IMPLEMENTING HIGH-PERFORMANCE COMPLEX MATRIX MULTIPLICATION VIA THE 1M METHOD

FIELD G. VAN ZEE*

Abstract. Almost all efforts to optimize high-performance matrix-matrix multiplication have been focused on the case where matrices contain real elements. The community's collective assumption appears to have been that the techniques and methods developed for the real domain carry over directly to the complex domain. As a result, implementors have mostly overlooked a class of methods that compute complex matrix multiplication using only real matrix products. This is the second in a series of articles that investigate these so-called induced methods. In the previous article, we found that algorithms based on the more generally applicable of the two methods—the 4M method—lead to implementations that, for various reasons, often underperform their real domain counterparts. To overcome these limitations, we derive a superior 1M method for expressing complex matrix multiplication, one which addresses virtually all of the shortcomings inherent in 4M. Implementations are developed within the BLIS framework, and testing on microarchitectures by three vendors confirms that the 1M method yields performance that is generally competitive with solutions based on conventionally implemented complex kernels, sometimes even outperforming vendor libraries.

Key words. high-performance, complex, matrix, multiplication, microkernel, kernel, BLAS, BLIS, 1m, 2m, 4m, induced, linear algebra, DLA

AMS subject classifications. 65Y04

1. Introduction.

Over the last several decades, matrix multiplication research has resulted in methods and implementations that primarily target the real domain. Recent trends in implementation efforts have condensed virtually all matrix product computation into relatively small *kernels*—building blocks of highly optimized code (typically written in assembly language) upon which more generalized functionality is constructed via various levels of nested loops [23, 5, 3, 22, 2]. Because most effort is focused on the real domain, the complex domain is either left as an unimplemented afterthought—perhaps because the product is merely a proof-of-concept or prototype [5], or because the project primarily targets applications and uses cases that require only real computation [2]—or it is implemented in a manner that mimics the real domain down to the level of the assembly kernel [23, 3, 4].[1] Most modern microarchitectures lack machine instructions for directly computing complex arithmetic on complex numbers, and so when the effort to implement these kernels is undertaken, kernel developers encounter additional programming challenges that do not manifest in the real domain. Specifically, these kernel developers must explicitly orchestrate computation on the real and imaginary components in order to implement multiplication and addition on complex scalars, and they must do so in terms of vector instructions to ensure high performance is achievable.

This low-level kernel approach carries distinct benefits. Pushing the nuances and complexities of complex arithmetic down to the level of the kernel allows the higher-level loop infrastructure within the matrix multiplication to remain largely the same

*Oden Institute for Computational Engineering & Sciences, The University of Texas at Austin, Austin, TX (field@cs.utexas.edu)

Funding: This research was partially sponsored by grants from Intel Corporation and the National Science Foundation (Award ACI-1550493). *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).*

[1] Because they exhibit slightly less favorable numerical properties, we exclude Strassen-like efforts from this characterization.

as its real domain counterpart. (See Figure 1.1.) Another benefit leverages a key difference between the real and complex forms of nearly all matrix computations: arithmetic intensity. Complex matrix multiplication (regardless of how it is implemented) requires four times the number of floating-point operations but only twice the number of memory operations. Encoding complex arithmetic within assembly code allows the real and imaginary components of the multiplication operands to be loaded into registers and then resued at virtually no cost. The impact of the two-fold increase in memory operations is further minimized thanks to the standard format for storing complex matricies, which places an element's real and imaginary parts adjacent to one another[2], allowing both to be accessed conveniently via contiguous load and store instructions. Thanks to this low-cost reuse and accommodating storage format, implementations based on assembly-based complex kernels are capable of achieving a somewhat larger fraction of the hardware's peak performance relative to real domain kernels.

However, this low-level approach also doubles the number of assembly kernels that must be written in order to fully support computation in either domain (real or complex) for the desired floating-point precisions. And while computation in the complex domain may not be of interest to all developers, it is absolutely essential for many fields and applications in part because of complex numbers' unique ability to encode both the phase and magnitude of a wave. Thus, the maintainers of general-purpose matrix libraries—such as those that export the Basic Linear Algebra Subprograms (BLAS) [1]—are typically compelled by their diverse user bases to support general matrix multiplication (GEMM) on complex matrices despite the implementation and maintenance costs it may impose.

Because of how software developers have historically designed their implementations, many assume that supporting complex matrix multiplication operations first requires writing complex domain kernels. To our pleasant surprise, we have discovered a new way for developers to implement high-performance complex matrix multiplication *without* those kernels.

The predecessor to the current article investigates whether (and to what degree of effectiveness) real domain matrix multiplication kernels can be repurposed and leveraged toward the implementation of complex matrix multiplication [21]. The authors develop a new class of algorithms that implement these so-called "induced methods" for matrix products in the complex domain. Instead of relying on an assembly-coded complex kernel, as a conventional implementation would, these algorithms express complex matrix multiplication only in terms of real domain primitives.[3] We consider the current article a companion and follow-up to that previous work [21].

In this article, we will consider a new method for emulating complex matrix multiplication using only real domain building blocks, and we will once again show that a clever rearrangement of the real and imaginary elements within the internal "packed" matrices is key to facilitating high performance. The novelty behind this new method is that the semantics of complex arithmetic are encoded entirely within a special data layout, which allows each call to the complex matrix multiplication kernel to be replaced with just *one* call to a real matrix multiplication kernel. This substitution is possible because a real matrix multiplication on the reorganized data

[2] The widely-accepted BLAS interface requires use of this standard format.

[3] In [21], the authors use the term "primitive" to refer to a functional abstraction that implements a single real matrix multiplication. Such primitives are often not general purpose and may come with significant prerequisites to facilitate their use.

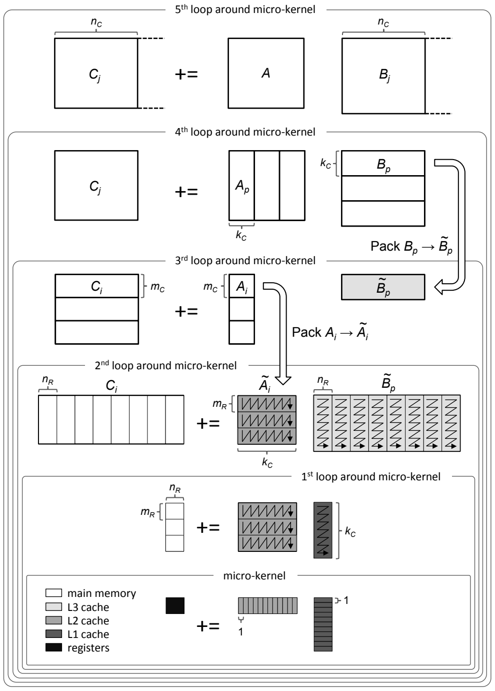

FIG. 1.1. *An illustration of the algorithm for computing high-performance matrix multiplication, taken from [21], which expresses computation in terms of a so-called "block-panel" subproblem.*

mimics the computation and I/O of a comparable complex matrix multiplication on the unaltered data. Because of this one-to-one equivalence, we call it the 1M method.

1.1. Contributions. This article makes the following contributions:

- It introduces[4] the 1M method along with two algorithmic variants and an analysis of issues germane to their high-performance implementations, including workspace, packing formats, cache behavior, multithreadability, and programming effort. A detailed review shows how 1M avoids all of the major challenges observed of the 4M method.
- It promotes code reuse and portability by continuing the previous article's focus on solutions which may be cast in terms of real matrix multiplication kernels. Such solutions have clear implications for developer productivity, as they allow kernel authors to focus their efforts on fewer and simpler kernels.
- It builds on the theme of the BLIS framework as a productivity multiplier [22], further demonstrating how complex matrix multiplication may be implemented with relatively minor modifications to the source code and in such a way that results in immediate instantiation of complex implementations for *all* level-3 BLAS-like operations.
- It demonstrates performance of 1M implementations that is not only superior to the previous effort based on the 4M method but also competitive with solutions based on complex matrix kernels.
- It serves as a reference guide to the 1M implementations for complex matrix

[4] This proposed 1M method was first published [19].

multiplication found within the BLIS framework, which is available to the community under the open-source 3-clause BSD software license.

We believe these contributions are consequential because the 1M method effectively obviates the previous state-of-the-art established via the 4M method. Furthermore, we believe the thorough treatment of induced methods encompassed by the present article and its predecessor will have lasting archival as well as pedagogical value.

