A Proposal for a Next-Generation BLAS

Contributors: J. Demmel, M. Gates, G. Henry, X.S. Li, J. Riedy, P.T.P. Tang

August 17, 2017

1. Introduction

2. Data Layout

3. Checking and Reporting Exceptions in the BLAS
- 3.1. BLAS checking and reporting needs
- 3.2. Recommendation Summary
- 3.3. BLAS error checking options
- 3.4. Reporting errors
  - 3.4.1. Recommendations
- 3.5. Performance impacts of argument checking
  - 3.5.1. Recommendations
- 3.6. Inconsistent exception handling in the current reference BLAS, and a proposed consistent alternative
  - 3.6.1. Recommendations

4. BLAS G2

5. The Semantics of Reproducible BLAS
- 5.1. Type and Value Restrictions
- 5.2. Invoking Reproducibility
- 5.3. Mixed Precisions and Types
- 5.4. Support Functions
- 5.5. Reproducible LAPACK and ScaLAPACK

6. Batch BLAS

7. Fixed-Point BLAS

8. Recommended reference implementation
- 8.1. Support for Mixed and Extended Precision
- 8.2. Support for Reproducible LAPACK
  - 8.2.1. Routines in Current BLAS That Support Reproducible LAPACK
  - 8.2.2. Level-1 BLAS G2 Routines
  - 8.2.3. Level-2 BLAS G2 Routines
  - 8.2.4. Level-3 BLAS G2 Routines

9. C++ Interface
- 9.1. Constants
- 9.2. Extended precision
- 9.3. Exceptions
- 9.4. Types
- 9.5. Matrix Layout
- 9.6. Real vs. Complex Routines
- 9.7. Routines

10. Open Forum

11. Appendix: Notes from the February 2017 Workshop

1. Introduction

The Basic Linear Algebra Subroutines, BLAS Levels 1 through 3, have for four decades enabled the numerical linear algebra community to provide portable and high-performance implementations of important linear algebra packages. With advances of computational algorithms, wide adoption of parallel computing platforms and emergence of heterogeneous hardware environments, the BLAS API is no longer sufficient as basic building blocks that support algorithm expressions.

In 1996, a set of extensions to BLAS, called XBLAS, was formulated and incorporated into the overall BLAS documentation in 2001 and has been used in LAPACK since version 3.2 (2008). XBLAS supports a set of algorithmic advances that improve numerical accuracy through appropriate uses of extra and mixed precision. More recently, the rise of heterogeneous computing motivated some independent efforts to further extend BLAS. One group of extensions aims to accommodate mixed data types of matrix and vector arguments within a single BLAS function. The data types also include hitherto unsupported types such as IEEE-754 half precision and signed/unsigned integer. Another group aims to extend BLAS to allow for computation on multiple matrices and vectors with a single invocation of a BLAS function, the so called batch BLAS routines. Finally, recent advances on reproducible numerical computation necessitate a new kind of data structure for numerical values. Consequently, one should consider extending the BLAS routines to be building blocks for reproducible computations by supporting this special data structure.

This report presents a new BLAS interface.

1. We describe the naming convention of our interface, which is somewhat different from that of the classical BLAS. The main motivation is to have them accommodate the largely expanded need in modern numerical linear algebra software.
2. This interface accommodates all of the functionalities of BLAS as well as XBLAS.
3. The interface supports mixed-precision computations, as well as half-precision and a special data structure to facilitate reproducible computations (see the next item). It also allows for expansion in the future for additional types such fixed-point type sketched out toward the end of this document.
4. Reproducible LAPACK/ScaLAPACK can be implemented using this new interface, where the crucial data structure for reproducibility is well supported.
5. The Batch BLAS are still under debate in other forums, largely over whether “groups” should be enabled or not. Regardless of that decision, we suggest the batch API modifications be consistent with the other BLAS modifications here.
6. The new interface is quite flexible, and so permits specifying a large number of routines for which there are no currently known use cases. We therefore present a short list of reference routines that include the present in-use functionality of XBLAS as well as those that will be able to support a reproducible version of LAPACK. In other words, we do not

propose providing all the routines the new interface can specify, but just the useful subset, which may grow over time.
7. We propose an interface that consists of two levels: a lower level defining the symbols in libraries that explicitly spells out the precisions involved, so each variant has a different name, and high-level generic interfaces in C++ and Fortran that abstract the data types, drastically reducing the number of names and simplifying the application developer's use of BLAS. As an example, instead of calling sgemm, dgemm, cgemm, or zgemm, there would be a single gemm interface, overloaded for various data types, which can now be mixed, extended, reduced precision or even quantized. A suggested C++ API is provided in a later Section. Other higher level C++ APIs may be built upon this one, for instance those using expression templates (Eigen, MTL, Boost's uBLAS, etc.).

Note that we are not proposing a *replacement* for the current BLAS interface but rather a new naming scheme that can support upcoming extensions. Nevertheless, this new proposal's functionality covers that of the current BLAS. We envision vendors will continue supporting the existing BLAS, and possibly replacing them as a set of wrappers when eventually the new BLAS interfaces are well supported and offer high performance.

There are a number of places in this document that describe a set of reasonable design choices, and ask for feedback from the reader. These issues are highlighted in the Open Forum at the end.

2. Data Layout

Data layout and interface are tied together, so before discussing new interfaces, we describe some data layout options. For the original BLAS, there were four different data types (S, D, C, Z) and at least one data layout for matrices using these types. The "general" data layout for matrices was a two-dimensional array, and assuming column-major ordering, it included a "column stride", or leading dimension, to indicate the distance between elements in the same row, but in consecutive columns. If we assume all matrices in the next generation BLAS also specify a single column stride, then we can create an interface that looks similar to the current BLAS, where one passes in a matrix by passing the address of the first element, followed by the column stride (leading dimension) and separately elsewhere, the row and column size for that matrix.

Our data layout becomes complicated when we consider new data structures to handle things like extra precision or reproducible arrays. That is, if I now represent a matrix as a sum of matrices, for either extra precision or reproducibility, should elements be stored consecutively or not? Similarly, if I represent a complex matrix, should the real and imaginary parts be stored as two separate arrays, or should they be interleaved so each imaginary component of an entry follows immediately after the real component? For the standard BLAS, complex data is

interleaved. That is, if A is a complex matrix, then the current BLAS interleaves the data so that the imaginary part of A(1,1) is stored right after the real part.

Suppose we have an existing algorithm that works with A in double precision and we wish to see the impact on the developer's coding effort of presenting the data as A and A_lo, where A_lo is used to represent double-double errors not captured by A alone. The easiest thing for a developer might be just to add the new matrix A_lo to the existing algorithm. However, the data will not be "interleaved", and to use this strategy one needs to now pass A and A_lo into BLAS functions. Before one tackles the interface, we need to decide things like: should we require A and A_lo to both have the same leading dimension (column stride)?

For vectors, this is not as crucial, not only because the data movement on vectors is lower-order for BLAS levels 2 and 3, but also because the existing stride parameters already cover this. Suppose our vector is x and it's double-double so there's an x_hi and x_lo. Interleaved memory would indicate that the data would be stored as {x = x_hi(1), x_lo(1), x_hi(2), x_lo(2), etc..} But one can still call a kernel where they are separated with a call to &x(1), and incx_hi = 2 for x_hi and &x(2) and incx_lo=2 for x_lo. The non-interleaved kernels can easily do both interleaved and non-interleaved cases. The non-interleaved kernels separate out the vectors and those only interested in an interleaved option would still be fine using this stride trick.

For matrices, however, we cannot deal with both interleaved and non-interleaved with the same kernels, unless we employ the same trick we did with vectors. However, to perform that trick, we need to consider also a row-stride in addition to column-stride (leading dimension). That is, if {A = {A_hi(1,1), A_lo(1,1), A_hi(2,1), A_lo(2,1), etc.}, we can always specify a "row-stride" of 2, and now we can call a low-level kernel where the high and low part of the matrices are separated. Of course, if A_hi and A_lo have m rows and we interleave them into a single data type A representing double-double data, then the column-stride would also be at least 2*m, so this single decision of whether to interleave data impacts how to think of strides.

An advantage of adding row strides to the standard existing column-strides (leading dimension) is that this will enable kernels more applicable to tensor contractions. As a side note, several packages, such as BLIS, already expect users to pay attention to both row and column strides. This also provides another way to handle transposition.

This leaves us with three different paths:

1. Stick with the current BLAS limitation of enforcing all data to be interleaved.
    a. PRO: It aligns easier with higher level interfaces such as when one wants to use C++ to define a "double-double matrix type".
    b. PRO: It reduces the number of cases an implementer has to worry about.
    c. CON: It makes it difficult to do operations where one wants to just add some extra precision to an existing algorithm. So some users may be out of luck.
    d. CON: It's less flexible, so tensor operations may involve more copying.

2. Enforce all **new** data types (not the pre-existing interleaved complex types) to be non-interleaved — so for instance, in the complex*16-complex*16 type of C64x2, while the "leading'' and "trailing" matrices are separate, the complex-valued matrix elements are each stored with its real and imaginary parts next to each other (we propose that the column strides for multiple matrices can be different; using two separate leading dimensions is useful for placing a "tail" in workspace). Keeping the matrices non-interleaved and separate matches our current LAPACK coding with XBLAS.
    a. CON: It doesn't align as easily for higher level interfaces
    b. PRO: It reduces the number of cases an implementer has to worry about.
    c. PRO: It makes it easier to modify existing code, where extra precision can be pursued without recopying all the data into an entirely new format.
    d. CON: It's less flexible, so tensor operations may involve more copying.
    e. CON: It treats data types differently, so doesn't work well for templated code.
3. Put no restrictions on how users access the BLAS (interleaved or not), just add row-strides
    a. PRO: One can still create kernels that assume non-interleaved data, but call them efficiently from interleaved-data simply by playing with the row-stride.
    b. CON: It could potentially double the number of cases implementers may have to worry about, however this is not mandatory. One can still optimize for only a few special cases.
    c. PRO: It makes it easier to modify existing code, where extra precision can be pursued without recopying all the data into an entirely new format.
    d. PRO: It's more flexible, so operations like tensor contractions can be implemented more easily.

Much of the current document was written with option 2 in mind (many examples have non-interleaved assumptions when passing matrices), however we've been leaning toward option 3 because it's the most flexible and supports newer applications like in-place tensor contractions (no copying).

Each option changes what an interface using that option might look like. For instance, for option 1, we would have calls very similar to the current BLAS. For option 2, for each extended precision matrix in the new equivalent functionality of a current BLAS call, we'd need to pass two or more matrices. While the dimensions of the matrix don't need repeating, the leading dimension (column stride) may be different (unless we enforce it's the same). For option 3, for each kernel an implementer creates, we would have to pass in two matrices just like option 2, but also add a row stride to the current column stride (however, the higher level interfaces may be simplified tremendously). A single "matrix" may have different column / row strides if the input matrix is allocated from the higher-level application but the trailing matrix in the double-double sense is allocated from workspace.

Note that we continue to assume that old data in the current BLAS, like complex data, continues to be in interleaved format. We are treating double-double and analogs differently because

current LAPACK code keeps the trailing portion of a vector in contiguous workspace while the leading (and returned) portion is input with arbitrary increment.

3. Checking and Reporting Exceptions in the BLAS

3.1. BLAS checking and reporting needs

There are two primary error / exception checking and reporting needs in the BLAS:

1. catching semantic errors in arguments, and
2. reporting exceptional values in the input, computation, and/or output.

For the first, the reference BLAS checks that matrix and vector dimensions are consistent. Other implementations may check other issues including pointer alignment and aliasing, overlapping arrays, *etc*. The reference BLAS reports inconsistencies through a very heavy-handed method: The reference BLAS calls a provided `XERBLA` routine and returns. The reference `XERBLA` prints an error and `HALT`s. This definitely is not appropriate for the batched BLAS.

Managing and reporting exceptional values is not supported currently by the reference BLAS. Different users desire different combinations of checks on inputs, outputs, and/or intermediate computations. They also desire different levels of reporting and handling. Handling exceptional values also requires considering how such values are propagated through BLAS routines.

We would like feedback on our recommendations, summarized below. Later sections provide detailed descriptions of the large number of options, benefits and drawbacks for each, and the impact on optimized BLAS libraries. We intend these recommendations to apply to a new BLAS API covering the current usage, batched BLAS, and extended precision BLAS.

3.2. Recommendation Summary

Our recommendation is that the new BLAS interface should

1. check arguments for dimensional consistency (the second argument checking option below),
2. return an error code rather than use XERBLA or using an output INFO parameter,
3. provide XERBLA-like functionality through a separate debugging wrapper library,
4. rely on the platform's mechanism for tracking exceptional arithmetic (*e.g.* IEEE-754 flags),
5. provide routines to check matrices and vectors for exceptional values, and
6. guarantee consistent exception propagation (see Section 3.6 below).

These recommendations require wider discussion.

3.3. BLAS error checking options

We describe a design space of different ways to check for errors and exceptions when calling the BLAS and include our new recommendations. After discussing their pros and cons, we ask the reader for advice on which one(s) to provide in this new standard. The options range from fastest

to slowest:
(1) checking nothing (fastest, when it works), to
(2) just checking input parameters for inconsistent values like N<0 or LDA < max(N, 1) for matrix operations like GEMM (current practice in the BLAS, and in the default interface proposed elsewhere in this document), to
(3) checking input arrays for pointer validity (this may be in addition to (2)), to
(4) insisting on consistent floating point exception handling and propagation (this may be in addition to (2) and/or (3)), to
(5) reporting the existence of any exceptions or exceptional values (Infs and NaNs) to the users (this may be in addition to (2), (3) and/or (4); most performance expensive).

