Supporting Mixed-domain Mixed-precision Matrix Multiplication within the BLIS Framework

FIELD G. VAN ZEE, DEVANGI N. PARIKH, and ROBERT A. VAN DE GEIJN,
The University of Texas at Austin

We approach the problem of implementing mixed-datatype support within the general matrix multiplication (GEMM) operation of the BLAS-like Library Instantiation Software framework, whereby each matrix operand A, B, and C may be stored as single- or double-precision real or complex values. Another factor of complexity, whereby the matrix product and accumulation are allowed to take place in a precision different from the storage precisions of either A or B, is also discussed. We first break the problem into orthogonal dimensions, considering the mixing of domains separately from mixing precisions. Support for all combinations of matrix operands stored in either the real or complex domain is mapped out by enumerating the cases and describing an implementation approach for each. Supporting all combinations of storage and computation precisions is handled by typecasting the matrices at key stages of the computation—during packing and/or accumulation, as needed. Several optional optimizations are also documented. Performance results gathered on a 56-core Marvell ThunderX2 and a 52-core Intel Xeon Platinum demonstrate that high performance is mostly preserved, with modest slowdowns incurred from unavoidable typecast instructions. The mixed-datatype implementation confirms that combinatorial intractability is avoided, with the framework relying on only two assembly microkernels to implement 128 datatype combinations.

CCS Concepts: • **Mathematics of computing → Mathematical software performance**; **Computations on matrices**;

Additional Key Words and Phrases: Dense, linear algebra, DLA, high-performance, real, complex, mixed, datatype, type, domain, precision, matrix, multiplication, microkernel, BLAS, BLIS, libraries, framework

ACM Reference format:
Field G. Van Zee, Devangi N. Parikh, and Robert A. van de Geijn. 2021. Supporting Mixed-domain Mixed-precision Matrix Multiplication within the BLIS Framework. *ACM Trans. Math. Softw.* 47, 2, Article 12 (April 2021), 26 pages.
https://doi.org/10.1145/3402225

This research was partially sponsored by grants from Oracle, Huawei, and the National Science Foundation (Awards No. ACI-1550493 NSF, No. ACI-1714091 NSF, and No. CSSI-2003921 NSF). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).
Authors' addresses: F. G. Van Zee, D. N. Parikh, and R. A. van de Geijn, The University of Texas at Austin, Oden Institute for Computational Engineering and Sciences, Department of Computer Sciences, 2317 Speedway, Stop D9500, Austin, TX 78712; emails: {field, dnp, rvdg}@cs.utexas.edu.

0098-3500/2021/04-ART12 $15.00
https://doi.org/10.1145/3402225

1 INTRODUCTION

The Basic Linear Algebra Subprograms (BLAS) [8] define the general matrix-matrix multiplication (GEMM) operation to support any of the following computations:

$$\begin{array}{lll} C := \alpha AB + \beta C, & C := \alpha AB^T + \beta C, & C := \alpha AB^H + \beta C, \\ C := \alpha A^T B + \beta C, & C := \alpha A^T B^T + \beta C, & C := \alpha A^T B^H + \beta C, \\ C := \alpha A^H B + \beta C, & C := \alpha A^H B^T + \beta C, & C := \alpha A^H B^H + \beta C, \end{array}$$

where C is $m \times n$, the left-hand matrix product operand (A, A^T, or A^H) is $m \times k$, the right-hand matrix product operand (B, B^T, or B^H) is $k \times n$, and α and β are scalars.

This matrix multiplication functionality is made available to software developers via the following application programming interfaces, or APIs:

```
sgemm( transa, transb, m, n, k, alpha, A, ldA, B, ldB, beta, C, ldC )
dgemm( transa, transb, m, n, k, alpha, A, ldA, B, ldB, beta, C, ldC )
cgemm( transa, transb, m, n, k, alpha, A, ldA, B, ldB, beta, C, ldC )
zgemm( transa, transb, m, n, k, alpha, A, ldA, B, ldB, beta, C, ldC )
```

The first letter of the routine name uniquely encodes the datatype—that is, the domain and precision—of the matrix and scalar operands as well as the computation: single-precision real (`s`); double-precision real (`d`); single-precision complex (`c`); and double-precision complex (`z`). The parameters `transa` and `transb` indicate if A and/or B, respectively, should be computed upon as if they were transposed or conjugate-transposed. The interfaces implicitly require that matrices be stored in column-major order. Accordingly, the parameters `ldA`, `ldB`, and `ldC` convey the so-called "leading dimensions" of the arrays `A`, `B`, and `C`, respectively—that is, the number of elements that separate matrix element (i, j) from element $(i, j + 1)$ in memory.

While this interface has served the HPC community well, it has also become constraining. For example, when computing tensor contractions (which often resemble matrix multiplications), one may need to refer to a sub-tensor that cannot be represented with column-major storage without making a temporary copy. Similarly, some situations may call for conjugating (but not transposing) a matrix operand. Indeed, such functionality is already supported by the BLIS framework, which exports BLAS-like operations and APIs [28]. However, even BLIS only supports computation on operands with identical datatypes. Consider the following:

- There exist applications that may wish to compute upon matrices stored in different domains. For example, consider the update of a complex matrix by the product of a complex matrix and a real matrix, which occurs within applications involving damped response [6, 17], Green's functions methods [20], Complex Absorbing Potential (CAP), and Complex Scaling (CS) [15]. These mixed-domain instances of GEMM are currently improvised either by casting the operation in terms of `cgemm` or `zgemm`, in which case half of the floating-point operations are superfluous, or by performing two passes with `sgemm` or `dgemm`, which tends to be cumbersome and error-prone, requires extra workspace in which to make temporary copies of the real and imaginary parts of the complex matrix operands, and likely yields suboptimal performance. Another quantum chemistry application involves calculating the real or imaginary density of a complex wave function. In this scenario, the user aims to compute a complex matrix product from $A, B \in \mathbb{C}$ but then discard either the imaginary or real component of AB when storing the result to C. Currently, the BLAS does not allow this computation to be expressed except via `cgemm` or `zgemm`, either of which unnecessarily doubles the number of flops that must be executed.

- Similarly, there exist applications that could benefit from storing matrix operands in different precisions, and/or computing in a precision that is lower or higher than the storage precision of A and B. These include NWChem [1, 25] performing Coupled Cluster Singles and Doubles with Perturbative Triples, or CCSD(T) computations [7, 24], and various applications in machine learning [16]. NWChem and other quantum chemistry applications may derive particular benefit from two scenarios of mixed precision computation: (1) computing GEMM in single-precision arithmetic using double-precision matrix operands; and (2) computing GEMM in single precision using single-precision matrices A and B to update a double-precision matrix C. A third scenario calls for computing GEMM in single precision using a double-precision matrix A and a single-precision B to update a single- or double-precision matrix C. This may arise when using double-precision integrals and a ground state wavefunction, but computing the excited state(s) in single precision (since the latter can often tolerate a larger relative error). Another well-known example that uses mixed-precision computation is encountered when solving $Ax = b$ via a decomposition method (e.g., LU factorization with partial pivoting) and iterative refinement. In that setting, the LU factorization with pivoting is performed in a lower (e.g., single) precision and used to compute an approximate solution. Higher (e.g., double) precision is subsequently employed to compute the residual, which then is used to compute a correction to the approximation. This process can be repeated to further improve the precision of the solution. Currently, these mixed-precision computations must be performed in an ad hoc manner similar to the mixed-domain case and with similar workspace and performance drawbacks.
- Perhaps even more intriguing, some current or future applications may wish to use some combination of the aforementioned mixed-domain and mixed-precision functionality simultaneously.

Thus, there is likely a fair amount of pent-up demand for high-performance implementations to datatype-flexible BLAS-like APIs.

As alluded to above, the naive approach to supporting mixed-datatype functionality within the GEMM operation comes with obvious memory, performance, and productivity drawbacks: typecasting matrix operands to a common domain and/or precision outside of the original implementation requires considerable workspace; the memory access patterns engendered by monolithic casting almost certainly acts as a drag on performance; and programming an ad hoc solution in terms of the BLAS GEMM interfaces sometimes requires non-trivial skills. Indeed, some would find providing the full combinatorial space of functionality daunting, and in response might attempt to survey the community and then only implement those cases for which interest was expressed. Instead, our goal from the outset is to implement support for *all* cases, and to do so in a manner that delivers high or near-high performance.[1] Our approach, which builds on the BLAS-like Library Instantiation Software (BLIS) framework, yields a comprehensive reference implementation with which consumers of this functionality can explore the benefits of mixed-domain and mixed-precision GEMM computation without being constrained by limitations in the interface, incomplete coverage within the implementation, or unnecessarily inefficient performance.

[1]Understandably, some readers may question the utility of some mixed-datatype cases discussed in this article. Skeptics may argue that one only needs to focus on the cases that "make sense." We reason about the issue as follows. While we can identify certain cases that *are today* useful to *some* people or applications, we cannot say with certainty which cases *will never be* used by *any* person or application. And because we cannot *a priori* identify the cases that will never be needed, we take the position that we must treat all cases as important enough to merit implementation.

1.1 Notation

Our notation should be mostly self-evident to most readers of high-performance dense linear algebra literature. We use uppercase Roman letters, such as A, B, and C to denote matrices, and lowercase Greek letters α and β to represent scalars.

Real and complex domains are indicated by $\mathbb{R}$ and $\mathbb{C}$, respectively. Occasionally, we refer to the real part of a matrix or matrix expression X with $\mathcal{R}e(X)$ and to the imaginary part with $\mathcal{I}m(X)$. In other places, such as where this notation would be too cumbersome, we use superscripts, such as χ^r and χ^i for the real and imaginary components, respectively, of a scalar χ.

When representing elements within a matrix, we use a subscript to encode the row and column indices. For example, a scalar α_{13} would reference the element located in the second row and fourth column of a matrix A.[2]

2 BACKGROUND

In this section, we review the approach to matrix multiplication taken within the BLIS framework as well as some related implementation details that will provide important context to discussions later in this article.

2.1 Matrix Multiplication in BLIS

The GotoBLAS algorithm [9] for performing matrix multiplication underlies the highest-performing BLAS libraries for current general-purpose microprocessors. The BLIS framework, which implements GEMM and other matrix-matrix operations, refactors the GotoBLAS algorithm as pictured in Figure 1. BLIS isolates the code that needs to be optimized (in assembly code or with vector intrinsics) for different architectures in a *microkernel* that updates a very small submatrix, or microtile, of C with a sequence of rank-1 updates that are accumulated in registers [28]. All other loops and supporting kernels are implemented portably in C99. By contrast, Goto's implementation—also adopted by the OpenBLAS fork [21] of the GotoBLAS library—casts the computation into a larger assembly coded kernel. This larger unit of code, which corresponds to what BLIS refers to as the *macrokernel*, consists of the microkernel plus the logic that falls within the first two loops around the microkernel.

