P1674R1: Evolving a Standard C++ Linear Algebra Library from the BLAS

Authors

- Mark Hoemmen (mark.hoemmen@gmail.com) (NVIDIA)
- D. S. Hollman (me@dsh.fyi) (Google)
- Christian Trott (crtrott@sandia.gov) (Sandia National Laboratories)

Date: 2022-04-15

Revision history

- Revision 0 submitted 2019-06-17
- Revision 1 to be submitted 2022-04-15
  - Clean up Markdown formatting
  - Update links and references
  - Account for P2128 (multidimensional `operator[]`), which WG21 adopted on 2021-10
  - Account for changes to P1673 and other proposals in flight

Introduction

Many different applications depend on linear algebra. This includes machine learning, data mining, web search, statistics, computer graphics, medical imaging, geolocation and mapping, and physics-based simulations. Good implementations of seemingly trivial linear algebra algorithms can perform asymptotically better than bad ones, and can get much more accurate results. This is why authors (not just us!) have proposed adding linear algebra to the C++ Standard Library.

Linear algebra builds on over 40 years of well-accepted libraries, on which science and engineering applications depend. In fact, there is an actual multiple-language standard for a set of fundamental operations, called the Basic Linear Algebra Subprograms (BLAS) Standard. The BLAS Standard defines a small set of hooks that vendors or experts in computer architecture can optimize. Application developers or linear algebra algorithm experts can then build on these hooks to get good performance for a variety of algorithms. The only thing missing from the BLAS is a modern C++ interface. Nevertheless, its wide availability and many highly optimized implementations make it a good starting point for developing a standard C++ library.

In this paper, we will walk through the design iterations of a typical C++ linear algebra library that builds on the BLAS. Our point is to show how developers might "grow" a C++ linear algebra library incrementally, solving problems as encountered and gradually increasing the level of abstraction. This process will highlight challenges that C++ developers using the BLAS currently face. It will also show features C++ developers need that the BLAS and its derivatives, like the LAPACK Fortran library, do not provide. Finally, it will give our views about how simple a C++ "BLAS wrapper" could be and still be useful.

This paper implicitly argues for inclusion of linear algebra in the C++ Standard Library. It is meant to be read as part of the design justification for our C++ Standard Library proposal P1673, "A free function linear algebra interface based on the BLAS."

We base this work on our years of experience writing and using linear algebra libraries, and working with people who have done so for much longer than we have. Readers may wish to refer to P1417R0 for a history of linear algebra library standardization. P1417R0 cites first-hand histories of the development of the BLAS, related libraries like LINPACK and LAPACK that use the BLAS, and Matlab (which started as a teaching tool that wrapped these libraries).

Wrapping the BLAS

Suppose that a developer wants to write an application in portable C++, that needs to compute dense matrix-matrix products and some other dense linear algebra operations efficiently. They discover that implementing matrix multiply with a naïve triply nested loop is slow for their use cases, and want to try a library.

The C++ Standard Library currently lacks linear algebra operations. However, our hypothetical C++ developer knows that the Basic Linear Algebra Subroutines (BLAS) exists. The BLAS is a standard -- not an ISO standard like C++, but nevertheless agreed upon by discussion and votes, by people from many institutions. The BLAS has been used for decades by many scientific and engineering libraries and applications. Many vendors offer optimized implementations for their hardware. Third parties have written their own optimized implementations from a combination of well-understood first principles (Goto and van de Geijn 2008) and automatic performance tuning (Bilmes et al. 1996, Whaley et al. 2001). All this makes the BLAS attractive for our developer.

The BLAS standard has both C and Fortran bindings, but the reference implementation comes only in Fortran. It's also slow; for example, its matrix-matrix multiply routine uses nearly the same triply nested loops that a naïve developer would write. The intent of the BLAS is that users who care about performance find optimized implementations, either by hardware vendors or by projects like ATLAS (Whaley et al. 2001), the GotoBLAS, or OpenBLAS.

Suppose that our developer has found an optimized implementation of the BLAS, and they want to call some of its routines from C++. Here are the first two steps they might take.

1. Access the BLAS library.

2. Write a generic BLAS wrapper in C++.

Access the BLAS library

"Accessing" the BLAS library means both linking to it, and figuring out how to call it from C++. For the latter, our developer may start either with the BLAS' C binding, or with its Fortran binding.

Linking to the library

Our developer has two options for the BLAS: its C binding, or its Fortran binding. In both cases, they must add the library or libraries to their link line. One implementation (Intel's Math Kernel Library) offers so many options for this case that they offer a web site (formerly here, now here) to generate the necessary link line. The multiplicity of options comes in part from the vendor's interest in supporting different architectures (32- or 64-bit), operating systems, compilers, and link options. However, the BLAS implementation itself has

options, like whether it uses OpenMP or some other run-time environment for parallelism inside, or whether it uses an LP64 (32-bit signed integer matrix and vector dimensions) or ILP64 (64-bit signed integer matrix and vector dimensions) interface. In the best case, getting these wrong could cause link-time errors. Use of the wrong interface through `extern "C"` declarations could cause run-time errors.

Some developers only need to support one platform, so they rarely need to figure out how to link to the BLAS. Other developers need to support many different platforms, and get good performance on all of them. Such developers end up writing build system logic for helping installers automatically detect and verify BLAS libraries. The author has some experience writing and maintaining such build system logic as part of the Trilinos project. Other projects' build system logic takes at least this much effort. For example, CMake 3.23.1's `FindBLAS` module has 1343 lines of CMake code, not counting imported modules. Other build systems, like GNU Autoconf, have BLAS detection with comparable complexity. (This example just includes finding the BLAS library, not actually deducing the ABI.)

Typical logic for deducing name mangling either tries to link with all known name manglings until the link succeeds, or inspects symbol names in the BLAS library. Since the ABI for function arguments and return value is not part of the mangled symbol name, some build logic must actually try to run code that calls certain BLAS functions (e.g., complex dot product), and make sure that the code does not crash or return wrong results. This hinders cross compilation. In general, it's impossible to make these tests generic. Developers normally just have special cases for known platforms and BLAS ABI bugs.

C binding

The C BLAS is not as widely available as Fortran BLAS, but many vendors have optimized implementations of both, so requiring the C BLAS is not unreasonable. If they choose the C binding, its header file will import many symbols into the global namespace. Best practice would be to write wrapper functions for the desired C BLAS functions, and hide the C binding's header file include in a source file. This prevents otherwise inevitable collisions with the same `extern "C"` declarations in applications or other libraries.

If our developer later wants LAPACK functionality, such as solving linear systems, they will discover that there is no standard C LAPACK binding. Implementations that provide a C BLAS generally also allow calling the Fortran BLAS from the same application, but developers would still need a C or C++ interface to LAPACK.

One advantage of the C BLAS binding is that it works for both row-major and column-major matrix data layouts. Native multidimensional arrays in C and C++ have row-major layout, but the BLAS' Fortran binding only accepts column-major matrices. If users want code that works for both the Fortran and C BLAS interfaces, they will need to use column-major matrices. The Standard Library currently has no way to represent column-major rank-2 arrays (but see our `mdspan` (P0009) and `mdarray` (P1684) proposals). This leads C++ developers to one of the following solutions:

1. Write C-style code for column-major indexing (e.g., `A[i + stride*j]`);
2. use nonstandard matrix or array classes that use column-major storage; or
3. trick the BLAS into working with row-major matrices, by specifying that every operation use the transpose of the input matrix. (This only works for real matrix element types, since BLAS functions have no "take the conjugate but not the transpose" option.)