We note that the current reference BLAS, and presumably many vendor BLAS, do not propagate exceptions consistently (examples below). All options on the above cost spectrum have been requested by some users, perhaps just as a user-selectable option, to enable debugging. User selection may be done in several ways, for example
(1) by an input argument at run-time,
(2) by calling a BLAS routine with a different name (eg gemm_check instead of gemm),
(3) interposing pre-defined "validation layers"[1] in debugging builds,
(4) setting environment or other run-time system variables, or
(5) by linking a different version of the library.
The first two options permit selectively checking only specific calls.

Finally, the results of any error checking may be reported back to the user in several ways, including
(1) as an output argument (typically called INFO in libraries like LAPACK),
(2) as a value returned by the function (as in the default interface proposed earlier in this document),
(3) by calling a subroutine that may print an error message and possibly halt (current practice in the BLAS, calling subroutine XERBLA), or
(4) relying on a platform's exception flags (*e.g.* on platforms that support IEEE-754).
Reporting is tied to what will be checked and reported.

3.4. Reporting errors

The current reference BLAS (and LAPACK) routines call a subroutine XERBLA to report dimensional errors. A few practical issues have appeared over the years:

1. Only one instance can be linked without heroic, low-level effort. Different libraries calling the BLAS within one application may make different assumptions about XERBLA.
2. Some environments will want their own style of exceptions, and passing those across language/library boundaries can be difficult.
3. The use within LAPACK (and other users?) needs auditing to check if routines assume XERBLA HALTs. If a replacement does not abort, do the calling routines work?

[1] See Vulkan's validation layers:
https://github.com/KhronosGroup/Vulkan-LoaderAndValidationLayers/tree/master/layers

Passing an error code back to the caller is our preferred option. Providing an error code uses either a function's return value or an output INFO parameter. Both are feasible. We prefer returning an error code. Adding an output INFO parameter to *all* subroutines would permit DOT and I?AMAX to continue returning values. But supporting double-double DOT or returning the result of DOT in a reproducible structure already requires moving DOT's "return value" to an output argument. Using the return value as the reporting mechanism would provide a uniform mechanism that is easier to handle automatically.

The grouped batched BLAS add another twist. Different groups may have different dimensional errors. This is covered in depth in the separate proposal for the batched BLAS and does require an INFO output (and possibly input) array parameter. We recommend that the batched BLAS still return a scalar error code that gives an indication that some error occurs. This will save the conscientious user from checking the entire INFO array on every call.

Feedback from NAG brought up that first-pass debugging benefits from XERBLA's halt. To support this and other possible debugging uses, we recommend that there be a separate debugging wrapper library provided on platforms where that is feasible. Some platforms may support a very thin wrapper library like an ELF dispatcher that looks up the symbol, performs general argument checking, and then calls the primary BLAS library. This mechanism would be similar to typical relocatable library dispatch.

3.4.1. Recommendations

- Errors from dimension mis-matches, *etc* are returned. Every routine becomes an integer-returning function. Return values are encoded as in LAPACK's INFO. This implies the DOT and AMAX routines provide their results through an output argument (as in the BLAS Technical Forum standard). [Recommendation #2]
- A separate debugging wrapper library should be provided where feasible. [Recommendation #3]
- That the batched BLAS return the "worst" of the errors to be checked. The batched BLAS still need at least an output INFO argument to report errors in different groups or arrays.

3.5. Performance impacts of argument checking

Each form of argument checking imposes some run time costs. We feel the cost of checking dimensional arguments generally is acceptable. Providing yet another interface to remove that checking is not the best option for improving performance.

The performance cost of dimension checking appears small in many cases:

- Intel's `MKL_DIRECT_CALL` option appears to achieve a 3× FLOPS improvement for small matrices. One of its optimizations is removing argument checks, so this provides an upper bound on the performance improvement seen in MKL. In the same range, the dynamic generation approach of LIBXSMM achieves a 5× improvement through other techniques and does not avoid dimension checks. So while some performance is gained

by removing checks, more is possible by taking a different approach.

- If there are many small matrices, the batched interface may amortize the dimensional check cost across the batch.

The value of the dimension checks is not as easy to quantify. However, LAPACK still receives bug reports about surprising dimensions on input (often zero). Disabling internal checks may be premature.

Providing `nocheck` variants imposes some software development cost. That cost may be small but the longer term maintenance costs are high. Some vendors match the reference BLAS behavior precisely, and each additional interface will impose a maintenance burden on those vendors.

The Vulkan graphics API presents another interesting option likely inspired by MPI's profiling interface. Vulkan does not include argument checking by default, but it can be enabled through a validation layer. Layers intercept function calls via the Vulkan loader mechanism. In some ways, this is exactly what we want, but it requires a platform with sufficient capabilities to use LIBXSMM's generation technique for even better performance. There are BLAS implementations targeting embedded environments, FPGAs, and other systems where a layer system may require too much overhead.

The performance cost of scanning for exceptional values is much larger, $O(M \times N)$ for matrices. The tests may only be used for deciding to use a "consistent" version of the lower-level function. While a full survey of the BLAS work-avoiding tests remains in the future, known uses (see section 6) skip $O(M)$ or $O(N)$ vector operations when they are to be multiplied with zero. The cost of not skipping these operations is approximately cost of the scan. Users may best be served by having consistent propagation by default. Impact inside of LAPACK and ScaLAPACK is not yet estimated.

3.5.1. Recommendations

- Continue checking the dimensions by default. Implementations are free to provide mechanisms to disable dimension checks, but there likely are better mechanisms for achieving high performance. [Recommendation #1]
- Rather than checking for exceptional arithmetic throughout the BLAS, we can rely on external mechanisms like IEEE-754 flags. Quality BLAS implementations using internal parallelism will handle merging different threads' states. Current vector units support merged flags. [Recommendation #4]
- Add trivial routines that check if "exceptional values" like Infs or NaNs appear in vectors and matrices. [Recommendation #5]

3.6. Inconsistent exception handling in the current reference BLAS, and a proposed consistent alternative

Here we give some examples of inconsistent exception handling in the current reference BLAS, why they may be problematic, and possible alternatives for the new standard. These examples illustrate that "consistency" may depend on a consistent interpretation of NaNs, which could be

(1) "missing data" (as in the R statistical environment),

(2) “any possible finite number”,
(3) “any possible finite or infinite number”, or
(4) “anything, including invalid results like 0*Inf, 0/0, sqrt(-1) or a grape.”

For example 0*“any possible finite number” is 0, but 0*“any possible finite or infinite number” is “anything”. We note that the IEEE 754 Floating Point Standard Committee revisits this question occasionally, asking whether NaNs should be further specified to indicate whether they represent “missing data” or “other”, but a consensus has not arisen so far.

Throughout the following, we consider only consistent propagation of exceptional values. Any internal generation of these values may vary between runs (see the discussion of GEMM below for an example). Reproducibility is handled through providing reproducible BLAS which handle exceptional values consistently as a by-product. Some users may want input exceptional values to be propagated consistently without the potential performance penalty of reproducibility. We do not make guarantees about exception flags (if supported).

The examples below illustrate why it is difficult to define consistent exception handling in a way that is compatible with the competing demands of (1) compatibility with (inconsistent) past practice and resulting user expectations, (2) differing language standards, and (3) high performance. Our suggestion is to **have consistent exceptional value propagation by default**. Codes expecting the old behavior will use the old BLAS names. **With consistent exceptional value propagation, we propose using NaN interpretation (4) above.** In particular, NaNs should always propagate to the output, i.e. 0*NaN = NaN, with the following exceptions: First, when multiplying a matrix or a vector by a zero scalar, eg in gemm: C = 0*A*B + 0*C, then it is ok to skip the multiplication and assume the result is zero. Second, to accommodate different language standards’ definitions of complex arithmetic (see below), we accept exception propagation of either Infs or NaNs, since either one provides a warning to the user. Having consistent propagation by default reduces the number of interfaces and should assist highly vectorized implementations as well as the batched BLAS.

I{S,C}AMAX. The ISAMAX routine in the reference BLAS1 takes a real input array A(1:n) and returns the index of the entry of maximum absolute value; if there is more than one entry of maximum absolute value, the reference ISAMAX implementation returns the index of the first one:

```
isamax=1
smax = abs(A(i))
for i = 2,n
   if ( abs(A(i)) .gt. smax ) then
     isamax = i
     smax = abs(A(i))
   end if
end for
```

Since a comparison x.gt.y always returns false if either x or y is a NaN, this code returns 3 on input array A = [0, NaN, 2], i.e. the third entry is largest, but returns 1 on the permuted input

array A = [NaN, 0, 2]. We regard this exception handling behavior as inconsistent. There are several possible alternatives. For example, changing the comparison to

(abs(A(i)) .gt. smax) .or. isnan(smax)

guarantees that the code will return the index of the largest non-NaN entry of A, if one exists. If one considers NaN to mean "missing data", then this option is appropriate. On the other hand, using

(abs(A(i)) .gt. smax) .or. isnan(A(i))

would always return a NaN if one exists. If one wants to ensure that NaNs propagate to the output, so exceptions are not "lost", as we proposed above, then this option is appropriate. If one interprets NaN to mean any number, finite or infinite, and A contains both an Inf and a NaN, then one might want to return the index of the Inf, not the NaN, requiring a more complicated test, such as

(abs(A(i)) .gt. smax) .or. ((isinf(A(i)) .or. isnan(A(i))) .and. .not. isinf(smax)).

We note that using .not. (abs(A(i)) .le. smax), which seems most similar to the original code, behaves differently yet again.m

An alternative is to initialize `smax` to -1.0, `isamax` to 1, and start the loop at `i = 1`. If there are no numbers in the vector, `isamax` will return 1, the index of the first NaN. Otherwise, isamax will return the entry of the largest magnitude *number* (considering the extended reals) in the vector. This may be preferred if the entry is to be used as a scaling factor. NaN-oblivious code would scale by the found NaN, wiping out other information that may otherwise be useful in determining the source of the NaN. A similar argument could be made for ignoring infinities, but that requires an additional check within the loop. Note that even if a number is returned, platforms implementing IEEE-754 arithmetic will signal invalid for comparisons with any NaNs in the vector.

Our recommendation is that `I?AMAX` returns the index of the first NaN if any are present, and otherwise the index of the first largest magnitude number (including infinities if any are present).

`ICAMAX`, which uses `abs(real(A(i)))+abs(imag(A(i)))` instead of the more expensive `abs(A(i))`, can return the wrong answer even when all inputs are finite numbers: If `abs(real(A(i)))+abs(imag(A(i)))` overflows for two different values of `i`, then the first value of `i` will be returned, even if the second value is correct. For the purposes of choosing a pivot in Gaussian elimination, this (rare) result is still good enough.

TRSV. The TRSV routine in the reference BLAS2 solves a triangular system of equations T*x = b for x; T may be upper or lower triangular, and unit diagonal (T(i,i)=1) or not. One may also ask TRSV to solve the transposed linear system T^T*x=b. The reference TRSV returns x = [1; 0] when asked to solve U*x = b with U=[1,NaN; 0,NaN] and b=[1; 0], because it checks for trailing zeros in b and does not access the corresponding columns of U (which it would multiply by zero). More generally, TRSV overwrites b with x and checks for zeros appearing anywhere in the updated x, to avoid multiplying the corresponding columns of U by zero. This means solving U*x = b with U = [1,NaN,1; 0,1,1; 0,0,1] and b = [2; 1; 1] yields x = [1; 0; 1]. So in both cases the NaN(s) in U do not propagate to the result x (neither would an Inf, which if multiplied by zero should also create a NaN). However, if L is the two-by-two transpose of the first U above,

then calling TRSV to solve L^T *x=b, with b = [1; 0] returns x=[NaN;NaN]. This is the same linear system as above, but the reference implementation does not check for zeros in this case; this is inconsistent. Again, one could imagine vendor TRSV behaving differently (in Matlab, the NaN does propagate to the solution in these examples). In these cases, if NaN were interpreted to mean "some unknown but finite number", so that 0*NaN were always 0, then not propagating the NaN would be correct. But if NaN meant "some unknown, and possibly infinite number", then 0*NaN should be a NaN (the default IEEE 754 behavior), and not propagating the NaN is incorrect. Our proposal for consistent exception handling would disallow such checking for zeros.

GER. The GER routine in the reference BLAS2 computes A = A + alpha*x*y^T , where A is a matrix, alpha is a scalar, and x and y are column vectors. In the reference implementation of GER, if alpha = 0, the code returns immediately, so no Infs or NaNs in x or y propagate to A. If alpha is nonzero, and if any y(i) = 0, the code skips multiplying y(i) by alpha and x. But it does not check for zeros in x. So an Inf or NaN in y(i) will propagate to all entries in column i of A, but an Inf or NaN in x(j) may not propagate to all entries in row j of A; this is inconsistent. Vendor GER may again be different. Our proposal for consistent exception handling would allow checking for alpha=0, but not checking for zeros inside x or y.

GEMM. The GEMM routine in the reference BLAS3 computes C = alpha*A*B + beta*C. We discuss GEMM to illustrate that certain kinds of "inconsistencies" are in fact expected and acceptable. The reference implementation has special cases for beta in {0,1} and alpha = 0, to avoid multiplying by 0 and 1; when alpha = beta = 0 GEMM is used to initialize C to the zero matrix at minimal cost without doing any arithmetic, or propagating any Infs or NaNs, at all; this is expected by users and acceptable. Analogously to GER, the reference GEMM checks for zero entries in B, but not A, to avoid multiplication by zero; this inconsistency may be useful, if it leads to large speed-ups on sparse B matrices. It is expected that GEMM implementors will tune its performance, usually involving changing the order of summation to better match the underlying computer architecture. It is easy to see that this can change the computed answer because of roundoff (since floating point addition is not associative), but also because of different exceptions: Consider adding the four numbers x, x, -x and -x, where x is a positive finite number larger than half the overflow threshold. It is easy to see that the result of summing these four numbers could be zero (correct), +Inf, -Inf or NaN, depending on the order of summation. Indeed, all these values could be returned by different runs of the same program on the same computer, if the computer chooses to use a different number of parallel processors from run to run. Again, we believe this is expected and acceptable for consistency; users wanting reproducibility should use the reproducible BLAS.