A key element of the GotoBLAS algorithm is that high-performance implementation of GEMM incorporates the packing of submatrices of B (into buffer $\widetilde{B}$) and of A (into buffer $\widetilde{A}$) to improve data locality during the execution of the microkernel.[3] This is achieved by partitioning matrices A and B into submatrices using carefully selected cache blocksizes m_C, n_C, k_C, which partition the m, n, and k dimensions, respectively. This packing has been used in the past to consolidate other functionality into the same framework: implementation of other matrix-matrix operations (level-3 BLAS) [10, 28], fusing sequences of matrix operations of importance to Machine Learning [30], and implementation of practical Fast Matrix Multiplication (Strassen-like) algorithms [13, 14]. A final insight comes from Van Zee [26], in which it is shown how complex matrix multiplication can be cast in terms of only microkernels designed for real domain GEMM without a significant performance penalty.

Given a target architecture, instantiating the traditional functionality of GEMM with BLIS requires only two microkernels, one each for single- and double-precision real domain computations, with the insight from Reference [26], inducing the functionality typically provided by complex domain microkernels. It also requires packing functions, which by default take the form of architecture-agnostic (C99) implementations provided by the framework, as well

[2] Our subscript notation starts counting from 0.

[3] This reorganization of matrices A and B can also improve TLB performance [11].

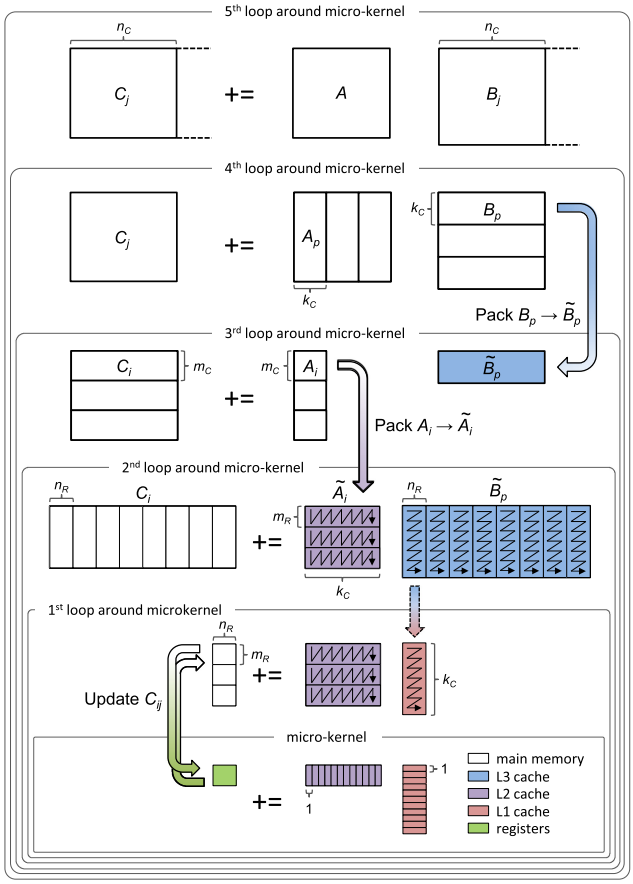

Fig. 1. The BLIS refactoring of the Goto algorithm as five loops around the microkernel. This diagram was modified from a similar image first published in Reference [27] and is used with permission.

as architecture-specific cache and register blocking parameters. BLIS exposes several other configure-time options, though they all default to values that typically need no further tweaking.

2.2 Managing Complexity in BLIS

The combinatorial complexity of the GEMM operation in BLIS is mitigated in several ways.

2.2.1 Storage. BLIS tracks separate row and column strides for each matrix object.[4] Using these two stride parameters, BLIS supports three matrix storage formats in its end-user APIs: row-major storage (where the column stride is unit); column-major storage (where the row stride is unit); and general storage (where neither stride is unit).[5] Each GEMM operand in BLIS may individually be stored in *any* of the three aforementioned storage formats. If we consider all possible variations of general storage as a single format, then this results in a total of $3^3 = 27$

[4]In BLIS, the row stride—like the leading dimension in row-major storage—expresses the number of elements that separate matrix element (i, j) and element $(i + 1, j)$ in memory. The column stride—like the leading dimension in column-major storage—expresses the number of elements that separate elements (i, j) and $(i, j + 1)$.
[5]We often refer to row-major matrices as being row-stored, and column-major matrices as being column-stored.

different storage combinations. The GEMM operation supports these 27 storage combinations as follows: the packing function is written generically to allow it to read from any of the three storage formats when reading matrices A and B (during the packing of $\widetilde{A}$ and $\widetilde{B}$), and the microkernel is required to handle input/output of C in any of the three supported formats.

2.2.2 Transposition. BLIS easily accommodates transposition of matrices A or B by swapping the row and column strides (and their corresponding dimensions) just prior to packing. This technique merely affects how the matrices are traversed rather than how they are stored; thus, we call this an "induced" (or logical) transposition, as no matrix elements are actually copied or moved.

2.2.3 Conjugation. In the case of complex matrices, optional conjugation[6] of A and B is handled during the packing into $\widetilde{A}$ and $\widetilde{B}$.

2.2.4 Multithreaded Parallelization. The authors of Reference [22] discuss how BLIS exposes many loops, each of which can be parallelized. Crucially, the complexity over matrix storage datatypes (domain and precision), transposition/conjugation parameters (`transA` and `transB`), and matrix storage formats (row, column, generalized) are orthogonal to the issues pertaining to extracting multithreaded parallelism from GEMM. Therefore, the insights of Reference [22] carry over to the parallelization when mixing domains and/or precisions.

2.2.5 Optimizing Input/Output on C. High-performance microkernels accumulate their intermediate results using vector registers. Thus, the microkernel author must decide whether to semantically assign the vector registers to contain contiguous rows or contiguous columns of the microtile submatrix. We refer to this as the microkernel's *register orientation*. Interestingly, the register orientation necessarily biases the microkernel toward loading and storing elements of C as either rows or columns, since performing IO on elements in the opposite orientation would require a sequence of costly permutation instructions. BLIS tracks this intrinsic property, or IO preference, of the microkernel so that the framework can transform the matrix problem to the microkernel's liking. For example, if our microkernel is row-preferential and the GEMM implementation is executed on a column-stored matrix C, then BLIS will employ a high-level transposition of the entire operation (to $C^T := B^T A^T$), so that, from the perspective of the microkernel, C^T *appears* to be stored in its preferred format—that is, a manner consistent with its vector register orientation.[7] Thus, regardless of whether the microkernel is row- or column-preferential, the GEMM implementation will, generally speaking, yield similar performance on row- and column-stored matrices C.[8]

2.3 For the Busy Reader

We acknowledge that some readers may wish for a very short synopsis of the insights presented later in this article. We sum up the techniques that allow implementing all cases of mixed-domain and mixed-precision GEMM within the BLIS framework as follows: The mixing of domains can be handled by enumerating the eight cases (six of them new), which largely reduce to either manipulating matrix metadata, and/or exposing the real and imaginary elements in a complex matrix

[6] In BLIS, all input matrix operands to GEMM and most other operations may be conjugated *without* transposition, which corresponds to a new `trans` parameter value absent in the BLAS.

[7] If, for whatever reason, this optimization is not employed, then the microkernel in this example would use the general storage case to read and write to a column-stored C, which would incur a small performance penalty due to an increased number of assembly instructions.

[8] Typically, the general-storage case in the microkernel must be handled separately from the contiguous row- or column-storage case that is preferred. This case usually incurs a performance penalty that is mostly unavoidable due to decreased spatial locality and an increased number of assembly instructions.

```
gemm( m, n, k, alpha, transA, A, domainA, precA, rstrideA, cstrideA,
                      transB, B, domainB, precB, rstrideB, cstrideB,
                 beta,        C, domainC, precC, rstrideC, cstrideC,
                                          precAB )
```

Fig. 2. A hypothetical BLAS-like API for mixed-domain, mixed-precision GEMM.

```
void bli_gemm( obj_t* alpha, obj_t* a, obj_t* b, obj_t* beta, obj_t* c );
```

Fig. 3. Function for computing GEMM provided by the object-based API in the BLIS framework.

in such a way that a real matrix multiplication may be performed to induce the desired result, in part motivated by insights from the 1M method [26]. The mixing of precisions can be handled by typecasting between precisions, as needed, during the packing of A and B (into $\widetilde{A}$ and $\widetilde{B}$), while the typecasting during the accumulation of the intermediate product AB may occur in special code wrappers that update C appropriately. And while the single-threaded performance of our datatype-mixing GEMM implementation performs on-par or slightly better than a so-called ad hoc implementation (i.e., one that is expressed in terms of existing BLAS, performing data copies to/from temporary workspace as needed), the advantages of our approach become more apparent when obtaining multithreaded parallelization, where we observe speedups ranging from approximately 15–100% depending on the hardware, datatype combination case, and problem size.

3 API CONSIDERATIONS

In the spirit of the BLAS API, a more complete interface that supports this richer, mixed-datatype environment could take the form of Figure 2. In Figure 2, `transX` indicates whether X should be computed upon as if it were (optionally) transposed, conjugate-transposed, or conjugated *without* transposition; `X` is the address where matrix X is stored; `domainX` indicates whether X is stored as a matrix of real or complex elements; `precX` indicates the precision in which elements of X are stored (e.g., half-, single-, double-, or quad-precision); `rstrideX` and `cstrideX` indicate the row and column strides, respectively, for storing[9] X; and `precAB` indicates the precision in which the matrix multiplication takes place (possibly implying promotion or demotion from the storage precision of either A or B). Note that column-major order and row-major order are the special cases where either `rstrideX` is unit or `cstrideX` is unit, respectively.

The hypothetical API shown in Figure 2 is given only to illustrate all of the dimensions of functionality along which the library developer must provide implementations. However, we feel that such an interface is not very useful toward *inspiring* those implementations, as it subtly nudges the developer toward extending the use of separate interfaces for each parameter combination down the function stack, leading to solutions that are vertically siloed from each other, even if various subsets share many similarities.

BLIS preempted this problem by starting with an object-based foundation for encoding and expressing matrix (and vector) operands. Each linear algebra entity (such as a matrix or vector) is encapsulated within a data structure, or more specifically, a custom `struct` type. For example, BLIS currently exports the following object-based function prototype for invoking the GEMM operation in Figure 3. In Figure 3, the function `bli_gemm()` takes five arguments of type `obj_t*`, each of which corresponds to the address of a `struct` representing the floating-point operands traditionally passed into the GEMM operation. The function exposes no other arguments, because

[9]Generally speaking, we consider these separate strides to support three storage formats—column-major, row-major, and so-called generalized storage (where neither the row stride nor column stride is unit).

```
typedef struct obj_s
{
    dim_t     offset_m, offset_n;
    dim_t     dim_m, dim_n;
    inc_t     rstride, cstride;
    doff_t    diag_off;
    siz_t     elem_size;
    objbits_t info;
    char*     buffer;
    // Other fields as necessary...
} obj_t;
```

Fig. 4. A simplified definition for the `struct` underlying the `obj_t` type used within BLIS.

all of the conventional parameters (such as `transA` and `transB`) may be interpreted as properties of one of the floating-point operands.