C++ lacks multidimensional arrays with run-time dimensions, so even if developers want to use row-major indexing, they would still need to write C-style code or use a matrix or array class. This takes away some of the value of a BLAS that can support row-major matrices.

Fortran binding

The experience of developers of large scientific libraries is that the C interface is less often available than the Fortran interface. For example, the Reference BLAS only comes in a Fortran version. Thus, the most portable approach is to rely on Fortran. If our developer links against a Fortran BLAS, they will face the difficult task of deducing the library's Fortran BLAS ABI. This is just close enough to the C ABI to lead innocent developers astray. For example, Fortran compilers mangle the names of functions in different ways, that depend both on the system and on the compiler. Mangled names may be all-capitals or lower-case, and many have zero to two underscores somewhere at the beginning or end of the mangled name. The BLAS may have been built with a different compiler than the system's Fortran compiler (if one is available), and may thus have a different mangling scheme. Fortran interfaces pass all input and output arguments, including integers, by the C ABI equivalent of pointer, unless otherwise specified. Fortran ABIs differ in whether they return complex numbers on the stack (by value) or by initial output argument, and how they pass strings, both of which come up in the BLAS. We have encountered and worked around other issues, including actual BLAS ABI bugs. For instance, functions that claim to return `float` might actually return `double`, as this Trilinos CMake function attempts to detect.

We have written hundreds of lines of build system code to deduce the BLAS ABI, have encountered all of these situations, and expect to encounter novel ones in the future. Other projects do the same. If our developer wants to use LAPACK as well as the BLAS, they will be stuck with these problems.

In summary:

1. Users may need to deduce the BLAS library's Fortran ABI. There is no platform-independent way to do this.

2. Ignorance of some Fortran ABI issues (e.g., return of complex values) could cause bugs when porting code to different hardware or software.

Once our developer has deduced the Fortran BLAS ABI, they will need to do the following:

1. Write `extern "C"` declarations for all the BLAS functions they want to call, using the deduced ABI to mangle function names and correctly handle return values (if applicable).

2. Hide the `extern "C"` declarations and write wrappers to avoid polluting the global namespace and colliding with other libraries or applications. (It's likely that the kinds of large applications that would want to use this BLAS wrapper would already have their own well-hidden `extern "C"` declarations. The authors have encountered this issue multiple times.)

3. Change wrappers to pass read-only integers or scalars by value or const reference, instead of by pointer, so that users can pass in integer or scalar literals if desired.

Generic C++ wrapper

Suppose that our developer has solved the problems of the above section. They have access to the BLAS in their source code, through a wrapped C API. They will next encounter three problems:

- this C API is not type safe;
- it is not generic; and
- it is tied to availability of an external BLAS library.

Our developer's next step for solving these problems would be to write a generic C++ wrapper, with a fall-back implementation for unsupported matrix or vector element types, or if the BLAS library is not available.

Developers who just want to compute one matrix-matrix multiply for matrices of `double` might be satisfied with the BLAS' C binding. However, if they want to use this interface throughout their code, they will discover the following problems. First, the interface is not type safe. For example, the C BLAS doesn't know about `_Complex` (as in Standard C) or `std::complex` (as in Standard C++). Instead, it takes pointers to complex numbers as `void*`. This violates C++ Core Guidelines I.4: "Make interfaces precisely and strongly typed." Second, the interface is not generic. It only works for four matrix or vector element types: `float`, `double`, `std::complex<float>`, and `std::complex<double>`. What if our developer wants to compute with matrices of lower-precision or higher-precision floating-point types, integers, or custom types? Most BLAS operations only need addition and multiplication, yet the BLAS does not work with matrices of integers. The C and Fortran 77 function names also depend on the matrix or vector element type, which hinders developing generic algorithms. Fortran 95's generic BLAS functions, while convenient for Fortran programmers, would only create more ABI deduction problems for C++ developers. Third, what if the BLAS is not available? Users might like a fall-back implementation, even if it is slow, as a way to remove an external library dependency or to test an external BLAS.

The logical solution is to write a generic C++ interface to BLAS operations. "Generic C++ interface" means that users can call some C++ function templated on the matrix or vector element type `T`, and the implementation will either dispatch to the appropriate BLAS call if `T` is one of the four types that the BLAS supports, or fall back to a default implementation otherwise. The fall-back implementation can even replace the BLAS library.

Libraries like the BLAS and LAPACK were written to be as generic as possible. (See our paper P1417R0 for a survey of their history and the authors' intentions.) Thus, once one has figured out the ABI issues, it's not too much more effort to write a generic C++ wrapper. For example, Trilinos has one, `Teuchos::BLAS`, that has served it for over 15 years with few changes.

Some subtle issues remain. For example, some corresponding interfaces for real and complex numbers differ. Our developer must decide whether they should try to present a single interface for both, or expose the differences. We have not found this to be a major issue in practice.

Introduce C++ data structures for matrices & vectors

Many developers may find the above solution satisfactory. For example, the `Teuchos::BLAS` class in Trilinos is a generic C++ wrapper for the BLAS, as we described above. It has seen use in libraries and applications since the mid 2000's, with rare changes. However, many developers are not familiar with the BLAS, and/or find its interface alien to C++. The BLAS has no encapsulation of matrices or vectors; it uses raw pointers. Its functions take a long list of pointer, integer, and scalar arguments in a mysterious order. Mixing up the arguments can cause memory corruption, without the benefit of debug bounds checking that C++ libraries offer. Furthermore, users may need to work with column-major data layouts, which are not idiomatic to C++ and

may thus cause bugs. This suggests that the next step is to develop C++ data structures for matrices and vectors, and extend the above BLAS wrapper to use them.

BLAS routines take many, unencapsulated arguments

The `extern "C"` version of the BLAS interface violates the following C++ Core Guidelines:

- I.23, "Keep the number of function arguments low," and
- I.24, "Avoid adjacent parameters that can be invoked by the same arguments in either order with different meaning."

Its functions take a large number of function arguments, and put together unrelated parameters of the same type. A big reason for this is that neither the C nor the Fortran binding of the BLAS gives users a way to encapsulate a matrix or vector in a single data structure. The four BLAS routines for matrix-matrix multiply all have the following form:

```
xGEMM(TRANSA, TRANSB, M, N, K, ALPHA, A, LDA, B, LDB, BETA, C, LDC)
```

where the initial x is one of S, D, C, or Z, depending on the matrix element type. The arguments have the following types:

- `TRANSA` and `TRANSB` are character arrays of length at least 1;
- `M`, `N`, `K`, `LDA`, `LDB`, and `LDC` are integers;
- `ALPHA` and `BETA` are scalar values of the same type as the matrix element type, and
- `A`, `B`, and `C` are rank-2 Fortran arrays with run-time dimensions.

These routines can perform several different operations, which their documentation represents as `C = alpha*op(A)*op(B) + beta*C`. Here,

- `C` is the input and output matrix;
- `A` and `B` are the two input matrices;
- `C` is `M` by `N`, `A` is `M` by `K`, and `B` is `K` by `N`; and
- `op(X)` represents `X`, the transpose of `X`, or the conjugate transpose of `X`, depending on the corresponding `TRANSX` argument. `TRANSA` and `TRANSB` need not be the same. For real matrix element types, the conjugate transpose and transpose mean the same thing.