Complex Multiplication. This last example shows that programming language standards do not always agree on exception handling in operations as basic as complex multiplication. The C99 and C11 standards define a complex number to be "infinite" if either component is infinite, even if one component is a NaN, and define multiplication so that infinite*(finite-nonzero or infinite) is infinite, even if the 754 rules would yield both components of the product being NaN. For example, straightforward evaluation of complex multiplication yields (Inf + 0*i)(Inf + Inf*i) = NaN + NaN*i, but the C-standard-conforming compiler yields (Inf + 0*i)(Inf + Inf*i) = Inf +

Inf*i. The C standard includes a 30+ line procedure for complex multiplication. There are similar rules for complex division (but no procedure is provided in the C standard document).[2] In contrast, the IEEE 754 standard and recent Fortran standards say nothing about handling exceptions in complex arithmetic. This is obviously a challenge for consistent exception handling as well. To accommodate this, we propose to allow exceptions to propagate either as Infs or NaNs, since either one provides a warning to the user.

3.6.1. Recommendations

- The new BLAS API should propagate exceptional values consistently. [Recommendation #6]
- The BLAS_AMAX_I32_* routine (the consistent version of the BLAS routines I{C,S,D,Z}AMAX) should output (via an OUTPUT parameter) the index of the first NaN if any are in the vector or else the first largest magnitude number (including infinities if any are present).

4. BLAS G2

We begin by recalling the existing BLAS interface and its limitations. The convention established by BLAS and adopted in XBLAS is to use the letters S, D, C, Z to denote both the mathematical type (real or complex) and the precision (single or double). The Fortran 77 name of a BLAS routine takes the form

<x>*blasFunction*(… parameters …)

where *blasFunction* is the functionality such as GEMM or DOT. The "x" that precedes *blasFunction* is one of S, D, C, Z and corresponds to the data type of the input and output vectors or matrices.

The Fortran 77 name of an XBLAS routine takes the form

BLAS_<x>*blasFunction*[_a_b][_X](… parameters …)

where *blasFunction* is the functionality such as GEMM or DOT. The "x" that precedes *blasFunction* is one of S, D, C, Z and corresponds to the data type of the output. The "_a_b", if present, represents a mixed precision calculation, with "a" and "b" being one of S, D, C, or Z. The "_X", if present, corresponds to internal computation being done in extra precision.

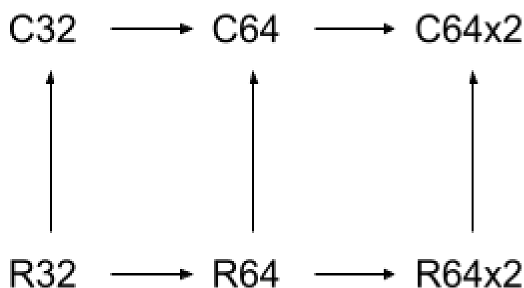
A number of limitations can be seen through these naming conventions. The input as well as output data types are restricted to one of the four choices, thus not allowing the input or output

[2] For other similar situations, see Kahan and Thomas, "Augmenting a Programming Language with Complex Arithmetic." https://www2.eecs.berkeley.edu/Pubs/TechRpts/1992/6127.html

to be in extended or mixed precision. Along the same line, the half-precision data type which is important in machine learning computations is not explicitly supported either. The fact that one can at most specify the data types for one output and two input data types, means that one cannot support, for example, the GEMM operation C_out ← ALPHA * A * B + BETA * C_in where one wishes to have the input C_in to be of a different data type than the output C_out. The fact that one only specifies the data type of the matrices imply that the data types of the scalars such as ALPHA and BETA are inferred. The inference rule will need to be updated should the matrices and vectors be of different data types. Beyond the limitations tied to the naming conventions of the routines, XBLAS also adopted the restriction that one can only either have mixed mathematical types (real and complex), or mixed precision (single and double), but not both simultaneously.

The next-generation BLAS proposed here, BLAS G2 (Generation 2), alleviates the above limitations in several ways.

- In lieu of S, D, C, Z to describe the data types, BLAS G2 adopts a more expressive convention. Examples of this convention -- detailed later -- are C64 and R64x2. C64 corresponds to complex, double precision (the traditional Z) and R64 corresponds to double precision (the traditional D) where the "x2" means "a pair," thus R64x2 is the familiar extra precision type double-double.
- Data types of matrices, vectors and scalars are specified and/or inferred in 3 possible ways. First, if only one data type is listed in the routine name, all matrices, vectors and scalars are inferred to be of the same type (excluding, obviously, integer parameters like N and LDA). Second, if this does not fit the intention of that routine, types of all the matrices and vectors must be specified individually in the name of the routine. From the types of all the matrices and vectors, the inference rule for the scalars is that all of them take the same data type that corresponds to the highest mathematical type (complex > real) and highest precision levels of all the input and output matrices or vectors (see the figure below). Thus for example the inference rule for the scalars mandate the mathematical type of all scalars will be complex if one of the matrices or vectors is of complex mathematical type, and real otherwise. The precision length is similarly the highest of the precision lengths of all the matrices and vectors. Third, if this inference result does not fit the intention of the routine, then the types of all the scalars must also



be specified individually in the name of the routine. We remark that many common routines have only one type specification in their names, and all except a handful of routines (e.g. the analogues of current BLAS routines crotg and csrot, and crot in LAPACK/SRC) can infer the scalars' types from the types of input matrices and vectors. Ideally these rules will simplify generating bindings from high-level languages.

- We propose deprecating XERBLA. Instead, all BLAS G2 routines are Integer typed functions whose returned values report exceptions. A zero return value indicates normal completion. A negative returned value of -K means the K-th input argument is problematic (for example violating some constraints that the function imposes), and a positive value can encode exceptions encountered during the execution of the specific function in question.[3] For the rest of this document, all BLAS G2 names are understood to be of type INTEGER FUNCTION. A debugging wrapper library can replace XERBLA's functionality for initial debugging.
- Grammar governing the naming convention will be given later; but a few examples here first. One example of the Fortran 77 name of a BLAS G2 is

  BLAS_GEMM_R32R32R64(…. ALPHA, A,..., B,..., BETA, C,...).

  Here C ← ALPHA * A * B + BETA * C where A and B are real single precision. C on input and output is real double precision. The scalars ALPHA and BETA are inferred to be real double precision. Extending the functionalities of BLAS and XBLAS, the GEMM_OUT function (_OUT stands for out-of-place) is defined where the output matrix C can be referenced differently from the input matrix C, as illustrated below:

  BLAS_GEMM_OUT_R64R64R64R64x2(…. ALPHA_hi, ALPHA_lo, A, lda, B, ldb, BETA_hi,
  BETA_lo, C_in, ldc_in, C_out_hi, ldc_out_hi,
  C_out_lo, ldc_out_lo)

  Note that this assumes a data layout where C_in and C_out are non-interleaved. See the Data Layout section for additional details on how this might change. Here, for

  (C_out_hi, C_out_lo) ← (ALPHA_hi,ALPHA_lo) * A * B + (BETA_hi,BETA_lo) * C_in,

  the input C matrix, C_in, and output C matrix, C_out, have different data types, the latter being real double-double. By inference, ALPHA and BETA are in real double-double precision (not double) as the highest precision length of all inputs and output corresponds to double-double. The user can of course use ALPHA and BETA to carry simple double-precision values. A typical implementation is expected to check if the input scalars are really just double precision in content and carry out natural performance

[3] An obvious alternative is to have all BLAS G2 routines be subroutines, and always carry an "INFO" parameter in the calling sequence. We solicit feedback and discussions on this design issue.

optimizations. As a third example, consider

BLAS_GEMM_R64(…. ALPHA, A,..., B,..., BETA, C,...)

With only one data type description, all input and output matrices/vectors are of the same type, which is real double precision in this case. The reader may readily recognize this is just DGEMM in the current BLAS. As a final example, the XBLAS extra precision capability is preserved by an extra token in the calling name

BLAS_GEMM_R64_64x2(…. ALPHA, A,..., B,..., BETA, C,...)

While all inputs/outputs matrices and vectors and scalars are real double precision, the operation is expected to be performed internally in double-double.

- As mentioned before, the number of names visible to the user may be reduced by using the generic interface feature of Fortran 90 and function overloading in C++, which allow the same subroutine name to be used with arguments of varying types. We note that specifications like "_64x2" for extra internal precision would still need to appear explicitly in the subroutine name. Sections 4–8 focus on the low-level, C/Fortran 77 interface, while section 9 provides a high-level, C++ interface. We leave a high-level Fortran 90 interface for future work.
- All integer dimension and increment parameters as well as integer return values are 64-bit integers. These are scalars, so this should not overly impact 32-bit platforms. Future wider integer platforms may want wider parameters, and we welcome comments on this choice.

We now give a more formal description of the low-level naming *scheme*. The specific names will be listed later on. We use a grammatical form for BLAS G2 names[4]. While a legal name must follow this grammar, not all functions with grammatically correct names are required, or in some cases, even allowed. These restrictions as well as a reference list of required functions will be detailed later. Note that we intend these names to be the symbols in the generated library and do not consider issues like Fortran compilers adding underscores, *etc*.

The grammar for a BLAS G2 function takes the form

BLAS_<blasFunction>_<typeSequence>[_<precisionLength>[<multiplier>]][_<suppl>]
(...*parameter list…*)

[4] There is related work for C++ in Trilinos: https://trilinos.org/docs/dev/packages/thyra/doc/html/LinearAlgebraFunctionConvention.pdf

<blasFunction> := dot | gemm | gemm_out | gemm_batch | trsv | … *etc* …
<typeSequence> := <type> | <typeSequence>
<type> := <mathType><precisionLength>[<multiplier>]
<multiplier> := x2 | Repro3 | … *different specific names as needed* ...
<mathType> := C | R … *the listed ones here are complex and real; this scheme allows for additions in the future, such as for example fixed point, signed or unsigned integers (as quantizations of some range), etc. See the section on fixed-point BLAS for a proposal.*
<precisionLength> := 8 | 16 | 32 | 64 | 80 | … *etc* …
<suppl> := Repro3 | … *other possibilities* …

The token <blasFunction> designates the particular mathematical operation in question and is familiar to BLAS users except for the fact that the scope of operations is expanded.

The token of <typeSequence> can consist of just one type, in which case all matrices, vectors and scalars share this same type, or consist of exactly as many as the total number of matrices and vectors (in which case the scalar types are inferred), or consist of exactly as many as the total number of matrices, vectors and scalars (and so specifying all the types individually). See the previous discussion on type specification and inference.

The <mathType> is currently C for complex, or R for real. A note on the leading dimension parameter: In a calling sequence, any array reference, such as A, is immediately followed by its leading dimension, such as LDA. Independent of the data type (eg R32, or C64x2), LDA counts the number of such entries in each column of A (so we can determine the distance between A(1,1) and A(1,2) in a column major formating), so independent of the number of words of memory each entry actually occupies. For example, when A is an 5x5 submatrix of a 10x10 array of type R64x2 while B is a 5x6 submatrix of a 10x20 array of R32, then the storage for A and B has 10 rows of R64x2 and R32 elements, respectively, and LDA as well as LDB is 10. Similarly, suppose A is R64Repro3, our special structure for reproducible computation, described below. Then an LDA of 10 means that each column of A’s storage has 10 R64Repro3 items (regardless of the sizes of M or N or K.)

The <multiplier> is meant to support both multiple-precision arguments and arguments represented using “reproducible accumulators” [5]. For example, R64x2 means each number is represented by two double-precision real arguments containing the high and low order bits of a real number, computed using double-double arithmetic [6]. And the type R64Repro3 means each number is represented by a reproducible accumulator of 3 “bins” (the structure of which is detailed in Section III below). The number 3 is the minimum number of bins, but can be higher.

---

[5] P. Ahrens, J. Demmel, and H. D. Nguyen, “Efficient Reproducible Floating Point Summation and BLAS,” UC Berkeley EECS Tech Report UCB/EECS-2016-121, June 2016, https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-121.html
[6] D. Bailey, J. Demmel, G. Henry, Y. Hida, J. Iskandar, W. Kahan, S. Kang, A. Kapur, X. Li, M. Martin, B. Thompson, T. Tung and D. Yoo, “Design, Implementation and Testing of Extended and Mixed Precision BLAS,” ACM Trans. Math. Soft. v. 28, issue 2, June 2002

The Repro option is only needed for floating point arithmetic, for which addition is not associative because of roundoff. It is not needed for integer arithmetic, for which addition is associative as long as modular (2s complement) arithmetic is used. Note however that If saturation arithmetic is used, then even integer addition is not necessarily associative in the face of overflow.

The tokens within the square brackets […] are optional. The first optional token specified by <precisionLength>[multiplier] corresponds to the precision length in which internal computations are performed. This matches the functionality of XBLAS and corresponds to the input argument of *PREC*.

The second optional token <suppl> denotes supplemental requirements. This is for possible enhancements such as a requirement for reproducible results, described next.

