing Laboratory
[2]Intel Corporation
[3]NVIDIA Corporation
[4]Advanced Micro Devices, Inc.

February 21, 2018

ICL INNOVATIVE COMPUTING LABORATORY
THE UNIVERSITY OF TENNESSEE KNOXVILLE

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early testbed platforms, in support of the nation's exascale computing imperative.

Revision	Notes
06-2017	first publication
02-2018	• copy editing, • new cover.
03-2018	• adding a section about GPU support, • adding Ahmad Abdelfattah as an author.

```
@techreport{gates2017cpp,
  author={Gates, Mark and Luszczek, Piotr and Abdelfattah, Ahmad and
          Kurzak, Jakub and Dongarra, Jack and Arturov, Konstantin and
          Cecka, Cris and Freitag, Chip},
  title={{SLATE} Working Note 2: C++ {API} for {BLAS} and {LAPACK}},
  institution={Innovative Computing Laboratory, University of Tennessee},
  year={2017},
  month={June},
  number={ICL-UT-17-03},
  note={revision 03-2018}
}
```

Contents

CHAPTER 1

Introduction and Motivation

The Basic Linear Algebra Subprograms[1] (BLAS) and the Linear Algebra PACKage[2] (LAPACK) have been around for many decades and serve as *de facto* standards for performance-portable and numerically robust implementations of essential linear algebra functionality. Both are written in Fortran with C interfaces provided by CBLAS and LAPACKE, respectively.

BLAS and LAPACK will serve as building blocks for the Software for Linear Algebra Targeting Exascale (SLATE) project. However, their current Fortran and C interfaces are not suitable for SLATE's templated C++ implementation. The primary issue is that the data type is specified in the routine name—`sgemm` for single, `dgemm` for double, `cgemm` for complex-single, and `zgemm` for complex-double. A templated algorithm requires a consistent interface with the same function name to be called for all data types. Therefore, we are proposing a new C++ interface layer to run on top of the existing BLAS and LAPACK libraries.

We start with a survey of traditional BLAS and LAPACK libraries, with both the Fortran and C interfaces. Then we review various C++ linear algebra libraries to see the trends and features available. Finally, Chapter 3 covers our proposed C++ API for BLAS and LAPACK.

[1] http://www.netlib.org/blas/
[2] http://www.netlib.org/lapack/

CHAPTER 2

Standards and Trends

2.1 Programming Language Fortran

2.1.1 FORTRAN 77

The original FORTRAN [1] BLAS first proposed level-1 BLAS routines for vector operations with $O(n)$ work on $O(n)$ data. Level-2 BLAS routines were added for matrix-vector operations with $O(n^2)$ work on $O(n^2)$ data. Finally, level-3 BLAS routines for matrix-matrix operations benefit from the surface-to-volume effect of $O(n^2)$ data to read for $O(n^3)$ work.

Routines are named to fit within the FORTRAN 77 naming scheme's six-letter character limit. The prefix denotes the precision, like so:

- `s` single (float)
- `d` double
- `c` complex-single
- `z` complex-double

For level-2 BLAS and level-3 BLAS, a two-letter combination denotes the type of matrix, like so:

[1] FORTRAN refers to FORTRAN 77 and earlier standards. The capitalized spelling has since been abandoned, and first-letter-capitalized spelling is now preferred and used uniformly throughout the standard documents.

`ge`	general rectangular
`gb`	general band
`sy`	symmetric
`sp`	symmetric, packed storage
`sb`	symmetric band
`he`	Hermitian
`hp`	Hermitian, packed storage
`hb`	Hermitian band
`tr`	triangular
`tp`	triangular, packed storage
`tb`	triangular band

Finally, the root specifies the operation:

`axpy`	$y = \alpha x + y$
`copy`	$y = x$
`scal`	scaling $x = \alpha x$
`mv`	matrix-vector multiply, $y = \alpha Ax + \beta y$
`mm`	matrix-matrix multiply, $C = \alpha AB + \beta C$
`rk`	rank-k update, $C = \alpha AA^T + \beta C$
`r2k`	rank-$2k$ update, $C = \alpha AB^T + \alpha BA^T + \beta C$
`sv`	matrix-vector solve, $Ax = b$, A triangular
`sm`	matrix-matrix solve, $AX = B$, A triangular

So, for example, `dgemm` would be a double-precision, general matrix-matrix multiply.

The original Fortran interfaces had a number of limitations, listed below.

- **Portability issues calling Fortran:** Since Fortran is case insensitive, compilers variously use `dgemm`, `dgemm_`, and `DGEMM` as the actual function name in the binary object file. Typically, macros are used to abstract these differences in C/C++.
- **Portability issues for routines returning numbers, such as nrm2 and dot (norm and dot product)**: The Fortran standard does not specify how numbers are returned (e.g., on the stack or as an extra hidden argument), so compilers return them in various ways. The f2c and old g77 versions also returned singles as doubles, and this issue remains when using macOS Accelerate, which is based on the f2c version of LAPACK/BLAS.
- **Lacks mixed precision:** Mixed precision (e.g., $y = Ax$, where A is single, and x is double) is important for mixed-precision iterative refinement routines.
- **Lacks mixed real/complex routines:** Mixed real/complex routines (e.g., $y = Ax$, where A is complex, and x is real) occur in some eigenvalue routines.
- **Rigid naming scheme:** Since the precision is encoded in the name, the Fortran interfaces cannot readily be used in precision-independent template code (either C++ or Fortran 90).

2.1.2 BLAST XBLAS

The BLAS technical forum[2] (BLAST) added extended and mixed-precision BLAS routines, called XBLAS, with suffixes added to the routine name to indicate the extended datatypes. Using `gemm` as an example, the initial precision (e.g., `z` in `zgemm`) specified the precision of the output matrix C and scalars (α, β). For mixed precision, a suffix of the form *_a_b* was added, where each of *a* and *b* is one of the letters `s`, `d`, `c`, or `z` indicating the types of the A and B matrices, respectively. For example, in `blas_zgemm_d_z`, A is double precision (`d`), while B and C are complex-double (`z`). For extended precision, a suffix `_x` was added that specified it internally used extended precision, for example, `blas_sdot_x` is a single-precision dot product that accumulates internally using extended precision.

While these parameters added capabilities to the BLAS, several issues remain:

- **Extended precision was internalized:** Output arguments were in standard precision. For parallel algorithms, the output matrix needed to be in higher precision for reductions. For instance, a parallel `gemv` would do `gemv` on each node with the local matrix and then do a parallel reduction to find the final product. To effectively use higher precision, the result of the local `gemv` had to be in higher precision, with rounding to lower precision only after the parallel reduction. XBLAS did not provide extended precision outputs.
- **Superfluous routines:** Many of the XBLAS routines were superfluous and not useful in writing LAPACK and ScaLAPACK routines, thereby making implementation of XBLAS unnecessarily difficult.
- **Limited precision types:** XBLAS had no mechanism for supporting additional precision and did not support half-precision (16-bit); integer or quantized; fixed-point; extended precision (e.g., double-double—two 64-bit quantities representing one value), or quad precision (128-bit).
- **Not widely adopted:** The XBLAS was not widely adopted or implemented, although LAPACK can be built using XBLAS in some routines. The Intel® Math Kernel Library (Intel® MKL) also provides XBLAS implementations.

2.1.3 Fortran 90

The BLAST forum also introduced a Fortran 90 interface, which includes precision-independent wrappers around all of the routines and makes certain arguments optional with default values (e.g., assume $\alpha = 1$ or $\beta = 0$ if not given).

[2] http://www.netlib.org/blas/blast-forum/blast-forum.html

2.2 Programming Language C

2.2.1 Netlib CBLAS

The BLAS technical forum also introduced CBLAS, a C wrapper for the original Fortran BLAS routines. CBLAS addresses a couple of inconveniences that a user would face when using the Fortran interface directly from C. CBLAS allows for passing of scalar arguments by value, rather than by reference, replaces character parameters with enumerated types, and deals with the compiler's mangling of the Fortran routine names. CBLAS also supports the row-major matrix layout in addition to the standard column-major layout. Notably, this is handled without actually transposing the matrices, but is accomplished instead by changing the transposition, upper/lower, and dimension arguments. Netlib CBLAS declarations reside in the `cblas.h` header file. This file contains declarations of a handful of types:

```
typedef enum {CblasRowMajor=101, CblasColMajor=102} CBLAS_LAYOUT;
typedef enum {CblasNoTrans=111, CblasTrans=112, CblasConjTrans=113} CBLAS_TRANSPOSE;
typedef enum {CblasUpper=121, CblasLower=122} CBLAS_UPLO;
typedef enum {CblasNonUnit=131, CblasUnit=132} CBLAS_DIAG;
typedef enum {CblasLeft=141, CblasRight=142} CBLAS_SIDE;
```

and contains signatures of all the functions:

```
void cblas_dtrsm(CBLAS_LAYOUT layout, CBLAS_SIDE Side,
                 CBLAS_UPLO Uplo, CBLAS_TRANSPOSE TransA,
                 CBLAS_DIAG Diag, const int M, const int N,
                 const double alpha, const double *A, const int lda,
                 double *B, const int ldb);

void cblas_ztrsm(CBLAS_LAYOUT layout, CBLAS_SIDE Side,
                 CBLAS_UPLO Uplo, CBLAS_TRANSPOSE TransA,
                 CBLAS_DIAG Diag, const int M, const int N,
                 const void *alpha, const void *A, const int lda,
                 void *B, const int ldb);
```

Notably, Netlib CBLAS does not introduce a complex type, due to the lack of a standard C complex type at that time. Instead, complex parameters are declared as `void*`. Routines that return a complex value in Fortran are recast as subroutines in the C interface, with the return value being an output parameter added to the end of the argument list, which allows them to also be of type `void*`. Also, the name is suffixed by _sub, as shown below.

```
void   cblas_cdotu_sub(const int N, const void *X, const int incX,
                       const void *Y, const int incY, void *dotu);
void   cblas_cdotc_sub(const int N, const void *X, const int incX,
                       const void *Y, const int incY, void *dotc);
```

CBLAS contains one function, `i_amax`, in 4 precision flavors, that returns an integer value used for indexing an array. Keeping with C language conventions, it indexes from 0, instead of from 1 as the Fortran `i_amax` does. The type is `int` by default and can be changed to `long` by setting the amusing flag `WeirdNEC`, like so:

```
#ifdef WeirdNEC
   #define CBLAS_INDEX long
#else
   #define CBLAS_INDEX int
#endif

CBLAS_INDEX cblas_isamax(const int N, const float  *X, const int incX);
```

```
8  CBLAS_INDEX cblas_idamax(const int N, const double *X, const int incX);
9  CBLAS_INDEX cblas_icamax(const int N, const void   *X, const int incX);
10 CBLAS_INDEX cblas_izamax(const int N, const void   *X, const int incX);
```

In terms of style, CBLAS uses capital snake case for type names, lower snake case for function names—prefixed with `cblas_`—and Pascal case for constant names. In function signatures, CBLAS uses lower case for scalars, single capital letter for arrays, and Pascal case for enumerations. Also, CBLAS uses **`const`** for all read-only input parameters for both scalars and arrays.

To address the issue of Fortran name mangling, CBLAS allows for Fortran routine names to be upper case, lower case, or lower case with an underscore (e.g., `DGEMM`, `dgemm`, or `dgemm_`). Appropriate renaming is done by C preprocessor macros.

2.2.2 Intel MKL CBLAS

Intel MKL CBLAS follows most of the conventions of the Netlib CBLAS with two main exceptions. First, `CBLAS_INDEX` is defined as `size_t`. Second, all integer parameters are of type `MKL_INT`, which can be either 32-bit or 64-bit precision. Also, header files in Intel MKL are prefixed with `mkl_`, and, therefore, the CBLAS header file is `mkl_cblas.h`.

2.2.3 Netlib lapack_cwrapper

The lapack_cwrapper was an initial attempt to develop a C wrapper for LAPACK, similar in nature to the Netlib CBLAS. Like CBLAS, the lapack_cwrapper replaced character parameters with enumerated types, replaced passing of scalars by reference with passing by value, and dealt with Fortran name mangling. The name of the main header file was `lapack.h`.