A simplified version of the `obj_t` type definition may be given as in Figure 4. In Figure 4, `dim_t`, `inc_t`, `doff_t`, `siz_t`, and `objbits_t` represent various integer types defined within BLIS for representing dimensions, strides and increments, diagonal offsets, byte sizes, and object property bitfields, respectively. The idea behind the `struct` example in Figure 4 is that matrices may be represented by a collection of properties, or metadata, and that these properties may be set—for example, when the object is initialized and its underlying data buffer is allocated—and then subsequently queried or modified using a collection of object-based accessor functions. Encapsulating matrix properties within objects helps hide details that need not be exposed at certain levels of the implementation.

The key observation to make now is that the `domainX` and `precX` arguments shown in Figure 2 can be completely hidden within the object API of BLIS. Indeed, the current definition of `obj_t` within the framework *already* includes domain and precision bits within the `info` bitfield. We only need to add an additional parameter, or designate additional bits within the `info` property, to support the computation precision (labeled in Figure 2 as `precAB`). Thus, it is possible to add mixed-datatype support to GEMM without any modification to the function interface to `bli_gemm()`.

A more thorough walkthrough of BLIS's object API is well beyond the scope of this article.[10] The main takeaway from this discussion is that the original author of BLIS designed the framework around an object-based core with the keen understanding that additional APIs of arbitrary format, including (but not limited to) those in the style of the BLAS, could always be layered on top of this more general abstraction. Consequently, any such APIs built above and beyond the underlying object layer are only incidental to the framework; they merely constitute syntactic re-expressions of some subset of the functionality made possible by the object foundation.[11]

4 SUPPORTING MIXED DOMAIN COMPUTATION

We consider the storage domain (real or complex) of the matrix to be orthogonal to the storage precision (single, double, etc). In this section, we consider how to handle mixing matrix operands

[10] Curious readers may find tutorial-like example codes included alongside the BLIS source code, which is primarily distributed via GitHub [4]. Markdown documentation is also made available, and may be conveniently rendered via GitHub with modern web browsers.

[11] The BLAS API provided by BLIS serves as a classic example of this kind of layering, as it builds on the object API to arrive an an interface that exactly mimics the BLAS, even if doing so precludes access to functionality and features that would otherwise be available.

of different domains. For now, the reader should assume that the storage precision is held constant across all matrix operands, and therefore can be ignored. We also ignore scalars α and β for the time being, which simplifies the general matrix multiplication operation to $C := AB + C$.

4.1 The 1m Method

The author of Reference [26] recently presented a novel method of computing complex matrix multiplication without relying upon kernels that explicitly perform complex arithmetic at the scalar level, as is typically the case in high-performance BLAS libraries. Instead, the so-called 1M method relies *only* upon matrix primitives (kernels) that compute real matrix multiplication. And unlike the older and more easily understood 4M method [27], 1M replaces each complex matrix multiplication with only a *single* real matrix multiplication.

The key to 1M is a pair of special packing formats, which Van Zee denotes 1E and 1R. The author illustrates the role of these two packing formats using the following example of complex matrix multiplication $C \mathrel{+}= AB$ where $m = 3$, $n = 4$, and $k = 2$:

$$\begin{pmatrix} \gamma_{00}^r & \gamma_{01}^r & \gamma_{02}^r & \gamma_{03}^r \\ \gamma_{00}^i & \gamma_{01}^i & \gamma_{02}^i & \gamma_{03}^i \\ \gamma_{10}^r & \gamma_{11}^r & \gamma_{12}^r & \gamma_{13}^r \\ \gamma_{10}^i & \gamma_{11}^i & \gamma_{12}^i & \gamma_{13}^i \\ \gamma_{20}^r & \gamma_{21}^r & \gamma_{22}^r & \gamma_{23}^r \\ \gamma_{20}^i & \gamma_{21}^i & \gamma_{22}^i & \gamma_{23}^i \end{pmatrix} \mathrel{+}= \begin{pmatrix} \alpha_{00}^r & -\alpha_{00}^i & \alpha_{01}^r & -\alpha_{01}^i \\ \alpha_{00}^i & \alpha_{00}^r & \alpha_{01}^i & \alpha_{01}^r \\ \alpha_{10}^r & -\alpha_{10}^i & \alpha_{11}^r & -\alpha_{11}^i \\ \alpha_{10}^i & \alpha_{10}^r & \alpha_{11}^i & \alpha_{11}^r \\ \alpha_{20}^r & -\alpha_{20}^i & \alpha_{21}^r & -\alpha_{21}^i \\ \alpha_{20}^i & \alpha_{20}^r & \alpha_{21}^i & \alpha_{21}^r \end{pmatrix} \begin{pmatrix} \beta_{00}^r & \beta_{01}^r & \beta_{02}^r & \beta_{03}^r \\ \beta_{00}^i & \beta_{01}^i & \beta_{02}^i & \beta_{03}^i \\ \beta_{10}^r & \beta_{11}^r & \beta_{12}^r & \beta_{13}^r \\ \beta_{10}^i & \beta_{11}^i & \beta_{12}^i & \beta_{13}^i \end{pmatrix}. \tag{1}$$

In this example, it is assumed that matrix C is column-stored, which prescribes that the 1E format be applied to A and the 1R format be applied to B. This can be confirmed by inspection: applying a real matrix multiplication to the left- and right-hand matrix product operands would not correctly compute the complex matrix multiplication if C were row-stored, because the intermediate elements of AB would update the wrong elements of C. However, 1M provides a cure to this situation. Namely, symmetry in the method allows for a row-oriented variant where C is row-stored:

$$\begin{pmatrix} \gamma_{00}^r & \gamma_{00}^i & \gamma_{01}^r & \gamma_{01}^i & \gamma_{02}^r & \gamma_{02}^i \\ \gamma_{10}^r & \gamma_{10}^i & \gamma_{11}^r & \gamma_{11}^i & \gamma_{12}^r & \gamma_{12}^i \\ \gamma_{20}^r & \gamma_{20}^i & \gamma_{21}^r & \gamma_{21}^i & \gamma_{22}^r & \gamma_{22}^i \\ \gamma_{30}^r & \gamma_{30}^i & \gamma_{31}^r & \gamma_{31}^i & \gamma_{32}^r & \gamma_{32}^i \end{pmatrix} \mathrel{+}= \begin{pmatrix} \alpha_{00}^r & \alpha_{00}^i & \alpha_{01}^r & \alpha_{01}^i \\ \alpha_{10}^r & \alpha_{10}^i & \alpha_{11}^r & \alpha_{11}^i \\ \alpha_{20}^r & \alpha_{20}^i & \alpha_{21}^r & \alpha_{21}^i \\ \alpha_{30}^r & \alpha_{30}^i & \alpha_{31}^r & \alpha_{31}^i \end{pmatrix} \begin{pmatrix} \beta_{00}^r & \beta_{00}^i & \beta_{01}^r & \beta_{01}^i & \beta_{02}^r & \beta_{02}^i \\ -\beta_{00}^i & \beta_{00}^r & -\beta_{01}^i & \beta_{01}^r & -\beta_{02}^i & \beta_{02}^r \\ \beta_{10}^r & \beta_{10}^i & \beta_{11}^r & \beta_{11}^i & \beta_{12}^r & \beta_{12}^i \\ -\beta_{10}^i & \beta_{10}^r & -\beta_{11}^i & \beta_{11}^r & -\beta_{12}^i & \beta_{12}^r \end{pmatrix}. \tag{2}$$

In this case, the application of the packing formats is reversed, such that 1E is applied to B and 1R is applied to A. Van Zee later points out that it is not actually the storage of C that determines whether Equation (1) or Equation (2) is employed, but rather the SIMD register orientation of the underlying real domain microkernel—which determines the input/output preference of the microtile and thus the natural method of performing SIMD reads and write instructions on C.

While the 1M method was originally articulated as a way to implement complex domain matrix multiplication on operands of identical datatypes, we will soon show that not only can it be extended to mixed-precision complex domain computation, but it also indirectly supports a particular combination of mixed-domain operands.

Case	C	A	B	Description/Approach
0	$\mathbb{R}$	$\mathbb{R}$	$\mathbb{R}$	Supported by the original framework. Performs $2mnk$ flops.
1B	$\mathbb{R}$	$\mathbb{R}$	$\mathbb{C}$	Interpreted as $C := A\mathcal{R}e(B) + C$. Ignore $\mathcal{I}m(B)$ and compute as Case 0. Performs $2mnk$ flops.
1A	$\mathbb{R}$	$\mathbb{C}$	$\mathbb{R}$	Interpreted as $C := \mathcal{R}e(A)B + C$. Ignore $\mathcal{I}m(A)$ and compute as Case 0. Performs $2mnk$ flops.
2AB	$\mathbb{R}$	$\mathbb{C}$	$\mathbb{C}$	Interpreted as $C := \mathcal{R}e(AB) + C$. Use 1R packing format from 1M method to compute only $\mathcal{R}e(AB)$. Performs $4mnk$ flops.
1C	$\mathbb{C}$	$\mathbb{R}$	$\mathbb{R}$	Compute AB and accumulate into $\mathcal{R}e(C)$. Performs $2mnk$ flops.
2BC	$\mathbb{C}$	$\mathbb{R}$	$\mathbb{C}$	Compute as if $A \in \mathbb{C}$, but avoid all computations with $\mathcal{I}m(A)$. Represent C and B with real and imaginary elements indistinguishable within a $m \times 2n$ and $k \times 2n$ real matrices, respectively. Requires a row-preferential microkernel. Performs $4mnk$ flops.
2AC	$\mathbb{C}$	$\mathbb{C}$	$\mathbb{R}$	Compute as if $B \in \mathbb{C}$, but avoid all computations with $\mathcal{I}m(B)$. Represent C and A with real and imaginary elements indistinguishable within $2m \times n$ and $2m \times k$ real matrices, respectively. Requires a column-preferential microkernel. Performs $4mnk$ flops.
3	$\mathbb{C}$	$\mathbb{C}$	$\mathbb{C}$	Supported by the original framework (via the the 1M method and/or assembly-coded complex microkernels). Performs $8mnk$ flops.

Fig. 5. A table summarizing the eight possible cases of mixed-domain computation within the GEMM operation. The first column identifies a name for each case, with the number identifying the number of complex matrix operands and the letters identifying the matrices that are complex. The second, third, and fourth columns explicitly identify the domains of each matrix operand. The last column describes the interpretation of each case within BLIS along a brief comment on how the case is implemented (where applicable) and a (minimum) flop count when implemented optimally.

4.2 Enumerating the Cases

Consider for simplicity $C := AB + C$, ignoring the scaling factors α and β. Each of $\{A, B, C\}$ can be stored in either the real or complex domains, leading to $2^3 = 8$ different combinations. Figure 5 enumerates and names each possible case. We will now discuss each case, how it is interpreted, and how it is implemented within the BLIS framework.