The BLAS has a consistent system for ordering these arguments, that a careful read of the BLAS Quick Reference chart should suggest:

1. Character arrays that modify behavior, if any;
2. Matrix and/or vector dimensions; then

3. Constants and matrices / vectors, in the order in which they appear in the right-hand side of the algebraic expression to evaluate. After each matrix / vector comes its stride argument.

However, most users are not familiar with the BLAS' conventions. We *are*, and nevertheless we have found it easy to mix up these arguments. Users need to read carefully to see which of M, N, and K go with A, B, or C. Should they reverse these dimensions if taking the transpose of A or B? In some cases, the BLAS will check errors for you and report the first argument (by number) that is wrong. (The BLAS' error reporting has its own issues; see "Error checking and handling" below.) In other cases, the BLAS may crash or get the wrong answer. Since the BLAS is a C or Fortran library, whatever debug bounds checking you have on your arrays won't help. It may not have been built with debug symbols, so run-time debuggers may not help. Developers who haven't done a careful job wrapping the BLAS in a type-safe interface will learn the hard way, for example if they mix up the order of arguments in the `extern "C"` declarations and their integers get bitwise reinterpreted as pointers.

Matrix and vector data structures

The BLAS takes matrices and vectors as raw pointers. This violates C++ Core Guidelines I.13: "Do not pass an array as a single pointer." However, C++ does not currently provide a BLAS-compatible matrix data structure, with column-major storage, including dimensions and stride information. Thus, C++ does not currently give us a standard way *not* to break that rule. The language's native two-dimensional arrays cannot have run-time dimensions and also promise contiguous or constant-stride (row-major or column-major) storage. Other array-like data structures in the C++ Standard Library are only one dimensional. The `valarray` class and its slice classes were meant to serve as a building block for vectors and matrices (see footnote in **[template.valarray.overview]**). **[class.slice.overview]** calls `slice` "a BLAS-like slice from an array." However, none of these classes can represent a matrix without additional information. Also, they require using `valarray` for data storage, rather than whatever container the application might like to use.

The lack of general multidimensional array data structures in C++ led us to propose `mdspan` (P0009). An `mdspan` with `layout_left` layout can represent a view of a dense column-major matrix with leading dimension (stride) equal to the number of rows. The BLAS permits matrices with leading dimension greater than the number of rows, for which case one can use either `mdspan`'s `layout_stride` layout, or the `layout_blas_general` proposed in P1673. Thus, `mdspan` can encapsulate all the pointer, dimensions, and stride arguments to BLAS functions that represent a view of a matrix. Unlike `valarray` slices, `mdspan` can even represent dimensions and strides as compile-time values. More importantly, `mdspan` can view matrices and vectors stored in the application's preferred container types. Applications need not commit to a particular container type, unlike with `valarray`. Therefore, we think it makes sense to use `mdspan` as the minimal common interface for many different data structures to interact with libraries like the BLAS.

BLAS library reports user errors incompatibly with C++

The BLAS library checks user input, and reports user errors in a way that is incompatible with C++ error handling. Section 1.8 of the BLAS Standard requires error checking and reporting for the Fortran 95, Fortran 77, and C bindings. Checking looks at dimensions, strides, and bandwidths. Users of the Fortran Reference BLAS see this as program exit in the `XERBLA` routine; users of an implementation conformant with the BLAS Standard see this as program exit in `BLAS_error` or `blas_error`. There are a few issues with the BLAS' error reporting approach.

1. There is no way to recover from an error.

2. There is no stack unwinding.

3. Users can replace the default error handler at link time, but it must "print an informative error message describing the error, and halt execution." There is no way to return control to the caller.

4. Fortran did not define an equivalent of "exit with a nonzero error code" until version 2008 of the Standard. Thus, programs that use the BLAS' Fortran binding and invoke the BLAS error handler could exit with no indication to an automated caller that something went wrong.

Developers writing their own C++ wrapper around an existing Fortran or C BLAS implementation have at least two options to address these issues.

1. Detect user errors in the wrapper. Try to catch all cases that the BLAS would catch, and report them in a way friendly to C++ code.

2. Declare that user errors invoke undefined behavior, just like passing the wrong pointers to `memcpy`. Let the BLAS library detect errors if it wants.

The problem with Option 1 is that it adds overhead by duplicating the BLAS library's existing error checks. For our proposal P1673, we have chosen Option 2, by defining user run-time errors as Precondition violations. This would let a Standard Library author quickly stand up an implementation by wrapping an existing BLAS library. Later, they can go back and modify the BLAS library to change how it detects and handles errors.

Function argument aliasing and zero scalar multipliers

Summary:

1. The BLAS Standard forbids aliasing any input (read-only) argument with any output (write-only or read-and-write) argument.

2. The BLAS uses `INTENT(INOUT)` (read-and-write) arguments to express "updates" to a vector or matrix. By contrast, C++ Standard algorithms like `transform` take input and output iterator ranges as different parameters, but may let input and output ranges be the same.

3. The BLAS uses the values of scalar multiplier arguments ("alpha" or "beta") of vectors or matrices at run time, to decide whether to treat the vectors or matrices as write only. This matters both for performance and semantically, assuming IEEE floating-point arithmetic.

4. We recommend separately, based on the category of BLAS function, how to translate `INTENT(INOUT)` arguments into a C++ idiom:

   a. For in-place triangular solve or triangular multiply, we recommend translating the function to take separate input and output arguments that shall not alias each other.

   b. Else, if the BLAS function unconditionally updates (like `xGER`), we recommend retaining read-and-write behavior for that argument.

   c. Else, if the BLAS function uses a scalar `beta` argument to decide whether to read the output argument as well as write to it (like `xGEMM`), we recommend providing two versions: a write-only version (as if `beta` is zero), and a read-and-write version (as if `beta` is nonzero).

Both the BLAS Standard and the C++ Standard Library's algorithms impose constraints on aliasing of their pointer / array arguments. However, the BLAS Standard uses different language to express these constraints, and has different interface assumptions. In this section, we summarize the BLAS' constraints in C++ Standard terms, and explain how an idiomatic C++ interface would differ.

Aliasing in the BLAS

The BLAS Standard says that its functions do not permit any aliasing of their arguments. In order to understand this restriction, we must look both at Fortran (the BLAS' "native language"), and at the `INTENT` specifications of BLAS functions. It turns out that this restriction is no more onerous than that on the input and output arguments of `copy` or `memcpy`. Furthermore, `INTENT(INOUT)` permits the BLAS to express linear algebra "update" operations in an idiom that linear algebra experts find more natural. This differs from the idiom of C++ Standard Library algorithms like `transform`, that can be used to perform update operations.

The Fortran standard rarely refers to "aliasing." The proper Fortran term is *association*. In C++ terms, "association" means something like "refers to." For example, Fortran uses the term *argument association* to describe the binding of the caller's *actual argument*, to the corresponding *dummy argument*. C++ calls the former an "argument" **[defns.argument]** -- what the caller puts in the parentheses when calling the function -- and the latter a "parameter" **[defns.parameter]** -- the thing inside the function that gets the value. (Fortran uses "parameter" in a different way than C++, to refer to a named constant.) *Sequence association* means (among other things) that an array argument can "point to a subset" of another array, just like it can in C++. We omit details about array slices, but C++ developers can get a rough idea of this behavior by thinking of Fortran arrays as pointers.