Enumerated types included all of the types defined in CBLAS and, notably, preserved their integer values, as shown below.

```
1  enum lapack_order_type {
2              lapack_rowmajor = 101,
3              lapack_colmajor = 102 };
4
5  enum lapack_trans_type {
6              lapack_no_trans   = 111,
7              lapack_trans      = 112,
8              lapack_conj_trans = 113 };
9
10 enum lapack_uplo_type  {
11             lapack_upper       = 121,
12             lapack_lower       = 122,
13             lapack_upper_lower = 123 };
14
15 enum lapack_diag_type {
16             lapack_non_unit_diag = 131,
17             lapack_unit_diag     = 132 };
18
19 enum lapack_side_type {
20             lapack_left_side  = 141,
21             lapack_right_side = 142 };
```

At the same time, many new types were introduced to cover all the other cases of character constants in LAPACK, e.g.:

```
enum lapack_norm_type {
            lapack_one_norm       = 171,
            lapack_real_one_norm  = 172,
            lapack_two_norm       = 173,
            lapack_frobenius_norm = 174,
            lapack_inf_norm       = 175,
            lapack_real_inf_norm  = 176,
            lapack_max_norm       = 177,
            lapack_real_max_norm  = 178 };

enum lapack_symmetry_type {
            lapack_general                  = 231,
            lapack_symmetric                = 232,
            lapack_hermitian                = 233,
            lapack_triangular               = 234,
            lapack_lower_triangular         = 235,
            lapack_upper_triangular         = 236,
            lapack_lower_symmetric          = 237,
            lapack_upper_symmetric          = 238,
```

Like CBLAS, the lapack_cwrapper also used the `void*` type for passing complex arguments and applied the `const` keyword to all read-only parameters for both scalars and arrays.

Notably, lapack_cwrapper preserved all of the original application programming interface's (API's) semantics, did not introduce support for row-major layout, did not introduce any extra checks (e.g., NaN checks), and did not introduce automatic workspace allocation.

In terms of style, all names were snake case, including those for types, constants, and functions. In function signatures, lapack_cwrapper used small letters only. Function names were prefixed with `lapack_`. Notably, the name CLAPACK and prefix `clapack_` were not used, to avoid confusion with an incarnation of LAPACK that was expressed in C, by automatically translating the Fortran codes using the F2C tool. The confusing part was that, while being implemented in C, CLAPACK preserved the Fortran calling convention.

2.2.4 LAPACKE

LAPACKE is another C language wrapper for LAPACK, originally developed by Intel and later incorporated into LAPACK. Like CBLAS, LAPACKE replaces passing scalars by reference with passing scalars by value. LAPACKE also deals with Fortran name mangling in the same manner as CBLAS. Unlike CBLAS and lapack_cwrapper, though, LAPACKE did not replace character parameters with enumerate types.

And, unlike other C APIs for LAPACK, LAPACKE actually uses complex types for complex parameters and introduces `lapack_complex_float` and `lapack_complex_double`, which are set by default to `float _Complex` and `double _Complex`, respectively, relying on the definition of `_Complex` in `complex.h`.

For integers, LAPACKE uses `lapack_int`, which is defined as `int` by default and defined as `long` if the `LAPACK_ILP64` flag is set.

Also like CBLAS, the the matrix layout is the first parameter in the LAPACKE calls. Two constants are defined with CBLAS compliant integer values, shown below.

```
#define LAPACK_ROW_MAJOR               101
#define LAPACK_COL_MAJOR               102
```

However, unlike in CBLAS, support for row-major layout cannot be implemented by changing the values of transposition and lower/upper arguments. Here, the matrices have to be actually transposed.

LAPACKE offers two interfaces: (1) a higher-level interface with names prefixed by `LAPACKE_`; and (2) a lower-level interface with names prefixed by `LAPACKE_` and suffixed by `_work`.

For example:

```
lapack_int LAPACKE_zgecon( int matrix_layout, char norm, lapack_int n,
                           const lapack_complex_double* a, lapack_int lda,
                           double anorm, double* rcond );

lapack_int LAPACKE_zgecon_work( int matrix_layout, char norm, lapack_int n,
                                const lapack_complex_double* a, lapack_int lda,
                                double anorm, double* rcond,
                                lapack_complex_double* work, double* rwork );
```

In the case of `matrix_layout=LAPACK_COL_MAJOR`, the lower level interface (`_work` suffix) serves only as a simple wrapper with no extra functionality added. In the case of `matrix_layout=LAPACK_ROW_MAJOR`, the lower-level interface performs out-of-place transpositions of all the input arrays and corresponding allocations and deallocations. At the same time, the lower level interface preserves the LAPACK convention of leaving it up to the user to allocate the required workspaces.

The higher-level interface (no `_work` suffix) eliminates the requirement for the user to allocate workspaces. Instead, the workspace allocation is done inside the routine after the appropriate query for the required size.

At the same time, the higher-level interface performs NaN checks for all of the input arrays, which can be disabled if LAPACKE is compiled from source, by setting the `LAPACK_DISABLE_NAN_CHECK` flag; notably, this is not possible with the binary distribution.

2.2.5 Next-Generation BLAS: "BLAS G2"

There is a new, ongoing effort to develop the next generation of BLAS, called "BLAS G2." This BLAS G2 effort, first presented at the Batched, Reproducible, and Reduced Precision BLAS workshop,[3, 4] introduces a new naming scheme for the lower-level BLAS routines. This new scheme, which is more flexible than the single-prefix character scheme used in the original BLAS and XBLAS, uses suffixes for data types:

`r16`	half (16-bit float)
`r32`	single
`r64`	double
`c32`	complex-single
`c64`	complex-double
`r64x2`	extended double-double

[3] https://docs.google.com/document/d/1DY4ImZT1coqri2382GusXgBTTTVdBDvtD5I14QHp9OE
[4] http://www.netlib.org/utk/people/JackDongarra/WEB-PAGES/Batched-BLAS-2017/

Arguments can either share the same precision (e.g., all `r64` for traditional `dgemm`) or have mixed precisions (e.g., `blas_gemm_r32r32r64`, which has two single-precision matrices [A and B] and a double-precision matrix [C]). The new scheme also defines extensions for having different input and output matrices (e.g., C_{in} and C_{out}) and has reproducible accumulators that give the same answer regardless of the runtime choices in parallelism or evaluation order.

These additions to the scheme provide a mechanism to name the various routines. However, not all names that fit the mechanism would be implemented, and a set of recommended routines for implementation will also be defined.

While these low-level names are rather cumbersome (e.g., `blas_gemm_r64_r64_r32`), BLAS G2 also defines high-level interfaces in C++ and Fortran that overload the basic operations to simplify use. For example, in C++, `blas::gemm` would call the correct low-level routine depending on the precisions of its arguments.

2.3 C++ Programming Language

A number of C++ linear algebra libraries also exist. Most of these provide actual implementations of BLAS-like functionality in C++ rather than being simple wrappers like CBLAS and LAPACKE. Some of these libraries can also call the high-performance, vendor-optimized (traditional) BLAS. The following subsections describe some of these C++ libraries.

2.3.1 Boost and uBLAS

Boost is a widely used collection of C++ libraries covering many topics. Some of the features developed in Boost have later been adopted into the C++ standard template library (STL). As one library within Boost, uBLAS[5] provides level-1, level-2, and level-3 BLAS functionality for dense, banded, and sparse matrices. This functionality is implemented using expression templates with lazy evaluation. Basic expressions on whole matrices are easy to specify. Example `gemm` calls include:

```
1    // C = alpha A B
2    C = alpha * prod( A, B );
3
4    // C = alpha A^H B + beta C
5    noalias(C) = alpha * prod( herm(A), B ) + beta * C;
```

Here, `noalias` prevents the creation of a temporary result. While using `noalias` in this case is a bit dubious, since C is on the right hand side, the result appears to be correct. uBLAS can also access submatrices, both contiguous ranges and slices with stride between rows and columns. However, the syntax is rather cumbersome:

```
1    noalias( project( C, range(0,i), range(0,j) ))
2        = alpha * prod( project( A, range(0,i), range(0,k) ),
3                        project( B, range(0,k), range(0,j) ) )
4        + beta * project( C, range(0,i), range(0,j) );
```

[5] http://www.boost.org/doc/libs/1_64_0/libs/numeric/ublas/doc/index.html

Because the code is templated, any combination of precisions and real and complex values will work. The uBLAS interface *mostly* conforms with C++ STL containers and iterators. Triangular, symmetric, and Hermitian matrices are stored in a packed configuration, thereby saving significant space but also making operations slower. For example, uBLAS implements `spmv` rather than `symv`. It can also use full matrices for triangular solves to do both `trmm` and `tpmm`.

However, uBLAS is not multi-threaded, nor does it interface fully with vendor BLAS, although there is a way to get the matrix multiply to call MKL,[6] and there is an experimental binding to work with Automatically Tuned Linear Algebra Software (ATLAS).

There does not appear to be a way to wrap existing matrices and vectors (i.e., existing matrices and vectors have to be copied into new uBLAS matrices and vectors). Per the uBLAS FAQ, development has stagnated since 2008, so it is missing the latest C++ features and is not as fast as other libraries. Benchmarks showed it is $12–15\times$ slower than sequential Intel MKL for $n = 500$ `dgemm` on a machine running a Linux operating system, an Intel Sandy Bridge CPU, Intel icpc and GNU g++ compilers, `-O3` and `-DNDEBUG` flags, and cold cache.

Below is an example of a blocked Cholesky algorithm.

```
#include <boost/numeric/ublas/matrix.hpp>
#include <boost/numeric/ublas/vector.hpp>
#include <boost/numeric/ublas/matrix_proxy.hpp>
#include <boost/numeric/ublas/vector_proxy.hpp>
#include <boost/numeric/ublas/triangular.hpp>

using namespace boost::numeric::ublas;

template< typename T, typename Layout >
int potrf( matrix< T, Layout >& A )
{
    // Assume uplo == lower. This is a left-looking version.
    // Compute the Cholesky factorization A = L*L^H.
    int n = A.size1(), lda = n, nb = 8, info = 0;
    for (int j = 0; j < n; j += nb) {
        // Update and factorize the current diagonal block and test
        // for non-positive-definiteness.
        int jb = std::min( nb, n-j );
        // herk: A(j:j+jb, j:j+jb) -= A(j:j+jb, 0:j) * A(j:j+jb, 0:j)^H
        noalias( project( A, range(j, j+jb), range(j, j+jb) ))
            -= prod(       project( A, range(j, j+jb), range(0, j) ),
                     herm( project( A, range(j, j+jb), range(0, j) )));
        lapack_potrf( 'l', jb, &A(j, j), lda, &info );
        if (info != 0) {
            info += j;
            break;
        }
        if (j+jb < n) {
            // Compute the current block column.
            // gemm: A(j+jb:n, j:j+jb) -= A(j+jb:n, 0:j) * A(j:j+jb, 0:j)^H
            noalias( project( A, range(j+jb, n), range(j, j+jb) ))
                -= prod(       project( A, range(j+jb, n), range(0, j) ),
                         herm( project( A, range(j, j+jb), range(0, j) )));

            // trsm: A(j+jb:n, j:j+jb) = A(j+jb:n, j:j+jb) / A(j:j+jb, j:j+jb)^H  # lower
            //  ==>  A(j+jb:n, j:j+jb)^H = A(j:j+jb, j:j+jb) \ A(j+jb:n, j:j+jb)^H
            // inplace_solve doesn't compile ... don't know why
            // out-of-place solve will create a temporary. sigh.
            project( A, range(j+jb, n), range(j, j+jb) )
                = solve( project( A, range(j, j+jb), range(j, j+jb) ),
```

[6] https://software.intel.com/en-us/articles/how-to-use-boost-ublas-with-intel-mkl

```
                        project( A, range(j+jb, n), range(j, j+jb) ),
                        lower_tag() );
        }
    }
    return info;
}
```

2.3.2 Matrix Template Library: MTL 4

Matrix Template Library 4[7] (MTL 4) is a C++ library that supports dense, banded, and sparse matrices. For dense matrices, it supports row-major (default), column-major, and Morton recursive layouts. MTL 4 uses parts of Boost, and the default installation even places MTL in a subfolder of Boost. For sparse matrices, MTL 4 supports compressed row storage (CRS)/compressed sparse row (CSR), compressed column storage (CCS)/compressed sparse column (CSC), coordinate, and ELLPACK formats.

Many of the functions are global functions rather than member functions. For instance, MTL 4 uses `num_rows(A)` instead of `A.num_rows()`.

MTL 4 has extensive documentation with numerous example codes. Still, the documentation is somewhat difficult to follow, and it can be difficult to find procedures on how to do certain things or find out what features are explicitly supported.