5. The Semantics of Reproducible BLAS

As floating-point arithmetic is not associative, even a simple dot product evaluation on the same input arguments can yield different results on different platforms or even the same platform on different invocations, if the summation is done in different orders. Algorithms have been devised recently to produce bitwise identical results that involves summation of floating-point values (see the publication in footnote 4, referred to below as [1]). As an example, the following BLAS G2 routine guarantees an identical result for identical input arguments

BLAS_DOT_R64_Repro3(N, ALPHA, X, INCX, BETA, Y, INCY, R)

(see below for possible restrictions on the values of the scalars ALPHA and BETA). If one wishes to use extra precision internally, the following can be used

BLAS_DOT_R64_64x2_Repro3(N, ALPHA, X, INCX, BETA, Y, INCY, R)

Since reproducibility is a new functionality that BLAS G2 is introducing, we discuss here in more detail exactly what reproducibility means, and what functions we propose to support. There are a number of design choices to make, so we present and explain our choices. We solicit advice from readers in cases when several choices seem reasonable.

Reproducibility is carefully defined in section 2 of [1], so we just summarize here. Our ultimate goal is being able to get bitwise identical answers on any computer, no matter what hardware resources are available, or how they are scheduled, for any size and ordering of inputs, that would get identical results in exact arithmetic. This is called “complete reproducibility” in section 2 of [1]. The algorithm in [1] has this property, assuming only that the number of double (resp. single) precision summands (e.g., the length of two vectors in a dot product) is at most 2^64

(resp. 2^33), and that a limited subset of IEEE 754 floating point standard operations are available. Furthermore, exception handling is also reproducible, i.e. infinities (Infs) and Not-a-Numbers (NaNs) are returned the same way[7]. If any computations are done in double-double (or single-single) precision, as in the second subroutine call above, then we must also make sure that the same double-double arithmetic algorithms are used (this should not be an issue on a single homogeneous computer, but could be on a heterogenous computer, or across different computers, or different compilers). We note that we must also precisely specify how arithmetic with complex numbers is done, since a "single" operation like multiplication or absolute value is actually a sequence of operations, which could potentially be done differently.

Therefore, the most straightforward way to achieve complete reproducibility, would be for users to use the identical reference implementation, which we would supply. A user or vendor who then modified the reference implementation to improve performance on a particular platform, in a way that might change the answer while maintaining reproducibility on that platform, should then document that their implementation only guarantees reproducibility on that platform.

For our reference implementation, we rely on the algorithm presented in detail in [1], so we summarize it briefly here, along with its error bounds. We take the entire floating point exponent range, and divide it into intervals with the same fixed width *w* (e.g. *w*=40 bits in double) and with fixed endpoints (e.g. the topmost interval starts at the largest exponent). Then we
(1) take each summand, and break its mantissa bits into the intervals they lie in,
(2) sum all the bits in each interval exactly (we call the accumulator corresponding to an interval a *bin*)
(3) keep only the topmost *k* consecutive bins, where *k* is a parameter chosen by the user (in the subroutine calls on the previous page, the suffix "_Repro3" means *k*=3). The algorithm represents a bin by 2 floating point numbers (in the same underlying format as the summands, or possibly wider[8]; for simplicity we assume the same format in this brief summary). We call the *k* consecutive bins (2*k* floating point numbers) a *reproducible accumulator.* When the sum is complete, we round the reproducible accumulator back to one floating point number. (We address the question of returning a format like double-double later.)

If S is the true sum of n summands x_i, S_repro is the sum computed by the algorithm, and epsilon is machine precision for the floating point format used for the x_i, bins and S_repro, then the error is bounded by

| S - S_repro | <= n * 2^((1-*k*)*w*) * max | x_i | + 7 * epsilon * | S |

The second term in this error bound is close to the minimum possible, which is one rounding error in the true sum (epsilon * |S|). The first term in the error bound is proportional to the

[7] This means, for example, that a NaN will always be returned for a given set of inputs, or never returned. Since the IEEE 754 Floating Point Standard does not specify the contents of the NaN (i.e. all its mantissa bits), we also cannot guarantee the reproducibility of the contents.

[8] In other words, if any input or intermediate result is double or double-double, then bins are constructed from doubles. Otherwise, if all inputs or intermediate results are single or single-single, then bins may be constructed either from singles or doubles.

largest summand in absolute value, as is the error bound for conventional summation, but by choosing *k* larger, it can be made smaller. For example, for double precision the smallest useful value is *k* = 3, and epsilon = 2^(-53), leading to the error bound (recall *w*=40)

$$| S - S_{repro} | \le n \cdot 2^{-80} \cdot \max | x_i | + 7 \cdot 2^{-53} \cdot | S |$$

which can be smaller than the conventional error bound by a factor 2^(80-53) = 2^(27) ~ 10^8 or even n*2^(27), depending on how much cancellation there is in the sum.

In summary, with the use of the reproducible accumulator, we can sum reproducibly summands that are either floating-point values of the same precision, or reproducible accumulators of the same precision, or a mixture of them. In BLAS G2, we use Repro*k* (*k* is a specific integer value, e.g. Repro3) to denote an accumulator with *k* bins.

With the reproducible summation algorithm in mind, we can now discuss various design issues involved in the BLAS G2 routines that support reproducible computations. These are

- *Type and value restrictions:* What data types to support and restrictions to impose on input/output matrices and vectors, and their associated scalars ALPHA and BETA. In particular, we describe several possible interfaces that restrict the possible values of ALPHA and BETA when reproducibility is requested, and solicit feedback on which interface is preferred.
- *Invoking reproducibility:* How to invoke reproducibility in BLAS G2, e.g. whether by specifying arguments as being stored as reproducible accumulators, or by independently asking for a reproducible result, or a combination.
- *Mixed precisions and types:* To what extent do we accommodate mixed precisions and mixed types in the input/output arguments when reproducible routines are invoked.
- *Support functions:* What additional routines do we propose in order to support parallel reproducible sum-reductions, and conversion between formats.

We discuss each of these in turn. At the end, we (1) discuss the design point used by the implementation (called ReproBLAS) described in [1], and (2) summarize a detailed analysis of LAPACK and ScaLAPACK, and what changes would be required to make them reproducible given the restricted interfaces that we propose. See also Section 8.2.

5.1. Type and Value Restrictions

We examine here how a reproducible matrix-matrix multiplication kernel that computes C_out ← ALPHA*A*B + BETA*C_in would be used.

Consider the data types that are involved. First, we know of no use cases where the scalars ALPHA or BETA, or matrices A or B, need to be represented by reproducible accumulators. If these were computed by a previous reproducible computation, then they may be rounded back to standard floating point numbers before calling GEMM, without loss of reproducibility. Next, C_in may be a floating-point valued matrix; but it can be a matrix of reproducible accumulators.

This latter scenario occurs, for example, in a parallel reduction where C_in is computed on a different processor. Moreover, neither the number of such reductions ALPHA*A*B+BETA*C_in nor the order in which they are performed are deterministic. For the same reasons, the output result C_out may be a floating-point valued or reproducible-accumulator valued matrix. We therefore impose the restrictions that ALPHA, BETA, A and B can only be floating-point valued, while C_in and C_out can also be matrices of reproducible accumulators.

Now consider the possible values of the (floating-point) scalars ALPHA and BETA. First, if the matrix C_in is of floating-point type, BETA*C_in can be computed reproducibly for general-valued BETA. On the other hand, we do not yet have a reproducible algorithm for multiplying a general scalar value by a reproducible accumulator. Hence if C_in is one such object, we would need to restrict the value of BETA. Fortunately, we know of no use case other than BETA in {+1,-1,0} when C_in is in fact a matrix of reproducible accumulators. Next we consider the values that the scalar ALPHA may have. We ask how we can compute ALPHA*A*B reproducibly, given that all are of standard floating-point values. If ALPHA is in {+1,-1,0}, the answer is already at hand: using reproducible accumulators for computing the inner products between rows of A and columns of B. If however ALPHA is not in {+1,-1,0}, we note that multiplying a partial sum represented by a reproducible accumulator by a floating point number not in {+1,-1,0} may create rounding errors that destroy reproducibility. This means that all multiplications by ALPHA need to occur either before summing with a reproducible accumulator, or after summing. Multiplying before summing means we compute each ALPHA*A(i,k)*B(k,j) as a floating-point value reproducibly, that is, computing the products in ordinary floating-point arithmetic in a predictable order. This could be done in one of the three ways below; assuming A is M-by-K and B is K-by-N, we also indicate the number of multiplications:

(1.1) sum_k (ALPHA*A(i,k))*B(k,j)   #mults = M*N*K + M*K
(assumes we can precompute and store at least one row of ALPHA*A). This method can support the case where C_in is a reproducible accumulator (with BETA restricted to {+1,-1,0} as discussed earlier). The result can also be a reproducible accumulator.

(1.2) sum_k A(i,k)*(ALPHA*B(k,j))   #mults = M*N*K + N*K
(assumes we can precompute and store at least one column of ALPHA*B). Same as (1.1), this method can support the case where C_in is a reproducible accumulator (with BETA restricted to {+1,-1,0} as discussed earlier). The result can also be a reproducible accumulator.

(1.3) sum_k (ALPHA*(A(i,k)*B(k,j)))   #mults = 2*M*N*K   (no extra storage needed).
Same as (1.1), this method can support the case where C_in is a reproducible accumulator (with BETA restricted to {+1,-1,0} as discussed earlier). The result can also be a reproducible accumulator.

In contrast to (1.1) through (1.3), we can multiply by ALPHA after summing:

(1.4) ALPHA * (sum_k A(i,k)*B(k,j)) #mults = M*N*K + M*N

where the reproducible sum sum_k A(i,k)*B(k,j) MUST first be rounded to a floating-point number before multiplication by ALPHA. Consequently, (1.4) cannot support the case when either C_in or the result C_out is a reproducible accumulator.

These many variations and restrictions prompt us to propose the BLAS_G2 matrix-matrix routines BLAS_GEMM_<typeSequence>_Repro*k* with restrictions on the possible values of ALPHA and BETA, but general enough to allow a user to implement any of (1.1), (1.2) or (1.4) in the parallel case (there does not appear to be a use case for (1.3), which may do twice as many multiplications as the other approaches).

We recommend one design choice for our BLAS G2 matrix-matrix multiplication routines, abbreviated as GEMM_Repro*k* Interface 1 (for ease of distinction here) and list two other progressively more restrictive alternatives for the reader's consideration. Note that all of these interfaces restrict the matrices A, B, and the scalars ALPHA and BETA to be floating-point (per previous discussions). Only C_in or C_out can possibly be represented by reproducible accumulators, denoted as Repro*k*.

In the table below, we summarize the constraints on each of the interfaces before giving more detail on the rationale and implementation methods. In the table, a type for C_in or C_out can be floating-point, denoted by FP, a reproducible accumulator, denoted by Repro*k*, or Any, meaning it can be either. For ALPHA and BETA, no constraints means they can be of general floating-point values, and constrained means only taking on values of +1, -1, or 0.

Interfaces	Type of (C_in, C_out)	Constraints on (ALPHA,BETA)?
Interface 1 ***Recommended*** Makes reproducible LAPACK easy to build	*Allow (Any,Any) with different constraints on (ALPHA,BETA)* (Any, Repro*k*) (Repro*k*, Any) (FP, FP) … used by LAPACK	 (Yes, Yes) (Yes, Yes) (No, No)
Interface 2 (slightly more restrictive)	*Allow (Any,Any)* (Any, Any)	 (Yes, Yes)
Interface 3 (most restrictive)	*Only allow type(C_in)=type(C_out)* (FP, FP) (Repro*k*, Repro*k*)	 (Yes, Yes) (Yes, Yes)

GEMM_Repro*k* Interface 1 (our recommended choice):
Each of C_in and C_out can independently be a floating-point type or a reproducible accumulator. The value of ALPHA and BETA are both restricted to {+1,-1,0} ***except*** when both

C_in and C_out are floating-point valued. (We emphasize here the restrictions on ALPHA and BETA apply only to the reproducible routines.) The ability to handle general-valued ALPHA and BETA in the case both C_in and C_out are floating-point valued will support a reproducible LAPACK more straightforwardly, not having to make changes to the current LAPACK code -- such changes would otherwise be needed had there been restrictions on ALPHA and BETA.

For completeness, we describe the 4 possible cases of GEMM_Repro*k* Interface 1, depending on whether C_in or C_out are stored as floating point numbers (i.e. R32, R64, C64, C64x2 etc.) or reproducible accumulators:

(2.1) This is the case when C_in and C_out are both floating-point values. When ALPHA is not in {+1,-1,0}, this means that A*B needs to be reproducibly computed first, rounded to floating point numbers, multiplied by ALPHA, and then BETA*C_in computed and added last. In other words, the implicit parentheses in the evaluation are C_out = (ALPHA*(A*B)) + (BETA*C_in), which corresponds to option (1.4) above. To guarantee reproducibility, we note that this final evaluation should not be done using FMA. When ALPHA is in {+1,-1}, then BETA*C_in should be added or subtracted from the reproducible accumulators representing A*B before rounding to floats; this means the superior error bound of reproducible summation will apply to the overall operation. This can be done whether or not C_in and C_out are the same matrix (they are the same in the conventional BLAS). Some buffer space may be needed for this, but this is already the case when using optimized (tiled) implementations.

(2.2) When both C_in and C_out are stored as reproducible accumulators, then each A(i,k)*B(k,j) is again rounded to a floating point number and added to (or subtracted from) the reproducible accumulator ±C_in(i,j), and returned as a reproducible accumulator. This again makes sense whether or not C_in and C_out are the same matrix. This is a natural building block for a parallel reproducible reduction.

(2.3) Suppose C_in is stored as reproducible accumulators, but C_out is not (so they cannot be the same array). This may be used as the final step in a reduction. Then the computation proceeds as in (2.2), and then is rounded to floating point and returned in C_out.

(2.4) Finally, suppose C_out is stored as reproducible accumulators, but C_in is not, so again they cannot be the same array. Then A*B is computed reproducibly, and added to (or subtracted from) ±C_in reproducibly, and returned as reproducible accumulators. This makes sense as the starting step of a reduction (leaves of the reduction tree).