4.2.1 Cases 0, 3. The trivial case where all matrices are stored in the real domain, which we refer to as Case 0, is already supported by the framework via algorithms based on real domain microkernels. Similarly, Case 3, which applies when all matrices are stored in the complex domain, is also already supported. Support for Case 3 is provided in BLIS via conventional algorithms based on complex domain microkernels as well as via the 1M method, which is particularly useful when complex microkernels are not available. Cases 0 and 3 incur $2mnk$ and $8mnk$ flops, respectively.

4.2.2 Cases 1A, 1B. Case 1A captures situations where C and B are real while A is complex. We interpret such an operation as $C := \mathcal{R}e(A)B + C$. Implementing this case in BLIS is rather straightforward: We ignore the imaginary part of A and compute as if all matrices were real. Because BLIS tracks both row and column strides for each matrix operand, ignoring the imaginary

elements amounts to a temporary change to the dimension and stride metadata contained within the object representing A. Case 1B involves a complex matrix B and real matrices C and A, but is otherwise handled similarly. Since these case are ultimately implemented in terms of Case 0, they both performs $2mnk$ flops.

4.2.3 Case 2AB. Case 2AB is applicable when A and B are complex while the matrix to which they accumulate, C, is real. We interpret this somewhat curious scenario as a matrix product that takes place in the complex domain, but one for which the imaginary result is discarded: $C := \mathcal{R}e(AB) + C$. Since $\mathcal{I}m(AB)$ is not needed, only $4mnk$ flops need to be performed. Thus, this case provides an opportunity for computational savings when properly implemented. BLIS implements 2AB by borrowing the 1R packing format used by the 1M method.[12] Specifically, BLIS packs *both* matrices A and B using the 1R format while simultaneously conjugating B. This has the effect of allowing a subsequent real matrix multiplication over the packed matrices to correctly compute only the real half of the complex matrix multiplication update.

4.2.4 Case 1C. The opposite of 2AB—Case 1C—refers to settings in which matrix C is complex while A and B are real. Since the matrix product takes place entirely in the real domain, the natural interpretation is that AB updates only $\mathcal{R}e(C)$, and $\mathcal{I}m(C)$ is left untouched. Generally speaking, BLIS implements 1C using a strategy similar to the one used with 1A and 1B. That is, a temporary change to the object metadata describing matrix C allows us to isolate $\mathcal{R}e(C)$, which once again reduces the problem to Case 0. Accordingly, this case requires only $2mnk$ flops.

4.2.5 Cases 2AC, 2BC. Consider Case 2AC, in which matrices C and A are complex and matrix B is real. We interpret this situation as performing a complex matrix product AB to update both real and imaginary parts of C. However, all computation involving the imaginary part of B, which is implicitly zero, may be ignored. This means that the computation requires only $4mnk$ flops. Now, if C and A were guaranteed to be column-stored, BLIS could handle this case with a simple change of metadata that recasts those complex matrices as real, with the real and imaginary elements treated equally and indistinguishably. However, BLIS also allows row storage (and general storage) for all matrices, and thus the solution is not quite so simple. Instead, BLIS handles 2AC as follows.

If the original problem fits into Case 2AC and the microkernel is column-preferential, then A is packed as a real matrix (with imaginary elements stored traditionally, in element-wise interleaved fashion) and B is packed normally. A real domain macrokernel (Case 0) is then executed, which will properly update C. However, if the microkernel is row-preferential, the operation is logically transposed and Case 2BC is executed instead (whereby matrices C and B are complex and A is real). Thus, the effective case employed, 2AC or 2BC, depends on the register orientation of the microkernel, *not* the storage of C, and does so for the same reason that the 1M method, discussed previously in Section 4.1, depends on the same property.

After the aforementioned logical transposition is applied (or not), the microkernel input/output preference may differ from the storage of matrix C. If this is the case, then BLIS calls a *virtual* microkernel,[13] instead of calling the microkernel directly, allowing for logic that will use a very small amount of temporary storage—equivalent to one microtile—to facilitate the proper use of

[12] This is the indirect support alluded to at the conclusion of Section 4.1.

[13] In BLIS, virtual microkernels share the same type signature of conventional ("native") microkernels and ultimately compute the same operation. The only difference is that virtual microkernels typically implement the microkernel operation in the form of additional logic before (and sometimes after) the call to the native microkernel. Sometimes, as with the 1M method, the native microkernel being called is the real-domain equivalent relative to the virtual microkernel's type signature (e.g., a virtual `zgemm()` microkernel implemented in terms of the native `dgemm()` microkernel). Thus, the term is somewhat general-purpose and additional context is needed to identify its specific nature.

the microkernel (whether it be row- or column-preferential) and then copy and/or accumulate the temporary microtile result back to the appropriate location within the output matrix C.

4.3 Computation Domain

Unlike the computation precision, which is discussed in the next section, the computation domain is implied according to the case-specific interpretations covered in Section 4.2 and summarized in Figure 5. Alternate interpretations exist, however. For example, consider Case 2AB, which handles situations where $C \in \mathbb{R}$ and $A, B \in \mathbb{C}$. Our mixed-datatype GEMM implementation currently interprets this case as $C := \mathcal{R}e(AB) + C$. That is, AB is computed in the complex domain, but then only the real part is accumulated into C. Alternatively, the computation of Case 2AB could be interpreted as taking place entirely in the real domain, which would result in $\mathcal{I}m(A)$ and $\mathcal{I}m(B)$ being ignored before computation even began. Similar interpretations—all of which would change the update to C—could be applied to Cases 2AC, 2BC, or even 3. However, we do not immediately see significant utility in exposing these cases.[14] Thus, our implementation in BLIS does not presently allow the caller to explicitly specify the computation domain.

5 SUPPORTING MIXED PRECISION COMPUTATION

Now that variation among the storage domains has been fully explored, we turn our attention to the storage precision of the matrices A, B, and C. Once again, we set aside the scalars α and β, focusing only on variation among matrix operands. We also limit the initial discussion to variation within the set of precisions that includes only single and double precision.

5.1 Supporting All Cases

Each of $\{A, B, C\}$ can be stored in single or double precision. Furthermore, we define a separate *computation* precision to identify the precision in which the matrix product takes place. Combining the storage precision of each matrix with the computation precision, we find that there exist $2^4 = 16$ different cases.

At first glance, it may seem worthwhile to enumerate all mixed-precision cases as we did for mixed-domain computation in Section 4. However, there is a more concise and systematic way of describing how to support all cases, one that happens to coincide closely with how mixed-precision support was ultimately implemented in BLIS.

While not part of the runtime logic for implementing mixed-precision computation, we must first modify the packing facility so that the source and destination precisions may differ. For example, we must be able to pack from a single-precision real matrix to a double-precision real matrix, or vice versa. Once the mixed-precision functionality is in place for the packing operation, the runtime logic may be encoded in three broad steps, as follows.

5.1.1 Identify the Computation Precision. First, we must identify the computation precision. In BLIS, we provide a field for the computation precision within the metadata of any matrix object. However, semantically, we deem only the field within the object for matrix C to be relevant. Thus, upon calling the GEMM operation, we query the computation precision from C. Note that in BLIS, when objects are created, the computation precision is initialized to be equal to the object's storage precision. Consequently, if it is not explicitly set by the caller prior to invoking GEMM, then the computation precision will automatically default to the storage precision of C.

[14] An intrepid user who wished to access such functionality could easily implement it within BLIS. Of course, if users show interest in this functionality, we will reconsider official support within the framework.

5.1.2 Construct the Target Datatypes for A and B. Next, we embed *target* datatypes within the metadata for objects representing matrices A and B. BLIS defines the target datatype for A and B as being the storage datatype (domain and precision) of the matrix during its packed state—in other words, the datatype to which the matrix must be typecast before it can be computed upon.[15] The target datatype for matrix A is constructed by combining the storage domain of A with the operation's computation precision, with a similar process resulting in the target datatype for matrix B. The target datatype for each matrix is then embedded within the metadata of the corresponding object.

5.1.3 Determine Whether Typecasting is Needed on Accumulation. If the computation precision differs from the storage precision of C, then the intermediate result from computing the product AB must be typecast before it is accumulated into C. This may be implemented outside the microkernel by implementing additional macrokernels that write the microkernel result to a temporary microtile (allocated on the function stack) before typecasting and accumulating the temporary values back to C. Alternatively, this logic may be hidden within a virtual microkernel. BLIS opts for the former solution, which somewhat reduces function call overhead at the cost of a somewhat higher object (binary) code footprint.

If the computation precision is identical to the storage precision of C, then AB does not require any typecasting before being accumulated. This corresponds to use of the traditional macrokernel.

5.2 Using the 1m Method for Case 3

As alluded to in the closing of Section 4.1, the 1m method can be extended to encompass all combinations of mixed-precision operands—that is, all precision combinations that fall within mixed-domain Case 3. This amounts primarily to (1) adding the ability to pack to the 1e and 1r[16] formats when the target datatype differs from the storage datatype and (2) adding the ability to scale by α (when $\alpha^i \neq 0$) during mixed-precision packing of A and B. With these changes in place, the 1m virtual microkernel may be used as-is, since its functioning is undisturbed by the typecasting logic encoded in the additional macrokernels mentioned previously in Section 5.1.3.[17]

5.3 Summary

By addressing the three areas described in Sections 5.1.1 through 5.1.3, and adding support for typecasting within the packing function, we can handle all 16 cases of mixing single and double precisions. Similarly, relatively minor changes to BLIS's implementation of the 1m method enable all mixed-precision instances of Case 3 to be handled even if conventional, assembly coded complex microkernels are unavailable.

Interestingly, the hypothetical impact of adding support for an additional floating-point precision, such as half-precision, would manifest almost exclusively in the form of additional support to the packing function.[18] The mixed-precision runtime logic, described above, would then trivially extend to support the two additional datatypes (one each for real and complex domains).

[15] Note that we do not need to track the target datatype for C, since the storage datatype of C does not change in the course of the mixed-datatype gemm operation.

[16] The ability to typecast while packing to the 1r format is also required by mixed-precision instances of Case 2ab.

[17] In the course of our work, the BLIS testsuite was updated to allow testing of its 1m method implementation with mixed-precision operands.

[18] A real domain microkernel that performs computations in the new precision would also be needed.

6 OPTIMIZATIONS

After retrofitting the mixed-domain/mixed-precision functionality into the GEMM implementation, it is possible to apply various optimizations to certain cases. Specifically, one may: (1) avoid virtual microkernel overhead by implementing one microkernel each for row- and column IO (or one that does both); (2) avoid repeated non-contiguous IO on C; (3) avoid repeated typecasting (and round-off error) on C; and (4) in the absence of (1), avoid virtual microkernel overhead with a workspace matrix. All of these optimizations are optional, and optimizations (2) through (4), if desired, require the use of workspace.

In this section, we briefly explore these optimizations in some detail.

6.1 Avoid Virtual Microkernel Overhead with Two Microkernels

If and when BLIS adopts a regime whereby each hardware architecture is supported simultaneously by *two* microkernels per datatype, one with a row preference and one with a column preference, then the virtual microkernel logic becomes unnecessary, and both 2AC and 2BC may be fully implemented without additional overhead.