MTL 4 has native C++ implementations for BLAS operations like matrix-multiply, so it is not limited to the four precisions of traditional BLAS. By defining `MTL_HAS_BLAS`, it will interface with traditional BLAS routines for `gemm`. Upon searching the code, it does not appear that other traditional BLAS routines are called. However, benchmarks did not reveal any difference in MTL 4's matrix-multiply performance whether `MTL_HAS_BLAS` was defined or not.

As far as obtaining MTL 4, it has an MIT open-source license, as well as a commercial Supercomputing Edition with parallel and distributed support.

Compared to uBLAS, MTL 4's syntax for accessing sub-matrices is nicer. An example is shown below.

```
    dense2D<T> Asub = sub_matrix( A, i1, i2, j1, j2 );
    // or
    dense2D<T> Asub = A[ irange(i1, i2) ][ irange(j1, j2) ];
```

Like uBLAS, MTL uses expression templates, providing efficient implementations of BLAS operations in a convenient syntax. The syntax is nicer than uBLAS, avoiding the `noalias()` and `prod()` functions.

Here are some example calls:

```
    C = alpha*A*B;

    // gemm: C = alpha A^T B + beta C
    C = alpha * trans(A)*B + beta * C;

    // gemv
    y = alpha*A*x + beta*y;
```

[7] http://new.simunova.com/index.html#en-mtl4-index-html

MTL 4 uses "move semantics" to more efficiently return matrices from functions (i.e., it does a shallow copy when returning matrices). Aliasing arguments can be an issue, however, and if MLT 4 detects some aliasing methods, it will throw an exception (e.g., in `A = A*B`). However, if there is partial overlap, aliasing will not be detected, and this must be resolved by the user by adding a temporary matrix for the intermediate result. It should be noted that traditional BLAS will not detect aliasing either. MTL 4 also throws exceptions if matrix sizes are incompatible. The user can disable exceptions by defining `NDEBUG`.

MTL 4 has triangular-vector solves (`trsv`) available in `upper_trisolve` and `lower_trisolve`, but it does not appear to support a triangular-matrix solve (`trsm`). This is an impediment to even a simple blocked Cholesky implementation. However, MTL 4 provides a recursive Cholesky implementation example. It supports recursive algorithms by providing an `mtl::recursator` that divides a matrix into quadrants (A_{11}, A_{12}, A_{21}, and A_{22}), named `north_west`, `north_east`, `south_west`, and `south_east`, respectively.

MTL 4 also supports symmetric eigenvalue problems, but it is otherwise unclear if it supports operations on symmetric matrices like `symm`, `syrk`, `syr2k`, etc. Outside of the symmetric eigenvalue problem, there is little mention of symmetric matrices, but there is an `mtl::symmetric` tag. MTL 4 interfaces with UMFPACK for sparse non-symmetric systems.

MTL also includes the following matrix solver capabilities:

- LU, with and without pivoting;
- QR orthogonalization;
- eigenvalue problems (QR iteration);
- SVD; and
- ILU(0), IC(0), IMF(s) incomplete LU, Cholesky, and multi-frontal sparse solvers.

MTL 4 optionally supports some modern C++ 11 features, including:

- move semantics (`std::move, std::forward`);
- static asserts (`static_assert`) for compile-time checks of templates (e.g., that a template type is compatible);
- initializer lists: `dense2D<T> A = {{ 3, 4 }, { 5, 6 }}`; and
- for loops using range: `for (int i : irange(size(v))) { ... }`.

Similar to uBLAS, benchmarks showed MTL 4 is around $14\times$ slower than sequential Intel MKL for $n = 500$ `dgemm` on a machine running a Linux operating system, an Intel Sandy Bridge CPU, Intel icpc and GNU g++ compilers, `-O3` and `-DNDEBUG` flags, and cold cache.

Below is an example of a blocked-Cholesky algorithm lacking `trsm`.

```
#include <boost/numeric/mtl/mtl.hpp>

template< typename T, typename Layout >
int potrf( mtl::dense2D< T, Layout >& A )
{
    // Assume uplo == lower. This is a left-looking version.
    // Compute the Cholesky factorization A = L*L^H.
    int n = num_rows(A), lda = n, nb = 8, info = 0;
    for (int j = 0; j < n; j += nb) {
        // Update and factorize the current diagonal block and test
        // for non-positive-definiteness.
        int jb = std::min( nb, n-j );
        // herk: A(j:j+jb, j:j+jb) -= A(j:j+jb, 0:j) * A(j:j+jb, 0:j)^H
```

```
14          if (j > 0) {  // throws exception on empty matrices
15              sub_matrix( A, j, j+jb, j, j+jb )
16                  -=          sub_matrix( A, j, j+jb, 0, j ) *
17                     adjoint( sub_matrix( A, j, j+jb, 0, j ));
18          }
19          lapack_potrf( 'l', jb, &A(j, j), lda, &info );
20          if (info != 0) {
21              info += j;
22              break;
23          }
24          if (j+jb < n) {
25              // Compute the current block column.
26              // gemm: A(j+jb:n, j:j+jb) -= A(j+jb:n, 0:j) * A(j:j+jb, 0:j)^H
27              if (j > 0) {
28                  sub_matrix( A, j+jb, n, j, j+jb )
29                      -=          sub_matrix( A, j+jb, n, 0, j ) *
30                         adjoint( sub_matrix( A, j, j+jb, 0, j ));
31              }
32              // trsm: A(j+jb:n, j:j+jb) = A(j+jb:n, j:j+jb) / A(j:j+jb, j:j+jb)^H  # lower
33              //  ==>  A(j+jb:n, j:j+jb)^H = A(j:j+jb, j:j+jb) \ A(j+jb:n, j:j+jb)^H
34              // This solve doesn't compile: ambiguous (perhaps a bug in their API).
35              // Also, only trsv is supported, not trsm.
36              // lower_trisolve( sub_matrix( A, j, j+jb, j, j+jb ),
37              //                 sub_matrix( A, j+jb, n, j, j+jb ),
38              //                 sub_matrix( A, j+jb, n, j, j+jb ) );
39          }
40      }
41      return info;
42  }
```

2.3.3 Eigen

Like uBLAS and MTL4, Eigen[8], another C++ template library for linear algebra, is based on C++ expression templates. Eigen seems to be a more mature product than uBLAS and MTL 4.

In addition to BLAS-type expressions, the Eigen library includes: (1) linear solvers (e.g., LU with partial pivoting or full pivoting, Cholesky, Cholesky with pivoting [for semidefinite], QR, QR with column pivoting (rank revealing), and QR with full pivoting); (2) eigensolvers (e.g., Hermitian ["Self Adjoint"], generalized Hermitian [$Ax = \lambda Bx$ where B is HPD], and non-symmetric); and (3) SVD solvers (e.g., two-sided Jacobi and bidiagonalization).

Eigen does not include a symmetric-indefinite solver (e.g., Bunch-Kaufman pivoting, Rook pivoting, or Aasen's algorithm).

Eigen's block syntax is more succinct than other libraries:

uBLAS: `project( A, range( i, i+mb ), range( j, j+nb ))`
MTL 4: `sub_matrix( A, i, i+mb, j, j+nb )`
Eigen: `A.block( i, j, mb, nb )`

[8] http://eigen.tuxfamily.org/

However, when using member functions in a template context, the syntax requires extra "template" keywords, which are annoying and clutter the code:

Eigen: `A.template block( i, j, mb, nb )`

Eigen also provides both triangular and Hermitian (self-adjoint) views on matrices, though it does not appear to offer complex-symmetric views, which are less frequently used but do occur in some applications.

As with uBLAS and MTL 4, aliasing can be an issue in Eigen. Component-wise operations, where the $C(i,j)$ output entry depends only on the $C(i,j)$ input entry of C and other matrices, are unaffected by aliasing. Some operations like transpose have an in-place version available, and Eigen detects obvious cases of aliasing in debug mode. Like in uBLAS, Eigen assumes matrix-multiply uses an alias and generates a temporary intermediate matrix unless the user adds the `.noalias()` call.

Therefore, while it makes expressions like `C = A*B` simple, more complex expressions are quickly bogged down by extra function calls (e.g., `block`, `triangularView`, `selfadjointView`, `solveInPlace`, `noalias`) and C++ syntax.

Eigen has a single class that covers both matrices and vectors. This single class also covers compile-time fixed size (good for small matrices) and runtime dynamic sizes. Rows or columns, or both, can be fixed at compile-time. Default storage is column-wise, but the user can change that via a template parameter. Eigen also has an array class for component-wise operations, like `x .* y` (in Matlab notation), and an easy conversion between matrix and array classes, as shown below.

```
1     VectorXd x(n), y(n);
2     double   r = x.transpose() * y;      // dot product
3     VectorXd w = x * y;                  // assertion error: invalid matrix product
4     VectorXd z = x.array() * y.array();  // component-wise product
```

Incompatible matrix dimensions in matrix-multiply are detected, at runtime, with an assert in debug mode. Other errors like aliasing are (sometimes) detected, and execution is then aborted with an assert. If desired, these errors can be redefined to throw C++ exceptions.

Unlike uBLAS and the open-source MTL 4 release, Eigen supports multi-threading through the Open Multi-Processing (OpenMP) API. Eigen's performance is also better than uBlas, though it is still outperformed by the vendor-optimized code in Intel's MKL. For single-threaded `dgemm`, Intel's MKL is about $2\times$ faster than Eigen for $n = 500$ (compared to MKL being $14\times$–$15\times$ faster than uBLAS and MTL 4), while for multi-threaded runs, MKL is $2\times$–$4\times$ faster than Eigen on a machine running a Linux operating system, an Intel Sandy Bridge CPU, GNU g++ compiler, `-O3` and `-DNDEBUG` flags, and cold cache. Performance is notably worse with the Intel icpc compiler.

However, Eigen can directly call BLAS and LAPACK functions by setting `EIGEN_USE_MKL_ALL`, `EIGEN_USE_BLAS`, or `EIGEN_USE_LAPACKE`. With these options, Eigen's performance ranges from nearly the same as Intel's MKL to $2\times$ slower.

Below is an example of a Cholesky factorization using the Eigen template.

```
1  #include <Eigen>
2
3  template< typename T, int Rows, int Cols, int Layout >
4  int potrf( Eigen::Matrix< T, Rows, Cols, Layout >& A )
```

```
{
    // Assume uplo == lower. This is a left-looking version.
    // Compute the Cholesky factorization A = L*L^H.
    int n = A.rows(), lda = n, nb = 8, info = 0;
    for (int j = 0; j < n; j += nb) {
        // Update and factorize the current diagonal block and test
        // for non-positive-definiteness.
        int jb = std::min( nb, n-j );
        // herk: A(j:j+jb, j:j+jb) -= A(j:j+jb, 0:j) * A(j:j+jb, 0:j)^H
        A.template block( j, j, jb, jb )
         .template selfadjointView< Eigen::Lower >()
         .rankUpdate( A.template block( j, 0, jb, j ), -1.0 );
        lapack_potrf( 'l', jb, &A(j, j), lda, &info );
        if (info != 0) {
            info += j;
            break;
        }
        if (j+jb < n) {
            // Compute the current block column.
            // gemm: A(j+jb:n, j:j+jb) -= A(j+jb:n, 0:j) * A(j:j+jb, 0:j)^H
            A.template block( j+jb, j, n - (j + jb), jb ) -=
                A.template block( j+jb, 0, n - (j + jb), j ) *
                A.template block( j, 0, jb, j ).adjoint();
            // trsm: A(j+jb:n, j:j+jb) = A(j+jb:n, j:j+jb) * A(j:j+jb, j:j+jb)^{-H}  # lower
            A.template block( j, j, jb, jb )
             .template triangularView< Eigen::Lower >().adjoint()
             .template solveInPlace< Eigen::OnTheRight >(
                A.template block( j+jb, j, n - (j + jb), jb ) );
        }
    }
    return info;
}
```

2.3.4 Elemental

Elemental[9] is a Message Passing Interface (MPI) based, distributed-memory linear algebra library that includes a C++ interface for BLAS (excluding band and packed formats) and a selection of LAPACK routines. Elemental's BLAS interface is in the `El::blas` namespace, and functions are named after the traditional BLAS routines, with the exception of the precision prefix, in Pascal case. For the four standard precisions (single, double, complex-single, and complex-double), Elemental calls an optimized BLAS library. It also offers a templated C++ reference implementation for arbitrary numeric datatypes like `int` or `double-double`.