In the latest (2018) Fortran standard, Section 15.5.2.3 gives argument association rules, and Section 19.5 defines the different kinds of association. Section 15.5.1.2 explains that "[a]rgument association can be sequence association." This blog post by Steve Lionel explains argument association and aliasing in detail.

A first-order C++ approximation of Fortran's default behavior is "pass scalar values by reference -- nonconst unless declared `INTENT(IN)` -- and pass arrays by pointer." Thus, aliasing rules matter even more in Fortran than in C++. Fortran only permits associating the same entity with two different dummy arguments if the dummy arguments are both explicitly marked read only, through the `INTENT(IN)` attribute. Function arguments can have four different `INTENT`s:

- `INTENT(IN)`, read only;
- `INTENT(OUT)`, write only;
- `INTENT(INOUT)`, read and write; or
- unspecified, so the function's behavior defines the actual "intent."

For example, the BLAS' `xAXPY` function performs the vector sum operation `Y = Y + ALPHA*X`. It has the following arguments:

- `ALPHA` (scalar) is `INTENT(IN)`,
- `X` (vector) is `INTENT(IN)`, and
- `Y` (vector) is `INTENT(INOUT)`.

"No aliasing" here means that when users call `xAPXY`, they promise that neither of their first two actual arguments aliases any elements in the third actual argument. Aliasing two `INTENT(IN)` arguments is legal. For

instance, `xGEMM` (matrix-matrix multiply) permits its `INTENT(IN)` arguments `A` and `B` to be the same, as long as they do not alias the `INTENT(INOUT)` argument `C`.

Aliasing in C++ Standard algorithms

The `transform` (see **[alg.transform]**) algorithm is a good analog to updating functions like `xAXPY`. `transform` does not take the C++ equivalent of an `INTENT(INOUT)` argument. Instead, `transform` takes input and output iterator ranges as separate arguments, but lets its output range be equal to either input range. If so, then `transform` implements an "update" that accesses the output in read-and-write fashion. This is the C++ way of expressing a read-and-write argument for update operations.

In general, each C++ Standard algorithm (see e.g., **[alg.modifying.operations]**) states its own constraints on its input and output iterator ranges. For example, in **[alg.copy]**,

- Three-argument `copy` requires that the output iterator `result` "shall not be in the range `[first, last)`" (the input iterators).
- The overload of `copy` that takes an `ExecutionPolicy` requires that the "ranges `[first, last)` and `[result, result + (last - first))` shall not overlap."
- `copy_if` in general requires that the input range `[first, last)` and the output range `[result, result + (last - first))` "shall not overlap."

Note the mismatch between the BLAS and the C++ Standard Library. The BLAS has `INTENT(INOUT)` arguments to express the idea of "an input that is also an output." C++ Standard Library algorithms that have both input and output ranges take separate arguments for those ranges. In some cases, the separate input and output arguments may refer to the same ranges, but they are still separate arguments.

Read-and-write access is idiomatic

Many linear algebra algorithms assume read-and-write access. For example, Krylov subspace methods compute an update to an existing solution or residual vector. Cholesky, LU, and QR factorizations apply a low-rank (outer product) update to a trailing matrix. Most of these algorithms have no risk of parallel race conditions, as long as users follow the rule that `INTENT(INOUT)` arguments may not alias `INTENT(IN)` arguments.

The exceptions in the BLAS are the triangular solves `xTRSM` and `xTRMM`, in which the right-hand side vector(s) are `INTENT(INOUT)` arguments that the algorithm overwrites with the solution vector(s) on output. In practice, the authors often need to keep the original right-hand side vectors, and end up making a copy before the triangular solve. This interface also precludes parallel implementations, since the BLAS is not allowed to allocate memory for temporary copies.

BLAS has special access rules for zero scalar prefactors

BLAS functions have special access rules when their operations multiply a vector or matrix by a scalar prefactor. Whenever the scalar prefactor is zero, the BLAS does not actually read the vector's or matrix's entries. For example, the `xGEMM` function performs the matrix-matrix multiply update `C := alpha * A * B + beta * C`, where `A`, `B`, and `C` are matrices and `alpha` and `beta` are scalars. If `alpha` is zero, the BLAS does not read `A` or `B` and treats that entire term as zero. The `C` argument has declared `INTENT(INOUT)`, but if `beta` is

zero, the BLAS does not read `C` and treats the `C` argument as write only. This is a run-time decision, based on the value(s) of the scalar argument(s).

The point of this rule is so that "multiplying by zero" has the expected result of dropping that term in a sum. This rule matters semantically; it is not just a performance optimization. In IEEE floating-point arithmetic, `0.0 * A[i,j]` is `NaN`, not zero, if `A[i,j]` is `Inf` or `NaN`. If users have not initialized an `INTENT(INOUT)` argument, then it's possible that some of the uninitialized values may be `Inf` or `NaN`. Linear algebra algorithm developers depend on this behavior. For example, textbook formulations of some Krylov subspace methods assume this rule for `xAPXY`, as a way to avoid a special case for the first iteration (where the input vector may not be initialized).

Overloading is more idiomatically C++

The above special access rule is not idiomatic C++ for the following reasons:

1. C++ standard algorithms should be generic, but the rule makes sense only for special cases of a particular arithmetic system.

2. The rule forces a branch with a major behavior change based on run-time input values. This violates both the zero overhead requirement, and the Single Responsibility Principle.

For instance, when we implemented BLAS-like computational kernels in the Trilinos project, Reason 2 required us either to put a branch in the inner loop, or to have an outer branch that calls into one of two separate kernels. Neither has zero overhead, especially if the vectors or matrices are very small. Optimized BLAS implementations likely take the latter approach of implementing two separate kernels, since they do not prioritize performance for very small problems.

A more idiomatic C++ linear algebra library could express write-only vs. read-and-write semantics by overloading. This would remove the dependence of semantics on possibly run-time scalar values, and it would match the convention in **[algorithms.requirements]** that "[b]oth in-place and copying versions are provided for certain algorithms." For example, the library would have two overloads of `xGEMM`:

1. an overload that takes `C` as a strictly write-only argument and performs the operation `C := alpha * A * B`, without regard for the value of `alpha`; and

2. an overload that performs the operation `C := alpha * A * B + beta * D`, permits `D` to be the same as `C` (compare to `transform`), and does so without regard for the values of `alpha` and `beta`.

This would have the side benefit of extending the set of operations "for free." For example, the overloading approach would give users a `xWAXPY` operation `W := alpha*X + Y` without adding a new function name or increasing implementer effort. (In our proposal P1673, we show how `mdspan`'s accessor policy would let us remove scalar arguments like `alpha` and `beta` as well.)

The disadvantage of this approach is that implementations could no longer just call the existing BLAS interface directly. They would need to introduce run-time checks (beyond what the BLAS already does) for `alpha = 0` or `beta = 0` cases. This is one justification for proposing the removal of these special cases if adding linear algebra to the C++ standard library (see our proposal P1673). BLAS implementations (that some vendors write already) or other BLAS-like libraries likely have internal functions for implementing the different cases, in order to avoid branches in inner loops. A standard library written by vendors could access those

internal functions. Furthermore, a C++ linear algebra library for vectors and matrices with very small compile-time sizes would in any case not want to have these run-time branches.