1.2. Notation.

In this article, we continue the notation established in [21]. Specifically, we use uppercase Roman letters (e.g. A, B, and C) to refer to matrices, lowercase Roman letters (e.g. x, y, and z) to refer to vectors, and lowercase Greek letters (e.g. χ, ψ, and ζ) to refer to scalars. Subscripts are used typically to denote sub-matrices within a larger matrix (e.g. $A = \left(\begin{array}{c|c|c|c} A_0 & A_1 & \cdots & A_{n-1} \end{array} \right)$) or scalars within a larger matrix or vector.

We make extensive use of superscripts to denote the real and imaginary components of a scalar, vector, or (sub-)matrix. For example, $\alpha^r, \alpha^i \in \mathbb{R}$ denote the real and imaginary parts, respectively, of a scalar $\alpha \in \mathbb{C}$. Similarly, A^r and A^i refer to the real and imaginary parts of a complex matrix A, where A^r and A^i are real matrices with dimensions identical to A. Note that while this notation for real, imaginary, and complex matrices encodes information about content and origin, it does not encode how the matrices are actually stored. We will explicitly address storage details as implementation issues are discussed.

At times we find it useful to refer to the real and imaginary elements of a complex object indistinguishably as *fundamental elements* (or F.E.). We also abbreviate floating-point operations as "flops" and memory operations as "memops". We define the former to be a MULTIPLY or ADD (or SUBTRACT) operation whose operands are F.E. and the latter to be a load or store operation on a single F.E.. These definitions allow for a consistent accounting of complex computation relative to the real domain.

We also discuss cache and register blocksizes that are key features of the matrix multiplication algorithm discussed elsewhere [22, 20, 21]. Unless otherwise noted, blocksizes n_C, m_C, k_C, m_R, and n_R refer to those appropriate for computation in the real domain. Complex domain blocksizes will be denoted with a superscript ^{z}.

This article discusses several hypothetical algorithms and functions. Unless otherwise noted, a call to function FUNC that implements $C := C + AB$ appears as [C] := FUNC(A, B, C). We will also reference functions that access properties of matrices. For example, M(A) and N(A) would return the m and n dimensions of a matrix A, while RS(B) and CS(B) would return the row and column strides of B.

2. Background and review.

2.1. Motivation.

In [21], the authors list three primary motivating factors behind their effort to seek out methods for inducing complex matrix multiplication via real domain kernels:

- **Productivity.** By inducing complex matrix multiplication from real domain kernels, the number of kernels that must be supported would be halved. This allows the DLA library developers to focus on a smaller and simpler set of real domain kernels. This benefit would manifest most obviously when instantiating BLAS-like functionality on new hardware [20].
- **Portability.** Induced methods avoid dependence on complex domain kernels because they encode the idea of complex matrix product at a higher level. This would naturally allow us to encode such methods portably within a framework such as BLIS [22]. Once integrated into the framework, developers

and users would benefit from the immediate availability of complex matrix multiplication implementations whenever real matrix kernels were present.

- **Performance.** Implementations of complex matrix multiplication that rely on real domain kernels would likely inherit the high-performance properties of those kernels. Any improvement to the real kernels would benefit both real and complex domains.

Thus, it is clear that finding a suitable induced method would carry significant benefit to DLA library and kernel developers.

2.2. The 3m and 4m methods. The authors of [21] investigated two general ways of inducing complex matrix multiplication: the 3M method and the 4M method. These methods are then contrasted to the conventional approach, whereby a blocked matrix multiplication algorithm is executed with a complex domain kernel—one that implements complex arithmetic at the scalar level, in assembly language.

The 4M method begins with the classic definition of complex scalar multiplication and addition in terms of real and imaginary components of $\alpha, \beta, \gamma \in \mathbb{C}$:

$$\begin{aligned} \gamma^r &:= \gamma^r + \alpha^r\beta^r - \alpha^i\beta^i \\ \gamma^i &:= \gamma^i + \alpha^i\beta^r + \alpha^r\beta^i \end{aligned} \tag{2.1}$$

We then observe that we can apply such a definition to complex matrices $A \in \mathbb{C}^{m\times k}$, $B \in \mathbb{C}^{k\times n}$, and $C \in \mathbb{C}^{m\times n}$, provided that we can reference the real and imaginary parts as logically separate submatrices:

$$\begin{aligned} C^r &:= C^r + A^rB^r - A^iB^i \\ C^i &:= C^i + A^iB^r + A^rB^i \end{aligned} \tag{2.2}$$

This definition expresses a complex matrix multiplication in terms of four matrix products (hence the name 4M) and four matrix accumulations (i.e., additions or subtractions).

The 3M method relies on a Strassen-like algebraic equivalent of Eq. 2.2:

$$\begin{aligned} C^r &:= C^r + A^rB^r - A^iB^i \\ C^i &:= C^i + \left(A^r + A^i\right)\left(B^r + B^i\right) - A^rB^r - A^iB^i \end{aligned}$$

This re-expression reduces the number of matrix products to three at the expense of increasing the number of accumulations from four to seven. However, when the cost of a matrix product greatly exceeds that of an accumulation, this trade-off can result in a net reduction in computational runtime.

The authors of [21] observe that both methods may be applied to any particular level of a blocked matrix multiplication algorithm, resulting in several algorithms, each exhibiting somewhat different properties. Furthermore, they show how either method's implementation is facilitated by reordering real and imaginary elements within the internal storage format used when making packed copies of the current matrix blocks.[5] The blocked algorithm used in that article is shown in Figure 1.1 and revisited in Section 2.4 of the present article.

Algorithms that implement the 3M method were found to yield "effective flops per second" performance that not only exceeded that of 4M, but also approached or

[5] Others have exploited the careful design of packing and computational primitives in an effort to improve performance, including in the context of Strassen's algorithm [7, 9, 10, 11], the computation of the K-Nearest Neighbors [24], tensor contraction [8], and Fast Fourier Transform [17].

exceeded the theoretical peak rate of the hardware.[6] Unfortunately, these compelling results come at a cost: the numerical properties of implementations based on 3M are slightly less robust than that of algorithms based on the conventional approach or 4M. And although the author of [6] found that 3M was stable enough for most practical purposes, many applications will be unwilling to stray from the numerical expectations implicit in conventional matrix multiplication. Thus, going forward, we will focus on 4M as the standard reference method against which we will compare.

It is worth briefly considering the simplest approach to implementing the 4M method, which is hinted at by Eq. 2.2. This straightforward algorithm would invoke the real domain GEMM four times, computing A^rB^r and A^iB^i to update C^r and A^iB^r and A^rB^i to update C^i. This is possible as long as the matrices' real and imaginary parts are separately addressable, as they are in modern BLAS-like frameworks such as BLIS [22]. The previous article studied this high-level instance of the 4M method, dubbed Algorithm 4M_HW, and classified it into a family of related algorithms. Unfortunately, Algorithm 4M_HW does not perform well with the standard format for storing complex matrices. The reason: Algorithm 4M_HW computes with A^r, A^i, B^r, B^i, C^r, and C^i as separate logical matrices, but since their F.E. are stored non-contiguously, those F.E. cannot be accessed efficiently on modern hardware. If Algorithm 4M_HW instead computed upon matrices that split the storage of their real and imaginary parts into two separate matrices, each with contiguous rows or columns, then its expected performance would rise to match that of real matrix multiplication. However, even with this somewhat exotic "split" complex storage format, Algorithm 4M_HW would carry some disadvantages relative to the new induced method discussed later in this article.[7]

Thus, our aim is to develop an induced method that (1) yields performance that is at least as high as that of a corresponding real domain GEMM while also (2) allowing applications to continue using the standard storage format[8] and (3) avoiding key disadvantages inherent in the various 4M algorithms, including 4M_HW.

2.3. Previous findings. For the reader's convenience, we will now summarize the key findings, observations, and other highlights from the previous article regarding algorithms and implementations based on the 4M method [21].

- Since all algorithms in the 4M family execute the same number of flops, the algorithms' relative performance depends entirely on (1) the number of memops executed and (2) the level of cache from which F.E. of the packed matrices $\tilde{A}_i$ and $\tilde{B}_p$ are reused[9]. The number of memops is affected only by a halving of certain cache blocksize needed in order to leave cache footprints of $\tilde{A}_i$ and

[6] Note that 3M and other Strassen-like algorithms are able to exceed the hardware's theoretical peak performance when measured in *effective* flops per second: that is, the 3M implementation's wall clock time—now shorter because of avoided matrix products—divided into the flop count of a *conventional* algorithm.

[7] For example, parallelizing Algorithm 4M_HW may be limited by the three implicit synchronization points that would occur between the four invocations of real domain GEMM. Also, it was shown in the previous article that Algorithm 4M_HW inherently can only be applied to two-operand level-3 operations such as TRMM and TRSM by using $m \times n$ workspace [21].