Hermitian and symmetric routines are extended to all precisions. For example, Herk ($C = \alpha AA^H + \beta C$, C is Hermitian) and Syrk ($C = \alpha AA^T + \beta C$, C is symmetric) are both available for real and complex data types. Dot products are also defined for both real and complex. This allows for templated code to use the same name for all data types.

Arguments in the Elemental wrappers are similar to the traditional BLAS and LAPACK arguments, including options, dimensions, leading dimensions, and scalars. Dimensions use `int`, and there is experimental support for 64-bit integers. Options are a single character, corresponding to the traditional BLAS options; this differs from CBLAS, which uses enums for options. For instance, a `NoTrans`, `Trans` matrix-matrix multiply ($C = \alpha AB^T + \beta C$) is expressed as:

```
    El::blas::Gemm( 'N', 'T', m, n, k, alpha, A, lda, B, ldb, beta, C, ldc );
```

[9] http://libelemental.org/

Elemental wraps a handful of LAPACK routines, with most of these dealing with eigenvalue and singular value problems. Instead of functions using the LAPACK acronym names (e.g., `syevr`), Elemental uses descriptive English names (e.g., `HermitianEig`).

In LAPACK, eigenvalue routines have a job parameter that specifies whether to compute just eigenvalues or to also compute eigenvectors. Some routines also have range parameters to specify computing only a portion of the eigen/singular value spectrum. In Elemental's wrappers, these different jobs are provided by overloaded functions, thereby avoiding the need to specify the job parameter and unused dummy arguments. See below.

```
// factor A = Z lambda Z^H, eigenvalues lambda and eigenvectors Z
HermitianEig( uplo, n, A, lda, lambda,                 tol=0 )  // lambda only
HermitianEig( uplo, n, A, lda, lambda, Z, ldz,         tol=0 )  // lambda and Z
HermitianEig( uplo, n, A, lda, lambda,         il, iu, tol=0 )  // il-th to iu-th lambda
HermitianEig( uplo, n, A, lda, lambda, Z, ldz, il, iu, tol=0 )  // il-th to iu-th lambda and Z
HermitianEig( uplo, n, A, lda, lambda,         vl, vu, tol=0 )  // lambda in (vl, vu]
HermitianEig( uplo, n, A, lda, lambda, Z, ldz, vl, vu, tol=0 )  // lambda in (vl, vu] and Z
```

Elemental also provides wrappers around certain functionalities provided in MPI, the Scalable Linear Algebra PACKage (ScaLAPACK), Basic Linear Algebra Communication Subprograms (BLACS), Parallel Basic Linear Algebra Subprograms (PBLAS), libFLAME, and the Parallel Multiple Relatively Robust Representations (PMRRR) library.

Elemental throws the (`SingularMatrixException` and `NonHPDMatrixException`) C++ exceptions for runtime numerical issues.

Elemental defines a dense matrix class (`Matrix`), a distributed-memory matrix class (`DistMatrix`), and sparse matrix classes (`SparseMatrix` and `DistSparseMatrix`). The `Matrix` class is templated on data type only. Elemental uses a column-major LAPACK matrix layout, with a leading dimension that may be explicitly specified as an option—unlike most other C++ libraries reviewed here.

A Matrix can also be constructed as a view to an existing memory buffer, as shown below.

```
    Matrix<double> A( m, n, data, lda );
```

Numerous BLAS, BLAS-like, LAPACK, and other algorithms are defined for Elemental's matrix types. In contrast to the lightweight wrappers described above, the dimensions are implicitly known from matrix objects, rather than being passed explicitly. Options are specified by enums instead of by character values; however, the enums are named differently than they are in CBLAS. In particular, Elemental has an `Orientation` enum instead of `Transpose`, with values `El::NORMAL`, `El::TRANSPOSE`, and `El::ADJOINT` corresponding to `NoTrans`, `Trans`, and `ConjTrans`, respectively. In addition to standard BLAS routines, Elemental provides the following routines, among others.

Adjoint	out-of-place conjugate transpose, $B = A^H$
Axpy	add matrices, $Y = \alpha X + Y$
Broadcast	parallel broadcast
DiagonalScale	$X = \text{op}(D)X$
Dot	matrix Hilbert-Schmidt inner product, $\text{vec}(A)^H \text{vec}(B)$
Hadamard	element-wise product, $C = A \circ B$
QuasiTrsm	Schur-form quasi-triangular solve
Reduce	parallel reduction
Transpose	out-of-place transpose, $B = A^T$
Trrk	Rank-k update limited to triangular portion (e.g., useful for syrk-like update $C = \alpha AB + \beta C$ when AB is known to be symmetric; cf. syrkx in cuBLAS and gemmt in Intel MKL)
TwoSidedTrmm	$A = L^H AL$
TwoSidedTrsm	$A = L^{-1} AL^{-H}$

In addition to standard LAPACK algorithms, Elemental provides pivoted Cholesky, non-pivoting LU, and complete-pivoting LU. It also has a number of other matrix factorizations and applications like as pseudospectra, polar decomposition, matrix square root, and matrix sign function.

The syntax for accessing submatrices is very concise, using the `IR( low, hi )` integer range class, which provides the half-open range (`low`, `hi`), shown below.

```
1     Matrix<double> A( m, n );
2     auto Asub = A( IR(j, j+jb), IR(j, n) );
```

Because C++ cannot take a non-`const` reference of a temporary, the output submatrix of each call must be a local variable. For example, one cannot write:

```
1         El::Herk( El::LOWER, El::NORMAL,
2                   -1.0, A( IR(j,j+jb), IR(0,j) ),
3                    1.0, A( IR(j,j+jb), IR(j,j+jb) ) );
```

Instead, one must make the local variable `Ajj`:

```
1         auto Ajj = A( IR(j,j+jb), IR(j,j+jb) );
2         El::Herk( El::LOWER, El::NORMAL,
3                   -1.0, A( IR(j,j+jb), IR(0,j) ),
4                    1.0, Ajj );
```

Elemental could resolve this issue by adding an overloaded version of Herk and other routines using a C++ 11 `rvalue` reference (&&) for the output matrix. Thanks to Vincent Picaud for pointing this out.

Below is an example of a Cholesky factorization in Elemental.

```
1  #include <El.h>
2
3  // throws NonHPDMatrixException
4  template< typename T >
5  void potrf( El::Matrix<T>& A )
6  {
7      assert( A.Height() == A.Width() );
8      int n = A.Height();
9      int nb = 8;
10
11     using El::IR;
```

```
    // Assume uplo == lower. This is a left-looking version.
    // Compute the Cholesky factorization A = L*L^H.
    for (int j = 0; j < n; j += nb) {
        // Update and factorize the current diagonal block and test
        // for non-positive-definiteness.
        int jb = std::min( nb, n-j );
        // herk: A(j:j+jb, j:j+jb) -= A(j:j+jb, 0:j) * A(j:j+jb, 0:j)^H
        auto Ajj = A( IR(j,j+jb), IR(j,j+jb) );
        El::Herk( El::LOWER, El::NORMAL,
                  -1.0, A( IR(j,j+jb), IR(0,j) ),
                   1.0, Ajj );
        El::Cholesky( El::LOWER, Ajj );
        if (j+jb < n) {
            // Compute the current block column.
            // gemm: A(j+jb:n, j:j+jb) -= A(j+jb:n, 0:j) * A(j:j+jb, 0:j)^H
            auto Acol = A( IR(j+jb,n), IR(j,j+jb) );
            El::Gemm( El::NORMAL, El::ADJOINT,
                      -1.0, A( IR(j+jb,n), IR(0,j) ),
                            A( IR(j,j+jb), IR(0,j) ),
                       1.0, Acol );
            // trsm: A(j+jb:n, j:j+jb) = A(j+jb:n, j:j+jb) * A(j:j+jb, j:j+jb)^{-H}  # lower
            El::Trsm( El::RIGHT, El::LOWER, El::ADJOINT, El::UNIT,
                      1.0, A( IR(j,j+jb), IR(j,j+jb) ),
                           Acol );
        }
    }
}
```

2.3.5 Intel DAAL

Intel's Data Analytics Acceleration Library (DAAL)[10] provides highly optimized algorithmic building blocks for data analysis and includes provisions for preprocessing, transformation, analysis, modeling, and validation. DAAL also provides routines for principal component analysis, linear regression, classification, and clustering. DAAL is designed to handle data that is too big to fit in memory, and, instead, it processes data as chunks—a mode of operation that can be referred to as "out-of-core." DAAL is also designed for distributed processing using popular data analytics platforms like Hadoop, Spark, R, and Matlab, and can access data from memory, files, and Structured Query Language (SQL) databases.

Intel DAAL calls BLAS through wrappers, which are defined as static members of the `Blas` class template. For example, a call to the `SYRK` function in the `computeXtX` method of the `ImplicitALSTrainKernelCommon` class looks like this:

```
#include "service_blas.h"
template <typename algorithmFPType, CpuType cpu>
void computeXtX(size_t *nRows, size_t *nCols, algorithmFPType *beta,
                algorithmFPType *x, size_t *ldx,
                algorithmFPType *xtx, size_t *ldxtx)
{
    char uplo = 'U';
    char trans = 'N';
    algorithmFPType alpha = 1.0;
    Blas<algorithmFPType, cpu>::xsyrk(&uplo, &trans,
                                      (DAAL_INT *)nCols, (DAAL_INT *)nRows,
                                      &alpha, x, (DAAL_INT *)ldx,
                                      beta, xtx, (DAAL_INT *)ldxtx);
}
```

[10] https://software.intel.com/en-us/intel-daal

The `service_blas.h` header file (shown below) contains the definition of the `Blas` class template.

```
#include "service_blas_mkl.h"
template<typename fpType, CpuType cpu, template<typename, CpuType> class _impl=mkl::MklBlas>
struct Blas
{
    typedef typename _impl<fpType,cpu>::SizeType SizeType;
    static void xsyrk(char *uplo, char *trans, SizeType *p, SizeType *n,
                      fpType *alpha, fpType *a, SizeType *lda,
                      fpType *beta, fpType *ata, SizeType *ldata)
    {
        _impl<fpType,cpu>::xsyrk(uplo, trans, p, n, alpha, a, lda, beta, ata, ldata);
    }
```

This file, in turn, relies on the `mkl::MklBlas` class template, defined in `service_blas_mkl.h`, which contains partial specializations of the BLAS routines for double precision and single precision.

Double precision:

```
template<CpuType cpu>
struct MklBlas<double, cpu>
{
    typedef DAAL_INT SizeType;
    static void xsyrk(char *uplo, char *trans, DAAL_INT *p, DAAL_INT *n,
                      double *alpha, double *a, DAAL_INT *lda,
                      double *beta, double *ata, DAAL_INT *ldata)
    {
        __DAAL_MKLFN_CALL(blas_, dsyrk, (uplo, trans, p, n, alpha, a, lda, beta, ata, ldata));
    }
```

Single precision:

```
template<CpuType cpu>
struct MklBlas<float, cpu>
{
    typedef DAAL_INT SizeType;
    static void xsyrk(char *uplo, char *trans, DAAL_INT *p, DAAL_INT *n,
                      float *alpha, float *a, DAAL_INT *lda,
                      float *beta, float *ata, DAAL_INT *ldata)
    {
        __DAAL_MKLFN_CALL(blas_, ssyrk, (uplo, trans, p, n, alpha, a, lda, beta, ata, ldata));
    }
```

The call passes through a couple of macro definitions, as shown below.

```
#define __DAAL_MKLFN_CALL(f_pref,f_name,f_args)  __DAAL_MKLFN_CALL1(f_pref,f_name,f_args)

#define __DAAL_MKLFN_CALL1(f_pref,f_name,f_args)                    \
    if(avx512 == cpu)                                               \
    {                                                               \
        __DAAL_MKLFN(avx512_,f_pref,f_name) f_args;                 \
    }                                                               \

#define __DAAL_MKLFN(f_cpu,f_pref,f_name)        __DAAL_CONCAT4(fpk_,f_pref,f_cpu,f_name)

#if !defined(__DAAL_CONCAT4)
    #define __DAAL_CONCAT4(a,b,c,d) __DAAL_CONCAT41(a,b,c,d)
    #define __DAAL_CONCAT41(a,b,c,d) a##b##c##d
#endif
```

The call then reaches the actual reference to an Intel MKL function (e.g., `avx512_blas_syrk()`).

Calls to LAPACK are handled in a similar manner. Intel DAAL calls LAPACK through wrappers, which are defined as static members of the `Lapack` class template.