When C_out is of type R64x2, then we need to be able to round a reproducible accumulator to double-double in an accurate and reproducible way. A natural proposal is to use the same Algorithm 6.12 in [1], but just do all the arithmetic in double-double. The corresponding error analysis in Lemma 6.9 would need to be modified accordingly; this is future work. Another possibility is to use Algorithm 6.12 to get an accurate double precision approximation x of the

value in the reproducible accumulator, subtract x from the reproducible accumulator, and use Algorithm 6.12 again.

For discussion and feedback from readers[9], we describe two other possible design points that progressively put more constraints on the calling sequences and more burden on the user to work around these constraints. The first one, which we call GEMM_Repro*k* Interface 2, while allowing C_in and C_out to be either a floating-point or reproducible-accumulator type, always restricts ALPHA and BETA to be in the set {+1, -1, 0}. This interface supports the reproducible computation of ±A*B (or ±A*B ± C_in), but puts the burden on the user when computing C_out = ALPHA*A*B + BETA*C_in for more general ALPHA and BETA. Compared to Interface 1, Interface 2 moves some of the implementation burden from the BLAS G2 builder to the user. The second design point, which we call GEMM_Repro*k* Interface 3, is the most restrictive of all. Not only ALPHA and BETA must only take on values from {+1,-1,0}, but the type of C_in and C_out must be the same. Nevertheless, we have confirmed that all three design points are adequate for supporting reproducible versions of LAPACK, and probably ScaLAPACK (given some implementation effort described below).

GEMM_Repro*k* Interface 2:
This interface is very nearly the same as the previous one. The only difference is that even when C_in and C_out are both of floating-point type, the values of ALPHA and BETA are still restricted to {+1,-1,0}. This interface supports the reproducible computation of ±A*B (or ±A*B ± C_in), but puts the burden on the user of computing C_out = ALPHA*A*B + BETA*C_in for more general ALPHA and BETA. The user may also implement either (1.1) or (1.2) this way, by precomputing some (or all) columns (resp. rows) of Bs = ALPHA*B (resp. As = ALPHA*A) and then multiplying A*Bs (resp. As*B).

For the four combinations of the types of C_in and C_out, (2.2) to (2.4) detailed previously apply. We only need to elaborate here the case when both C_in and C_out are of floating-point type, which is rather simple:

(3.1) C_in and C_out are of floating-point type, and ALPHA and BETA both restricted. Here, the summands ±A(i,k)*B(k,j) will be rounded and added using a reproducible accumulator, to which ±C_in(i,j) may also be added, before being rounded back to a floating point number C_out(i,j). This makes sense whether or not C_in and C_out are the same matrix. (See section 2 of [1] for a discussion of applying the usual tiling optimizations used in conventional matrix multiplication to the reproducible case.)

GEMM_Repro*k* Interface 3:
Our last design point is an even simpler one: C_in and C_out are restricted to be stored the same way (so either in floating point or as reproducible accumulators, and they may be the

[9] More formally, we ask the readers whether they prefer us to provide GEMM_Repro*k* Interface 1 (options (2.1) through (2.4)), the more restricted GEMM_Repro*k* Interface 2 (options (3.1), (2.2), (2.3) and (2.4)), or the even more restricted GEMM_Repro*k* Interface 3 (just options (3.1) and (2.2)).

same arrays or different), and ALPHA and BETA are both restricted to lie in {+1,-1,0}. This corresponds to options (3.1) and (2.2) above.

We note that all 3 design points, ReproBLAS_Interface_1, ReproBLAS_Interface_2 and ReproBLAS_Interface_3, allow the (parallel) reproducible implementations of (1.1), (1.2) and (1.4): To implement (1.1), the user precomputes and rounds some or all rows of ALPHA*A (call the result As, for scaled A), and then reproducibly computes C_out = As*B + BETA*C_in. Similarly, to implement (1.2), the user precomputes and rounds some or all columns of Bs = ALPHA*B, and then reproducibly computes C_out = A*Bs + BETA*C_in. Finally, to implement (1.4), the user reproducibly computes Ctemp = A*B, rounds Ctemp to floating point, and then does C_out = (ALPHA*Ctemp) + (BETA*C_in). We note that these straightforward approaches may require extra workspace, for As, Bs or Ctemp, so our analyses of LAPACK and ScaLAPACK, described later, were in part intended to identify when this workspace was needed; fortunately the answer is very rarely.

5.2. Invoking Reproducibility

In order to support parallel reproducible BLAS (a primary motivation for reproducibility), it is necessary to output the result stored as reproducible accumulators, which can then participate in a reproducible parallel reduction. This implies that data stored as reproducible accumulators also needs to be accepted as input, at least to contribute to a sum reduction.

We note that in the proposed subroutine naming grammar, reproducibility can arise in a BLAS G2 call in 3 ways: C_out may be stored as reproducible accumulators, C_in may be stored as reproducible accumulators, and the user may independently request reproducibility (with the suffix "_Repro*k*", *k* being the number of bins, in the subroutine name). For simplicity we will assume that if either C_out or C_in is stored as reproducible accumulators, then _Repro*k* must appear in the subroutine name, since we don't see a use case otherwise.

5.3. Mixed Precisions and Types

The next design question is compatibility of precision specifications of input/output variables (e.g. R64, R64x2 or R64Repro3, where the latter means a reproducible accumulator consisting of *k*=3 real double precision bins), and any extended precision specification (e.g. _64x2). First, we propose that all variables that are reproducible accumulators (e.g. C_in and C_out in GEMM) must have identical specifications, since we don't know of a use case otherwise, and in fact that all appearances of _Repro*k* in the subroutine name (the *k* being the number of bins in the reproducible accumulator) have the same value of *k*, e.g. _Repro3. Second, we disallow specifications of the form R64x2Repro*k*, i.e. that combine double-double (or single-single) with reproducibility. This is because the algorithms in [1] are only designed to work with IEEE 754 standard rounding modes, not the ones provided by double-double, and because one can adjust

the precision even more flexibly by choosing the number of bins *k*, so supporting double-double would not add any more functionality. Note also that in fact, a type R64x2Repro*k* is not supported by our proposed grammar: Both “x2” and “Repro*k*” are considered a multiplier when used in part of a type, for which only a single multiplier is allowed. Third, we propose that the minimum precision specified by *k* (so proportional to 2^((1-*k*)**w*), where *w* is the width of a bin, so *w*=40 in double precision and *w*=13 in single precision), be at least as precise as any other specified precision. For simplicity we also propose that the format of the bins be at least as wide as the format of the other variables; e.g. if A is R64, then the bins must also be represented using R64 (or wider), and the value of *k* must be at least 3. If some variables are R64x2, or the intermediate precision is specified as 64x2, then *k* must be at least 4. If the largest variables or intermediate precision is 32x2, then *k* must be at least 5 (see Table 2 in [1] for details).

Even with these constraints, the number of possible combinations of mixed precisions is large. We recommend implementing (at most) the subset that is identified in Section 8 of this document, “Recommended reference implementation”. [10]

5.4. Support Functions

Finally, we propose having some simpler routines to enable parallel reproducible summation, namely

C_out = ALPHA*C_in + BETA*C_out

where both ALPHA and BETA must be in {+1,-1,0}, at least one of C_in or C_out is stored as a reproducible accumulator, and the sum is done using the reproducible accumulator. The same precision compatibility rules as in (C) apply.

We now describe the design decisions made for the preliminary implementation described in [1] (called ReproBLAS, which should be subsumed into BLAS G2 here), and how they correspond to the choices described above. There is only one input/output argument C in matrix multiply, which can either consist of floating point numbers or reproducible accumulators. ALPHA can be any floating point number (ReproBLAS uses option (1.1) above). If C is floating point, then BETA can be any floating point number. If C consists of reproducible accumulators, then BETA is assumed to be 1. If necessary, C can be initialized to zero before calling matrix multiply. In other words, the design choices made by ReproBLAS differ slightly from our recommended ones above, but it would be straightforward to modify the ReproBLAS to match them.

5.5. Reproducible LAPACK and ScaLAPACK

Finally, we describe our analysis of LAPACK, and then ScaLAPACK, to determine whether the above proposed interfaces, in particular with ALPHA and BETA restricted to {+1,-1,0}, would be

[10] We invite input from the readers on which subset is worth implementing.

adequate to provide reproducible versions, with the additional goal of not changing any interfaces, i.e. invisibly to users. We first consider the sequential reference implementation of LAPACK, so that the only potential sources of nonreproducibility would be in BLAS calls. Ben Mehne of UC Berkeley generously provided a parsing tool that extracted all the calls to the BLAS in the LAPACK/SRC and LAPACK/TESTING directories (and their subdirectories), and their ALPHA and BETA parameters, so we could evaluate whether our proposed interfaces restricting ALPHA and BETA to lie in {+1,-1,0} would entail significant reprogramming efforts, and/or changes to interfaces to provide additional workspace (for As, Bs or Ctemp arrays as described above). To summarize, the LAPACK/SRC routines would be relatively easy to change, and require no interface changes, but the LAPACK/TESTING routines are harder, and would likely require interface changes. We believe this requirement is acceptable as most users do not directly call LAPACK/TESTING code. Our analysis of ScaLAPACK/SRC was less thorough, but it appears to be even easier to convert than LAPACK/SRC. A detailed routine-by-routine spreadsheet is available on request.

Here is a little more detail on the changes required in LAPACK/SRC. Fortunately, most of the 800+ BLAS calls had ALPHA and BETA in {+1,-1,0}, leaving roughly 100 to look at more carefully. Of these, we distinguish those with one scalar parameter ALPHA (ger, gerc, geru, her, hpr, spr, syr, trsm) and those with ALPHA and BETA (gemm, gemv, hemm, hemv, hpmv, spmv, symm, symv).

Of those LAPACK/SRC BLAS calls just with ALPHA not in {+1,-1,0}, all except trsm have just one or two additions and a few multiplications per output entry, for example ger: A = ALPHA*x*y^T + A, so reproducibility should follow from a deterministic implementation. For trsm, B = ALPHA*inv(T)*B, reproducibility should follow by premultiplying B by ALPHA (as in the reference implementation) and then implementing the triangular solve using reproducible summation.

Of those LAPACK/SRC BLAS calls with ALPHA and BETA not both in {+1,-1,0}, the following simple transformations let us convert ALPHA and BETA to both lie in {+1,-1,0} in most cases:

1) Replace y = 1*A*x + BETA*y by y = BETA*y, y = 1*A*x + 1*y
2) Replace y = ALPHA*A*x + 0*y by y = 1*A*x + 0*y, y = ALPHA*y

In a few cases, ALPHA and BETA were equal and could be factored out and multiplied by y later. Sometimes (in tptqrt2) this factoring was across more than one BLAS call. In a few other cases (in sygst and hegst), with ALPHA = .5 and BETA = 1, we could replace C = .5*A*B + C by

C = 2*C, C = A*B+C, C = .5*C

More generally, we could replace C = ALPHA*A*B + BETA*C by

C = (BETA/ALPHA)*C, C = A*B + C, C = ALPHA*C

but this is not always numerically reliable, depending on the magnitude of BETA/ALPHA. There is also an alternative algorithm for {sy,he}gst presented on page 77 of the following paper, which may eliminate this problem: http://epubs.siam.org/doi/pdf/10.1137/1.9780898718058.ch3

Now we consider LAPACK/TESTING/{LIN,EIG,MATGEN} routines. These do not test for reproducibility of LAPACK now, of course, and so would require further modifications to do so. We note again that changing interfaces or other workspace sizes in the TESTING code will not be noticed by most users, and so may be done with less risk than LAPACK/SRC. We consider only the few cases not addressed by the techniques mentioned above.

1) Routine LIN/drvrf4 tests BLAS3-like routines sfrk and hfrk, which perform C = ALPHA*A*A^T + BETA*C on a symmetric or hermitian matrix in RFP format, and calls BLAS3 routines sprk and hprk with general ALPHA and BETA to do so. This would most naturally be implemented using formula (1.4) above, requiring workspace Ctemp of size O(n^2), which is not available, and so would require a new workspace parameter (or local allocation). We note that sfrk and hfrk are not called by any other LAPACK/SRC routines. Since sfrk and hfrk are BLAS3-like routines, we may consider whether adding them to the new BLAS is worthwhile[11].
2) Routine EIG/get35 tests trsyl, which solves the Sylvester equation, and calls gemm with general ALPHA and BETA to do so. But all arrays are small (6x6 or 10x10) and declared locally, so declaring another one for workspace would be easy.
3) Routine EIG/get52 tests ggev, the generalized nonsymmetric eigensolver, and calls gemv with y = ALPHA*A*x + 1*y. This can be replaced by

   tmp = A*x, tmp = ALPHA*tmp, y = tmp + y

   Sufficient additional workspace is available in the real versions (WORK(N^2+1:N^2+N)), but may need to be passed or allocated in the complex versions.

Now we consider the changes needed to make ScaLAPACK reproducible. We only consider BLAS and PBLAS calls in ScaLAPACK/SRC, not other sources (e.g. calls to MPI_Reduce). This analysis is also less thorough than the LAPACK analysis because we may not have analyzed all calls to PBLAS-like routines, nor have we looked at ScaLAPACK/TESTING. That said, all BLAS calls can be handled by the simple transformations mentioned above, eg replacing
y = ALPHA*A*x + 0*y by y = 1*A*x, y = ALPHA*y, and replacing y = 1*A*x + BETA*y by
y = BETA*y, y = 1*A*x + 1*y. All PBLAS calls requiring changes are also either of the form
y = 1*A*x +BETA*y, or C = .5*A*B + C, as described above.