[8] An implementation may use the split format internally while still requiring the standard storage format at the user level. However, this technique, which is employed on more granular scale by Algorithm 4M_1A in the previous article, would incur a noticeable increase in memory operations and serve as a net drag on performance [21].

[9] Here, the term "reuse" refers to the reuse of F.E. that corresponds to the recurrence of A^r, A^i, B^r, and B^i in Eq. 2.2, not the reuse of whole (complex) elements that naturally occurs in the execution of the GEMM algorithm in Figure 1.1.

$\tilde{B}_p$ unchanged. The level of cache from which F.E. are reused is determined by the level of the GEMM algorithm to which the 4M method was applied.

- The lowest-level application, algorithm 4M_1A, efficiently moves F.E. of A, B, and C from main memory to the L1 cache only once per rank-k_C update and reuses F.E. from the L1 cache. It relies on a relatively simple packing format and requires negligible, fixed-size workspace, is well-suited for multithreading, and is minimally disruptive to the BLIS framework. Algorithm 4M_1A can also be extended relatively easily to all other level-3 operations.
- The conventional assembly-based approach to complex matrix multiplication can be viewed as a special case of 4M in which F.E. are reused from registers rather than cache. In this way, a conventional implementation embodies the lowest-level application of 4M possible, in which the method is applied to individual scalars (and then optimally encoded via vector instructions).
- The way complex numbers are stored has a significant effect on performance. The standard format adopted by the community (and required by the BLAS), which uses an interleaved pair-wise storage of real and imaginary values, naturally favors conventional implementations because they can reuse F.E. from vector registers. However, this storage is awkward for algorithms based on 4M (and 3M) because it stymies the use of vector instructions for loading and storing F.E. of C^r and C^i. The 4M_1A algorithm already suffers from a *quadrupling*[10] of the number of memops on C in addition to being forced to access these F.E. in a non-contiguous manner.
- While the performance of Algorithm 4M_1A exceeds that of its simpler sibling, 4M_HW, it not only falls short of a comparable conventional solution, it also falls short of its real domain "benchmark"—that is, the performance of a similar problem size in the real domain computed by an optimized algorithm using the same real domain kernel.

2.4. Revisiting the matrix multiplication algorithm. In this section, we review a common algorithm for high-performance matrix multiplication on conventional microprocessor architectures. This algorithm was first reported on in [3] and further refined in [22]. Figure 1.1 illustrates the key features of this algorithm.

The current state-of-the-art formulation of the matrix multiplication algorithm consists of six loops, the last of which resides within a microkernel that is typically highly optimized for the target hardware. These loops partition the matrix operands using carefully chosen cache (n_C, k_C, and m_C) and register (m_R and n_R) blocksizes that result in submatrices residing favorably at various levels of the cache hierarchy so as to allow data to be reused many times. In addition, submatrices of A and B are copied ("packed") to temporary workspace matrices ($\tilde{A}_i$ and $\tilde{B}_p$, respectively) in such a way that allows the microkernel to subsequently access matrix elements contiguously in memory, which improves cache and TLB performance. The cost of this packing is amortized over enough computation that its impact on overall performance is negligible for all but the smallest problems. At the lowest level, within the microkernel loop, an $m_R \times 1$ micro-column and a $1 \times n_R$ micro-row are loaded from the current micropanels of $\tilde{A}_i$ and $\tilde{B}_p$, respectively, so that the outer product of these vectors may be computed to update the corresponding $m_R \times n_R$ submatrix, or micro-tile, of C. The individual floating-point operations that constitute these tiny rank-1 updates

[10] A factor of two comes from the fact that, as shown in Eq. 2.2, 4M touches C^r and C^i twice each, while another factor of two comes from the cache blocksize scaling required on k_C in order to maintain the cache footprints of micropanels of $\tilde{A}_i$ and $\tilde{B}_p$.

are oftentimes executed via vector instructions (if the architecture supports them) in order to maximize utilization of the floating-point unit(s).

The algorithm captured by Figure 1.1 forms the basis for all level-3 implementations found in the BLIS framework (as of this writing). This algorithm is based on a so-called block-panel matrix multiplication.[11] The register (m_R, n_R) and cache (m_C, k_C, n_C) blocksizes labeled in the algorithmic diagram are typically chosen by the kernel developer as a function of hardware characteristics, such as the vector register set, cache sizes, and cache associativity. The authors of [15] present an analytical model for identifying suitable (if not optimal) values for these blocksizes.

3. 1m method.

The primary motivation for seeking a better induced method comes from the observation that 4M inherently must update real and imaginary F.E. of C: (1) in separate steps, and may not use vector instructions to do so (due to the standard interleaved storage format); and (2) twice as frequently, in the case of 4M_1A, due to the algorithm's half-of-optimal cache blocksize k_C. As reviewed in Section 2.3, this imposes a significant drag on performance. If there existed an induced method that could update real and imaginary elements in one step, it may conveniently avoid both issues.

3.1. Derivation.

Consider the classic definition of complex scalar multiplication and accumulation, shown in Eq. 2.1, refactored and expressed in terms of matrix and vector notation:

$$\begin{pmatrix} \gamma^r \\ \gamma^i \end{pmatrix} \mathrel{+}= \begin{pmatrix} \alpha^r & -\alpha^i \\ \alpha^i & \alpha^r \end{pmatrix} \begin{pmatrix} \beta^r \\ \beta^i \end{pmatrix} \tag{3.1}$$

Here, we have a singleton complex matrix multiplication problem that can naturally be expressed as a tiny real matrix multiplication where $m = k = 2$ and $n = 1$. Let us assume we implement this very small matrix multiplication according to the high-performance algorithm discussed in Section 2.4.

From this, we make the following key observation: If we pack α to $\tilde{A}_i$ in such a way that duplicates α^r and α^i to the second column of the micropanel (while also swapping the placement of the duplicates and negating the duplicated α^i), and if we pack β to $\tilde{B}_p$ such that β^i is stored to the second row of the micropanel (which, granted, only has one column), then a real domain GEMM microkernel executed on those micropanels will compute the correct result in the complex domain and do so with a *single* invocation of that microkernel.

Thus, Eq. 3.1 serves as a packing template that hints at how the data must be stored. Furthermore, this template can be generalized. We augment α, β, γ with conventional row and column indices to denote the complex elements of matrices A, B, and C, respectively. Also, let us apply the Eq. 3.1 to the special case of $m = 3$, $n = 4$, and $k = 2$ to better observe the general pattern.

$$\begin{pmatrix} \gamma^r_{00} & \gamma^r_{01} & \gamma^r_{02} & \gamma^r_{03} \\ \gamma^i_{00} & \gamma^i_{01} & \gamma^i_{02} & \gamma^i_{03} \\ \gamma^r_{10} & \gamma^r_{11} & \gamma^r_{12} & \gamma^r_{13} \\ \gamma^i_{10} & \gamma^i_{11} & \gamma^i_{12} & \gamma^i_{13} \\ \gamma^r_{20} & \gamma^r_{21} & \gamma^r_{22} & \gamma^r_{23} \\ \gamma^i_{20} & \gamma^i_{21} & \gamma^i_{22} & \gamma^i_{23} \end{pmatrix} \mathrel{+}= \begin{pmatrix} \alpha^r_{00} & -\alpha^i_{00} & \alpha^r_{01} & -\alpha^i_{01} \\ \alpha^i_{00} & \alpha^r_{00} & \alpha^i_{01} & \alpha^r_{01} \\ \alpha^r_{10} & -\alpha^i_{10} & \alpha^r_{11} & -\alpha^i_{11} \\ \alpha^i_{10} & \alpha^r_{10} & \alpha^i_{11} & \alpha^r_{11} \\ \alpha^r_{20} & -\alpha^i_{20} & \alpha^r_{21} & -\alpha^i_{21} \\ \alpha^i_{20} & \alpha^r_{20} & \alpha^i_{21} & \alpha^r_{21} \end{pmatrix} \begin{pmatrix} \beta^r_{00} & \beta^r_{01} & \beta^r_{02} & \beta^r_{03} \\ \beta^i_{00} & \beta^i_{01} & \beta^i_{02} & \beta^i_{03} \\ \beta^r_{10} & \beta^r_{11} & \beta^r_{12} & \beta^r_{13} \\ \beta^i_{10} & \beta^i_{11} & \beta^i_{12} & \beta^i_{13} \end{pmatrix} \tag{3.2}$$

[11] This terminology describes the shape of the typical problem computed by the macro-kernel, i.e. the second loop around the microkernel. An alternative algorithm that casts its largest cache-bound subproblem in terms of panel-block matrix multiplication is discussed in [19].