A C++ library could alternately say that any `Inf` or `NaN` values (either in input arrays or as the intermediate result of computations, like `A * B` in the `alpha * A * B` term) give implementation-defined results. However, this would make the library's specification non-generic on the matrix element type, which would defeat our goal of a generic linear algebra library. Thus, we do not favor this approach.

Unconditionally read-and-write arguments

Many BLAS functions have unconditionally read-and-write behavior for their output arguments. This includes

1. Element-wise functions over vectors or matrices, like `xSCAL` (vector scale: `x := alpha * x`) and `xAXPY` (vector update: `y = alpha * x + y`);
2. rank-1 or rank-2 matrix update functions, like `xGER`;
3. in-place triangular matrix-vector multiply functions, like `xTRMV` and `xTRMM`; and in-place triangular solve functions, like `xTRSV` and `xTRSM`.

We consider each of these separately. First, any reasonable implementation of `xSCAL` should behave like `transform`, in that it should work whether or not the output and input vectors are the same (as long as they do not partially overlap). Similarly, an operation `w := alpha*x + beta*y` that permits `w` to be the same as `x` or `y` would behave like `transform`, and would cover all `xAXPY` use cases. The same argument applies to any element-wise function.

Second, rank-1 or rank-2 matrix update functions are idiomatic to the implementation of matrix factorizations, in particular for matrices with a small number of columns (the "panel" case in LAPACK). Users normally want to update the matrix in place. Furthermore, *not* updating makes a performance mistake. An outer product that overwrites a matrix destroys sparsity of the outer product representation. Users are better off keeping the vector(s), instead of forming their outer product explicitly. Updating an already dense matrix with an outer product does not destroy sparsity. The C++ Standard offers `sort` as precedent for only including the in-place version of an algorithm.

Third, the in-place triangular matrix functions cannot be made parallel without overhead (e.g., allocating intermediate storage). This means that, unlike most C++ Standard algorithms, they could not accept `ExecutionPolicy` overloads for parallel execution -- not even for non-threaded vectorization. Furthermore, in practice, users often need to keep the original right-hand side vectors when they do triangular solves, so they end up making a copy of the input / output vector(s) before calling the BLAS. Thus, an idiomatic C++ library would only include versions of these functions that take separate input and output objects, and would forbid aliasing of input and output.

Summary

A C++ version of the BLAS that wants idiomatic C++ treatment of input and output arguments may need to translate each BLAS function with `INTENT(INOUT)` arguments separately.

1. For in-place triangular solve or triangular multiply, the function would take separate input and output arguments that do not alias each other.

2. Else, if the BLAS function unconditionally updates (like `xGER`), the corresponding C++ function would have read-and-write behavior for that argument.

3. Else, if the BLAS function uses a scalar `beta` argument to decide whether to read the output argument as well as write to it (like `xGEMM`), the C++ library would provide two versions: a write-only version (as if `beta` is zero), and a read-and-write version (as if `beta` is nonzero).

Support for the BLAS' different matrix storage formats

Summary:

1. The dense BLAS supports several different dense matrix "types." Type is a mixture of "storage format" (e.g., packed, banded) and "mathematical property" (e.g., symmetric, Hermitian, triangular).

2. Some "types" can be expressed as custom `mdspan` layouts; others do not.

3. Thus, a C++ BLAS wrapper cannot overload on matrix "type" simply by overloading on `mdspan` specialization. The wrapper must use different function names, tags, or some other way to decide what the matrix type is.

BLAS dense matrix storage formats

The dense BLAS supports several different dense matrix "types" (storage formats). We list them here, along with the abbreviation that BLAS function names use for each.

- General (GE): Either column major or row major (C binding only).
- General Band (GB): Stored like General, but functions take two additional integers, one for the upper band width, and one for the lower band width.
- Symmetric (SY): Stored like General, but with the assumption of symmetry (`A[i,j] == A[j,i]`), so that algorithms only need to access half the matrix. Functions take an `UPLO` argument to decide whether to access the matrix's upper or lower triangle.
- Symmetric Band (SB): The combination of General Band and Symmetric.
- Symmetric Packed (SP): Assumes symmetry, but stores entries in a contiguous column-major packed format. The BLAS function takes an `UPLO` argument to decide whether the packed format represents the upper or lower triangle of the matrix.
- Hermitian (HE): Like Symmetric, but assumes the Hermitian property (the complex conjugate of `A[i,j]` equals `A[j,i]`) instead of symmetry. Symmetry and the Hermitian property only differ if the matrix's element type is complex.
- Hermitian Band (HB): The combination of General Band and Hermitian.
- Hermitian Packed (SP): Like Symmetric Packed, but assumes the Hermitian property instead of symmetry.
- Triangular (TR): Functions take an `UPLO` argument to decide whether to access the matrix's upper or lower triangle, and a `DIAG` argument to decide whether to assume that the matrix has an implicitly

stored unit diagonal (all ones). Both options are relevant for using the results of matrix factorizations like LU and Cholesky.

- Triangular Band (TB): The combination of General Band and Triangular.
- Triangular Packed (TP): Stores entries in a contiguous column-major packed format, like Symmetric Packed. BLAS functions take the same `UPLO` and `DIAG` arguments as Triangular.

Matrix storage format is not equivalent to mdspan layout

The BLAS' "matrix types" conflate three different things:

1. the arrangement of matrix elements in memory -- the mapping from a 2-D index space (row and column indices) to a 1-D index space (the underlying storage);
2. constraints on what entries of the matrix an algorithm may access -- e.g., only the upper or lower triangle, or only the entries within a band; and
3. mathematical properties of a matrix, like symmetric, Hermitian, banded, or triangular.

"The arrangement of matrix elements in memory" is exactly what `mdspan`'s layout intends to express. However, the layout cannot properly express "constraints on what entries of the matrix an algorithm may access." P0009 defines the "domain" of an `mdspan` -- its set of valid multidimensional indices -- as the Cartesian product of the dimensions ("extents," in P0009 terms).

This excludes the various Triangular formats, since they do not permit access outside the triangle. One could "hack" the layout, along with a special `mdspan` accessor, to return zero values for read-only access outside the triangle. However, `mdspan` does not have a way to express "read-only access for some matrix elements, and read-and-write access for other matrix elements." The `mdspan` class was meant to be a low-level multidimensional array, not a fully generalizable matrix data structure.

Similarly, there is no way to define a mathematically Hermitian matrix using `mdspan`'s layout and accessor. The layout could reverse row and column indices outside of a specific triangle; that is a way to define a mathematically symmetric matrix. However, there is no way for the layout to tell the accessor that the accessor needs to take the conjugate for accesses outside the triangle. Perhaps the accessor could reverse-engineer this from the 1-D index, but again, this is out of `mdspan`'s scope.

A C++ linear algebra library has a few possibilities for dealing with this.

1. It could use the layout and accessor types in `mdspan` simply as tags to indicate the matrix "type." Algorithms could specialize on those tags.
2. It could introduce a hierarchy of higher-level classes for representing linear algebra objects, use `mdspan` (or something like it) underneath, and write algorithms to those higher-level classes.
3. It could imitate the BLAS, by introducing different function names, if the layouts and accessors do not sufficiently describe the arguments.