For example, a call to the `POTRF` function in the `solve` method of the `ImplicitALSTrainKernelBase` class looks like this:

```
#include "service_lapack.h"
template <typename algorithmFPType, CpuType cpu>
void ImplicitALSTrainKernelBase<algorithmFPType, cpu>::solve(
    size_t *nCols,
    algorithmFPType *a, size_t *lda,
    algorithmFPType *b, size_t *ldb)
{
    char uplo = 'U';
    DAAL_INT iOne = 1;
    DAAL_INT info = 0;
    Lapack<algorithmFPType, cpu>::xxpotrf(&uplo, (DAAL_INT *)nCols,
                                          a, (DAAL_INT *)lda, &info);
```

The `service_lapack.h` header file (shown below) contains the definition of the `Lapack` class template.

```
#include "service_lapack_mkl.h"
template<typename fpType, CpuType cpu, template<typename, CpuType> class _impl=mkl::MklLapack>
struct Lapack
{
    typedef typename _impl<fpType,cpu>::SizeType SizeType;
    static void xxpotrf(char *uplo, SizeType *p,
                        fpType *ata, SizeType *ldata, SizeType *info)
    {
        _impl<fpType,cpu>::xxpotrf(uplo, p, ata, ldata, info);
    }
```

This, in turn, relies on the `mkl::MklLapack` class template, defined in `service_lapack_mkl.h`, which contains partial specializations of the LAPACK routines for double precision and single precision.

Double precision:

```
template<CpuType cpu>
struct MklLapack<double, cpu>
{
    typedef DAAL_INT SizeType;
    static void xpotrf(char *uplo, DAAL_INT *p, double *ata, DAAL_INT *ldata, DAAL_INT *info)
    {
        __DAAL_MKLFN_CALL(lapack_, dpotrf, (uplo, p, ata, ldata, info));
    }
```

Single precision:

```
template<CpuType cpu>
struct MklLapack<float, cpu>
{
    typedef DAAL_INT SizeType;
    static void xpotrf(char *uplo, DAAL_INT *p, float *ata, DAAL_INT *ldata, DAAL_INT *info)
    {
        __DAAL_MKLFN_CALL(lapack_, spotrf, (uplo, p, ata, ldata, info));
    }
```

In summary, Intel DAAL calls BLAS and LAPACK through static member functions of the `Blas` and `Lapack` class templates. Also, Intel DAAL uses the legacy BLAS calling convention (Fortran), where parameters are passed by reference, and there is no parameter to specify the layout

(column-major or row-major). Finally, Intel DAAL only contains templates for the BLAS and LAPACK functions that it actually uses, and it only contains specializations for single precision and double precision.

One potential problem with making the datatype a class template parameter is supporting mixed or extended precision—the class has only one datatype, and it is unclear how to extend it to multiple datatypes.

2.3.6 Trilinos

Trilinos[11] is a collection of open-source software libraries, called packages, linked together by a common infrastructure and intended to be used as building blocks for the development of scientific applications. Trilinos was developed at Sandia National Laboratories from a core group of existing algorithms and utilities. Trilinos supports distributed-memory parallel computation through MPI and has growing support for shared-memory parallel computation and GPUs. This happens by the means of the Kokkos package, which provides a common C++ interface over various parallel programming models, including OpenMP, POSIX threads (pthreads), and CUDA.

Trilinos provides two sets of wrappers that interface with BLAS and LAPACK. The more generic interface is contained in the Teuchos package, while a much more concrete implementation is included in the Epetra package. One worthwhile feature of both of these interfaces is that the actual BLAS or LAPACK function call is nearly identical between the two. The only difference is the instantiation of the library object. That object serves as a pseudo namespace for all the subsequent calls to the wrapper functions. See the examples below for more details.

Another shared aspect of both packages is that only the column-major order of matrix elements is supported, and no provisions are made for a row-major layout.

Teuchos

Teuchos is main package within Trilinos that provides the BLAS and LAPACK interfaces. More precisely, there are two subpackages that constitute an interface: (1) `Teuchos::BLAS` and (2) `Teuchos::LAPACK`. These two subpackages constitute a rather thin layer on top of the existing linear algebra libraries, especially when compared with the rest of the features and software services that Teuchos provides (e.g., memory management, message passing, operating system portability).

The interface is heavily templated. The first two template parameters refer to (1) the numeric data type for matrix/vector elements and (2) the integral type for dimensions. In addition, traits are used throughout Teuchos in a manner similar to the string character traits in the standard C++ library. `MagnitudeType` corresponds to the magnitude of scalars, with a corresponding trait method, `squareroot`, that enforces non-negative arguments through the type system. `ScalarType` is used for scalars, and its trait methods include `magnitude` and `conjugate`.

In addition to a generic interface and the wrappers around low-level BLAS and LAPACK

[11] https://trilinos.org/

routines, Teuchos also contains reference implementations of a majority of BLAS routines. The implementations are vector-oriented and unlikely to yield efficient code, but they are useful for instantiation of Teuchos for more exotic data types that are not necessarily supported by hardware.

An example code that calls level-1 BLAS is shown below.

```
#include "Teuchos_BLAS.hpp"
int example(int n, double alpha, double *x, int incx) {
  // instantiate BLAS class for integer dimensions and double-precision numerics
  Teuchos::BLAS<int, double> blas;
  blas.SCAL( n, alpha, x, incx );
  return blas.IAMAX( n, x, incx );
}
```

Below is an example code that invokes dense solver routines for a system of linear equations given by a square matrix.

```
#include "Teuchos_LAPACK.hpp"
void example(int n, int nrhs, double *A, int ldA, int *piv, double *B, int& info) {
  Teuchos::LAPACK<int, double> lapack;
  lapack.GETRF(n, n, A, ldA, piv, &info);
  lapack.GETRS('N', n, nrhs, A, ldA, piv, B, ldB, &info);
}
```

Note the use of character integral types instead of enumerated types for standard LAPACK enumeration parameters. Also, the error handling requires explicit use of an integral type commonly referred to as `info`.

The LAPACK routines available in the `Teuchos::LAPACK` class are called through member functions that are not "inlined."

```
// File Teuchos_LAPACK.hpp
namespace Teuchos {
  template<typename OrdinalType, typename ScalarType>
  class LAPACK
  {
  public:
   void POTRF(const char UPLO, const OrdinalType n,
   ScalarType* A, const OrdinalType lda, OrdinalType* info) const;
  }
}
```

This separates declaration from the implementation and adds the additional overhead of a non-virtual member call (see below).

```
// File Teuchos_LAPACK.cpp
namespace Teuchos {
void LAPACK<int, float>::POTRF(const char UPLO, const int n,
  float* A, const int lda, int* info) const {
    SPOTRF_F77(CHAR_MACRO(UPLO), &n, A, &lda, info);
  }
}
```

Note that the implementation contains a macro that resolves the name-mangling scheme generated by the FORTRAN 77 compiler. This creates an implicit coupling at link time between Teuchos and the LAPACK implementation that depends on the name-mangling scheme. As a result, multiple implementations of the Teuchos LAPACK wrapper must exist on the target platform for every name mangling scheme of interest—unless only one mangling scheme is enforced across all LAPACK implementations.

The `Teuchos::LAPACK` class is templated with dimension template types (`OrdinalType`) and storage template types (`ScalarType`) for LAPACK matrices, vectors, and scalars. This may lead to a problem with excessive growth of the compiler-generated object code: consider all integral types available in the C/C++ languages (**`short`**, **`int`**, **`long`**, and **`long long`**) combined with three floating-point precisions (single, double, and extended) combined with real or complex values. This would lead to $4 \times 3 \times 2 = 24$ implementations that may be instantiated by the sparse solver that uses Teuchos. The standard implementations of BLAS and LAPACK are only available for 32-bit and 64-bit integers/pointers in two precisions for real and complex values (eight versions in total).

The `Teuchos::LAPACK` class has to be instantiated explicitly before any linear algebra routines can be called. The cost of construction could be optimized by using static methods, but the support for such optimization might only be supported in the newer C++ standards and the compilers that implement them. More specifically, the two optimization choices could be a non-template static function in a templated class versus a templated static function in a non-templated class. Neither of these choices are available in Teuchos, which uses non-templated member methods to invoke the LAPACK implementation routines. To reduce the overhead of constructing a `Teuchos::LAPACK` class object for every calling scope, the user may choose to keep a global object for all calls. It is worth considering that—because the constructor is empty and defined in the header file—a simple code inlining would likely eliminate the construction overhead altogether.

A similar argument applies to the object destruction, with the caveat that the destructor was made virtual, which might trigger the creation of the *vtable*. This is despite the fact that it is hard to imagine the need for a virtual destructor, because deriving from the base `Teuchos::LAPACK` class is an unlikely route—owing to the lack of internal state and the fact that the LAPACK interface is stable in syntax and semantics, with only occasional additions of new routines. However, Teuchos contains an additional abstract interface layer that derives from the base `Teuchos::LAPACK` class to accommodate various matrix and vector objects. More concretely, the band, dense, QR, and SPD (symmetric positive definite) solvers derive from the base class to call the specific LAPACK routines' wrappers (see below).

```
namespace Teuchos {
  template<typename OrdinalType, typename ScalarType> class SerialBandDenseSolver
  : public CompObject,
    public Object,
    public BLAS<OrdinalType, ScalarType>,
    public LAPACK<OrdinalType, ScalarType> ;
  template<typename OrdinalType, typename ScalarType> class SerialDenseSolver
  : public CompObject,
    public Object,
    public BLAS<OrdinalType, ScalarType>,
    public LAPACK<OrdinalType, ScalarType> ;
  template<typename OrdinalType, typename ScalarType> class SerialQRDenseSolver
  : public CompObject,
    public Object,
    public BLAS<OrdinalType, ScalarType>,
    public LAPACK<OrdinalType, ScalarType> ;
  template<typename OrdinalType, typename ScalarType> class SerialSpdDenseSolver
  : public CompObject,
    public Object,
    public BLAS<OrdinalType, ScalarType>,
    public LAPACK<OrdinalType, ScalarType> ;
}
```

These derived classes contain generic methods for factorization (using `factor()`), solving-

with-factors (using `solve()`), and inversion (using `invert()`). Additional methods may include equilibration, error estimation, and conditioning estimation.

For completeness, it should be mentioned that Teuchos includes additional objects and functions that could be used to perform linear algebra operations. This additional interface layer is above the level of abstraction, which is the aim of the present document. An example code that calls a linear solve is shown below.

```
#include "Teuchos_SerialDenseMatrix.hpp"
#include "Teuchos_SerialDenseSolver.hpp"
#include "Teuchos_RCP.hpp" // reference-counted pointer
#include "Teuchos_Version.hpp"

void example(int n) {
  Teuchos::SerialDenseMatrix<int,double> A(n, n);
  Teuchos::SerialDenseMatrix<int,double> X(n,1), B(n,1);
  Teuchos::SerialDenseSolver<int,double> solver;
  solver.setMatrix( Teuchos::rcp( &A, false ) );
  solver.setVectors( Teuchos::rcp( &X, false ), Teuchos::rcp( &B, false ) );

  A.random();
  X.putScalar(1.0); // set X to all 1's
  B.multiply( Teuchos::NO_TRANS, Teuchos::NO_TRANS, 1.0, A, X, 0.0 );
  X.putScalar(0.0); // set X to all 0's

  info = solver.factor();
  info = solver.solve();
}
```

Epetra

Epetra abbreviates “essential Petra,” the foundational functionality of Trilinos that aims, above all else, for portability across hardware platforms and compiler versions. As such, Epetra shuns the use of templates, and thus its code is much closer to hardware and implementation artifacts.

Complex valued matrix elements are not supported by either `Epetra_BLAS` or `Epetra_LAPACK`. Only single-precision and double-precision real interfaces are provided.

An example is provided below.

```
#include <Epetra_BLAS.h>
void example(int n, float *fx, double *dx, int inc, float& fsum, double& dsum) {
  Epetra_BLAS() blas;
  fsum = blas.ASUM(n, fx, inc);
  dsum = blas.ASUM(n, dx, inc);
}
```

An example code that invokes dense solver routines for a system of linear equations given by a square matrix is shown below.

```
#include <Epetra_LAPACK.h>
void example(int n, int nrhs, double *A, int ldA, int *piv, double *B, int& info) {
  Epetra_LAPACK() lapack;
  lapack.GETRF(n, n, A, ldA, piv, &info);
  lapack.GETRS('N', n, nrhs, A, ldA, piv, B, ldB, &info);
}
```

CHAPTER 3

C++ API Design

3.1 Stateless Interface

The proposed API shall be stateless, and any implementation-specific setting will be handled outside of this interface. Initialization and library cleanup will be performed with calls that are specific to the BLAS and LAPACK implementations, if any such operations are required.