From this, we can make the following observations:

- The complex matrix multiplication $C := C + AB$ with $m = 3$, $n = 4$, and $k = 2$ becomes a real matrix multiplication with $m = 6$, $n = 4$, and $k = 4$. In other words, the m and k dimensions are doubled for the purposes of the real gemm primitive.
- If the primitive is the real gemm microkernel, and we assume that matrices A and B above represent column-stored and row-stored micropanels from $\tilde{A}_i$ and $\tilde{B}_p$, respectively, and also that the dimensions are conformal to the register blocksizes of this microkernel (i.e., $m = m_R$ and $n = n_R$) then the micropanels of $\tilde{A}_i$ are packed from a $\frac{1}{2}m_R \times \frac{1}{2}k_C$ submatrix of A, which, when expanded in the special packing format, appears as the $m_R \times k_C$ micropanel that the real gemm microkernel expects.
- Similarly, the micropanels of $\tilde{B}_p$ are packed from a $\frac{1}{2}k_C \times n_R$ submatrix of B, which, when reordered into a second special packing format, appears as the $k_C \times n_R$ micropanel that the real gemm microkernel expects.

It is easy to see by inspection that the real matrix multiplication implied by Eq. 3.2 induces the desired complex matrix multiplication. We will refer to the packing format used on matrix A above as the 1e format, since the f.e. are "expanded" (i.e., duplicated to the next column, with the duplicates swapped and the imaginary duplicate negated). Similarly, we will refer to the packing format used on matrix B above as the 1r format, since the f.e. are merely reordered (i.e., imaginary elements moved to the next row). Thus, the 1m method is fundamentally about reordering the matrix data so that a subsequent real matrix multiplication on that reordered data is equivalent to a complex matrix multiplication on the original data.[12]

3.2. Two variants. Notice that implicit in the 1m method suggested by Eq. 3.2 is the fact that matrix C is stored by columns. This assumption is important; when A and B are packed according to the 1e and 1r formats, respectively, C must be stored by columns in order to allow the real domain primitive (or microkernel) to correctly update the individual real and imaginary f.e. of C with the corresponding f.e. from the matrix product AB.

Suppose that we instead refactored and expressed Eq. 2.1 as follows:

$$\begin{pmatrix} \gamma^r & \gamma^i \end{pmatrix} \mathrel{+}= \begin{pmatrix} \alpha^r & \alpha^i \end{pmatrix} \begin{pmatrix} \beta^r & \beta^i \\ -\beta^i & \beta^r \end{pmatrix} \tag{3.3}$$

This gives us a different template, one that implies different packing formats for A and B. Applying Eq. 3.3 to the special case of $m = 4$, $n = 3$, and $k = 2$ yields:

$$\begin{pmatrix} \gamma^r_{00} & \gamma^i_{00} & \gamma^r_{01} & \gamma^i_{01} & \gamma^r_{02} & \gamma^i_{02} \\ \gamma^r_{10} & \gamma^i_{10} & \gamma^r_{11} & \gamma^i_{11} & \gamma^r_{12} & \gamma^i_{12} \\ \gamma^r_{20} & \gamma^i_{20} & \gamma^r_{21} & \gamma^i_{21} & \gamma^r_{22} & \gamma^i_{22} \\ \gamma^r_{30} & \gamma^i_{30} & \gamma^r_{31} & \gamma^i_{31} & \gamma^r_{32} & \gamma^i_{32} \end{pmatrix} \mathrel{+}= \begin{pmatrix} \alpha^r_{00} & \alpha^i_{00} & \alpha^r_{01} & \alpha^i_{01} \\ \alpha^r_{10} & \alpha^i_{10} & \alpha^r_{11} & \alpha^i_{11} \\ \alpha^r_{20} & \alpha^i_{20} & \alpha^r_{21} & \alpha^i_{21} \\ \alpha^r_{30} & \alpha^i_{30} & \alpha^r_{31} & \alpha^i_{31} \end{pmatrix} \begin{pmatrix} \beta^r_{00} & \beta^i_{00} & \beta^r_{01} & \beta^i_{01} & \beta^r_{02} & \beta^i_{02} \\ -\beta^i_{00} & \beta^r_{00} & -\beta^i_{01} & \beta^r_{01} & -\beta^i_{02} & \beta^r_{02} \\ \beta^r_{10} & \beta^i_{10} & \beta^r_{11} & \beta^i_{11} & \beta^r_{12} & \beta^i_{12} \\ -\beta^i_{10} & \beta^r_{10} & -\beta^i_{11} & \beta^r_{11} & -\beta^i_{12} & \beta^r_{12} \end{pmatrix} \tag{3.4}$$

In this variant, we see that matrix B, not A, is stored according to the 1e format (where columns become rows), while matrix A is stored according to 1r (where rows

[12] The authors of [17] also investigated the use of transforming the data layout during packing to facilitate complex matrix multiplication. And while they employ techniques similar to those of the 1m method, their approach differs in that it does not recycle the existing real domain microkernel.

Table 3.1
1M complex domain blocksizes as a function of real domain blocksizes

Variant	Blocksizes, in terms of real domain values, required for ...						
	k_C^z	m_C^z	n_C^z	m_R^z	m_P^z	n_R^z	n_P^z
1M_C	$\frac{1}{2}k_C$	$\frac{1}{2}m_C$	n_C	$\frac{1}{2}m_R$	m_P	n_R	n_P
1M_R	$\frac{1}{2}k_C$	m_C	$\frac{1}{2}n_C$	m_R	m_P	$\frac{1}{2}n_R$	n_P

Note: Blocksizes m_P and n_P represent the so-called “packing dimensions” for the micro-panels of $\tilde{A}_i$ and $\tilde{B}_p$, respectively. These values are analogous to the leading dimensions of matrices stored by columns or rows. In BLIS microkernels, typically $m_R = m_P$ and $n_R = n_P$, but sometimes the kernel author may find it useful for $m_R < m_P$ or $n_R < n_P$.

become columns). Also, we can see that matrix C must be stored by rows in order to allow the real GEMM microkernel to correctly update its F.E. with the corresponding values from the matrix product AB.

Henceforth, we will refer to the 1M variant exemplified in Eq. 3.2 as 1M_C since it is predicated on column storage of the output matrix C, and we will refer to the variant depicted in Eq. 3.4 as 1M_R since it assumes C is stored by rows.

3.3. Determining complex blocksizes. As we alluded in Section 3.1, the appropriate blocksizes to use with 1M are a function of the real domain blocksizes. This makes sense because the idea is to fool the real GEMM microkernel, and the various loops for register and cache blocking around the microkernel, into thinking that it is computing a real domain matrix multiplication. Which blocksizes must be modified (halved) and which are used unchanged depends on the variant of 1M being executed—or, more specifically, which matrix is packed according to the 1E format.

Table 3.1 summarizes the complex domain blocksizes prescribed for 1M_C and 1M_R as a function of the real domain values.

Those familiar with the matrix multiplication algorithm implemented by the BLIS framework, as depicted in Figure 1.1, may be unfamiliar with m_P and n_P, the so-called packing dimensions. These values are the leading dimensions of the micropanels. On most architectures, $m_P = m_R$ and $n_P = n_R$, but in some situations it may be convenient (or necessary) to use $m_R < m_P$ or $n_R < n_P$. In any case, these packing dimensions are never scaled, even when their corresponding register blocksizes *are* scaled to accommodate the 1E format, because the halving that would otherwise be called for is cancelled out by the doubling of F.E. that manifests in the 1E format.

3.4. Algorithms.

3.4.1. General algorithm. Before investigating 1M method algorithms, we will first provide algorithms for computing real matrix multiplication to serve as a reference for the reader. Specifically, in Figure 3.1 we provide pseudo-code for RMMBP, which depicts a real domain instance of the block-panel algorithm shown in Figure 1.1.