6.2 Avoid Non-contiguous/non-SIMD Access on C

Some mixed-domain cases—at least as described in Section 4.2—result in accessing complex matrix C by individual real and imaginary elements. The best example of this is Case 1C, in which $\mathcal{R}e(C)$ is updated by the product of real matrices A and B. However, to isolate $\mathcal{R}e(C)$, the metadata describing matrix C must be tweaked in such a way that the matrix then has non-unit stride in both dimensions. While BLIS microkernels may update such matrices, doing so on current hardware comes with an unavoidable performance penalty that does not manifest when accessing contiguous elements via SIMD load and store instructions. Thus, it may be advantageous to internally allocate a temporary $m \times n$ matrix C_{temp} in which to compute AB, after which the result is copied back to C. As long as C_{temp} is created (a) in the real domain and (b) as either row- or column-stored, the microkernel will be able to update C_{temp} efficiently with SIMD instructions.

This optimization tends to be most worthwhile when $k > k_C$, as it would imply that the computation of AB unfolds as multiple rank-k_C updates of C (that is, multiple iterations of the fourth loop around the microkernel), with the non-contiguous load/store penalty otherwise being incurred for each update.

6.3 Reduce Typecasting Costs and Round-off Error Accumulation

When the storage precision of C differs from the computation precision, the GEMM implementation executes typecasting instructions emitted by the compiler to properly convert from one floating-point datatype to another—single- to double-precision or double- to single-precision. These instructions can be costly, especially when $k > k_C$, since each element of C is typecast once per rank-k_C update. In addition to the performance cost of these typecasting instructions, they can also incur a numerical cost. Specifically, when the storage precision of C is lower than the computation precision, repeated round-off error can occur during accumulation of the intermediate matrix product, which is once again exacerbated for large values of k. But as with the repeated cost incurred from non-contiguous access described in Section 6.2, these costs can be avoided by allocating a temporary $m \times n$ matrix C_{temp}. The key difference is that, in this case, C_{temp} is created with its storage precision equal to the computation precision, which will avoid intermediate typecasting (and thus reduce round-off error), leaving only typecasts on input ($C_{temp} := C$) and on output ($C := C_{temp}$). The total number of typecasts needed may be further reduced to those on

output provided that C_{temp} is initialized to zero and the final AB product be accumulated, rather than copied, back to C.

6.4 Avoid Virtual Microkernel Overhead with C_{temp}

Notice that Cases 2AC and 2BC, as described in Section 4.2, may require use of a virtual microkernel if a row-preferential microkernel (needed by 2BC) must be used on a column-stored (or general stored) matrix, or if a column-preferential microkernel (needed by 2AC) must be used on a row-stored (or general stored) matrix. The overhead of the virtual microkernel, while small, may still be noticeable and is incurred for each rank-k_C update. The dual-microkernel strategy described in Section 6.1 solves this issue. However, if only a single microkernel (per datatype) is available, then a temporary $m \times n$ matrix C_{temp} may be allocated, with the important distinction being that C_{temp} is created with storage (by columns or rows) to match the preference of the available microkernel. This allows the implementation to avoid the virtual microkernel altogether during intermediate accumulation into C_{temp}.

6.5 Summary

The optimizations described above are optional. At the time of this writing, BLIS implements all except the dual-microkernel strategy described in Section 6.1. BLIS also allows the user to optionally disable all uses of C_{temp} at configure-time, which avoids the extra workspace that would otherwise be needed by the optimizations discussed in Sections 6.2 through 6.4. The consequences of enabling/disabling C_{temp} can be seen in the performance graphs shown in Section 8.2.

7 HANDLING SCALARS

Before concluding our discussion of how to implement and support mixed-datatype GEMM, we turn our attention to scalars α and β, which have been omitted from our discussion thus far.

7.1 Mixed Precision

If the precision of α differs from the computation precision, then a decision must be made as to how to proceed. Numerous possible policies exist for handling such situations. Three examples follow:

(1) Typecast α to match the computation precision.
(2) Typecast the computation precision to match that of α.
(3) Unconditionally promote the lower precision value to the precision of the other (higher precision) value.

A similar decision must be made for handling the precision of β. The choices here are even more numerous, because while we consider the precision of β and the storage precision of C to be the main inputs to the runtime logic, one could also argue for considering the disagreement with the computation precision that governs the computation of AB. A few possible policies are:

(1) Typecast β to match the storage precision of C.
(2) Perform the scaling βC in the higher precision of the two values, typecasting back to the storage precision of C, as necessary.
(3) Typecast both β and C to the computation precision so that all suboperations within $\beta C + AB$ can occur in the same precision before being typecast back to the storage precision of C.

For both α and β, our mixed-datatype extension to BLIS implements policy option (1), since by specifying the computation precision the user already establishes the precision in which they wish to compute.

7.2 Mixed Domain

Real values of β are easily handled for all eight cases enumerated in Section 4.2.

For Cases 0, 1A, 1B, and 1C, complex β may be projected into the real domain (which discards $\mathcal{I}m(\beta)$ entirely), since, with $C \in \mathbb{R}$, $\mathcal{I}m(\beta)$ cannot change the final result. Similarly, complex values of β are handled as expected in the four cases where $C \in \mathbb{C}$. Specifically, Case 3 already handles complex β while Cases 2AB, 2AC, and 2BC support $\mathcal{I}m(\beta) \neq 0$ via extra logic in the virtual microkernel.

Real and complex values of α are already handled in Cases 0 and 3, respectively. Real α are also already handled by Case 3, since $\mathbb{R} \subset \mathbb{C}$. In Case 0, a complex α can be projected to the real domain, since $\mathcal{I}m(\alpha)$ would not change the computation, even if we had storage in which to save the final result.

For Cases 1A, 1B, 1C, 2AB , 2AC, and 2BC, a non-zero imaginary component in α could presumably change the final computation under a literal interpretation of the computation—that is, one in which all five operands' domains are taken at face value. However, implementing this logic is non-trival. For example, consider our approach to handling Case 1A, as discussed in Section 4.2. In this case, we perform the computation according to Case 0, as if the imaginary components of A were zero. That case's handling is completely consistent with its mathematics, since in that scenario A is the only operand with non-zero imaginary values, and thus they would have no impact on the final result. However, if both A *and* α are complex, then the imaginary components could combine to change the real component of the scalar-matrix product αA. Adjusting for this new possible use case would require a different approach in the implementation, perhaps using temporary workspace to store a copy of A while it is scaled by α, after which the imaginary components may be ignored (assuming they were even computed to begin with).[19] However, while it is clear that going through such motions would maintain deeper fidelity to the literal mathematics expressed in the mixed-domain scenario, it's not clear to us that this additional functionality would be vital for most applications. As we continue to solicit feedback from the community, we will pay close attention to whether users expect or request support for non-zero imaginary values of α in Cases 1A, 1B, 1C, 2AB , 2AC, and 2BC. For now, our mixed-datatype solution supports only real values of α for those six cases, and prints an error message if the scalar is given with a non-zero imaginary component.

8 PERFORMANCE

In this section, we discuss performance results for our mixed-datatype implementations on two servers with modern hardware architectures.

8.1 Platform and Implementation Details

8.1.1 Marvell ThunderX2. The first system upon which we measured performance is a single compute node consisting of two 28-core Marvell ThunderX2 CN9975 processors.[20] Each core,

[19] Alternatively (and more preferably), logic that packs $\mathcal{R}e(\alpha A)$ where $\alpha, A \in \mathbb{C}$ may be encoded within a special packing function.

[20] While four-way symmetric multithreading (SMT) was available on this hardware, the feature was disabled at boot-time so that the operating system detects only one logical core per physical core and schedules threads accordingly.

running at a clock rate of 2.2 GHz, provides a single-core peak performance of 17.6 gigaflops (GFLOPS) in double precision and 35.2 GFLOPS in single precision. Each of the two sockets has a 32 MB L3 cache that is shared among its local cores, and each core has a private 256 KB L2 cache and 32 KB L1 (data) cache. The installed operating system was Ubuntu 16.04 running the Linux 4.15.0 kernel. Source code was compiled by the GNU C compiler (`gcc`) version 7.3.0.[21]

8.1.2 Intel Xeon Platinum. The second system is a single node consisting of two 26-core Intel Xeon Platinum 8167M processors.[22] Each core ran at a clock rate of 2.0 GHz, providing single-core peak performance of 64 gigaflops (GFLOPS) in double precision and 128 GFLOPS in single precision. Each of the two sockets has a 35.75 MB L3 cache that is shared among its local cores, and each core has a private 1 MB L2 cache and 32 KB L1 (data) cache. The installed operating system was Ubuntu 18.04.1 running the Linux 4.15.0 kernel. Source code was compiled by the GNU C compiler (`gcc`) version 7.3.0.[23]

8.1.3 Implementations. On both the ThunderX2 and the Xeon Platinum, the version of BLIS used was based on an inter-version release that preceded 0.5.1.[24] In both cases, BLIS was configured with OpenMP-based multithreading. Architecture-specific configuration, which determines settings such as kernel sets and cache blocksizes, was performed automatically via the `auto` target to the `configure` script.

We showcase two (or, in some cases, three) implementations, with a third (or fourth) provided for reference:

- **Internal without extra memory.** This refers to the BLIS implementation described in Sections 4 and 5 in which all logic for supporting the mixing of domains and precisions occurs opaquely inside of BLIS. This implementation does not, however, employ the use of a temporary matrix C_{temp} discussed in Sections 6.2–6.4.
- **Internal with extra memory.** This implementation is identical to the previous implementation, except that C_{temp} is made available, thus incurring extra workspace requirements in certain situations. It is worth pointing out that this implementation does not differ from its extra-memory-avoiding counterpart for all 128 mixed-datatype cases. Thus, we will only present these results when one of the conditions laid out in Sections 6.2–6.4 is applicable. Also, note that we do not employ any cache blocking or parallelism outside of the underlying call to GEMM, such as when copying/accumulating C_{temp} to/from C.
- **Ad hoc.** This refers to an implementation that is formulated *outside* of BLIS, using temporary workspace and matrix copies wherever needed. The purpose behind such an implementation is to show the best a knowledgeable computational scientist could reasonably expect to achieve using only a BLAS library. Thus, while we link this solution to BLIS, we do so via the framework's BLAS compatibility layer. Furthermore, we do not employ any cache blocking or parallelism outside of the underlying call to GEMM.

[21]The following optimization flags were used during compilation on the ThunderX2: `-O3 -ftree-vectorize -mtune=cortex-a57`. In addition to those flags, the following flags were also used when compiling assembly kernels: `-march=armv8-a+fp+simd -mcpu=cortex-a57`.

[22]Two-way SMT (which Intel refers to as "Hyperthreading") was available. However, we employed processor affinity settings that limited the operating system to utilizing only one logical core per physical core.

[23]The following optimization flags were used during compilation on the Xeon Platinum: `-O3`. In addition to those flags, the following flags were also used when compiling assembly kernels: `-mavx512f -mavx512dq -mavx512bw -mavx512vl -mfpmath=sse -march=skylake-avx512`.