6. Batch BLAS

Multiple research groups are considering “batch” extensions to the BLAS where using a single call one requests multiple BLAS runs with similar parameters on different matrices. The community has yet to come to a consensus on how these functions should behave. NVIDIA inside cuBLAS implemented a batch GEMM of one type, MAGMA introduced its own batch GEMM, Sandia’s KokkosKernels have a third approach, and Intel MKL has a batch GEMM of another type. Some differences consist in how the matrices are stored, but in our view, the

[11] We invite reader comments on the possible inclusion of this new RFP matrix format in the BLAS.

biggest functional difference is that the Intel MKL batch GEMM introduces the notion of a group of GEMMs within a batch. So if one has to do ten GEMMs with M=N=K=10 and ten other GEMMs with M=N=K=15, with a cuBLAS routine one would do two GEMM batch calls, whereas with Intel MKL, one could do a single GEMM batch call but with two different groups.

The current implementation for Intel MKL GEMM batch has this Fortran syntax (see https://software.intel.com/en-us/node/590016):

?gemm_batch (transa_array, transb_array, m_array, n_array, k_array, alpha_array, a_array, lda_array, b_array, ldb_array, beta_array, c_array, ldc_array, group_count, size_per_group)

These arrays are of size “group_count”: transa_array, transb_array, m_array, n_array, k_array, lda_array, ldb_array, alpha_array, beta_array, ldc_array, size_per_group. That means that all matrices in the same “group” have the same fixed values for each of these parameters.

These three arrays are the size given by the sum of all “group_count” entries of the array size_per_group: a_array, b_array, c_array. These are arrays of pointers to the actual arrays of data. Because there are sum of all size_per_group matrices to operate on, that means that there is one pointer per matrix that we use, and not one per group or otherwise. So each BLAS operation can be done with different matrices and different pointers.

A quick example might help illustrate. Suppose we have 2 groups. The first group is two different GEMMs where in all cases M=3, N=3, K=4, ALPHA=-1.0, BETA=2.0,TRANSA=’N’, TRANSB=’N’, LDA is 3, LDB is 4, LDC is 3. The second group is three different GEMMs where in all cases TRANSA=’T’, TRANSB=’N’, ALPHA=-1.0, BETA=0.0, and all integer parameters are fixed at M=N=K=LDA=LDB=LDC=4. There are 5 total GEMMs that need to be done. Using the non-grouping technique would require two different BATCH calls. Using the grouping technique, all five GEMMs could be specified in the same call, but with group_count set at two.

If group_count is one, then the Intel MKL functionality behind a “batch” BLAS operation is the same as the cuBLAS/MAGMA functionality. One easy way to spot this is that cublasDgemmBatched() call has M, N, K as fixed single parameters and not arrays like mentioned above. But again, this is only a special case of the Intel MKL functionality because passing a single integer by reference is similar to an array of size one. So when the group_count is one, the Intel MKL functionality matches.

We will defer the debate on whether multiple groups should be allowed to further in-depth discussion on batch BLAS in the general BLAS Forum. We point out that there are different or unique storage schemes (such as Sandia’s Kokkos method) related to this topic. There are also efforts to reduce metadata by storing matrices contiguously, but details of alternate data storage formats is beyond the scope of this present document.

However, we would like to point out how the naming would work for batch BLAS in this proposal.

Instead of dgemm_batch() the routine would be BLAS_GEMM_Batch_R64(). The parameters themselves would be the same. If one wished to do internally R64x2, one can still write: BLAS_GEMM_Batch_R64_64x2().

So the syntax we recommend is identical, and it's just a name redefinition:

BLAS_GEMM_Batch_R64 (transa_array, transb_array, m_array, n_array, k_array, alpha_array,
a_array, lda_array, b_array, ldb_array, beta_array, c_array,
ldc_array, group_count, size_per_group)[12]

Adopting the Intel MKL convention, "batch" routines just turn single integer/character parameters (such as transa, transb, m, n, k, lda, ldb, ldc, alpha, beta) into elements in an array. The order of parameters is maintained (and would be for any "batch" version discussed in this document) with additional "group_count" and "group size" at the end.

7. Fixed-Point BLAS

There are interests in fixed-point computations in support of Deep Learning computations. The common usage model is that the fixed point values are represented by integer data with a scaling factor in mind, and thus fixed point values are equivalent to "scaled'' integers. Additionally, all elements of a single vector or a single matrix typically share the same scale factor. One possible way to accommodate this usage model is to introduce "scaled integer type" QI and that a scale factor given by a single signed integer has to be provided in the calling sequence for each of the scalars, vectors and matrices present. The scaled integer QI is signed integer and arithmetic saturates on overflow. For another activity in defining fixed-point arithmetic, see http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1169.pdf . In close relation to the fixed-point type discussed here is an effort to introduce quantized arithmetic, which is more general. See https://github.com/google/gemmlowp for example of one such effort. Because quantized arithmetic is more general, our fixed-point proposal here is by design more specialized: we do not consider unsigned fixed-point or modular arithmetic.

A QI is encoded by an integer pair (I, Q) which represents the value I/2^Q. The following illustrates a fixed-point matrix-matrix product:

BLAS_GEMM_QI32(… ALPHA, ALPHAQ, A, AQ, …, B, BQ, BETA, BETAQ, C, CQ, …)

For example, if (ALPHA,ALPHAQ) = (100,8), alpha is the value 100/2^8 = 0.390625. The value of beta is represented similarly. All the entries of the A matrix share a single Q value, that is,

[12] Comments on the syntax and convention are welcome. We are merely using a proposal suggested elsewhere and discussing the naming convention that aligns with this proposal.

while A is an array, AQ is just one integer. The value of the matrix element, say, a_(2,3) is A(2,3)/2^AQ. The same scheme is used for matrices B and C.

In a common usage, BETA is {+1,-1,0} and SUM_k ALPHA*A(i,k)*B(k,j) can be accumulated without overflow in some integer accumulator, and cast to (C,CQ) format, before summing to C. For example, suppose (ALPHA,ALPHAQ) is (1,0), AQ = BQ = CQ = 8; the integer value of the sum, SUM_k A(i,k)*B(k,j) is right shifted and "rounded" by 8 bits before adding to C(i,j). As expected, the definition in BLAS G2 allows for many more possibilities than common usages actually exploit.

Two final comments. We can also use [_<suppl>] to specify specific rounding methods, the absence of which suggests a default rounding-to-nearest method; and one can use the extra precision token to guarantee some internal width, for example, BLAS_GEMM_QI32_64(…) signifies that accumulation will be performed with extra width.

8. Recommended reference implementation

At present, we do not yet have a reference implementation of BLAS G2. The section enumerates what we consider to be a minimal list of routines necessary to support important uses. Individual vendors wanting to provide their own optimized implementations for the set of functions listed here may need only to fine tune a core subset within this reference list.

BLAS G2 in theory subsumes the existing BLAS library: every routine in BLAS has a straightforward correspondence to one in BLAS G2.[13] Nevertheless, we do not intend to deprecate BLAS nor do we foresee vendors eliminating the BLAS and rewrite existing LAPACK to call BLAS G2 instead of BLAS. Our reference implementation here mainly (1) supports anticipated common use of mixed precision and internal extended precision, including the current uses for XBLAS within LAPACK, and (2) supports reproducible versions of LAPACK and ScaLAPACK.

8.1. Support for Mixed and Extended Precision

The reference list here consists a number of matrix-vector routines whose blasFunction tokens are {GE,GB,SY,SP,HE,HP}MV, AXPY, and their _Out analogues. We also detail the calling sequences here. These routines are used in extra-precise iterative refinement routines for Ax=b and overdetermined least squares.

{GE,GB}MV $y \leftarrow \alpha A x + \beta y$; $y \leftarrow \alpha A^T x + \beta y$; $y \leftarrow \alpha A^H x + \beta y$

[13] Sometimes the one in BLAS G2 is more general. For example, the dot product in BLAS G2 is not a function but returns the result in the parameter list.

The parameter lists for GEMV and GBMV are:
BLAS_GEMV_<typeSequence>[_<precisionLength>[<multiplier>]][_<suppl>]
(TRANS, M, N, ALPHA, [ALPHA_LO,] A, LDA, X, INCX, [X_LO, INCX_LO,]
BETA, [BETA_LO,] Y, INCY[, Y_LO, INCY_LO])

BLAS_GBMV_<typeSequence>[_<precisionLength>[<multiplier>]][_<suppl>]
(TRANS, M, N, KL, KU, ALPHA, [ALPHA_LO,] A, LDA, X, INCX, [X_LO, INCX_LO,]
BETA, [BETA_LO,] Y, INCY[, Y_LO, INCY_LO])

The arguments X_LO, INCX_LO (and Y_LO, INCY_LO) are needed if and only if the corresponding type requires a pair, an occasion that is obvious from the function name.

The reference list of {GE,GB}MV is:

```
// standard precision
BLAS_{GE,GB}MV_{R32,C32,R64,C64}( ...parameter list…)

// internal extended precision
BLAS_{GE,GB}MV_{R32,C32}_32x2( ...parameter list…)[14]
BLAS_{GE,GB}MV_{R64,C64}_64x2( ...parameter list…)

// y extended precision
BLAS_{GE,GB}MV_R32R32R32x2( ...parameter list…)
BLAS_{GE,GB}MV_C32C32C32x2( ...parameter list…)
BLAS_{GE,GB}MV_R64R64R64x2( ...parameter list…)
BLAS_{GE,GB}MV_C64C64C64x2( ...parameter list…)

// x extended precision
BLAS_{GE,GB}MV_R32R32x2R32( ...parameter list…)
BLAS_{GE,GB}MV_C32C32x2C32( ...parameter list…)
BLAS_{GE,GB}MV_R64R64x2R64( ...parameter list… )[15]
BLAS_{GE,GB}MV_C64C64x2C64( ...parameter list… )

// x and y extended precision
BLAS_{GE,GB}MV_R32R32x2R32x2( ...parameter list…)
BLAS_{GE,GB}MV_C32C32x2C32x2( ...parameter list…)
BLAS_{GE,GB}MV_R64R64x2R64x2( ...parameter list… )[16]
BLAS_{GE,GB}MV_C64C64x2C64x2( ...parameter list… )
```

---

[14] This allows an implementation to use _64 instead of _32x2 as the former gives better accuracy.
[15] for Ax=b, current BLAS_DGEMV2_X - in dla_gerfsx_extended.f for example
[16] for overdetermined least squares

Note that the arguments Y_LO, INCY_LO are needed in _R64R64R64x2, and the arguments X_LO, INCX_LO are needed in _R64R64x2R64. All four are needed in _R64R64x2R64x2. ALPHA_LO and BETA_LO are needed for all three of those routines, per the scalar type inference rules.

{GE,GB}MV_Out   z ← α op(A) x + β y where op(A)=A, A^T, or A^H

The parameter lists for GEMV_Out and GBMV_Out are:

BLAS_GEMV_Out_<typeSequence>[_<precisionLength>[<multiplier>]][_<suppl>]
(TRANS, M, N, ALPHA, [ALPHA_LO,] A, LDA, X, INCX, [X_LO, INCX_LO,]
BETA, [BETA_LO,] Y, INCY, [Y_LO, INCY_LO,] Z, INCZ[, Z_LO, INCZ_LO])

BLAS_GBMV_Out_<typeSequence>[_<precisionLength>[<multiplier>]][_<suppl>]
(TRANS, M, N, KL, KU, ALPHA, [ALPHA_LO,] A, LDA, X, INCX, [X_LO, INCX_LO,]
BETA, [BETA_LO,] Y, INCY, [Y_LO, INCY_LO,] Z, INCZ[, Z_LO, INCZ_LO])

The reference list is:

```
// y out extended precision
BLAS_{GE,GB}MV_Out_R32R32R32R32x2( ...parameter list…)
BLAS_{GE,GB}MV_Out_C32C32C32C32x2( ...parameter list…)
BLAS_{GE,GB}MV_Out_R64R64R64R64x2( ...parameter list…)
BLAS_{GE,GB}MV_Out_C64C64C64C64x2( ...parameter list…)

// y out extended precision, x extended precision
BLAS_{GE,GB}MV_Out_R32R32x2R32R32x2( ...parameter list…)
BLAS_{GE,GB}MV_Out_C32C32x2C32C32x2( ...parameter list…)
BLAS_{GE,GB}MV_Out_R64R64x2R64R64x2( ...parameter list…)
BLAS_{GE,GB}MV_Out_C64C64x2C64C64x2( ...parameter list…)
```

Note that in the reference list, we omit the _Out routines where the shape/type of Y and Z are the same. At the cost of a copy, one can just use the non “_Out” version.