Rationale: It is possible to include the state within the layer of the C++ interface, which could then be manipulated with calls not available in the original BLAS and LAPACK libraries. However, this creates confusion when the same call with the same call arguments behaves differently due to the hidden state, and so this idea was not implemented. The only way for the user to ensure consistent behavior for every call would be to switch the internal state to the desired setting. Even then, there is still the issue of threaded and asynchronous calls that could alter the internal state in between the state reset and, for example, the factorization call.

3.2 Supported BLAS and LAPACK Storage Types

In order to support templated algorithms, BLAS and LAPACK need to have precision-independent names (e.g., `gemm` instead of `sgemm`, `dgemm`, `cgemm`, or `zgemm`). This will also provide future compatibility with mixed and extended precisions, where the arguments have different precisions as proposed by the Next Generation BLAS (Section 2.2.5). A further goal is to make function calls consistent across all data types, thereby resolving any differences that currently exist.

Our C++ API defines a set of overloaded wrappers that call the traditional vendor-optimized

BLAS and LAPACK routines. Our initial implementation focuses on full matrices (with "ge," "sy," "he," and "tr" prefixes). It is also readily extendable to band matrices (with "gb," "sb," and "hb" prefixes) and packed matrices (with "sp," "hp," and "tp" prefixes).

3.3 Acceptable Constructs and C++ Language Standard

The C++ language standard has a long history, which results in practical considerations that we try to adapt into this document. In short, the very latest version of the standard is rarely implemented across the majority of compilers and supporting tools. Consequently, it is wise to restrict the range of constructs and limit the syntax in a working code to a subset of one of the standard versions. Accordingly, we will use only the features from the C++11 standard due to its wide acceptance by the software and hardware platforms that we target.

3.4 Naming Conventions

C++ interfaces to BLAS routines and associated constants are in the `blas` namespace, and they are made available by including the `blas.hh` header, as shown below.

```
1 #include <blas.hh>
2
3 using namespace blas;
```

The C++ interfaces to LAPACK routines are in the `lapack` namespace, and they are made available by including the `lapack.hh` header, as shown below.

```
1 #include <lapack.hh>
2
3 using namespace lapack;
```

Most C++ routines use the same names as they do in traditional BLAS and LAPACK, with the exception of precision, and are all lowercase (e.g., `blas::gemm`, `lapack::posv`). Arguments also use the same names as they do in BLAS and LAPACK. In general, matrices are uppercase (e.g., `A`, `B`), vectors are lowercase (e.g., `x`, `y`), and scalars are lower-case Greek letters spelled out in English (e.g., `alpha`, `beta`), following common math notation.

Rationale: The lowercase namespace convention was chosen per usage in standard libraries (`std namespace`), Boost (`boost namespace`), and other common use cases such as the Google style guide. For C++ only headers, the file extension `.hh` was chosen to distinguish it from C only `.h` headers. This goes against some HPC libraries such as Kokkos and Trilinos, which capitalize the first letter, but that naming does not fit any of the standards that are followed in our software.

3.5 Real vs. Complex Routines: The Case for Unified Syntax

Some routines in the traditional BLAS have different names for real and complex matrices (e.g., `herk` for complex Hermitian matrices and `syrk` for real symmetric matrices). This prevents

templating algorithms for both real and complex matrices. So, in these cases, both names are extended to apply to both real and complex matrices. For real matrices, herk and syrk are synonyms, with both meaning $C = \alpha AA^H + \beta C = \alpha AA^T + \beta C$, where C is symmetric. For complex matrices, herk means $C = \alpha AA^H + \beta C$, where C is complex Hermitian, while syrk means $C = \alpha AA^T + \beta C$, where C is complex symmetric. Some complex-symmetric routines, such as csymv and csyr, are not in the traditional BLAS standard but are provided by LAPACK. Some complex-symmetric routines are missing from BLAS and LAPACK, such as [cz]syr2, which can be performed using [cz]syr2k, albeit suboptimally. For consistency, we provide all of these routines in C++ BLAS. LAPACK routines prefixed with sy and he are handled similarly.

The dot product has different names for real and complex. We extend dot to mean dotc in complex, and extend both dotc and dotu to mean dot in real.

Additionally, in LAPACK the un prefix denotes a complex unitary matrix, and the **or** prefix denotes a real orthogonal matrix. For these cases, we extend the un-prefixed names to real matrices. The term "orthogonal" is not applicable to complex matrices, so **or**-prefixed routines apply only to real matrices.

Below is chart of the generic C++ names mapped to their respective traditional BLAS names.

C++ Name	Real	Complex
blas::hemv	[sd]symv	[cz]hemv
blas::symv	[sd]symv	[cz]symv †
blas::her	[sd]syr	[cz]her
blas::syr	[sd]syr	[cz]syr †
blas::her2	[sd]syr2	[cz]her2
blas::syr2	[sd]syr2	[cz]syr2 ‡
blas::herk	[sd]syrk	[cz]herk
blas::syrk	[sd]syrk	[cz]syrk
blas::her2k	[sd]syr2k	[cz]her2k
blas::syr2k	[sd]syr2k	[cz]syr2k
blas::hemm	[sd]symm	[cz]hemm
blas::symm	[sd]symm	[cz]symm
blas::dot	[sd]dot	[cz]dotc
blas::dotc	[sd]dot	[cz]dotc
blas::dotu	[sd]dot	[cz]dotu

†[cz]symv and [cz]syr are provided by LAPACK instead of BLAS.
‡[cz]syr2 is not available; can substitute [cz]syr2k with $k = 1$.

Below is chart of the generic C++ names mapped to their respective traditional LAPACK names. Note that this is not an exhaustive list.

C++ Name	Real	Complex
lapack::hesv	[sd]sysv	[cz]hesv
lapack::sysv	[sd]sysv	[cz]sysv
lapack::unmqr	[sd]ormqr	[cz]unmqr
lapack::ormqr	[sd]ormqr	—
lapack::ungqr	[sd]orgqr	[cz]ungqr
lapack::orgqr	[sd]orgqr	—

Where applicable, options that apply conjugate-transpose in complex are interpreted to apply transpose in real. For instance, in LAPACK's `zlarfb`, `trans` takes `NoTrans` and `ConjTrans` but not `Trans`, while in `dlarfb` it takes `NoTrans` and `Trans` but not `ConjTrans`. We extend this to allow `ConjTrans` in the real case to mean `Trans`. This is already true for BLAS routines such as `dgemm`, where `ConjTrans` and `Trans` have the same meaning.

In LAPACK, for non-symmetric eigenvalues, `dgeev` takes a split complex representation with two double-precision vectors for eigenvalues, one vector for real components, and one for imaginary components, while `zgeev` takes single vector of complex values. In C++, `geev` follows the complex routine in taking a single vector of complex values in both the real and complex cases.

Other instances where there are differences between real and complex matrices will be resolved to provide a consistent interface across all data types.

3.6 Use of const Specifier

Array arguments (matrices and vectors) that are read-only are declared **`const`** in the interface. Dimension-related arguments and scalar arguments are passed by value and are therefore not declared **`const`**, as there is no benefit at the call site.

3.7 Enum Constants

As in CBLAS, options like transpose, uplo (upper-lower), etc. are provided by enums. Strongly typed C++ 11 enums are used, where each enum has its own scope and does not implicitly convert to integer. Constants have similar names to those in CBLAS, minus the `Cblas` prefix, but the value is left unspecified and implementation dependent. Enums and constants are title case (e.g., `ColMajor`).

Enums for BLAS are listed below. Note that these values are for example only; also see implementation note below.

```
enum class Layout : char { ColMajor='C', RowMajor='R' };
enum class Op     : char { NoTrans ='N', Trans   ='T', ConjTrans='C' };
enum class Uplo   : char { Upper   ='U', Lower   ='L' };
enum class Diag   : char { NonUnit ='N', Unit    ='U' };
enum class Side   : char { Left    ='L', Right   ='R' };
```

Note that `CBLAS_ORDER` was renamed `CBLAS_LAYOUT` in LAPACK 3.6.0+.

In most cases, the name of the enum is also similar to the name in CBLAS. However, for transpose, because `Transpose::NoTrans` could easily be misread as *transposed* rather than *not transposed*, the enum is named `Op`, which is already frequently used in the documentation (e.g., for `zgemm`).

```
    TRANSA = 'N' or 'n',  op( A ) = A.
    TRANSA = 'T' or 't',  op( A ) = A^T.
    TRANSA = 'C' or 'c',  op( A ) = A^H.
```

In some cases, BLAS and LAPACK take identical options (e.g., `uplo`). For consistency within each library, typedef aliases for the five BLAS enums above are provided.

For some routines, LAPACK supports a wider set of values for an enum category than what is provided by BLAS. For instance, in BLAS, `uplo = Lower` or `Upper`, while in LAPACK, `laset` and `lacpy` take `uplo = Lower`, `Upper`, or `General`; and `lascl` takes eight different matrix types. Instead of having an extended enum, the C++ API consistently uses the standard prefixes (`ge`, `he`, `tr`, etc.) to indicate the matrix type, rather than using the `la` auxiliary prefix and differentiating matrix types based on an argument.

Below we introduce these new names and their mapping to the respective LAPACK names.

C++ API	LAPACK	Matrix Type
`gescl`	`lascl` with type=G	general
`trscl( uplo )`	`lascl` with type=uplo	triangular or Hermitian
`gbscl`	`lascl` with type=Z	general band
`hbscl( uplo )`	`lascl` with type=B (Lower) or Q (Upper)	Hermitian band
`hsscl`	`lascl` with type=H	Hessenberg
`gecpy`	`lacpy` with uplo=G	general
`trcpy( uplo )`	`lacpy` with same uplo	triangular or Hermitian
`geset`	`laset` with uplo=G	general
`trset( uplo )`	`laset` with same uplo	triangular or Hermitian

Implementation note: Three potential implementations are readily apparent. Enumeration values could be:

1. default values (0, 1, ...); this is used by cuBLAS;
2. same value as in CBLAS (e.g., `NoTrans = 111`); or
3. character values used in Fortran (e.g., `NoTrans = 'n'`, as shown above).

If the C++ API calls Fortran BLAS, then the first two options require a switch, `if-then`, or a lookup table to determine the equivalent character constant (e.g., `NoTrans=111` maps to `'n'`). The third option is trivially converted using a cast and is easier to understand if printed out for debugging.

If the C++ API calls CBLAS, obviously option 2 is the easiest.

If the C++ API calls some other BLAS library, such as cuBLAS or clBLAS, a `switch`, `if-then`, or lookup table is probably required in all three cases.

We leave the enumeration values unspecified and implementation dependent.

Rationale: In C++, the old style enumeration type, which was borrowed from C, is of integral type without exact size specified. This may cause problems for binary interfaces when the C compiler uses the default `int` representation, and the C++ compiler uses a different storage size. We do not face this issue here, as we only target C++ as the calling language and C or Fortran as the likely implementation language.

3.8 Workspaces

Many LAPACK routines take workspaces with both minimum and optimal sizes. These are typically of size $O(n \times n_b)$ for a matrix of dimension n and an optimal block size n_b. Notable exceptions are eigenvalue and singular value routines, which often take workspaces of size $O(n^2)$. As memory allocation is typically quick, the C++ LAPACK interface allocates optimal workspace sizes internally, thereby removing workspaces from the interface. Traditional BLAS routines do not take workspaces.

If this becomes a performance bottleneck, workspaces could be added as optional arguments—with a default value of `nullptr` indicating that the wrapper should allocate workspace—without breaking code written with the C++ LAPACK API.

Rationale: As needed, there is a possibility of adding an overloaded function call that takes a user-defined memory allocator as an argument. This may serve memory-constrained implementations that insist on controlled memory usage.

3.9 Errors

Traditional BLAS routines call `xerbla` when an error occurs. All errors that BLAS detects are bugs. LAPACK likewise calls `xerbla` for invalid parameters (which are bugs), but not for runtime numerical errors like a singular matrix in `getrf` or an indefinite matrix in `potrf`. The default implementation of `xerbla` aborts execution.[1]

Instead, we adopt C++ exceptions for errors, such as invalid arguments. Two new exceptions are also introduced: (1) `blas::error` and (2) `lapack::error`, which are subclasses of `std::exception`. The `what()` member function yields a description of the error.