In our proposal P1673, we take Approach 3. Our view is that a BLAS-like interface should be as low-level as possible. If a different library wants to implement a "Matlab in C++," it could then build on this low-level

library. We also do not want to pollute `mdspan` -- a simple class meant to be easy for the compiler to optimize -- with extra baggage for representing what amounts to sparse matrices.

BLAS General calls for a new mdspan layout

All BLAS matrix types but the Packed types actually assume the same layout as General. In our proposal P1673, we call General's layout `layout_blas_general`. It includes both row-major and column-major variants: `layout_blas_general<row_major_t>`, and `layout_blas_general<column_major_t>`.

`layout_blas_general` expresses a more general layout than `layout_left` or `layout_right`, because it permits a stride between columns resp. rows that is greater than the corresponding dimension. This is why BLAS functions take an "LDA" (leading dimension of the matrix A) argument separate from the dimensions of A. However, these layouts are slightly *less* general than `layout_stride`, because they assume contiguous storage of columns resp. rows.

One could omit `layout_blas_general` and use `layout_stride` without removing functionality. However, the advantage of these layouts is that subspans taken by many matrix algorithms preserve the layout type (if the stride is a run-time value). Many matrix algorithms work on "submatrices" that are rank-2 subspans of contiguous rows and columns of a "parent" matrix. If the parent matrix is `layout_left`, then in general, the submatrix is `layout_stride`, not `layout_left`. However, if the parent matrix is `layout_blas_general<column_major_t>` or `layout_blas_general<row_major_t>`, such submatrices always have the same layout as their parent matrix. Algorithms on submatrices may thus always assume contiguous access along one dimension.

Taking stock

Thus far, we have outlined the development of a generic C++ "BLAS wrapper" that uses `mdspan` for matrix and vector parameters. The library could call into the BLAS (C or Fortran) for layouts and data types for which that is possible, and would have fall-back implementations for other layouts and data types. We have also gradually adapted BLAS idioms to C++. For example, we talked about imitating the C++ Standard Algorithms with respect to function argument aliasing, and how that relates to the `alpha=0` and `beta=0` special cases for BLAS functions like matrix-matrix multiply.

Recall that we are arguing for inclusion of linear algebra in the C++ Standard Library. If the above interface were in the Standard, vendors could optimize it without much effort, just by calling their existing BLAS implementation, at least for the matrix and vector element types that the BLAS supports. The C++ library would also give vendors the opportunity to optimize for other element types, or even to drop the external BLAS library requirement. For example, it's possible to write a portable implementation of dense matrix-matrix multiply directly to a matrix abstraction like `mdspan`, and still get performance approaching that of a fully optimized vendor-implemented BLAS library. That was already possible given the state of C++ compiler optimization 20 years ago; see e.g., Siek and Lumsdaine 1998. The authors would be ecstatic to have a product like this available in the Standard Library.

Some remaining performance issues

Some C++ developers using a BLAS wrapper would encounter performance issues that relate to limitations in the design of the BLAS' interface itself. Here are three such issues:

1. It is not optimized for tiny matrices and vectors.

2. It has no way for users to control composition of parallelism, such as what parallelism policy to use, or to control nested parallelism.

3. Implementations have no way to optimize across multiple linear algebra operations.

The C++ interface outlined above has the right hooks to resolve these issues. In summary:

1. The interface already permits specializing algorithms for `mdspan` with compile-time dimensions. The `mdarray` container class (P1684) can eliminate any possible overhead from creating a view of a small matrix or vector. It also gives convenient value semantics for small matrices and vectors.

2. Like the C++ Standard Library's algorithms, an optional `ExecutionPolicy&&` argument would be a hook to support parallel execution and hierarchical parallelism, analogous to the existing parallel standard algorithms.

3. Optimizing across multiple linear algebra operations is possible, but adds complications. We talk below about different ways to solve this problem. It's not clear whether general solutions belong in the C++ Standard Library.

Tiny matrices and vectors

"Tiny" could mean any of the following:

- It's cheaper to pass the object by value than by pointer.
- Function call or error checking overhead is significant.
- The objects fit in registers or cache; memory bandwidth no longer limits performance.

The BLAS interface is not optimal for solving tiny problems as fast as possible, for the following reasons:

1. The BLAS is an external library; it cannot be inlined (at least not without costly interprocedural optimization).

2. The BLAS standard requires error checking and reporting (see above). For small problems, error checking might take more time than actually solving the problem.

3. The BLAS takes arrays by pointer, with run-time dimensions. Neither is optimal for very small matrices and vectors.

4. The BLAS only solves one problem at a time; it does not have a "batched" interface for solving many small problems at a time.

It turns out that our hypothetical C++ library has already laid the groundwork for solving all these problems. The C++ "fall-back" implementation is a great start for inlining, skipping error checks, and optimization. The `mdspan` class permits compile-time dimensions, or mixes of compile-time and run-time dimensions. It would be natural to specialize and optimize the C++ implementation for common combinations of compile-time dimensions.

The `mdspan` class is a view (**[views]**). Implementations store a pointer. Thus, it is not totally zero overhead for very small matrices or vectors with compile-time dimensions. A zero-overhead solution would only store the *data* at run time, not a pointer to the data; `std::array` is an example. Furthermore, it's awkward to use views

for very small objects (see example in P1684). Users of small matrices and vectors often want to handle them by value. For these reasons, we propose `mdarray` (P1684), a container version of `mdspan`.

Once we have C++ functions that take `mdspan`, it's not much more work to overload them to accept other matrix and vector data structures, like `mdarray`. This would also help our developer make their linear algebra library more like the C++ Standard Library, in that its algorithms would be decoupled from particular data structures.

The `mdspan` class also gives developers efficient ways to represent batches of linear algebra problems with the same dimensions. For example, one could store a batch of matrices as a rank-3 `mdspan`, where the leftmost dimension selects the matrix. The various layouts and possibility of writing a custom layout make it easier to write efficient batched code. For example, one could change the layout to facilitate vectorization across matrix operations. We have some experience doing so in our Kokkos and Trilinos libraries.

Composition of parallelism

Our developer may want to write a thread-parallel application. What happens if their BLAS implementation is thread parallel as well? This introduces two possible problems:

1. The BLAS' threads might fight with the application's threads, even while the application is not calling the BLAS.

2. The application may need to call the BLAS inside of a thread-parallel region. It may want to specify a subset of the computer's parallel hardware resources on which the BLAS should run, or it may not care what the BLAS does, as long as it doesn't make the calling parallel code slower.

BLAS implementations may use their own thread parallelism inside their library. This may involve OpenMP (as with some BLAS implementations) or a custom Pthreads-based back-end (as we have seen in earlier GotoBLAS releases). The only way to know is to read the implementation's documentation, and it may not be easy to control what it does at run time (e.g., how many threads it spins up).

What if our hypothetical developer wants to use thread parallelism in their own code? Their thread parallelism run-time system or implementation might fight with the BLAS' threads, even if our developer never calls the BLAS in a thread-parallel region. For example, one way to construct a thread pool is to pin threads to cores and have each thread spin-lock until work arrives. If both the BLAS and our developer do that, the two thread pools will fight constantly over resources.