[24]This version of BLIS may be uniquely identified, with high probability, by the first 10 digits of its `git` "commit" (SHA1 hash) number: `cbdb0566bf`.

- **Reference.** This refers to the `sgemm()`, `dgemm()`, `cgemm()`, and `zgemm()` provided by BLIS, whichever is appropriate, as determined by the mixed-datatype case presented by each graph. These Reference curves are provided as a visual "high-water mark" to show how much performance is ceded by the Internal and Ad hoc mixed-datatype implementations. As of this writing, we had not yet developed native complex domain GEMM microkernels for the ThunderX2 or Xeon Platinum microarchitectures, and therefore the `cgemm()` and `zgemm()` curves represent BLIS implementations based on the 1M method.

8.2 Results

8.2.1 Scope and Conventions. Performance data for sequential, multithreaded within one socket (28 threads), and multithreaded across two sockets (56 threads) was gathered on the ThunderX2. Similarly, sequential, single-socket (26 threads), and dual-socket (52 threads) data was gathered on the Xeon Platinum. This produced a rather large set of data, which we formatted into 768 individual graphs. As a practical matter, we have relegated this complete set of performance results to Online Appendix 10.3. Here, in the main body of the article, we limit our presentation to a slice of data that we feel is broadly representative of the full set. Readers interested in scalability may wish to skip this smaller sampling and focus on the full data set in Online Appendix 10.3.

For any given graph, the x-axis denotes the problem size (where $m = n = k$), the y-axis shows observed floating-point performance in units of GFLOPS per core, and the theoretical peak performance of the hardware coincides with the top of the graph. Problem sizes for sequential instances of GEMM were run from 40 to 2,000 in increments of 40 while multithreaded executions were run from 120 to 6,000 in increments of 120.[25] The data points in all performance graphs report the best of three trials.

Individual graphs are labeled according to the mixed-datatype case of its Internal and Ad hoc implementations. The datatypes are encoded as *cabx*, where the characters *c*, *a*, and *b* encode the storage datatypes of C, A, and B, respectively, while x encodes the computation precision. For example, a case labeled "zcsd" would refer to mixed-domain Case 2AC, where matrices A and B are stored in single-precision (complex and real, respectively), matrix C is double-precision complex, and the computation occurs in double-precision arithmetic.

All experiments reflect the use of randomized, column-stored matrices with GEMM scalars $\alpha = 1$ and $\beta = 1$.

8.2.2 Exposition. Figure 6 (top) reports sequential performance on the Marvell ThunderX2. The six graphs on the left half of Figure 6 (top) report performance for six select mixed-datatype cases that we felt are interesting: sdds, ddds, dsss, ccss, cscs, and csss. All of these mixed-datatype cases perform their computation in single-precision. On the right half, we display graphs that correspond to the precision-toggled analogues of the graphs on the left—dssd, sssd, sddd, zzdd, zdzd, and zddd—all of which perform their computation in double-precision. Both Internal implementations (with and without extra memory) are shown in four of the six graphs in each group, with the remaining two graphs displaying performance only for the implementation where extra memory is disabled, since the optimization is not applicable for those cases.

The legends are shown once for each group of six graphs. Within the legend, the curves labeled "Intern (+xm)" and "Intern (-xm)" refer to the Internal implementations with and without extra

[25] Some readers may wish that we had run our multithreaded experiments to a somewhat larger maximum problem sizes, perhaps 8,000 or 10,000. We sympathize with these readers. However, the results presented in this article, including Online Appendix 10.3, required a total of 302,400 invocations of GEMM performed over a period of several days. Limiting the maximum problem size was necessary so that the experiments would finish in a reasonable amount of time.

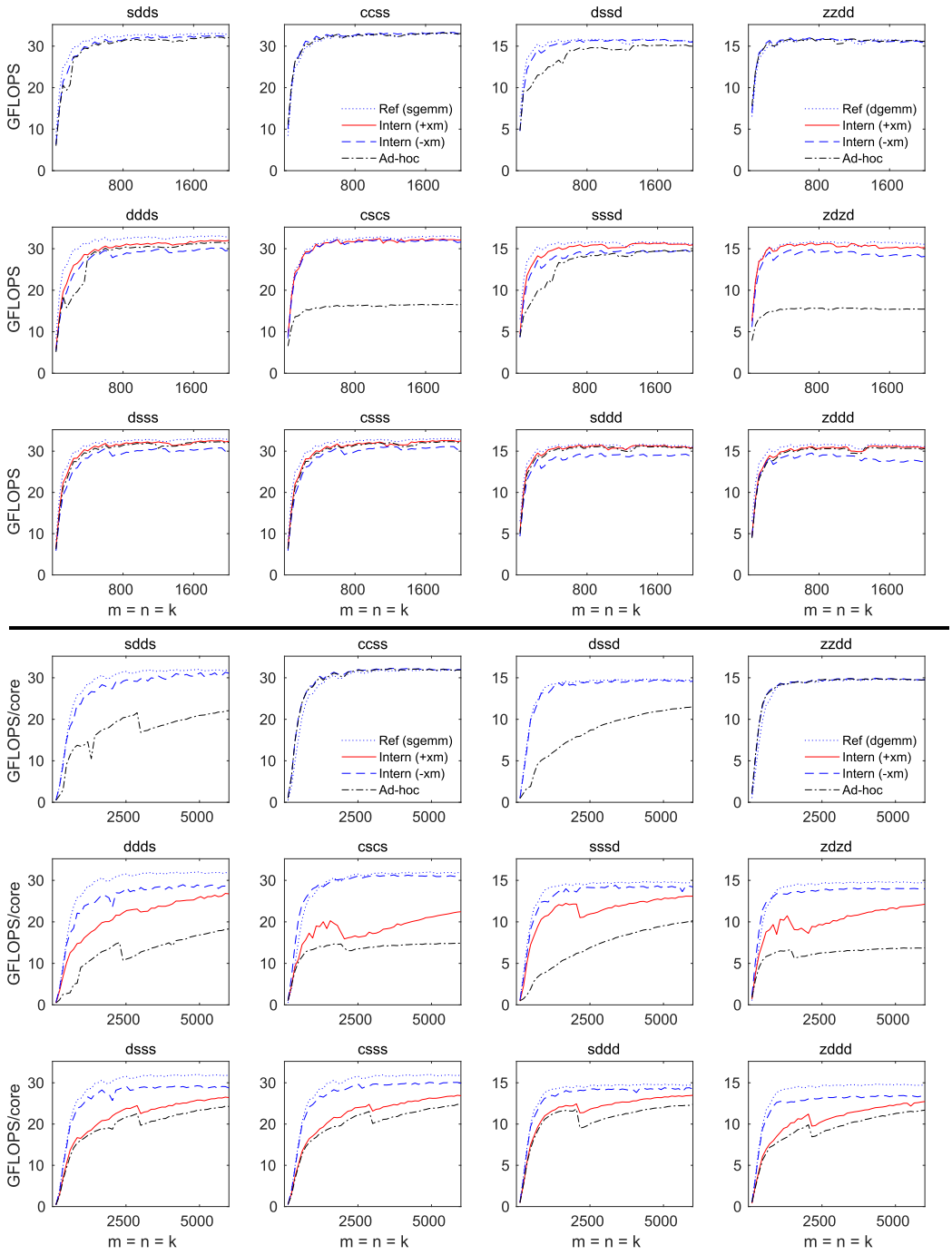

Fig. 6. Sequential (top) and multithreaded with 28 threads (bottom) performance of "Internal" and "Ad hoc" implementations of GEMM for select datatype combinations on a Marvell ThunderX2 CN9975 processor. 0 The 12 graphs on the left side and right sides report computation in single- and double-precision, respectively. The theoretical peak performance coincides with the top of each graph.

memory, respectively. Additionally, the label for the Reference implementation is augmented, in parentheses, with the conventional GEMM routine that serves as the reference curve within that group of six graphs.

Organized identically as those in the top, the graphs in Figure 6 (bottom) report multithreaded performance on one socket (28 threads) of the ThunderX2.

Finally, Figure 7 (top) and (bottom) report sequential and single-socket (26 threads) performance, respectively, on the Intel Xeon Platinum using the same mixed-datatype cases and organization as shown in Figure 6.

8.2.3 Observations and Analysis. Let us turn first to the sequential performance results from the ThunderX2.

The first thing we notice is that, in five of the six mixed-datatype cases, the Ad hoc approach is capable of performing quite well relative to the Internal implementations. The sixth case, which falls within mixed-domain Case 2BC, suffers, because unlike with 2AC, we are unable to cast the problem in terms of `sgemm()` or `dgemm()` by manipulating matrix metadata, and therefore must resort to using `cgemm()` and `zgemm()`. And because $A^i = 0$, this approach necessarily wastes half of the floating-point computations.

Next, we notice that employing C_{temp} often affords a modest but noticeable increase in performance relative to forgoing extra memory. This is expected for all of the reasons described in Sections 6.2–6.4.

Turning to the multithreaded performance on ThunderX2, we notice that the effect of using extra memory in the Internal implementation is not only reversed but also magnified. Here, the additional costs incurred within the virtual microkernel are overwhelmed by the cost of the accumulation of C_{temp} back to C that must be performed after the GEMM operation, which is not parallelized.[26] The performance of the Ad hoc implementation also suffers, for similar reasons to that of the Internal implementation using C_{temp}. The effect is even worse for Ad hoc, however, because that implementation must make whole copies of matrices up-front, and does so sequentially, before executing the underlying GEMM operation. By contrast, the Internal implementations benefit from typecasting A and B during packing, which is already parallelized.

Overall, the extra-memory-avoiding Internal implementation performs quite well relative to its `sgemm()` and `dgemm()` benchmarks.

Turning to the Intel Xeon Platinum results in Figure 7, we find the data largely tells the same story. Here, the multithreaded performance degradation caused by employing extra memory is even more severe, and the Ad hoc performance is similarly attenuated. Once again, in both sequential and multithreaded cases, one of the Internal implementations matches or exceeds (sometimes by a large margin) that of the Ad hoc approach. However, for some datatype cases, even the memory-avoiding Internal implementation lags noticeably behind its `sgemm()` or `dgemm()` benchmark. The cause of this is not immediately clear, but may be related to memory bandwidth becoming strained in cases where the the virtual microkernel is relied upon to update the output matrix in two steps, using a temporary microtile as intermediate storage.

These results strongly suggest that, in general, BLIS should employ the use of Internal implementation with C_{temp} for sequential invocations of GEMM, but avoid C_{temp} in the case of many-threaded execution. As a function of the number of threads, the crossover point between the two Internal implementations will likely depend on the amount of parallelism that can be extracted

[26] This copy/accumulation operation, while not parallelized for any of the implementations tested for this article, could in principle be parallelized. However, speedup for that component of the GEMM would likely be limited as the many threads quickly saturate the available memory bandwidth.