For runtime numerical errors, the traditional info value is returned. A zero indicates success. Note that these are often not fatal errors. For example, an application may want to know whether a matrix is positive definite, and the easiest, fastest test is to attempt a Cholesky factorization, which will return an error when it is not positive definite.

We do not implement NaN or Inf checks. These add $O(n^2)$ work and memory traffic with little added benefit. Ideally, a robust BLAS library would ensure that NaN and Inf values are propagated, meaning that if there is a NaN or Inf in the input, there is one in the output. Though, this might not be the case for optimizations where `alpha=0` or `beta=0`. In `gemm`, for instance, if `beta=0`, then it is specifically documented in the reference BLAS that C need not be initialized. The current reference BLAS implementation does not always propagate NaN and Inf; see the Next Generation BLAS (Section 2.2.5) for examples and proposed routines that are guaranteed to propagate NaN and Inf values.

Rationale: Occasionally, users express concern about the overhead of error checks. For even modestly sized matrices, error checks take negligible time. However, for very small matrices, with $n < 20$ or so, there can be noticeable overhead. Intel introduced `MKL_DIRECT_CALL` to

[1] See explanation in *C++ API for Batch BLAS, SLATE working note 4* as to why and how `xerbla` is a hideous monstrosity for parallel codes or multiple libraries. http://www.icl.utk.edu/publications/swan-004

disable error checks in these cases.[2] However, libraries compiled for specific sizes, either via templating or just-in-time (JIT) compilation, provide an even larger performance boost for these small sizes; for instance, Intel's libxsmm[3] for extra-small matrix-multiply or batched BLAS for sets of small matrices. Thus, users with such small matrices are encouraged to use special purpose interfaces rather than try to optimize overheads in a general purpose interface.

3.10 Return Values

Most C++ BLAS routines are void. The exceptions are `asum`, `nrm2`, `dot*`, and `iamax`, which return their result—as is also done in the traditional Fortran interface. The `dot` routine returns a complex value in the complex case (unlike CBLAS, where the complex result is an output argument). This makes the interface consistent across real and complex data types.

Most C++ LAPACK routines return an integer status code that corresponds to positive info values in LAPACK, thereby indicating numerical errors such as a singular matrix in `getrf`. A zero indicates success. LAPACK norm functions return their result.

3.11 Complex Numbers

C++ `std::complex` is used. Unlike CBLAS, complex scalars are passed by value, which is the same for real scalars. This avoids inconsistencies that would prevent templated code from calling BLAS. For type safety, arguments are specified as `std::complex` rather than as `void*`, which is what CBLAS uses.

3.12 Object Dimensions as 64-bit Integers

The interface will require 64-bit integers to specify object sizes using the `cstdint` header and the `int64_t` integral data type.

In recent years, 32-bit software has been in decline with both vendors and open-source projects dropping support for 32-bit versions and opting exclusively for 64-bit implementations. In fact, 32-bit software is more of a legacy issue with the increasing memory sizes and the demand of larger models that require large matrices and vectors.

BLAS and LAPACK libraries can easily address this issue, because sizing dense matrices and vectors has negligible cost. Even on a 32-bit system, an overhead of using 64-bit integers is not an issue—with the exception of storage for pivots, which arises in LU, pivoted QR, and accompanying routines that operate on these pivots (e.g., `laswp`). The overhead for those could be $\mathcal{O}(n)$, where n is the number of swapped rows.

[2] https://software.intel.com/en-us/articles/improve-intel-mkl-performance-for-small-problems-the-use-of-mkl-direct-call
[3] https://github.com/hfp/libxsmm

3.13 Matrix Layout

Traditional Fortran BLAS assumes column-major matrices, and CBLAS added support for row-major matrices. In many cases, this can be accomplished with essentially no overhead by swapping matrices, dimensions, upper-lower, and transposes, and then calling the column-major routine. For instance, `cblas_dgemv` simply changes `trans=NoTrans` to `Trans` or `trans=Trans` to `NoTrans`, swaps `m <=> n`, and then calls (column-major) `dgemv`. However, some routines require a little extra effort for complex matrices. For `cblas_zgemv`, `trans=ConjTrans` can be changed to `NoTrans`, but then the matrix isn't conjugated. This can be resolved by conjugating y and a copy of x, calling `zgemv` with `m <=> n` swapped and `trans=NoTrans`, and then conjugating y again. Several other level-2 BLAS routines have similar solutions. So, with minimal overhead, row-major matrices can be supported in BLAS.

We propose the same mechanism for the C++ BLAS API, either by calling CBLAS and relying on the row-major support in CBLAS or by reimplementing similar solutions in C++ and calling the Fortran BLAS.

We also build the same option into the C++ LAPACK API, for future support. However, this would not be implemented initially, which would cause an exception to be thrown. This is because, for some routines like `getrf`, there can be substantial overhead in calling the traditional Fortran LAPACK implementation, because a transpose is required. Other routines such as matrix norms, QR, LQ, SVD, and operations on symmetric matrices can be readily translated to LAPACK calls with essentially no overhead and without physically transposing the matrix in memory.

Row-major layout is specified the same way it is in CBLAS, using the `blas::Layout` or `lapack::Layout` enum as the first parameter of C++ BLAS and LAPACK functions. It could maybe be moved to the end to make it an optional argument with a default value of `ColMajor`.

3.14 Templated versions

As a future extension, in addition to overloaded wrappers around traditional BLAS routines, generic templated versions that work for any data type could be provided. For instance, these would support half, double-double, or quad precision and integer types. The data types need only basic arithmetic operations (e.g., $+ - */$) and functions (e.g., `conj`, `sqrt`, `abs`, `real`, `imag`) to be defined. Initially, such templated versions could be based on the reference BLAS, but these can be optimized using well-known techniques like blocking and vectorization.

3.15 Prototype implementation

To make our proposal concrete, we include a prototype implementation of wrappers for `blas::gemm` matrix-matrix multiply and `lapack::potrf` Cholesky factorization. For brevity, only the `complex<double>` datatype is shown; code for other precisions is similar. The only compile-time parameters are the Fortran name-mangling convention (here assumed to be lowercase with an appended underscore, "_") and `BLAS_ILP64`, which indicates it will be linked with an ILP64 (64-bit integer) BLAS/LAPACK library version.

blas.hh

```
#ifndef BLAS_HH
#define BLAS_HH

#include <cstdint>
#include <exception>
#include <complex>
#include <string>

namespace blas {

// ------------------------------------------------------------------------------
// Fortran name mangling depends on compiler, generally one of:
//     UPPER
//     lower
//     lower ## _
#ifndef BLAS_FORTRAN_NAME
#define BLAS_FORTRAN_NAME( lower, UPPER ) lower ## _
#endif

// ------------------------------------------------------------------------------
// blas_int is the integer type of the underlying Fortran BLAS library.
// BLAS wrappers take int64_t and check for overflow before casting to blas_int.
#ifdef BLAS_ILP64
typedef long long blas_int;
#else
typedef int blas_int;
#endif

// ------------------------------------------------------------------------------
enum class Layout : char { ColMajor='C', RowMajor='R' };
enum class Op     : char { NoTrans ='N', Trans   ='T', ConjTrans='C' };
enum class Uplo   : char { Upper   ='U', Lower   ='L' };
enum class Diag   : char { NonUnit ='N', Unit    ='U' };
enum class Side   : char { Left    ='L', Right   ='R' };

// ------------------------------------------------------------------------------
class Error: public std::exception
{
public:
    Error(): std::exception() {}
    Error( const char* msg ): std::exception(), msg_( msg ) {}
    virtual const char* what() { return msg_.c_str(); }
private:
    std::string msg_;
};

// ------------------------------------------------------------------------------
// internal helper function; throws Error if cond is true
// called by blas_throw_if macro
inline void throw_if( bool cond, const char* condstr )
{
    if (cond) {
        throw Error( condstr );
    }
}

// internal macro to get string #cond; throws Error if cond is true
#define blas_throw_if( cond ) \
        throw_if( cond, #cond )

// ------------------------------------------------------------------------------
// Fortran prototypes
// sgemm, dgemm, cgemm omitted for brevity
#define BLAS_zgemm BLAS_FORTRAN_NAME( zgemm, ZGEMM )
```

```
65
66  extern "C"
67  void BLAS_zgemm( char const* transA, char const* transB,
68                   blas_int const* m, blas_int const* n, blas_int const* k,
69                   std::complex<double> const* alpha,
70                   std::complex<double> const* A, blas_int const* lda,
71                   std::complex<double> const* B, blas_int const* ldb,
72                   std::complex<double> const* beta,
73                   std::complex<double>*       C, blas_int const* ldc );
74
75  // -----------------------------------------------------------------------------
76  // lightweight overloaded wrappers: converts C to Fortran calling convention.
77  // calls to sgemm, dgemm, cgemm omitted for brevity
78  inline void gemm_( char transA, char transB,
79                     blas_int m, blas_int n, blas_int k,
80                     std::complex<double> alpha,
81                     std::complex<double> const* A, blas_int lda,
82                     std::complex<double> const* B, blas_int ldb,
83                     std::complex<double> beta,
84                     std::complex<double>*       C, blas_int ldc )
85  {
86      BLAS_zgemm( &transA, &transB, &m, &n, &k,
87                  &alpha, A, &lda, B, &ldb, &beta, C, &ldc );
88  }
89
90  // -----------------------------------------------------------------------------
91  // templated wrapper checks arguments, handles row-major to col-major translation
92  template< typename T >
93  void gemm( Layout layout, Op transA, Op transB,
94             int64_t m, int64_t n, int64_t k,
95             T alpha,
96             T const* A, int64_t lda,
97             T const* B, int64_t ldb,
98             T beta,
99             T*       C, int64_t ldc )
100 {
101     // determine minimum size of leading dimensions
102     int64_t Am, Bm, Cm;
103     if (layout == Layout::ColMajor) {
104         Am = (transA == Op::NoTrans ? m : k);
105         Bm = (transB == Op::NoTrans ? k : n);
106         Cm = m;
107     }
108     else {
109         // RowMajor
110         Am = (transA == Op::NoTrans ? k : m);
111         Bm = (transB == Op::NoTrans ? n : k);
112         Cm = n;
113     }
114
115     // check arguments
116     blas_throw_if( layout != Layout::RowMajor && layout != Layout::ColMajor );
117     blas_throw_if( transA != Op::NoTrans && transA != Op::Trans && transA != Op::ConjTrans );
118     blas_throw_if( transB != Op::NoTrans && transB != Op::Trans && transB != Op::ConjTrans );
119     blas_throw_if( m < 0 );
120     blas_throw_if( n < 0 );
121     blas_throw_if( k < 0 );
122     blas_throw_if( lda < Am );
123     blas_throw_if( ldb < Bm );
124     blas_throw_if( ldc < Cm );
125
126     // check for overflow in native BLAS integer type, if smaller than int64_t
127     if (sizeof(int64_t) > sizeof(blas_int)) {
128         blas_throw_if( m   > std::numeric_limits<blas_int>::max() );
129         blas_throw_if( n   > std::numeric_limits<blas_int>::max() );
130         blas_throw_if( k   > std::numeric_limits<blas_int>::max() );
```

```
131         blas_throw_if( lda > std::numeric_limits<blas_int>::max() );
132         blas_throw_if( ldb > std::numeric_limits<blas_int>::max() );
133         blas_throw_if( ldc > std::numeric_limits<blas_int>::max() );
134     }
135
136     if (layout == Layout::ColMajor) {
137         gemm_( (char) transA, (char) transB,
138                (blas_int) m, (blas_int) n, (blas_int) k,
139                alpha,
140                A, (blas_int) lda,
141                B, (blas_int) ldb,
142                beta,
143                C, (blas_int) ldc );
144     }
145     else {
146         // RowMajor: swap (transA, transB), (m, n), and (A, B)
147         gemm_( (char) transB, (char) transA,
148                (blas_int) n, (blas_int) m, (blas_int) k,
149                alpha,
150                B, (blas_int) ldb,
151                A, (blas_int) lda,
152                beta,
153                C, (blas_int) ldc );
154     }
155 }
156
157 }  // namespace blas
158
159 #endif        //  #ifndef BLAS_HH
```