3.4.2. 1m-specific algorithm. Applying 1M_C and 1M_R to the block-panel algorithm depicted in Figure 1.1 yields two nearly identical algorithms, 1M_C_BP and 1M_R_BP, respectively. Their differences can be encoded within a few conditional statements within key parts of the high and low levels of code. Figure 3.2 shows a hybrid algorithm that encompasses both, supporting row- and column-stored C.

```
Algorithm: [ $C$ ] := rmmbp( $A$, $B$, $C$ )

for ( $j = 0 : n-1 : n_C$ )
  Identify $B_j, C_j$ from $B, C$
  for ( $p = 0 : k-1 : k_C$ )
    Identify $A_p, B_{jp}$ from $A, B_j$
    Pack $B_{jp} \rightarrow \tilde{B}_p$
    for ( $i = 0 : m-1 : m_C$ )
      Identify $A_{pi}, C_{ji}$ from $A_p, C_j$
      Pack $A_{pi} \rightarrow \tilde{A}_i$
      for ( $h = 0 : n_C - 1 : n_R$ )
        Identify $\tilde{B}_{ph}, C_{jih}$ from $\tilde{B}_p, C_{ji}$
        for ( $l = 0 : m_C - 1 : m_R$ )
          Identify $\tilde{A}_{il}, C_{jihl}$ from $\tilde{A}_i, C_{jih}$
          $C_{jihl}$ := rkern( $\tilde{A}_{il}$, $\tilde{B}_{ph}$, $C_{jihl}$ )
```

Fig. 3.1. *Abbreviated pseudo-code for implementing the general matrix multiplication algorithm depicted in Figure 1.1. Here,* rkern *calls a real domain* gemm *microkernel. The algorithm is left-justified to facilitate comparison with Algorithms* 1m_c_bp *and* 1m_r_bp *in Figure 3.2 (left).*

```
Algorithm: [ $C$ ] := 1m_?_bp( $A$, $B$, $C$ )

Set bool colStore if rs( $C$ ) = 1
for ( $j = 0 : n-1 : n_C$ )
  Identify $B_j, C_j$ from $B, C$
  for ( $p = 0 : k-1 : k_C$ )
    Identify $A_p, B_{jp}$ from $A, B_j$
    if colStore Pack1r $B_{jp} \rightarrow \tilde{B}_p$
    else        Pack1e $B_{jp} \rightarrow \tilde{B}_p$
    for ( $i = 0 : m-1 : m_C$ )
      Identify $A_{pi}, C_{ji}$ from $A_p, C_j$
      if colStore Pack1e $A_{pi} \rightarrow \tilde{A}_i$
      else        Pack1r $A_{pi} \rightarrow \tilde{A}_i$
      for ( $h = 0 : n_C - 1 : n_R$ )
        Identify $\tilde{B}_{ph}, C_{jih}$ from $\tilde{B}_p, C_{ji}$
        for ( $l = 0 : m_C - 1 : m_R$ )
          Identify $\tilde{A}_{il}, C_{jihl}$ from $\tilde{A}_i, C_{jih}$
          $C_{jihl}$ := vk1m( $\tilde{A}_{il}$, $\tilde{B}_{ph}$, $C_{jihl}$ )
```

```
[ $C$ ] := vk1m( $A$, $B$, $C$ )

Acquire workspace $W$
Determine if using $W$; set usew
if ( usew )
  Alias $C_{\text{use}} \leftarrow W$, $C_{\text{in}} \leftarrow 0$
else
  Alias $C_{\text{use}} \leftarrow C$, $C_{\text{in}} \leftarrow C$
Set bool colStore if rs($C_{\text{use}}$) = 1
if ( colStore ) cs($C_{\text{use}}$) $\times\!=$ 2
else            rs($C_{\text{use}}$) $\times\!=$ 2
n($A$) $\times\!=$ 2; m($B$) $\times\!=$ 2
$C_{\text{use}}$ := rkern( $A$, $B$, $C_{\text{in}}$ )
if ( usew )
  $C := W$
```

Fig. 3.2. *Left: Pseudo-code for Algorithms* 1m_c_bp *and* 1m_r_bp*, which result from applying* 1m_c *and* 1m_r *algorithmic variants to the block-panel algorithm depicted in Figure 1.1. Here,* Pack1e *and* Pack1r *pack matrices into the* 1e *and* 1r *formats, respectively. Right: Pseudo-code for a virtual microkernel used by all* 1m *algorithms.*

In Figure 3.2 (right), we illustrate the 1m *virtual* microkernel. This function, vk1m, consists largely of a call to the real domain microkernel rkern with some additional logic needed to properly induce complex matrix multiplication in all cases. Some of the details of the virtual microkernel will be addressed later.

TABLE 3.2
F.E. *memops incurred by various algorithms, broken down by stage of computation*

Algorithm	F.E. memops required to . . . [a]				
	update micro-tiles[b] C^r, C^i	pack $\tilde{A}_i$	move $\tilde{A}_i$ from L2 to L1 cache	pack $\tilde{B}_p$	move $\tilde{B}_p$ from L3 to L1 cache
4M_H	$8mn\frac{k}{k_C}$	$8mk\frac{n}{n_C}$	$4mk\frac{n}{n_R}$	$8kn$	$4kn\frac{m}{m_C}$
4M_1B	$8mn\frac{k}{k_C}$	$8mk\frac{2n}{n_C}$	$4mk\frac{n}{n_R}$	$8kn$	$4kn\frac{2m}{m_C}$
4M_1A	$8mn\frac{2k}{k_C}$	$8mk\frac{n}{n_C}$	$4mk\frac{n}{n_R}$	$8kn$	$4kn\frac{m}{m_C}$
assembly	$4mn\frac{k}{k_C}$	$4mk\frac{n}{n_C}$	$2mk\frac{n}{n_R}$	$4kn$	$2kn\frac{m}{m_C}$
1M_C_BP	$4mn\frac{2k}{k_C}$	$6mk\frac{n}{n_C}$	$4mk\frac{n}{n_R}$	$4kn$	$2kn\frac{2m}{m_C}$
1M_R_BP		$4mk\frac{n}{n_C}$	$2mk\frac{2n}{n_R}$	$6kn$	$4kn\frac{m}{m_C}$

[a] We express the number of iterations executed in the 5th, 4th, 3rd, and 2nd loops as $\frac{n}{n_C}$, $\frac{k}{k_C}$, $\frac{m}{m_C}$, and $\frac{n}{n_R}$. The precise number of iterations along a dimension x using a cache blocksize x_C would actually be $\lceil \frac{x}{x_C} \rceil$. Similarly, when blocksize scaling of $\frac{1}{2}$ is required, the precise value $\lceil \frac{x}{x_C/2} \rceil$ is expressed as $\frac{2x}{x_C}$. These simplifications allow easier comparison between algorithms while still providing meaningful approximations.

[b] As described in Section 3.6.2, $m_R \times n_R$ workspace sometimes becomes mandatory, such as when $\beta^i \neq 0$. When workspace is employed in a 4M-based algorithm, the number of F.E. memops incurred updating the micro-tile typically doubles from the values shown here.

3.5. Performance properties. Table 3.2 tallies the total number of F.E. memops required by 1M_C_BP and 1M_R_BP. For comparison, we also include the corresponding memop counts for a selection of 4M algorithms as well as a conventional assembly-based solution, as first published in Table III in [21].

Notice that 1M_C_BP and 1M_R_BP incur additional memops relative to a conventional assembly-based solution because, unlike the latter, 1M implementations cannot reuse[13] all real and imaginary F.E. from vector registers.

We can hypothesize that the observed performance signatures of 1M_C_BP and 1M_R_BP may be slightly different because each places the additional memop overhead that is unique to 1M on different parts of the computation. This stems from the fact that there exists an asymmetry in the assignment of packing formats to matrices in each 1M variant. Specifically, 50% more memops—relative to a conventional assembly solution—are required during the initial packing and the movement between caches for the matrix packed according to 1E since that format writes four F.E. for every two that it reads from the source operand. (Packing to 1R incurs the same number of memops as an assembly-based solution.) Also, if 1M_C_BP and 1M_R_BP use real microkernels with different micro-tile shapes (i.e., different values of m_R and n_R), those microkernels' differing performance properties will likely cause the performance signatures of 1M_C_BP and 1M_R_BP to deviate further.

Table 3.3 summarizes Table 3.2 and adds: (1) the level of the memory hierarchy from which each matrix is reused; and (2) a measure of memory movement efficiency.

[13] Here, the term "reuse" refers to the same reuse described in Footnote 9.