{SY,SP,HE,HP}MV   y ← α A x + β y; where A is Symmetric or Hermitian

The parameter lists for {SY,HE}MV :
BLAS_{SY,HE}MV_<typeSequence>[_<precisionLength>[<multiplier>]][_<suppl>]
(UPLO, N, ALPHA, [ALPHA_LO,] A, LDA, X, INCX, [X_LO, INCX_LO,]
BETA, [BETA_LO,] Y, INCY[, Y_LO, INCY_LO])

The parameter lists for {SP,HP}MV :
BLAS_{SP,HP}MV_<typeSequence>[_<precisionLength>[<multiplier>]][_<suppl>]

(UPLO, N, ALPHA, [ALPHA_LO,] A, LDA, X, INCX, [X_LO, INCX_LO,]
BETA, [BETA_LO,] Y, INCY[, Y_LO, INCY_LO])

The reference list of {SY,SP,HE,HP}MV is:

```
// standard precision
BLAS_{SY,SP}MV_{R32,C32,R64,C64}( ...parameter list… )
BLAS_{HE,HP}MV_{C32,C64}( ...parameter list… )

// internal extended precision
BLAS_{SY,SP}MV_{R32,C32}_32x2( ...parameter list…)
BLAS_{SY,SP}MV_{R64,C64}_64x2( ...parameter list…)
BLAS_{HE,HP}MV_C32_32x2( ...parameter list… )
BLAS_{HE,HP}MV_C64_64x2( ...parameter list… )

// y extended precision
BLAS_{SY,SP}MV_R32R32R32x2( ...parameter list…)
BLAS_{SY,SP}MV_C32C32C32x2( ...parameter list…)
BLAS_{SY,SP}MV_R64R64R64x2( ...parameter list…)
BLAS_{SY,SP}MV_C64C64C64x2( ...parameter list…)
BLAS_{HE,HP}MV_C32C32C32x2( ...parameter list… )
BLAS_{HE,HP}MV_C64C64C64x2( ...parameter list… )

// x extended precision
BLAS_{SY,SP}MV_R32R32x2R32( ...parameter list… )
BLAS_{SY,SP}MV_C32C32x2C32( ...parameter list… )
BLAS_{SY,SP}MV_R64R64x2R64( ...parameter list…)
BLAS_{SY,SP}MV_C64C64x2C64( ...parameter list…)
BLAS_{HE,HP}MV_C32C32x2C32( ...parameter list… )
BLAS_{HE,HP}MV_C64C64x2C64( ...parameter list… )

// x and y extended precision
BLAS_{SY,SP}MV_R32R32x2R32x2( ...parameter list… )
BLAS_{SY,SP}MV_C32C32x2C32x2( ...parameter list… )
BLAS_{SY,SP}MV_R64R64x2R64x2( ...parameter list…)
BLAS_{SY,SP}MV_C64C64x2C64x2( ...parameter list…)
BLAS_{HE,HP}MV_C32C32x2C32x2( ...parameter list… )
BLAS_{HE,HP}MV_C64C64x2C64x2( ...parameter list… )
```

{SY,SP,HE,HP}MV_Out $z \leftarrow \alpha A x + \beta y$; where A = A^T (SY) or A = A^H (HE)

The parameter lists for {SY,HE}MV_Out :

BLAS_{SY,HE}MV_Out_<typeSequence>[_<precisionLength>[<multiplier>]][_<suppl>]
(UPLO, N, ALPHA, [ALPHA_LO,] A, LDA, X, INCX, [X_LO, INCX_LO,]
BETA, [BETA_LO,] Y, INCY, [Y_LO, INCY_LO,] Z, INCZ [,Z_LO, INCZ_LO])

The parameter lists for {SP,HP}MV_Out :
BLAS_{SY,HE}MV_Out_<typeSequence>[_<precisionLength>[<multiplier>]][_<suppl>]
(UPLO, N, ALPHA, [ALPHA_LO,] A, X, INCX, [X_LO, INCX_LO,]
BETA, [BETA_LO,] Y, INCY, [Y_LO, INCY_LO,] Z, INCZ [,Z_LO, INCZ_LO])

The reference list is:

```
// y out extended precision
BLAS_{SY,SP}MV_Out_R32R32R32R32x2( ...parameter list…)
BLAS_{SY,SP}MV_Out_C32C32C32C32x2( ...parameter list…)
BLAS_{SY,SP}MV_Out_R64R64R64R64x2( ...parameter list…)
BLAS_{SY,SP}MV_Out_C64C64C64C64x2( ...parameter list…)
BLAS_{HE,HP}MV_Out_C32C32C32C32x2( ...parameter list…)
BLAS_{HE,HP}MV_Out_C64C64C64C64x2( ...parameter list…)

// y out extended precision, x extended precision
BLAS_{SY,SP}MV_Out_R32R32x2R32R32x2( ...parameter list…)
BLAS_{SY,SP}MV_Out_C32C32x2C32C32x2( ...parameter list…)
BLAS_{SY,SP}MV_Out_R64R64x2R64R64x2( ...parameter list…)
BLAS_{SY,SP}MV_Out_C64C64x2C64C64x2( ...parameter list…)
BLAS_{HE,HP}MV_Out_C32C32x2C32C32x2( ...parameter list…)
BLAS_{HE,HP}MV_Out_C64C64x2C64C64x2( ...parameter list…)
```

...same note as before: only gives those where Y and Z are of different type

AXPY $y \leftarrow \alpha x + y$

The parameter lists for AXPY:
BLAS_AXPY_<typeSequence>[_<precisionLength>[<multiplier>]][_<suppl>]
(N, ALPHA, [ALPHA_LO,] X, INCX, Y, INCY[, Y_LO, INCY_LO])

The reference list is:

```
// standard precision
BLAS_AXPY_{R32,R64,C32,C64}( ...parameter list… )

// y extended precision
BLAS_AXPY_R32R32x2( ...parameter list... )
BLAS_AXPY_R64R64x2( ...parameter list... )
BLAS_AXPY_C32C32x2( ...parameter list... )
```

BLAS_AXPY_C64C64x2(...parameter list...)

AXPY_Out z ← α x + y

The parameter list for AXPY is:
BLAS_AXPY_<typeSequence>[_<precisionLength>[<multiplier>]][_<suppl>]
(N, ALPHA, [ALPHA_LO,] X, INCX, Y, INCY, [Y_LO, INCY_LO], Z, INCZ[, Z_LO, INCZ_LO])

The reference list is:

```
// y extended precision, y_out extended precision
BLAS_AXPY_Out_R32R32x2R32x2( ...parameter list... )
BLAS_AXPY_Out_R64R64x2R64x2( ...parameter list... )
BLAS_AXPY_Out_C32C32x2C32x2( ...parameter list... )
BLAS_AXPY_Out_C64C64x2C64x2( ...parameter list... )
```

8.2. Support for Reproducible LAPACK

A reproducible version of LAPACK can be constructed by replacing some calls to the current BLAS library to a corresponding BLAS G2 reproducible routine. Depending on which GEMM_Repro*k* interface is adopted, varying amounts of minor modifications to the existing LAPACK source code will be required to accommodate the restrictions imposed by those interfaces. For example, if GEMM_Repro*k* Interface 2 is adopted, the LAPACK code that currently calls, say, DGEMM with ALPHA or BETA not in {+1,-1,0} has to be modified as outlined at the end of Section III. On the other hand, if GEMM_Repro*k* Interface 1 is adopted, we believe almost no modifications in LAPACK are required (beyond replacing existing BLAS calls with their reproducible counterparts, with the same arguments).

There are a number of routines in the current BLAS library that, when implemented naturally, and consistently, readily yield reproducible results. We list these routines first and recommend not including their BLAS G2 analogues in our reference BLAS G2 list. We point out however that some caution is needed. Non-reproducibility can arise due to inconsistent use or avoidance of the fused multiply-add instruction that is now part of IEEE754-2008. For instance, consider the rotation of the point (x,y) in 2-space by a cosine-sine pair (c,s):

x' ← (c * x) + (s * y) versus x' ← FMA(s, y, (c*x)), where FMA(a,b,c) computes a*b + c.

Non-reproducibility can also arise in simple computations such as the BLAS routine DGER, which computes A ← ALPHA x y^T + A where the core computation can be performed in at least two different ways:

A_ij ← x_i*(ALPHA*y_i) + A_ij versus A_ij ← (ALPHA*x_i)*y_j + A_ij.

8.2.1. Routines in Current BLAS That Support Reproducible LAPACK

These routines are reproducible, if implemented consistently as described above:

- {S,D,C,S,CS,ZD}ROT
- {S,D}ROTM
- {S,D,C,Z,CS,ZD}SCAL
- {S,D,C,Z}AXPY
- {S,D}{GER,SYR,SPR,SYR2,SPR2}
- {C,Z}{GERU,GERC,HER,HPR,HER2,HPR2}

We give the reference list of BLAS G2 routines needed to support a reproducible version of LAPACK.

8.2.2. Level-1 BLAS G2 Routines

- BLAS_DOT_R{32,64}_Repro3(N, ALPHA, X, INCX, BETA, Y, INCY, R)
- BLAS_DOT{C,U}_C{32,64}_Repro3(N, ALPHA, X, INCX, BETA, Y, INCY, R)
- BLAS_NRM2_R{32,64}_Repro3(N, X, INCX, R)
- BLAS_NRM2_C32R32_Repro3(N, X, INCX, R)
- BLAS_NRM2_C64R64_Repro3(N, X, INCX, R)
- BLAS_ASUM_R{32,64}_Repro3(N, X, INCX, R)
- BLAS_ASUM_C32R32_Repro3(N, X, INCX, R)
- BLAS_ASUM_C64R64_Repro3(N, X, INCX, R)

8.2.3. Level-2 BLAS G2 Routines

- BLAS_{GE,GB}MV_{R,C}{32,64}_Repro3
- BLAS_{SY,SB,SP,TR,TB,TP}MV_R{32,64}_Repro3
- BLAS_{SY,SB,SP}MV_C{32,64}_Repro3
- BLAS_{HE,HB,HP,TR,TB,TP}MV_C{32,64}_Repro3

Calling sequence for all the MV routines here are the same as those in the corresponding ones in the current BLAS library.

- BLAS_{TR,TB,TP}SV_{R,C}{32,64}_Repro3

Calling sequence for all the SV routines here are the same as those in the corresponding ones in the current BLAS library.

8.2.4. Level-3 BLAS G2 Routines

- BLAS_GEMM_{R,C}{32,64}_Repro3
- BLAS_{SYMM,SYRK,SYR2K}_R{32,64}_Repro3
- BLAS_{HERK,HER2K}_C{32,64}_Repro3
- BLAS_{TRMM,TRSM}_{R,C}{32,64}_Repro3

Calling sequence for all the MM, RK, R2K, and SM routines here are the same as those in the corresponding ones in the current BLAS library.

9. C++ Interface

While the low level API described in section 3 provides a mechanism that is fairly portable and callable from various languages, it is not very convenient for an end user to program with. C++ provides mechanisms to make the function name independent of the data type or precision, as well as to shorten the function name. For instance, with C++ we can use a single name blas::gemm, instead of the set of names: blas_gemm_r32, blas_gemm_r64, blas_gemm_c32, blas_gemm_c64, ... Making the function name independent of the data type is especially important to enable templated C++ code that works for any applicable data type. The specifications for the proposed C++ API are detailed below.

All the functions are in the blas namespace. Hence they require the blas:: prefix or a "using" statement:

```
blas::gemm( ... );
```

or

```
using blas::gemm;
gemm( ... );
```

or

```
using namespace blas;
gemm( ... );
```

Lowercase is chosen to match common C++ conventions such as the std namespace.

9.1. Constants

Character constants such as uplo and trans in the low-level API are replaced with C++ enum constants, with names similar to the ones in CBLAS. CBLAS defines these to be arbitrary integer values, such as CblasNoTrans=111. Instead, we define the name but leave the values undefined and implementation dependent.

> *Implementation note*: the arbitrary integer values in CBLAS require an if-then construct, switch, or lookup table to convert to the low-level character constants. By simply setting the enum constants to be the characters from the low-level API, only a cast at compile-time is required, with no runtime cost. For example:

```
typedef enum Op { NoTrans='n', Trans='t', ConjTrans='c' } Op;
```

The enum constants are as follows (except this does not define the values):

```
namespace blas {
enum class Layout : char { RowMajor, ColMajor };
enum class Op   : char { NoTrans, Trans, ConjTrans };
enum class Uplo : char { Upper, Lower };
enum class Diag : char { NonUnit, Unit };
enum class Side : char { Left, Right };
};  // end namespace blas
```

Title case was chosen to differentiate types and constants from variables, functions, and namespaces. The spelling and capitalization of constants ("NoTrans") is consistent with CBLAS, minus the Cblas prefix ("CblasNoTrans").

The enum name "Op" is suggested instead of "Transpose" to avoid confusion with the value Trans or NoTrans. The name Transpose::NoTrans could easily be misread as being transposed, rather than *not* transposed, whereas Op::NoTrans is clearer. The BLAS documentation, such as zgemm, already uses the term "op" in places:

```
C := alpha*op(A)*op(B) + beta*C
where op(X) is one of op(X) = X, op(X) = X**T, or op(X) = X**H.
```

9.2. Extended precision

For double-double arguments, the specification proposed in Section 3 uses two arrays, e.g., y_hi and y_lo. This choice facilitates adding y_lo to a routine that already has a vector y, without needing to interleave the vectors. In C++, two formats are supported. The first format is a std::pair of the two arrays, y_hi and y_lo (a struct-of-arrays implementation). Using a std::pair instead of two arguments differentiates the case of a double-double array from other cases where two arrays might be adjacent, such as the _out versions with y_in and y_out, or whether A or x is double-double in gemv. For the increment (incy) or leading dimension (lda), two versions are provided, one with incy as a std::pair of integers, another with a single integer for the common case when incy_hi and incy_lo are the same. For instance:

```
void blas::gemv( ..., std::pair< double*, double* > y, std::pair< int64_t, int64_t > incy );
void blas::gemv( ..., std::pair< double*, double* > y, int64_t incy );
```

These then call the low-level API with y_hi and y_lo arguments:

```
error = blas_gemv_r64r64r64x2( …, y.first, incy.first, y.second, incy.second );
error = blas_gemv_r64r64r64x2( …, y.first, incy, y.second, incy );
```

The second format is an array of std::pairs (an array-of-structs implementation). This format makes templated code easier, as the argument is still an array that supports subscripting, y[i],

just as with other data types. Compared to the first format, the difference here is where the pointer * is located, outside of std::pair< >.

```
blas::gemv( …, std::pair< double, double >* y, int64_t incy );
```

This then calls the low-level API with y_hi and y_lo arguments, doubling the incy to adjust for the stride between elements in the interleaved y array:

```
blas_dgemv_r64r64r64x2( …, &y[0].first, 2*incy, &y[0].second, 2*incy );
```

Unfortunately, this stride adjustment does not work for matrices, unless we add a row stride to the interface, as discussed in Section 2. Alternatively, if an array-of-structs version was added to the specification in Section 3, that could be called directly.