lapack.hh

```
1  #ifndef LAPACK_HH
2  #define LAPACK_HH
3
4  #include <cstdint>
5  #include <exception>
6  #include <complex>
7
8  #include "blas.hh"
9
10 namespace lapack {
11
12 // assume same int_type as BLAS
13 typedef blas::int_type int_type;
14
15 // alias types from BLAS
16 typedef blas::Layout Layout;
17 typedef blas::Op     Op;
18 typedef blas::Uplo   Uplo;
19 typedef blas::Diag   Diag;
20 typedef blas::Side   Side;
21
22 // omitted for brevity: lapack::Error, blas_throw_if similar to blas.hh
23
24 // ------------------------------------------------------------------------------
25 // Fortran prototypes
26 // spotrf, dpotrf, cpotrf omitted for brevity
27 #define LAPACK_zpotrf BLAS_FORTRAN_NAME( zpotrf, ZPOTRF )
28
29 extern "C"
30 void LAPACK_zpotrf( char const* uplo, int_type const* n,
31                     std::complex<double>* A, int_type const* lda,
32                     int_type* info );
33
```

```
34 // ------------------------------------------------------------------------------
35 // lightweight overloaded wrappers: converts C to Fortran calling convention.
36 // calls to spotrf, dpotrf, cpotrf omitted for brevity
37 inline void potrf_( char uplo, int_type n,
38                     std::complex<double>* A, int_type lda,
39                     int_type* info )
40 {
41     LAPACK_zpotrf( &uplo, &n, A, &lda, info );
42 }
43
44 // ------------------------------------------------------------------------------
45 // templated wrapper checks arguments, handles row-major to col-major translation
46 template< typename T >
47 int64_t potrf( Layout layout, Uplo uplo, int64_t n, T* A, int64_t lda )
48 {
49     // check arguments
50     blas_throw_if( layout != Layout::RowMajor && layout != Layout::ColMajor );
51     blas_throw_if( uplo != Uplo::Upper && uplo != Uplo::Lower );
52     blas_throw_if( n < 0 );
53     blas_throw_if( lda < n );
54
55     // check for overflow in native BLAS integer type, if smaller than int64_t
56     if (sizeof(int64_t) > sizeof(int_type)) {
57         blas_throw_if( n   > std::numeric_limits<int_type>::max() );
58         blas_throw_if( lda > std::numeric_limits<int_type>::max() );
59     }
60
61     int_type info = 0;
62     if (layout == Layout::ColMajor) {
63         potrf_( (char) uplo, n, A, lda, &info );
64     }
65     else {
66         // RowMajor: change upper <=> lower; no need to conjugate
67         Uplo uplo_swap = (uplo == Uplo::Lower ? Uplo::Upper : Uplo::Lower);
68         potrf_( (char) uplo_swap, (int_type) n, A, (int_type) lda, &info );
69     }
70     return info;
71 }
72
73 }  // namespace lapack
74
75 #endif        //  #ifndef LAPACK_HH
```

3.16 Support for Graphics Processing Units

So far, the proposed C++ API does not address any specific architecture. The prototype implementation shown in Section 3.15 works well for most CPU architectures. A question now arises: can we use the same API to provide a prototype implementation that supports accelerators (e.g., graphics processing units [GPUs] and similar devices)? It turns out that, while possible, using the same exact API hides a lot of GPU-specific features and takes away some useful controls that should be exposed to the user. Using the same API also creates confusion about targeting a certain hardware for execution. Such shortcomings are summarized below.

1. While it is possible to determine the memory space of a data pointer (i.e., whether it is in the CPU memory space or in the GPU memory space), it is confusing to the use the exact same API for both CPUs and GPUs. The semantics of calling the API become hidden within the pointer attributes, which leads to software readability issues, since a reader cannot determine whether a BLAS/LAPACK call is being executed on the CPU or on the

GPU.

2. There are issues with the behavior specifications of the routine based on the location of the data pointers. For example, we can assume that the routine automatically offloads the computation to the accelerator if only GPU pointers are passed to the routine. However, the behavior is undefined if the user passes a mix of CPU and GPU pointers.

3. Most of the GPU vendor libraries use handles to maintain some sort of context on the device. It is inconvenient to create and destroy the handle each time a BLAS/LAPACK routine is called. This might also lead to performance issues.

4. GPU accelerators introduce the notion of "queues" or "streams." A GPU kernel is always submitted to a queue; these queues can be the default (synchronous) queues used in NVIDIA GPUs, or they can be user defined. Queues are also used to launch concurrent workloads on the GPU by submitting these workloads into independent queues. The API shown in Section 3.15 does not expose such a control to the user, since a queue must be created and destroyed internally. This means that the user cannot express dependencies correctly among several GPU kernels. The use of the default queue, if it exists, is not a solution, because it is synchronous with respect to other queues.

5. In a multi-GPU environment, one must specify the GPU that will execute a certain kernel. This is enabled by some vendor runtime APIs that allow the user to *set current active device*. Such a control cannot be exposed through the proposed API.

Overloaded APIs for GPUs

Based on the aforementioned reasons, we decided to provide dedicated interfaces for GPUs. These interfaces have the same exact names specified in Section 3.5 but use a longer list of arguments. We take advantage of the C++ overloading capabilities and propose that the GPU interfaces for BLAS and LAPACK should contain an extra parameter that takes care of many GPU-specific details. The extra argument is a C++ class called `Queue`, which lives in the `blas` namespace. The code below shows the GPU interface for the GEMM routine.

```
template < typename T >
void gemm ( Layout layout , Op transA , Op transB ,
            int64_t m, int64_t n, int64_t k,
            T alpha ,
            T const * A, int64_t lda ,
            T const * B, int64_t ldb ,
            T beta ,
            T* C, int64_t ldc,
            Queue &queue );
```

Only GPU pointers are assumed for the device interfaces, and in case the user decides to compile the source with GPU support, the declaration of the device interfaces will be implicitly included in the `blas.hh` header.

The `Queue` Class

The `Queue` class encapsulates several functionalities that facilitate execution on the GPU. These include handling GPU-specific errors, encapsulating runtime calls, and initializing the vendor-supplied BLAS library. For now, the plan is to support cuBLAS for NVIDIA GPUs, and rocBLAS for AMD GPUs. Below is a simple implementation of the `Queue` class with the cuBLAS backend. The code shows some very basic functionalities. The final product will eventually expose significantly more controls to the end user.

```
namespace blas {

class Queue
{
public:
    // default constructor
    Queue(){
        blas::get_device( &device_ );
        device_error_check( cudaStreamCreate(&stream_) );
        device_blas_check( cublasCreate(&handle_) );
        device_blas_check( cublasSetStream( handle_, stream_ ) );
    }

    // constructor for a given device id
    Queue(blas::Device device){
        device_ = device;
        blas::set_device( device_ );
        device_error_check( cudaStreamCreate(&stream_) );
        device_blas_check( cublasCreate(&handle_) );
        device_blas_check( cublasSetStream( handle_, stream_ ) );
    }

    // member function to retrieve queue data members
    blas::Device   device()          { return device_;   }
    device_blas_handle_t   handle()   { return handle_;   }

    // member function to synchronize
    void sync(){
        device_error_check( cudaStreamSynchronize(this->stream()) );
    }

    // destructor
    ~Queue(){
        device_blas_check( cublasDestroy(handle_) );
        device_error_check( cudaStreamDestroy(stream_) );
    }


private:
    blas::Device         device_; // associated device ID
    cudaStream_t         stream_; // associated CUDA stream
    device_blas_handle_t handle_; // associated device blas handle

};

}        //  namespace blas
```

Some data types, such as `device_blas_handle_t` and others, encapsulate the vendor-specific data types. Similarly, `device_error_check()` and `device_blas_check()` are used to encapsulate device-specific errors and throw exceptions to the user if any are encountered.

Encapsulating Vendor Libraries

Most BLAS and LAPACK libraries that target CPU architectures agree, with minimal differences, on the routine naming conventions, data types, and constants. However, the corresponding libraries for GPUs have drastically different naming conventions and use vendor-defined data types and constants. This is why we build a simple abstraction layer that directly invokes device BLAS and LAPACK routines and wraps the vendor data types and constants. Such a layer prevents code changes to the high-level routine—the API of which is exposed to the user. It also facilitates adding interfaces to other libraries in the future as needed. As an example, the code below encapsulates the device `dgemm` routine.

```
void DEVICE_BLAS_dgemm(
    device_blas_handle_t handle,
    device_trans_t transA, device_trans_t transB,
    device_blas_int m, device_blas_int n, device_blas_int k,
    double alpha,
    double const *dA, device_blas_int ldda,
    double const *dB, device_blas_int lddb,
    double beta,
    double       *dC, device_blas_int lddc)
{
    #ifdef HAVE_CUBLAS
    cublasDgemm( handle, transA, transB,
                 m, n, k,
                 &alpha, dA, ldda, dB, lddb,
                 &beta,  dC, lddc );
    #elif defined(HAVE_ROCBLAS)
    /* equivalent rocBLAS call goes here */
    #endif
}
```

The types `device_blas_handle_t`, `device_trans_t`, and others are compiled to cuBLAS types if the flag `HAVE_CUBLAS` is enabled and are compiled to rocBLAS types if the flag `HAVE_ROCBLAS` is true. By building this simple layer, the device version of the `blas::gemm` routine always calls the `DEVICE_BLAS_xgemm`, regardless of the backend used for the vendor BLAS library.

Prototype Implementation

The code below shows a complete prototype for a device `dgemm` routine. The routine shares the same error checks shown in the CPU version. It also converts the input arguments from the types defined in the `blas` namespace to those defined in the vendor library. The routine automatically selects the GPU on which the routine is to be executed, so that the user does not have to explicitly manage the active device in the application code.

```
void gemm(
    blas::Layout layout,
    blas::Op transA,
    blas::Op transB,
    int64_t m, int64_t n, int64_t k,
    double alpha,
    double const *dA, int64_t ldda,
    double const *dB, int64_t lddb,
    double beta,
    double       *dC, int64_t lddc,
    blas::Queue &queue )
{
    // check arguments
```

```
    blas_error_if( layout != Layout::ColMajor &&
                   layout != Layout::RowMajor );
    blas_error_if( transA != Op::NoTrans &&
                   transA != Op::Trans &&
                   transA != Op::ConjTrans );



    blas_error_if( transB != Op::NoTrans &&
                   transB != Op::Trans &&
                   transB != Op::ConjTrans );

    blas_error_if( m < 0 );
    blas_error_if( n < 0 );
    blas_error_if( k < 0 );

    if (layout == Layout::ColMajor) {
        if (transA == Op::NoTrans)
            blas_error_if( ldda < m );
        else
            blas_error_if( ldda < k );

        if (transB == Op::NoTrans)
            blas_error_if( lddb < k );
        else
            blas_error_if( lddb < n );

        blas_error_if( lddc < m );
    }
    else {
        if (transA != Op::NoTrans)
            blas_error_if( ldda < m );
        else
            blas_error_if( ldda < k );

        if (transB != Op::NoTrans)
            blas_error_if( lddb < k );
        else
            blas_error_if( lddb < n );

        blas_error_if( lddc < n );
    }

    // check for overflow in native BLAS integer type, if smaller than int64_t
    if (sizeof(int64_t) > sizeof(device_blas_int)) {
        blas_error_if( m    > std::numeric_limits<device_blas_int>::max() );
        blas_error_if( n    > std::numeric_limits<device_blas_int>::max() );
        blas_error_if( k    > std::numeric_limits<device_blas_int>::max() );
        blas_error_if( ldda > std::numeric_limits<device_blas_int>::max() );
        blas_error_if( lddb > std::numeric_limits<device_blas_int>::max() );
        blas_error_if( lddc > std::numeric_limits<device_blas_int>::max() );
    }

    device_trans_t  transA_ = blas::device_trans_const( transA );
    device_trans_t  transB_ = blas::device_trans_const( transB );
    device_blas_int m_      = (device_blas_int) m;
    device_blas_int n_      = (device_blas_int) n;
    device_blas_int k_      = (device_blas_int) k;
    device_blas_int ldda_   = (device_blas_int) ldda;
    device_blas_int lddb_   = (device_blas_int) lddb;
    device_blas_int lddc_   = (device_blas_int) lddc;

    blas::set_device( queue.device() );
    if (layout == Layout::RowMajor) {
        // swap transA <=> transB, m <=> n, B <=> A
        DEVICE_BLAS_dgemm(
```

```
80                  queue.handle(), transB_, transA_,
81                  n_, m_, k_,
82                  alpha, dB, lddb_, dA, ldda_,
83                  beta,  dC, lddc_);
84      }
85      else {
86
87          DEVICE_BLAS_dgemm(
88                  queue.handle(), transA_, transB_,
89                  m_, n_, k_,
90                  alpha, dA, ldda_, dB, lddb_,
91                  beta,  dC, lddc_);
92      }
93  }
```