Table 3.3
Performance properties of various algorithms

Algorithm	Total F.E. memops required (Sum of columns of Table 3.2)	Level from which F.E. of matrix X are reused, and l_{L1}: # of times each cache line is moved into the L1 cache (per rank-k_C update).					
		C	l^C_{L1}	A	l^A_{L1}	B	l^B_{L1}
4M_H	$8mn\left(\frac{k}{k_C}\right)+4mk\left(\frac{2n}{n_C}+\frac{n}{n_R}\right)+2kn\left(4+\frac{2m}{m_C}\right)$	MEM	4	MEM	4	MEM	4
4M_1B	$8mn\left(\frac{k}{k_C}\right)+4mk\left(\frac{4n}{n_C}+\frac{n}{n_R}\right)+2kn\left(4+\frac{4m}{m_C}\right)$	L2	2[a]	L2	1	L1	1
4M_1A	$8mn\left(\frac{2k}{k_C}\right)+4mk\left(\frac{2n}{n_C}+\frac{n}{n_R}\right)+2kn\left(4+\frac{2m}{m_C}\right)$	L1	1[a]	L1	1	L1	1
assembly	$4mn\left(\frac{k}{k_C}\right)+2mk\left(\frac{2n}{n_C}+\frac{n}{n_R}\right)+2kn\left(2+\frac{m}{m_C}\right)$	REG	1	REG	1	REG	1
1M_C_BP	$4mn\left(\frac{2k}{k_C}\right)+2mk\left(\frac{3n}{n_C}+\frac{2n}{n_R}\right)+2kn\left(2+\frac{2m}{m_C}\right)$	REG	1	L2[b]	1	REG	1
1M_R_BP	$4mn\left(\frac{2k}{k_C}\right)+2mk\left(\frac{2n}{n_C}+\frac{2n}{n_R}\right)+2kn\left(3+\frac{2m}{m_C}\right)$	REG	1	REG	1	L1[b]	1

[a] This assumes that the micro-tile is not evicted from the L1 cache during the next call to RKERN.

[b] In the case of 1M algorithms, we consider F.E. of A and B to be "reused" from the level of cache in which the 1E-formatted matrix resides.

3.6. Algorithm details. This section lays out important details that must be handled when implementing the 1M method.

3.6.1. Microkernel I/O preference. Within the BLIS framework, microkernels are registered with a property that describes their input/output *preference*. The I/O preference describes whether the microkernel is set up to ideally use vector instructions to load and store elements of the micro-tile by rows or by columns. This property typically originates from the semantic orientation of vector registers used to accumulate the $m_R \times n_R$ micropanel product. Whenever possible, the BLIS framework will perform logical transpositions[14] so that the apparent storage of C matches the preference property of the microkernel being used. This guarantees that the microkernel will be able to load and store F.E. of C using vector instructions.

This preference property is merely an interesting performance detail for conventional implementations (real and complex). However, in the case of 1M, it becomes crucial for constructing a correctly-functioning implementation. More specifically, the microkernel's I/O preference determines whether the 1M_C or 1M_R algorithm is prescribed. Generally speaking, a 1M_C algorithmic variant must employ a microkernel that prefers to access C by columns, while a 1M_R algorithmic variant must use a microkernel that prefers to access C by rows.

3.6.2. Workspace. In some cases, a small amount of $m_R \times n_R$ workspace is needed. These cases fall into one of four scenarios: (1) C is row-stored and the real microkernel RKERN has a column preference; (2) C is column-stored and RKERN has a row preference; (3) C is general-stored (i.e., neither RS(C) nor CS(C) is unit); and (4) $\beta^i \neq 0$. If any of these conditions hold, then the 1M virtual microkernel will need to use workspace. This corresponds to the setting of USEW in VK1M (in Figure 3.2),

[14] This amounts to swapping the row and column strides and swapping the m and n dimensions.

which causes RKERN to compute the micropanel product normally but store it to the workspace W. Subsequently, the result in W is then accumulated back to C.

Cases (1) and (2), while supported, actually never occur in practice because BLIS will perform a logical transposition of the operation, when necessary, so that the storage of C will always appear to match the I/O preference of the microkernel.

Case (3) is needed because the real microkernel is programmed to support the updating of *real* matrices stored with general stride, which cannot emulate the updating of *complex* matrices stored with general stride. The reason is even when stored with general stride, complex matrices use the standard storage format, which interleaves real and imaginary F.E. in contiguous pairs. There is no way to coax this pattern of data access from a real domain microkernel, given its existing API. Thus, general stride support must be implemented outside RKERN, within VK1M.

Case (4) is needed because real domain microkernels are not capable of scaling C by complex scalars β when $\beta^i \neq 0$.

3.6.3. Handling alpha and beta scalars. As in the previous article, we have simplified the general matrix multiplication to $C := C + AB$. In practice, the operation is implemented as $C := \beta C + \alpha AB$, where $\alpha, \beta \in \mathbb{C}$. Let us use Algorithm 1M_C_BP in Figure 3.2 to consider how to support arbitrary values of α and β.

If no workspace is needed (because none of the four situations described in Section 3.6.2 apply), we can simply pass β^r into the RKERN call. However, if workspace *is* needed, then we must pass in a local $\beta_{\text{use}} = 0$ to RKERN, compute to local workspace W, and then apply β at the end of VK1M when W is accumulated to C.

When α is real, the scaling may be performed directly by RKERN. This situation is ideal since it usually incurs no additional costs.[15] Scaling by α with non-zero imaginary components can still be performed by the packing function when either $\tilde{A}_i$ or $\tilde{B}_p$ are packed. Though somewhat less than ideal, the overhead incurred by this treatment of α is minimal in practice since packing is a memory-bound operation.

3.6.4. Multithreading. As with Algorithm 4M_1A in the previous article, Algorithms 1M_C_BP and 1M_R_BP parallelize in a straightforward manner for multicore and many-core environments. Because these algorithms encode the 1M method entirely within the packing functions and the virtual microkernel, all other levels of code are completely oblivious to, and therefore unaffected by, the specifics of the new algorithms. Therefore, we expect that 1M_C_BP and 1M_R_BP will yield multithreaded performance that is on-par with that of RMMBP.

3.6.5. Bypassing the virtual microkernel. Because the 1M virtual microkernel serves as a function wrapper to the real domain microkernel, it incurs additional overhead. Thankfully, there exists a simple workaround, one that is viable as long as $\beta^i = 0$ and C is either row- or column-stored (but not general-stored). If these conditions are met, the real domain *macro*kernel can be called with modified parameters to induce the equivalent complex domain subproblem. This optimization allows the virtual microkernel (and its associated overhead) to be avoided entirely.

Because this optimization relies only on $\beta \in \mathbb{R}$ and row- or column storage of C, it may be applied automatically at runtime to the vast majority of use cases.

3.7. Other complex storage formats. The 1M method was developed specifically to facilitate performance on complex matrices stored using the standard storage format required by the BLAS. This interleaved storage convention for real and imag-

[15] This is because many microkernels multiply their intermediate AB product by α unconditionally

TABLE 4.1
Register and cache blocksizes used by various BLIS implementations of matrix multiplication, as configured for an Intel Xeon E5-2690 v3 "Haswell" processor

Precision/Domain	Implementation	m_R^z	n_R^z	m_C^z	k_C^z	n_C^z
single complex	BLIS 1M_C	16/2	6	144/2	256/2	4080
	BLIS 1M_R	6	16/2	144	256/2	4080/2
	BLIS assembly (c)	8	3	56	256	4080
	BLIS assembly (r)	3	8	75	256	4080
double complex	BLIS 1M_C	8/2	6	72/2	256/2	4080
	BLIS 1M_R	6	8/2	72	256/2	4080/2
	BLIS assembly (c)	4	3	44	256	4080
	BLIS assembly (r)	3	4	192	256	4080

Note: For 1M implementations, division by 2 is made explicit to allow the reader to quickly see both the complex blocksize values as well as the values that would be used by the underlying real domain microkernels when performing real matrix multiplication. The I/O preference of the assembly-based implementations is indicated by a "(c)" or "(r)" (for column- or row-preferring).

inary values is ubiquitous within the community and therefore implicitly assumed. However, some applications may be willing to tolerate API changes that would allow storing a complex matrix X as two separate real matrices X^r and X^i.For those applications, the best an induced method may hope to do is implement each specialized complex matrix multiplication in terms of *two* real domain matrix multiplications—since there are two real matrices that must be updated. Indeed, there exists a variant of the 1M method, which we call the 2M method, that targets updating a matrix C that separates (entirely or by blocks) its real and imaginary F.E. [19].

4. Performance.

In this section we present performance results for implementations of 1M algorithms on a recent Intel architecture. For comparison, we include results for a key 4M algorithm as well as those of conventional assembly-based approaches in the real and complex domains.

4.1. Platform and implementation details.

Results presented in this section were gathered on a single Cray XC40 compute node consisting of two 12-core Intel Xeon E5-2690 v3 processors featuring the "Haswell" microarchitecture. Each core, running at a clock rate of 3.2 GHz[16], provides a single-core peak performance of 51.2 gigaflops (GFLOPS) in double precision and 102.4 GFLOPS in single precision.[17] Each socket has a 30MB L3 cache that is shared among cores, and each core has a private 256KB L2 cache and 32KB L1 (data) cache. Performance experiments were gathered under the Cray Linux Environment 6 operating system running the Linux 4.4.103 (x86_64) kernel. Source code was compiled by the GNU C compiler (`gcc`) version 7.3.0.[18] The version of BLIS used in these tests was not officially released at

[16] This system uses Intel's Turbo Boost 2.0 dynamic frequency throttling technology. According to [14], the maximum the clock frequency when executing AVX instructions is 3.2 GHz when utilizing one or two cores, and 3.0 GHz when utilizing three or more cores.