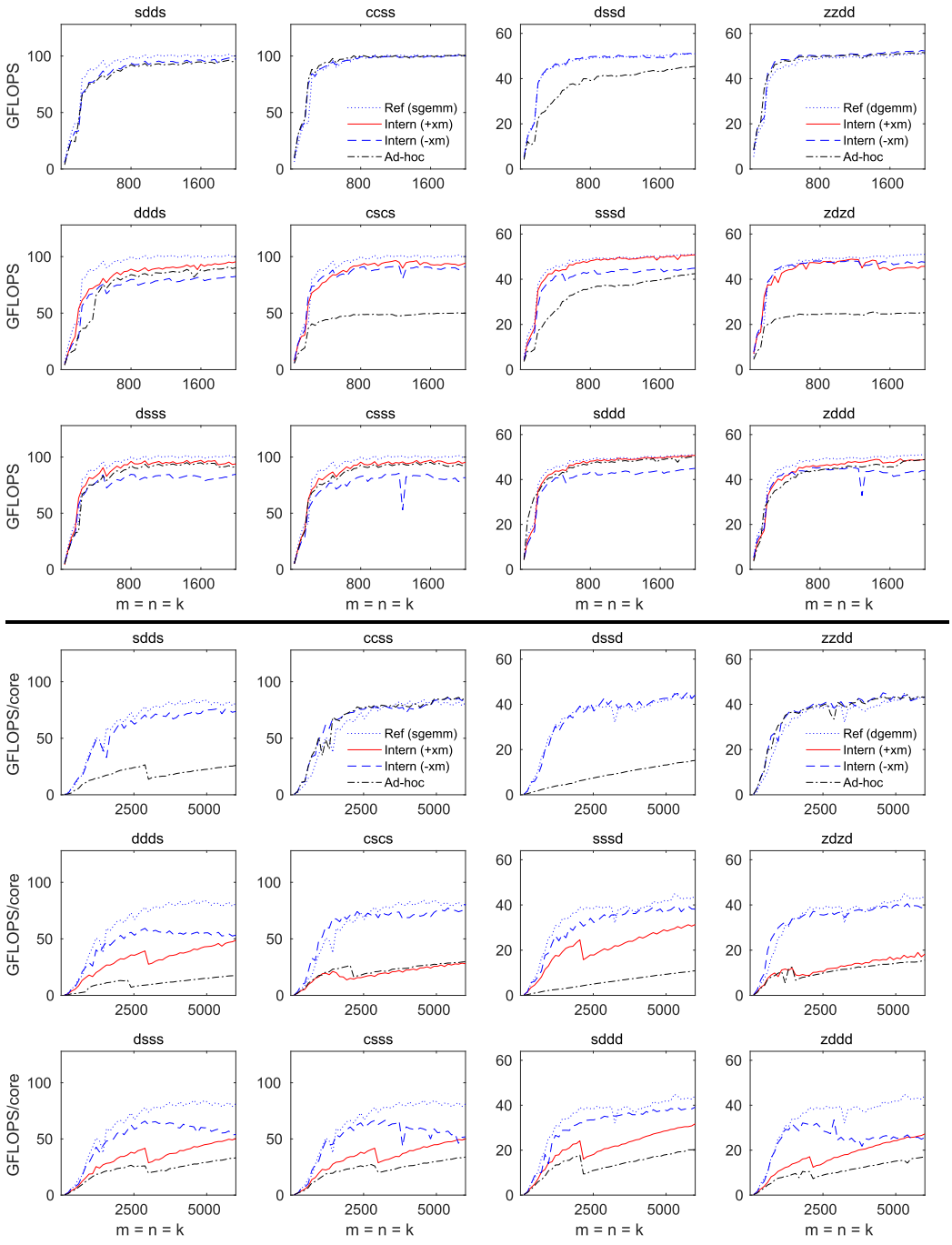

Fig. 7. Sequential (top) and multithreaded with 26 threads (bottom) performance of "Internal" and "Ad hoc" implementations of GEMM for select datatype combinations on a Intel Xeon Platinum 8167M processor. 1c2ac 2bc The 12 graphs on the left side and right sides report computation in single- and double-precision, respectively. The theoretical peak performance coincides with the top of each graph.

Framework source code	Total lines	Change	Total size (Kilobytes)	Change
Before	148,862		4,706	
After	154,962	+6,100	4,892	+186

Fig. 8. The total number of lines and the total size (in kilobytes) of source code in the BLIS framework (excluding the build system, kernels, and the testsuite) before (`git` commit `667d3929`) and after (`git` commit `5fec95b9`) support was added for mixed-datatype computation via the GEMM operation.

Testsuite source code	Total lines	Change	Total size (Kilobytes)	Change
Before	22,891		680	
After	24,356	+1,465	722	+42

Fig. 9. The total number of lines and the total size (in kilobytes) of source code in the BLIS testsuite before (`667d3929`) and after (`5fec95b9`) support was added for mixed-datatype computation via the GEMM operation.

within the accumulation of C_{temp} back to C before memory bandwidth is saturated. We leave this topic for future exploration.

9 MEASURING THE IMPACT ON CODE SIZE

Prior to reading the article, a casual reader might have been skeptical of the practicality of our solution. However, Sections 4 and 5 decompose the problem into mostly orthogonal use cases, giving hope that the ultimate impact on library code size is much more manageable.

Indeed, by multiple measures, the BLIS library grew only modestly after introducing mixed-datatype support.

The second column in Figure 8 shows the total number of lines[27] of code present in the BLIS framework proper—which excludes other components such as the build system, kernels, and the testsuite—before and after mixed-datatype support was added to the GEMM operation.[28] The fourth column shows the total size in kilobytes of the source code. The change between the "before" and "after" values for total lines and total size are shown in the third and fifth columns, respectively. Mixed-datatype support for the GEMM operation adds approximately 4% each to the total number of lines and total bytes of source code.

Figure 9 lists similar metrics for the BLIS testsuite, which is capable of testing the vast majority of BLIS's computational operations. Here, the support for testing all combinations of mixed-datatype execution, with any combination of matrix storage storage or transposition and/or combinations, increases the source code footprint by approximately 6% in both lines and total size.

Finally, Figure 10 shows the object (binary) code size for three build products: BLIS built as a static library; BLIS built as a shared library; and the BLIS testsuite linked against the static

[27] In this section, we uniformly report *total* lines of code, including blank lines and comments.

[28] The "before" and "after" snapshots of BLIS are uniquely identified with high probability by the first eight digits of the `git` commit (SHA1 hash) numbers. Commit `667d3929` identifies the code just before mixed-datatype support was added, while `5fec95b9` identifies the first commit in which mixed-datatype support is present. This latter commit includes virtually all changes discussed in this article with the exception of the mixed-precision support for the 1M method, which was added in a later commit (`375eb30b`).

Object code size (Kilobytes)	Static library	Change	Shared library	Change	Statically-linked testsuite	Change
Before	3,141		2,286		1,632	
After (disabled)	3,253	+112	2,382	+94	1,720	+88
After (enabled)	3,366	+225	2,486	+200	1,820	+188

Fig. 10. The total object code size for three build products: BLIS built as a static library; BLIS built as a shared library; and the BLIS testsuite linked against the aforementioned static library. Object code sizes are given using the: BLIS library just prior to mixed-datatype support (`667d3929`); with mixed-datatype support present (`5fec95b9`) but disabled at configure-time; and with mixed-datatype support present (`5fec95b9`) and enabled at configure-time.

library.[29] This figure shows three rows of values: before mixed-datatype support for GEMM was added (`667d3929`); after mixed-datatype support was added (`5fec95b9`) where the feature was disabled at configure-time; and after mixed-datatype support was added (`5fec95b9`) where the feature was enabled at configure-time. (In the the latter two cases, the "Change" columns represent the change from the "before" state.) With mixed-datatype support present and enabled, the size of the static library increases by only 255 KB, or 8% of the original library size. In the case of the shared library, the increase is just under 9%. And a statically linked instance of the BLIS testsuite increases by about 11%.

Thus, no matter the metric, the increase in code footprint is quite modest relative to the scope of functionality added.

10 FINAL THOUGHTS

We conclude this article by sharing with the reader insights and observations that we have drawn to-date about BLIS, BLAS, and the adoption of new software functionality by the broader HPC community.

10.1 Case Studies

In late 1990's, various community participants convened multiple meetings of the BLAS Technical (BLAST) Forum to discuss extensions to the original BLAS [3]. Some of these extensions targeted extended and mixed-precision functionality and were eventually implemented and branded as XBLAS [18, 29]. In the end, the mixed-precision extensions were not widely adopted. We speculate that this was in part due to the fact that the reference implementation falls short of achieving high performance. By contrast, when the reference codes of the original BLAS [2, 8] were introduced, compilers could easily translate them to binary implementations that would yield high performance on the vector supercomputers that were prominent at the time. Though computer architectures have evolved since then, we suspect that the initial availability of high-performance reference BLAS implementations helped give rise to a network of institutional experts and expertise, laying the foundation for today's well-established constellation of BLAS solutions.

A classic example of the release of a new standard hand-in-hand with a high quality implementation was the Message-passing Interface (MPI) [12, 23]. From the start, an implementation that later become known as MPICH provided a high-performance reference implementation [5]. Another example is TBLIS, a library and framework for performing efficient contractions and related

[29] These object codes were built using GNU `gcc` 5.4.0 on an Intel Xeon E3-1271 v3 (Haswell) workstation.

operations on tensors [19]. TBLIS borrows some of the insights of BLIS and then goes further to construct a general-purpose tensor library from scratch. This new software architecture avoids the drawbacks of ad hoc, BLAS-based solutions that must first reorder tensors into column-major storage simply for the purpose of calling `dgemm()`. The result is a comprehensive set of tensor functionality that exports flexible APIs (in C89 and C++11) while also facilitating higher performance for both sequential and multithreaded applications. In contrast to the BLAST extensions, we feel that MPICH and TBLIS serve as important examples of software projects that each made an impact in their community by coupling new APIs and functionality with a complete high-performance reference implementation.

10.2 Managing Complexity

Supporting the goal of providing a high-performance reference implementation with new APIs, while a laudable step in the right direction, is ultimately insufficient if the software is not designed to be practically implementable and maintainable. Suppose we attempted to solve the problem of mixed-datatype GEMM by implementing a solution in the style of the original reference BLAS. Such an approach tends to lead to implementations that are duplicative and vertically siloed from one another, with support for each datatype, storage case, and transposition/conjugation scenario resulting in a fully independent block of code. On top of supporting four (potentially differing) storage datatypes across all three matrix operands, a fully independent computation precision, and a fourth "conjugation only" transposition parameter value, BLIS also supports three different storage formats (row, column, and general storage) for each matrix operand. This would lead to 31,104 separate implementations. If two additional precisions are supported—half-precision and quad-precision, for example—then this number grows to 497,664.

Our takeaway from this analysis, and our past experiences, is that achieving a complete high-performance reference implementation for mixed-datatype GEMM requires careful management of complexity in the implementation. Complexity must be managed not only within interfaces (e.g., via object-based APIs) but also internally by allowing feature "decision points" (e.g., typecasting during packing) to work together in sequence, rather than in duplication, to enable the desired combinatorial space of functionality while collapsing its corresponding axial dimensions within the source code.[30] Otherwise, the solution quickly becomes unwieldy, if not hopelessly intractable, for its maintainers.