For extended precision routines, scalars alpha and beta are passed as a std::pair of scalars, following the type inference rules of Section 3.

TODO: confirm that std::pair guarantees elements are consecutive in memory.

TODO: to automatically convert scalar to double-double, it may be worthwhile to make our own class, say blas::extended, instead of using std::pair.

9.3. Exceptions

Instead of returning an integer error code, the C++ API throws an exception of type blas::Error, which is a subclass of std::exception. Throwing exceptions prevents accidentally ignoring errors by not checking the return value, and can simplify code by putting all the error checking code together, at the end, instead of interspersed. All errors that BLAS detects are programming bugs, such as wrong options or inconsistent dimensions, rather than runtime numerical issues, such as a singular matrix that LAPACK might detect.

9.4. Types

For complex values, C++ std::complex is used. Unlike CBLAS, complex scalars are passed by value, the same as 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* as CBLAS uses.

Array arguments that are read-only are declared const in the interface. Dimension-related and scalar arguments are passed by value, so are not declared const as there is no benefit.

Integers are passed as 64-bit signed values (int64_t).

9.5. Matrix Layout

Fortran BLAS assumes column-major matrices. CBLAS added support for row-major matrices. In most cases, this is accomplished with essentially no overhead by appropriately swapping

matrices, dimensions, uplo, transposes, etc., then calling the column-major routine. A few routines require extra effort, e.g., to conjugate gemv. CBLAS added a Layout argument (originally called Order) as the first argument, which we propose to also add for this purpose.

However, this may change depending on the layout discussion in Section 2. If row strides are introduced, row-major becomes supported in the low-level API, and there is no need to add an argument for it.

9.6. Real vs. Complex Routines

Some routines in the traditional BLAS have different names for real and complex matrices, for instance 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, we recommend both names are extended to apply to both real and complex matrices. For real matrices, herk and syrk are synonyms, 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. We recommend all these be included for completeness. The dot product has different names in real and complex. We extend dot to mean dotc in complex, and extend dotc and dotu both to mean dot in real.

Mapping of C++ generic name to BLAS G2 low-level names:

C++ name	**real**	**complex**
blas::hemv	symv_r{32,64}	hemv_c{32,64}
blas::symv	symv_r{32,64}	symv_c{32,64} †
blas::her	syr_r{32,64}	her_c{32,64}
blas::syr	syr_r{32,64}	syr_c{32,64} †
blas::her2	syr2_r{32,64}	her2_c{32,64}
blas::syr2	syr2_r{32,64}	syr2_c{32,64} ‡
blas::herk	syrk_r{32,64}	herk_c{32,64}
blas::syrk	syrk_r{32,64}	syrk_c{32,64}
blas::her2k	syr2k_r{32,64}	her2k_c{32,64}

blas::syr2k	syr2k_r{32,64}	syr2k_c{32,64}
blas::hemm	symm_r{32,64}	hemm_c{32,64}
blas::symm	symm_r{32,64}	symm_c{32,64}
blas::dot	dot_r{32,64}	dotc_c{32,64}
blas::dotc	dot_r{32,64}	dotc_c{32,64}
blas::dotu	dot_r{32,64}	dotu_c{32,64}

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

9.7. Routines

While each variant of the routines specified in Section 7 has a unique name, using C++ overloading, this set of names can be drastically reduced. In most cases, the variant to be called can be inferred from the argument data types. Exceptions to this are when internal behavior is specified, independent of the arguments. To use higher internal precision, while keeping regular precision input and output arguments, an _x suffix is appended. For reproducible BLAS, a _repro*k* suffix is appended. The _out versions, which have y_in and y_out as separate arguments, do not need an _out suffix in C++.

Here are examples for gemv. Other routines will follow in a similar manner.

```
// y = alpha * A * x + beta * y
void blas::gemv(
        blas::Layout layout, blas::Op trans, int64_t m, int64_t n,
        Talpha alpha,
        Ta const* A, int64_t lda,
        Tx const* x, int64_t incx,
        Tbeta beta,
        Ty* y, int64_t incy );

void blas::gbmv(
        blas::Layout layout, blas::Op trans, int64_t m, int64_t n, int64_t kl, int64_t ku,
        Talpha alpha,
        Ta const* A, int64_t lda,
        Tx const* x, int64_t incx,
        Tbeta beta,
        Ty* y, int64_t incy );
```

The data types Talpha, Ta, Tx, Tbeta, Ty are any supported data type, such as:

R32 float
R64 double
C32 std::complex<float>
C64 std::complex<double>

Extended precision arguments are in either format:

// pair of arrays
R32x2 std::pair< float*, float* >
R64x2 std::pair< double*, double* >
C32x2 std::pair< std::complex<float>*, std::complex<float>* >
C64x2 std::pair< std::complex<double>*, std::complex<double>* >
// currently, array of pairs for vectors only
R32x2 std::pair< float, float >*
R64x2 std::pair< double, double >*
C32x2 std::pair< std::complex< float >, std::complex< float > >*
C64x2 std::pair< std::complex< double >, std::complex< double > >*

The types Talpha and Tbeta follow the inference rules described above in Section 3. However, C++ will automatically promote data types, so often a user can use any lower precision data type. For instance, here alpha is automatically promoted from double to std::complex<double>:

```
std::complex<double> x(n), y(n);
double alpha = 1;
blas::axpy( n, alpha, x, 1, y, 1 );
```

For internal extended precision, _x is appended to the name, as in the XBLAS (see BLAST, Section 4.4.2). This maps to _32x2 and _64x2 variants.

```
void blas::gemv_x( layout, trans, m, n, alpha, A, lda, x, incx, beta, y, incy );
void blas::gbmv_x( layout, trans, m, n, kl, ku, alpha, A, lda, x, incx, beta, y, incy );
```

For _out variants, where z = alpha * A * x + beta * y, the output argument is added. An _out suffix is not needed.

```
void blas::gemv( layout, trans, m, n, alpha, A, lda, x, incx, beta, y, incy, z, incz )
void blas::gbmv( layout, trans, m, n, kl, ku, alpha, A, lda, x, incx, beta, y, incy, z, incz )
```

For reproducible variants, _repro*k* is appended, with the standard precision arguments.

```
void blas::gemv_repro3( … )
void blas::gbmv_repro3( … )
```

TODO: what is the data type for a reproducible accumulator? But it looked like none of the recommended routines in Section 7 had an argument that is a reproducible accumulator.

The recommended list of C++ routines are those that map directly to the recommended list from Section 7. Prototypes for gemv and gbmv are given below. For brevity, only double precision is provided; for other data types, replace double with float, std::complex<float>, or std::complex<double>.

```
// standard precision (same as dgemv, dgbmv)
{ge,gb}mv( …,   double alpha,
                double const* A, …,
                double const* x, …,
                double beta,
                double* y, … );

// internal extended precision, with _x suffix
ge,gb}mv_x( …,        double alpha,
                double const* A, …,
                double const* x, …,
                double beta,
                double* y, … );

// y extended precision, pair of arrays
// TODO: it would be helpful to have a version with alpha, beta as plain data types
// (float, double, …) instead of std::pair. Sigh, explosion of interfaces. Or some way
// that C++ can automatically upcast a data type to a std::pair of types — perhaps
// a custom blas::extended class instead of std::pair.
{ge,gb}mv( …,   std::pair<double, double> alpha,
                double const* A, …,
                double const* x, …,
                std::pair<double, double> beta,
                std::pair<double*, double*> y, … );

// y extended precision, array of pairs
{ge,gb}mv( …,   std::pair<double, double> alpha,
                double const* A, …,
                double const* x, …,
                std::pair<double, double> beta,
                std::pair<double, double>* y, … );

// x extended precision, pair of arrays
// TODO: best way to do const for x here? std::pair<float const*, float const*>?
// const works more naturally for array of pairs below.
{ge,gb}mv( …,   std::pair<double, double> alpha,
                double const* A, …,
                std::pair<double*, double*> x, …,
                std::pair<double, double> beta,
                double* y, … );

// x extended precision, array of pairs
```

```
{ge,gb}mv( …,   std::pair<double, double> alpha,
                double const* A, …,
                std::pair<double, double> const* x, …,
                std::pair<double, double> beta,
                double* y, … );

// x and y extended precision, pair of arrays
// TODO: const for x (see above)
{ge,gb}mv( …,   std::pair<double, double> alpha,
                double const* A, …,
                std::pair<double*, double*> x, …,
                std::pair<double, double> beta,
                std::pair<double*, double*> y, … );

// x and y extended precision, array of pairs
{ge,gb}mv( …,   std::pair<double, double> alpha,
                double const* A, …,
                std::pair<double, double> const* x, …,
                std::pair<double, double> beta,
                std::pair<double, double>* y, … );

// y_out extended precision, pair of arrays
{ge,gb}mv( …,   std::pair<double, double> alpha,
                double const* A, …,
                double const* x, …,
                std::pair<double, double> beta,
                double* y, …,
                std::pair<double*, double*> y_out, … );

// y_out extended precision, array of pairs
{ge,gb}mv( …,   std::pair<double, double> alpha,
                double const* A, …,
                double const* x, …,
                std::pair<double, double> beta,
                double const* y, …,
                std::pair<double, double>* y_out, … );

// y_out extended precision, x extended precision, pair of arrays
// TODO: const for x (see above)
{ge,gb}mv( …,   std::pair<double, double> alpha,
                double const* A, …,
                std::pair<double*, double*> x, …,
                std::pair<double, double> beta,
```

```
                double const* y, …,
                std::pair<double*, double*> y_out, … );

// y out extended precision, x extended precision, array of pairs
{ge,gb}mv( …,   std::pair<double, double> alpha,
                double const* A, …,
                std::pair<double, double> const* x, …,
                std::pair<double, double> beta,
                double const* y, …,
                std::pair<double, double>* y_out, … );

// reproducible, standard precision
{ge,gb}mv_repro3( …,        double alpha,
                        double const* A, …,
                        double const* x, …,
                        double beta,
                        double* y, … );
```

10. Open Forum

We gather a number of topics for which we solicit the larger community's input.

- On reproducible BLAS: In the previous section, we discussed the fact that multiplication of a reproducible accumulator with a scalar is difficult to accommodate. Three proposals were given on how to restrict the scalars ALPHA and BETA which were called GEMM_Repro*k* Interface{1,2,3}. While GEMM_Repro*k* Interface 1 is our recommendation, further suggestions, comments, discussions are most welcome.
- Is it worth adding the RFP matrix format to the BLAS?
- On the recommended list routines in our reference implementation list (section 8): Comments are solicited here on whether the list is complete, or can be streamlined.
- Batch BLAS is an ongoing activity, please give us feedback, especially on which of the two styles of interface (cf. the Batch BLAS section) is preferred; or perhaps both are considered needed.
- The fixed-point BLAS is very much a strawman proposal. We welcome discussions here.
- For all the cases where we now represent a matrix by multiple matrices, should we assume that they are interleaved into a single structure? If not, should each matrix have a different leading dimension (we strongly suggest "yes" because the original matrix dimensions may be large compared to a submatrix we're interested in)?
- Should we provide row strides to the current column strides to avoid the necessity of transposition, or specifying row-major matrices, and allow some operations like tensor contractions and the high-level interfaces to more easily support an interleaved data layout even when the kernels are non-interleaved?

- On error checking and exceptional value propagation, we have made the following potentially controversial recommendations that need wider consideration:
    - The new BLAS interface should always check arguments for dimensional consistency.
    - The BLAS should return an error code rather than using XERBLA or an INFO parameter, with a nod towards a different requirement for the batched BLAS.
    - We will rely on the platform’s mechanism for tracking exceptional arithmetic (*e.g.* IEEE-754 flags).
    - The new BLAS will guarantee consistent exceptional value propagation, while the older interface can do whatever.

11. Appendix: from the February 2017 Workshop

These are notes taken during the February 2017 workshop and may not reflect this draft. Some of the issues raised may already be addressed in the live document here.

- Note that extended-precision (internally) BLAS were first specified in Appendix B of Dongarra, *et al.* "An Extended Set of Fortran Basic Linear Algebra Subprograms." The implementations are under http://www.netlib.org/blas/#_extended_precision_level_2_blas_routines_2 .
- Note: _OUT routines may be inferable from the extra array.
- Do we want variants that do not check arguments for errors? What about errors during execution (zero diagonal entries in TRSV)? Possibilities using an INFO parameter below… (Do we need this at all?)
    - INFO = -1000 for turning off error checking (is this actually useful?)
    - INFO = -1001 for check LDA-style errors and NaNs input
    - INFO = -1002 for being consistent on checks for zeros, *etc*.?
- Concerns about symbol names v. dynamic arguments. Some attendees would prefer passing many of our "_suppl" portion as a dynamic argument similar to the current XBLAS. I (Jason) strongly disagree… But anyone could write a dispatch function that calls our specified routines.
- Other routines: cherk and cher2k have real alpha and/or beta, and so would require the full-length name specifying every argument. Note: We did handle this to some degree by removing a lot of the shorter names.
- From ARM's presentation: FP16 may need to fall back to FP32 arithmetic. Should we support a name for "maybe-wider" internal precision? It would be like the XBLAS "indigenous" precision, where 80-bit had the same role for x87 units.
- Asynchronous interfaces for generating task graphs / batches? [Having just reviewed the command buffer and pipeline complexity in Vulkan, from a community with experience, perhaps this best is deferred… -- Jason]
- Marat D. suggested defining "profiles" for different functionality. There could be a "classic BLAS" profile, a "low-precision" profile, *etc*.