[17] Accounting for the reduced AVX clock frequency, the peak performance when utilizing 24 cores is 48 GFLOPS/core in double precision and 96 GFLOPS/core in single precision.

[18] The following optimization flags were used during compilation of BLIS and its test drivers: `-O3 -mavx2 -mfma -mfpmath=sse -march=haswell`.

the time of this writing, and was adapted from version 0.6.0-11.[19]

Algorithms 1M_C_BP and 1M_R_BP were implemented in the BLIS framework as described in Section 3.4. We also refer to results based on existing conventional assembly-based microkernels written by hand (via GNU extended inline assembly syntax) for the Haswell microarchitecture.

All experiments were performed on randomized, column-stored matrices with GEMM scalars held constant: $\alpha = \beta = 1$. In all performance graphs, each data point represents the best of three trials.

Blocksizes for each of the BLIS implementations tested are provided in Table 4.1.

In all graphs presented in this section, the x-axes denote the problem size, the y-axes show observed floating-point performance in units of GFLOPS per core, and the theoretical peak performance coincides with the top of each graph.

4.2. Sequential results. Figure 4.1 reports performance results for various implementations of double- and single-precision complex matrix multiplication on a single core of the Haswell processor. For these results, all matrix dimensions were equal (e.g. $m = n = k$). Results for 1M_C_BP (which uses a column-preferring microkernel) appears on the left of Figure 4.1 while those of 1M_R_BP (which uses a row-preferring microkernel) appears on the right.

Each graph in Figure 4.1 also contains three reference implementations: BLIS's complex GEMM based on conventional assembly-coded kernels (e.g. "cgemm assembly"); BLIS's real GEMM (e.g. "sgemm assembly"); and the 4M_1A implementation found in BLIS.[20] We configured all three of these reference codes to use column-preferential microkernels on the left and row-preferential microkernels on the right, as indicated by a "(c)" or "(r)" in the legends, in order to provide consistency with the 1M results.

As predicted in Section 3.5, we find that the performance signatures of the 1M_C_BP and 1M_R_BP algorithms differ slightly. This was expected given that the 1E and 1R packing formats place different memory access burdens on different packed matrices, $\tilde{A}_i$ and $\tilde{B}_p$, which reside in different levels of cache. It was not previously clear, however, which would be superior over the other. It seems that, at least in the sequential case, the difference is somewhat more noticeable in double-precision, though even there it is quite subtle. This difference is almost certainly due to the individual performance characteristics of the underlying row- and column-preferential microkernels. We find evidence of this in the 4M_1A results, which was also affected by the change in microkernel I/O preference.

In all cases, the 1M implementations outperform 4M_1A, with the margin somewhat larger in single-precision.

The 1M implementations match or exceed the performance of their real domain GEMM benchmarks (the dashed lines in each graph) and are quite competitive with assembly-coded complex GEMM (the solid lines) regardless of the algorithm employed.

Finally, the curious reader may recall our brief hypothetical discussion of executing Algorithm 4M_HW on a split complex storage format from Section 2.2 and wonder where such an implementation would fall relative to the measured performance data. Since Algorithm 4M_HW on a split format would mimic the execution of four unrelated

[19] Despite not yet having an official version number, this version of BLIS may be uniquely identified, with high probability, by the first 10 digits of its `git` "commit" (SHA1 hash) number: `ceee2f973e`.

[20] Within any given graph of Figures 4.1 and 4.2, the 1M and 4M_1A implementations use the same real-domain microkernel as that of the real GEMM (e.g. "sgemm assembly" or "dgemm assembly").

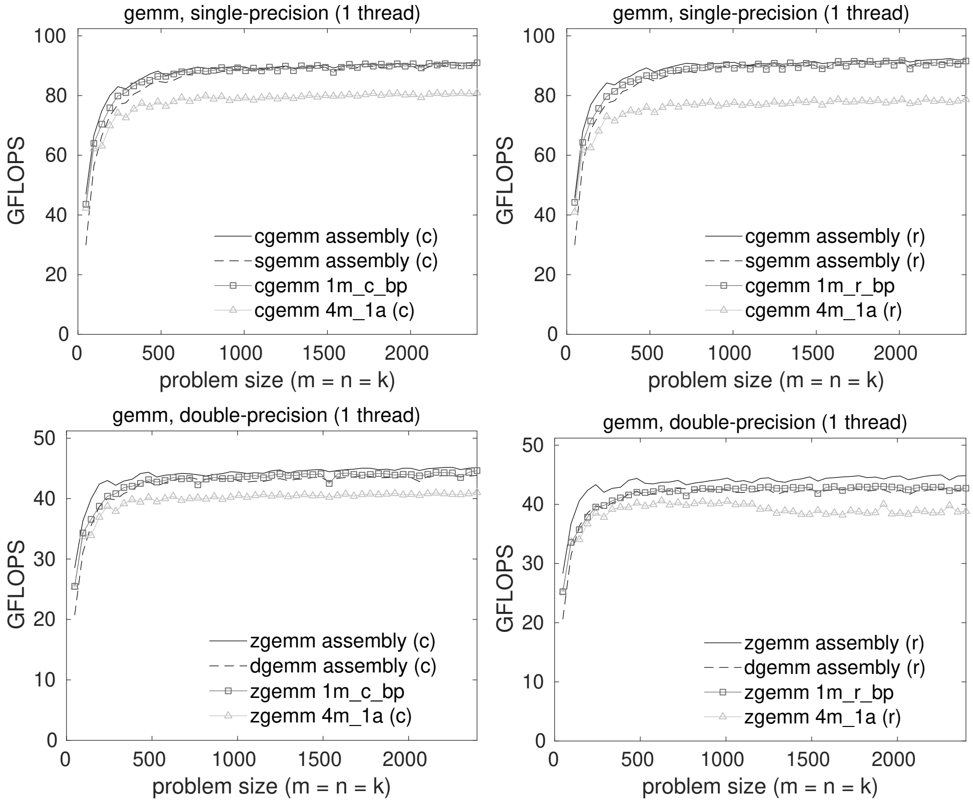

Fig. 4.1. *Single-threaded performance of various implementations of single-precision (top) and double-precision (bottom) complex* gemm *on a single core of an Intel Xeon E5-2690 v3 "Haswell" processor. The left and right graphs differ in which* 1m *implementation they report, with the left graphs reporting* 1m_c_bp *(which employs a column-preferring microkernel) and the right graphs reporting* 1m_r_bp *(which employs a row-preferring microkernel). The graphs also contain three reference curves for comparison: an assembly-coded complex* gemm*, an assembly-coded real* gemm*, and the* 4m_1a *implementation found in BLIS (with the latter two using the same microkernel as the* 1m *implementation shown in the same graph). For consistency with the* 1m *curves, these reference implementations differ from left to right graphs in the I/O preference of their underlying microkernel, indicated by a "(c)" or "(r)" (for column- or row-preferring) in the legends. The theoretical peak performance coincides with the top of each graph.*

real matrix multiplications, its performance would track nearly identically with that of the real domain gemm.

4.3. Multithreaded results. Figure 4.2 shows single- and double-precision performance using 24 threads, with one thread bound to each physical core of the processor. Performance is presented in units of gigaflops per core to facilitate visual assessment of scalability. For all BLIS implementations, we employed 4-way parallelism within the 5th loop, 3-way parallelism within the 3rd loop, and 2-way parallelism in the 2nd loop for a total of 24 threads. This parallelization scheme was chosen in a manner consistent with that of the previous article using a strategy set forth in [18].