In our experience, BLAS implementations that use OpenMP generally play nicely with the caller's use of OpenMP, as long as the caller uses the same compiler and links with the same version of the OpenMP run-time library. Even if the two thread pools play nicely together, what happens if our developer calls the BLAS inside of a parallel region? Intel's Math Kernel Library recognizes this by using the OpenMP API to determine whether it is being called inside of a parallel region. If so, it reverts to sequential mode. A smarter implementation could instead farm out work to other threads, if it makes sense for good performance. However, solving this at a system level, without our developer needing to change both their code and that of the BLAS implementation, is hard. For example, Pan 2010 shows how to nest parallel libraries and share resources between them without changing library code, but the approach is invasive in parallel programming run-time environments. "Invasive" means things like "reimplement Pthreads and OpenMP." Application developers may not have that level of control over the run-time environments they use.

Our developer could help by providing an optional execution policy (see **[execpol]** in the C++ Standard) that tells the BLAS what subset of parallel resources it may use. This is a logical extension of the C++ linear algebra interface we have been developing. Just like C++ Standard Algorithms, our developer's library could take optional execution policies. The optional execution policy parameter would also serve as an extension point for an interface that supports hierarchical parallelism (a "team-level BLAS"). That could also help with code that wants to solve many tiny linear algebra problems in parallel. This is the design choice we made in our linear algebra library proposal, P1673.

Optimize across operations

The functions in the BLAS only perform one linear algebra operation at a time. However, in many cases one can improve performance by doing multiple operations at a time. That would let the implementation fuse loops and reuse more data, and would also amortize parallel region launch overhead in a parallel implementation.

Provide specialized fused kernels?

One way to do this is simply to expand the set of operations in the interface, to include more specialized "fused kernels." The BLAS already does this; for example, matrix-vector multiply is equivalent to a sequence of dot products. BLAS 2 and 3 exist in part for this reason. The `xGEMM` routines fuse matrix-matrix multiply with matrix addition, in part because this is exactly what LAPACK's LU factorization needs for trailing matrix updates.

This approach can work well if the set of operations to optimize is small. (See e.g., Vuduc 2004.) The opaque interface to fused kernels gives developers complete freedom to optimize. However, it also makes the interface bigger and harder to understand. Users may miss optimization opportunities because they failed to read that last page of documentation with the fancy fused kernels. Thus, the more general the intended audience for a linear algebra library, the less value specialized fused kernels may have.

Expression templates?

Many linear algebra libraries use expression templates to optimize sequences of linear algebra operations. See P1417R0 for an incomplete list. Expression templates are a way to implement lazy evaluation, in that they defer evaluation of arithmetic until assignment. This also lets them avoid allocation of temporary results. When used with arithmetic operator overloading, expression templates give users a notation more like mathematics, with good performance. Some communities, like game developers, strongly prefer arithmetic operator overloading for linear algebra, so arithmetic operators on linear algebra objects strongly suggest expression templates.

We will consider two examples of linear algebra expressions:

1. Compute the 2-norm of `w`, where `w = alpha*x + beta*y + gamma*z` (`x`, `y`, `z`, and `w` are vectors, and `alpha`, `beta`, and `gamma` are scalar constants).

2. Compute `D = A*B*C`, where `A`, `B`, and `C` are matrices with run-time dimensions, `operator*` indicates the matrix-matrix product, and `B*C` has many more columns than `D`.

First example: Norm of a sum of scaled vectors

The first example is a classic use case for expression templates. As long as all vectors exist and have the same length, one can write an expression templates library that computes the entire expression in a single loop, without allocating any temporary vectors. In fact, one could even skip storage of the intermediate vector w, if one only wanted its norm.

This example may suggest to some readers that we don't even need this expression templates library. What if a future version of Ranges were to accept parallel execution policies, and work with `transform_reduce`? Wouldn't that make a expression templates library for linear algebra just a pinch of syntactic sugar? Such readers might go on to question the value of a BLAS 1 interface. Why do we need dot products and norms?

Computing norms and dot products accurately and without unwarranted underflow or overflow is tricky. Consider the norm of a vector whose elements are just a little bit larger than the square root of the overflow threshold, for example. The reference implementation of the BLAS does extra scaling in its 2-norm and dot product implementations, in order to avoid unwarranted overflow. Many BLAS 2 and 3 operations are mathematically equivalent to a sequence of dot products, and thus share these concerns.

Even if C++ already has the tools to implement something, if it's tricky to implement well, that can justify separate standardization. The C++ Standard Library already includes many "mathematical special functions" (**[sf.cmath]**), like incomplete elliptic integrals, Bessel functions, and other polynomials and functions named after various mathematicians. Special functions can be tricky to implement robustly and accurately; there are very few developers who are experts in this area. We think that linear algebra operations are at least as broadly useful, and in many cases significantly more so.

Second example: Matrix triple product

The second example shows the limits of expression templates in optimizing linear algebra expressions. The compiler can't know the optimal order to perform the matrix-matrix multiplications. Even if it knew the dimensions at compile time, optimizing the general problem of a chain of matrix multiplications is a nontrivial algorithm that could be expensive to evaluate at compile time. In this case, if the compiler evaluates `B*C` first, it would need to allocate temporary storage, since `D` is not large enough. Introducing `operator*` for matrix-matrix multiplcation thus forces the implementation to have storage for temporary results. That's fine for some users. Others could reach for named-function alternatives to arithmetic operators, to write lower-level code that does not allocate temporary storage.

When LAPACK moved from Fortran 77 to a subset of Fortran 90 in the 2000's, LAPACK developers polled users to ask if they wanted the interface to allocate scratch space internally (using Fortran 90 features). Users rejected this option; they preferred to manage their own scratch space. Thus, having the option of a lower-level interface is important for a general user community.

Expression rewriting with named functions

One way to avoid creation of temporaries in such expressions, is to combine named functions with expression templates. Such functions would pass an optional "continuation object" into the next expression. This approach would still enable high-level expression rewriting (like Eigen or uBLAS). It would also be agnostic to whether expression rewriting happens at compile time or run time. For example, for some operations with large sparse matrices and dense vectors, it could make sense to have a run-time expression rewriting engine. Compile-time expression templates might increase build time, so it would be helpful to have the option not to do them.

For example, suppose that someone wants to evaluate the dot product of two vectors `y` and `z`, where `z = A*x` for a matrix `A` and vector `x`. In Matlab, one would write `y'*(A*x)`. In our hypothetical named-function library, we could write something like this:

```
auto A_times_x = make_expression(z, prod(A, x));
double result = dot(y, A_times_x);
```

or perhaps like this instead:

```
double result = dot(y, prod(z, A, x));
```

The library would have the option to use `z` as temporary storage, but would not be required to do so.

There are two issues with this approach. First, users would need to query the library and the particular expression to know whether they need to allocate temporary storage. Second, expression templates in general have the issue of dangling references; see next section.

The dangling references problem

Expression templates implementations may suffer from the *dangling references problem*. That is, if expressions do not share ownership of their "parent" linear algebra objects' data, then if one permits (say) a matrix expression to escape a scope delimiting the lifetime of its parent matrix, the expression will have a dangling reference to its parent. This is why Eigen's documentation recommends against assigning linear algebra expressions to `auto` variables. This is an old problem: `valarray` has the same issue, if the implementation takes the freedom afforded to it by the Standard to use expression templates.