10.3 Thesis

It is our conjecture that interfaces to new functionality, such as the mixed-datatype GEMM presented in this article, should be accompanied by a corresponding high-performance reference implementation, and that the key to making such a goal attainable is managing combinatorial software complexity. Aside from serving several obvious purposes—a research proof-of-concept, a reference for other developers, a working solution upon which end-users can rely—the reference implementation, once instantiated, confers another, less tangible benefit. Namely, it creates the initial conditions in which a community can form and a tangible product around which its members can organize, collaborate, and advance toward its shared objectives. Therefore, this approach ultimately benefits all stakeholders.

[30] While we feel the current work is significant, we admit that managing certain complexity in a BLAS-like library such as BLIS depends on whether the implementations (and its developers) are amenable to packing (copying) matrix data into internal, intermediate formats. Implementing GEMM or similar operations (even absent the goal of mixing datatypes) on unpacked operands introduces serious challenges that are not addressed in the current article.

SOFTWARE AVAILABILITY

The software referenced in this article may be found at the BLIS project page [4] on GitHub along with documentation, examples, links to discussion forums, and other related resources.

SUPPLEMENTARY MATERIALS

Supplementary materials are available in the online version of this article.

ACKNOWLEDGMENTS

We kindly thank Jeff R. Hammond and Devin A. Matthews for referring us to appropriate citations for various applications and use cases that stand to benefit from implementing mixed-domain and mixed-precision matrix multiplication functionality. We also acknowledge Matthews for participating in and facilitating early discussions on the topic of mixed-domain and mixed-precision computation. Also, we thank Marvell and Oracle for providing access to the Marvell ThunderX2 CN9975 and Intel Xeon Platinum 8167M servers, respectively, on which performance data for this article was gathered. Finally, we thank members of the Science of High-Performance Computing (SHPC) group for their contributions throughout the research toward and drafting of this article.

REFERENCES

[1] E. Apra, E. J. Bylaska, W. A. de Jong, N. Govind, K. Kowalski, T. P. Straatsma, M. Valiev, H. J. J. van Dam, D. Wang, T. L. Windus, J. Hammond, J. Autschbach, K. Bhaskaran-Nair, J. Brabec, K. Lopata, S. A. Fischer, S. Krishnamoorthy, M. Jacquelin, W. Ma, M. Klemm, O. Villa, Y. Chen, V. Anisimov, F. Aquino, S. Hirata, M. T. Hackler, V. Konjkov, D. Mejia-Rodriguez, T. Risthaus, M. Malagoli, A. Marenich, A. Otero de-la Roza, J. Mullin, P. Nichols, R. Peverati, J. Pittner, Y. Zhao, P.-D. Fan, A. Fonari, M. J. Williamson, R. J. Harrison, J. R. Rehr, M. Dupuis, D. Silverstein, D. M. A. Smith, J. Nieplocha, V. Tipparaju, M. Krishnan, B. E. Van Kuiken, A. Vazquez-Mayagoitia, L. Jensen, M. Swart, Q. Wu, T. Van Voorhis, A. A. Auer, M. Nooijen, L. D. Crosby, E. Brown, G. Cisneros, G. I. Fann, H. Fruchtl, J. Garza, K. Hirao, R. A. Kendall, J. A. Nichols, K. Tsemekhman, K. Wolinski, J. Anchell, D. E. Bernholdt, P. Borowski, T. Clark, D. Clerc, H. Dachsel, M. J. O. Deegan, K. Dyall, D. Elwood, E. Glendening, M. Gutowski, A. C. Hess, J. Jaffe, B. G. Johnson, J. Ju, R. Kobayashi, R. Kutteh, Z. Lin, R. Littlefield, X. Long, B. Meng, T. Nakajima, S. Niu, L. Pollack, M. Rosing, K. Glaesemann, G. Sandrone, M. Stave, H. Taylor, G. Thomas, J. H. van Lenthe, A. T. Wong, and Z. Zhang. 2018. NWChem, A Computational Chemistry Package for Parallel Computers, Version 6.8 https://www.nwchem-sw.org/.

[2] BLAS. 2019. Retrieved from http://www.netlib.org/blas.

[3] BLAS. 2002. Basic linear algebra subprograms technical forum standard. *Int. J. High Perform. Appl. Supercomput.* 16, 1 (2002).

[4] BLIS. 2019. Retrieved from https://github.com/flame/blis.

[5] Patrick Bridges, Nathan Doss, William Gropp, Edward Karrels, Ewing Lusk, and Anthony Skjellum. 1995. *User's Guide for mpich, a Portable Implementation of MPI.* Argonne National Laboratory.

[6] Sonia Coriani, Ove Christiansen, Thomas Fransson, and Patrick Norman. 2012. Coupled-cluster response theory for near-edge x-ray-absorption fine structure of atoms and molecules. *Phys. Rev. A* 85 (Feb. 2012), 022507. Issue 2. DOI: https://doi.org/10.1103/PhysRevA.85.022507

[7] T. Daniel Crawford and Henry F. Schaefer. 2007. An introduction to coupled cluster theory for computational chemists. In *Reviews in Computational Chemistry*, Kenny B. Lipkowitz and Donald B. Boyd (Eds.). Wiley-Blackwell, 33–136. DOI: https://doi.org/10.1002/9780470125915.ch2 Retrieved from arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/9780470125915.ch2.

[8] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. 1990. A set of Level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.* 16, 1 (March 1990), 1–17.

[9] Kazushige Goto and Robert van de Geijn. 2008. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Soft.* 34, 3: Article 12, 25 pages (May 2008).

[10] Kazushige Goto and Robert van de Geijn. 2008b. High-performance implementation of the level-3 BLAS. *ACM Trans. Math. Soft.* 35, 1, Article 4 (July 2008b), 14 pages. DOI: https://doi.org/10.1145/1377603.1377607

[11] Kazushige Goto and Robert A. van de Geijn. 2002. *On Reducing TLB Misses in Matrix Multiplication.* Technical Report TR-02-55. Department of Computer Sciences, The University of Texas at Austin.

[12] W. Gropp, E. Lusk, and A. Skjellum. 1994. *Using MPI.* The MIT Press.

[13] J. Huang, L. Rice, D. A. Matthews, and R. A. van d Geijn. 2017. Generating families of practical fast matrix multiplication algorithms. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'17)*. 656–667. DOI: https://doi.org/10.1109/IPDPS.2017.56

[14] Jianyu Huang, Tyler M. Smith, Greg M. Henry, and Robert A. van de Geijn. 2016. Strassen's algorithm reloaded. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'16)*. IEEE Press, Piscataway, NJ, Article 59, 12 pages. Retrieved from http://dl.acm.org/citation.cfm?id=3014904.3014983.

[15] Thomas-C. Jagau, Ksenia B. Bravaya, and Anna I. Krylov. 2017. Extending quantum chemistry of bound states to electronic resonances. *Annu. Rev. Phys. Chem.* 68, 1 (2017), 525–553. DOI: https://doi.org/10.1146/annurev-physchem-052516-050622 arXiv:https://doi.org/10.1146/annurev-physchem-052516-050622 PMID: 28463649.

[16] Daya S. Khudia, Protonu Basu, and Summer Deng. 2018. Open-sourcing FBGEMM for state-of-the-art server-side inference. Retrieved from https://code.fb.com/ml-applications/fbgemm/.

[17] Kasper Kristensen, Joanna Kauczor, Thomas Kjærgaard, and Poul Jørgensen. 2009. Quasienergy formulation of damped response theory. *J. Chem. Phys.* 131, 4 (2009), 044112. DOI: https://doi.org/10.1063/1.3173828 arXiv:https://doi.org/10.1063/1.3173828

[18] Xiaoye S. Li, James W. Demmel, David H. Bailey, Greg Henry, Yozo Hida, Jimmy Iskandar, W. Kahan, Anil Kapur, Michael C. Martin, Brandon J. Thompson, Teresa Tung, and Daniel J. Yoo. 2000. *Design, Implementation and Testing of Extended and Mixed Precision BLAS*. LAPACK Working Note 149.

[19] Devin Matthews. 2018. High-performance tensor contraction without transposition. *SIAM J. Sci. Comput.* 40, 1 (2018), C1–C24. DOI: https://doi.org/10.1137/16M108968X arXiv:https://doi.org/10.1137/16M108968X

[20] Marcel Nooijen and Jaap G. Snijders. 1992. Coupled cluster approach to the single-particle Green's function. *Int. J. Quant. Chem.* 44, S26 (1992), 55–83. DOI: https://doi.org/10.1002/qua.560440808 Retrieved from arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/qua.560440808.

[21] OpenBLAS 2019. Retrieved from https://github.com/xianyi/OpenBLAS.

[22] Tyler M. Smith, Robert van de Geijn, Mikhail Smelyanskiy, Jeff R. Hammond, and Field G. Van Zee. 2014. Anatomy of high-performance many-threaded matrix multiplication. In *Proceedings of the IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 1049–1059.

[23] Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. 1996. *MPI: The Complete Reference*. The MIT Press.

[24] John F. Stanton. 1997. Why CCSD(T) works: A different perspective. *Chem. Phys. Lett.* 281, 1 (1997), 130–134. DOI: https://doi.org/10.1016/S0009-2614(97)01144-5

[25] M. Valiev, E. J. Bylaska, N. Govind, K. Kowalski, T. P. Straatsma, H. J. J. Van Dam, D. Wang, J. Nieplocha, E. Apra, T. L. Windus, and W. A. de Jong. 2010. NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations. *Comput. Phys. Commun.* 181, 9 (2010), 1477–1489. DOI: https://doi.org/10.1016/j.cpc.2010.04.018

[26] Field G. Van Zee. 2020. Implementing high-performance complex matrix multiplication via the 1m method. *SIAM J. Sci. Comput.* 42, 5 (Sep. 2020), C221–C244. https://doi.org/10.1137/19M1282040

[27] Field G. Van Zee and Tyler M. Smith. 2017. Implementing high-performance complex matrix multiplication via the 3m and 4m methods. *ACM Trans. Math. Soft.* 44, 1 (June 2017), 7:1–7:36. https://doi.org/10.1145/3086466

[28] Field G. Van Zee and Robert A. van de Geijn. 2015. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Trans. Math. Soft.* 41, 3, Article 14 (June 2015), 33 pages. DOI: https://doi.org/10.1145/2764454

[29] XBLAS 2019. Retrieved from http://www.netlib.org/xblas.

[30] Chenhan D. Yu, Jianyu Huang, Woody Austin, Bo Xiao, and George Biros. 2015. Performance optimization for the K-nearest neighbors kernel on x86 architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'15)*. ACM, New York, NY, Article 7, 12 pages. DOI: https://doi.org/10.1145/2807591.2807601

Received January 2019; revised December 2019; accepted May 2020