Copy elision makes it impossible for an expression type to tell whether it's being returned from a scope. Otherwise, the expression type could do something to claim ownership. One possible fix is for linear algebra objects to have `shared_ptr`-like semantics, so that expressions share ownership of the objects they use. However, that introduces overhead, especially for the case of tiny matrices and vectors. It's not clear how to solve this problem in general, especially if linear algebra objects may have container semantics.

The aliasing problem and opting out of lazy evaluation

Eigen's documentation also addresses the issue of aliasing and expression templates, with the following example. Suppose that `A` is a matrix, and users evaluate the matrix-matrix multiplication `A = A*A`. The expression `A = A + A` could work fine with expression templates, since matrix addition has no dependencies across different output matrix elements. Computing `A[i,j]` on the left-hand side only requires reading the value `A[i,j]` on the right-hand side. In contrast, for the matrix multiplication `A = A*A`, computing `A[i,j]` on the left-hand side requires `A[i,k]` and `A[k,j]` on the right-hand side, for all valid `k`. This makes it impossible to compute `A = A*A` in place correctly; the implementation needs a temporary matrix in order to get the right answer. This is why the Eigen library's expressions have an option to turn off lazy evaluation, and do so by default for some kinds of expressions. Furthermore, allocating a temporary result and/or eager evaluation of subexpressions may be faster in some cases.

This is true not necessarily just for expressions whose computations have significant reuse, like matrix-matrix multiply, but also for some expressions that "stream" through the entries of vectors or matrices. For example, fusing too many loops may thrash the cache or cause register spilling, so deciding whether to evaluate eagerly or lazily may require hardware-specific information (see e.g., Siek et al. 2008). It's possible to encode many such compilation decisions in a pure C++ library with architecture-specific parameters. See, e.g., this 2013 Eigen presentation, and "Lessons learned from `Boost::uBlas` development" in P1417R0. However, this burdens the library with higher implementation complexity and increased compilation time. Library designers may prefer a simpler interface that excludes expressions with these issues, and lets users decide where temporaries get allocated.

How general should expression rewriting be?

How many users actually write applications or libraries that have a large number of distinct, complicated linear algebra expressions, that are fixed at compile time? Would they be better served by specialized source-to-source translation tools, like those described in Siek et al. 2008?

Are arithmetic operators even a good idea?

Different application areas have different interpretations of "matrix" vs. "vector," row vs. column vectors, and especially `operator*` (dot? outer? Hadamard? element-wise? Kronecker?). This makes introducing arithmetic operators for matrices and vectors a bit risky. Many users want this, but it's important to get agreement on what the various operators mean. A general linear algebra library might prefer to stay out of this contentious debate.

Some remaining correctness issues

1. Does mixing precisions, real and complex, etc. do the right thing?

2. Some kinds of number types may call for a different interface.

Mixing precisions in linear algebra expressions

Suppose that a linear algebra library implements arithmetic operators on matrices, and users write `A * B` for a matrix A of `complex<float>`, and a matrix B of `double`. What should the matrix element type of the returned matrix be? The accuracy-preserving choice would be `complex<double>`. However, the `common_type` of `double` and `complex<float>` does *not* preserve accuracy in this case, as it is `complex<float>`. The most "syntactically sensible" choice would be `decltype(declval<complex<float>>() * declval<double>())`. However, this type does not exist, because C++ does not define the multiplication operator for these two types.

This issue may arise by accident. It's easy to get a `double` without meaning to, for example by using literal constants like 1.0. Implementers of mixed-precision libraries also need to watch out for bugs that silently reduce precision. See e.g., this issue in a library to which we have contributed. Internal expressions in implementations may need to deduce extended-precision types for intermediate values. For example, a double-precision expression may want to reach for a 128-bit floating-point type in order to reduce rounding error. This is especially important for the shorter floating-point types that people want to use in machine learning. This may call for traits machinery that doesn't exist in the C++ Standard Library yet.

There's no general solution to this problem. Both users and implementers of mixed-precision linear algebra need to watch out. The C++ interface we propose in P1673 avoids the issue of needing to deduce a return element type, because users must explicitly specify the types of output matrices and vectors. This means that users can do computations in extended precision, just by specifying an output type with higher precision than the input types. Implementations will still need to decide whether to use higher precision internally for intermediate sums.

Different interface for some kinds of number types?

Given a generic linear algebra library, users will put all sorts of number types into it. However, the BLAS was designed for real and complex floating-point numbers. Does a BLAS-like interface make sense for matrices of fixed-point numbers? Should such matrices carry along scaling factors, for example? What about "short" floating-point types that exclude `Inf` and `NaN` to make room for more finite numbers? Would that call for linear algebra operations that track underflow and overflow? How would such interfaces integrate with C++ versions of libraries like LAPACK? We are not experts on fixed-point arithmetic; for us, these are all open questions.

Summary

We started with a BLAS library, wrapped it in a C++ interface, and gradually adapted the interface for a better fit to C++ idioms. Without much effort, this development process fixed some serious performance and correctness issues that can arise when calling the BLAS from C++. Our paper P1673 proposes a linear algebra library for the C++ Standard that takes this approach. Even if such a library never enters the Standard, we think this style of interface is useful.

Acknowledgments

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Thanks to Damien Lebrun-Grandie for reviewing Revision 1 changes.

References

- J. Bilmes, K. Asanovíc, J. Demmel, D. Lam, and C. W. Chin, "PHiPAC: A Portable, High-Performance, ANSI C Coding Methodology and its application to Matrix Multiply," LAPACK Working Note 111, 1996.
- K. Goto and R. A. van de Geijn, "Anatomy of high-performance matrix multiplication", ACM Transactions of Mathematical Software (TOMS), Vol. 34, No. 3, May 2008.
- M. Hoemmen, D. Hollman, C. Trott, D. Sunderland, N. Liber, A. Klinvex, Li-Ta Lo, D. Lebrun-Grandie, G. Lopez, P. Caday, S. Knepper, P. Luszczek, and T. Costa, "A free function linear algebra interface based on the BLAS," P1673R6, Dec. 2021.
- C. Trott, D. Hollman, M. Hoemmen, and D. Sunderland, "`mdarray`: An Owning Multidimensional Array Analog of `mdspan`", P1684R1, Mar. 2022.
- Heidi Pan, "Cooperative Hierarchical Resource Management for Efficient Composition of Parallel Software,", PhD dissertation, Department of Electrical Engineering and Computer Science,

Massachusetts Institute of Technology, Jun. 2010.

- J. Siek, I. Karlin, and E. Jessup, "Build to order linear algebra kernels," in Proceedings of International Symposium on Parallel and Distributed Processing (IPDPS) 2008, pp. 1-8.
- J. Siek and A. Lumsdaine, "The Matrix Template Library: A Generic Programming Approach to High Performance Numerical Linear Algebra," in proceedings of the Second International Symposium on Computing in Object-Oriented Parallel Environments (ISCOPE) 1998, Santa Fe, NM, USA, Dec. 1998.
- R. Vuduc, "Automatic performance tuning of sparse matrix kernels," PhD dissertation, Electrical Engineering and Computer Science, University of California Berkeley, 2004.
- R. C. Whaley, A. Petitet, and J. Dongarra, "Automated Empirical Optimization of Software and the ATLAS Project," Parallel Computing, Vol. 27, No. 1-2, Jan. 2001, pp. 3-35.