Compared to the single-threaded case, we find a more noticeable difference in multithreaded performance between the 1m algorithms. Specifically, the 1m_r_bp implementation (based on a row-preferring microkernel) outperforms that of 1m_c_bp (based on a column-preferring microkernel), with the difference more pronounced

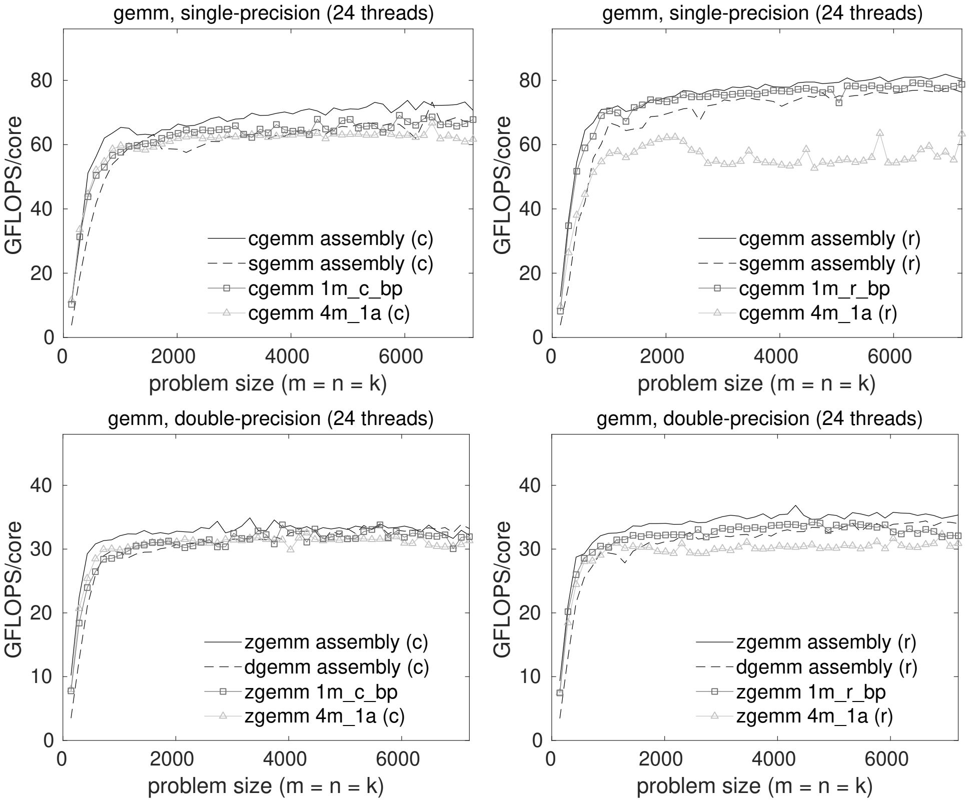

Fig. 4.2. *Multithreaded performance of various implementations of single-precision (top) and double-precision (bottom) complex* GEMM *on two Intel Xeon E5-2690 v3 "Haswell" processors, each with 12 cores. All data points reflect the use of 24 threads. The left and right graphs differ in which* 1M *implementation they report, with the left graphs reporting* 1M_C_BP *(which employs a column-preferring microkernel) and the right graphs reporting* 1M_R_BP *(which employs a row-preferring microkernel). The graphs also contain three reference curves for comparison: an assembly-coded complex* GEMM*, an assembly-coded real* GEMM*, and the* 4M_1A *implementation found in BLIS (with the latter two using the same microkernel as the* 1M *implementation shown in the same graph). For consistency with the* 1M *curves, these reference implementations differ from left to right graphs in the I/O preference of their underlying microkernel, indicated by a "(c)" or "(r)" (for column- or row-preferring) in the legends. The theoretical peak performance coincides with the top of each graph.*

in single-precision. We suspect this is rooted not in the algorithms *per se* but in the differing microkernel implementations used by each 1M algorithm. The 1M_R_BP algorithm uses a real microkernel that is 6×16 and 6×8 in the single- and double-precision cases, respectively, while 1M_C_BP uses 16×6 and 8×6 microkernels for single- and double-precision implementations, respectively. The observed difference in performance between the 1M algorithms is likely attributable to the fact that the microkernels' different values for m_R and n_R place different latency and bandwidth requirements when reading F.E. from the caches (primarily L1 and L2). More specifically, larger values of m_R place a heavier burden on loading elements from the L2 cache, which is usually disadvantageous since that cache may exhibit higher latency and/or lower bandwidth. By contrast, a microkernel with larger n_R loads more elements (per $m_R \times n_R$ rank-1 update) from the L1 cache, which resides closer to the

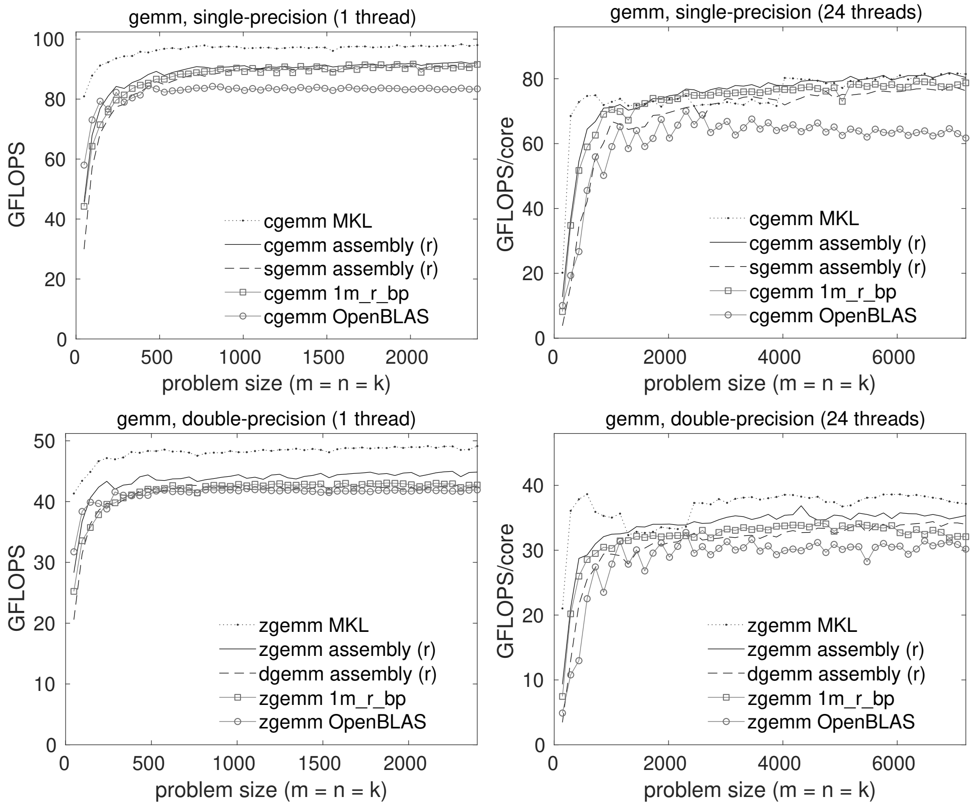

Fig. 4.3. *Single-threaded (left) and multithreaded (right) performance of various implementations of single-precision (top) and double-precision (bottom) complex* gemm *on a single core (left) or 12 cores (right) of an Intel Xeon E5-2690 v3 "Haswell" processor. All multithreaded data points reflect the use of 24 threads. The* 1m *curves are identical from those shown in Figures 4.1 and 4.2. The theoretical peak performance coincides with the top of each graph.*

processor and offers lower latency and/or higher bandwidth than the L2 cache.

The multithreaded 1m implementation approximately matches or exceeds its real domain counterpart in all cases.

The 1m algorithm based on a row-preferential microkernel, 1m_r_bp, outperforms 4m_1a, especially in single-precision where the margin is quite wide. The 1m algorithm based on column-preferential microkernels, 1m_c_bp, performs more poorly, barely edging out 4m_1a in single precision and tracking closely with 4m_1a in double precision. We suspect that 4m_1a is more resilient to the lower-performing column-preferential microkernel by virtue of the fact that the algorithm's virtual microkernel leans heavily on the L1 cache, which on this architecture is capable of being read from and written to at relatively high bandwidth (64 bytes/cycle and 32 bytes/cycle, respectively) [13].

4.4. Comparing to other implementations. While our primary goal is not to compare the performance of the newly developed 1m implementations with that of other established BLAS solutions, some basic comparison is merited and thus we have included Figure 4.3 (left). These graphs are similar to those in Figure 4.1, except that: we show only implementations based on row-preferential microkernels;

we omit 4M_1A; and we include results for complex GEMM implementations provided by OpenBLAS 0.3.6 [16] and Intel MKL 2019 Update 4 [12].

Figure 4.3 (right) shows multithreaded performance of the same implementations running with 24 threads.

These graphs show that BLIS's complex assembly-based and 1M implementations typically outperform OpenBLAS while falling short in most (but not all) cases when compared to Intel's MKL library.

4.5. Additional results. Additional performance results were gathered on a Marvell ThunderX2 compute server as well as an AMD EPYC (Zen) system. For brevity, we present and discuss that data in the appendix available online as supplementary materials. Those results reinforce the narrative provided here, lending even more evidence that the 1M method is capable of yielding high-performance implementations of complex matrix multiplication that are competitive with (and often outperform) other leading library solutions.

5. Observations.

5.1. 4m limitations circumvented. The previous article concluded by identifying a number of limitations inherent in the 4M method. We now revisit this list and briefly discuss whether, to what degree, and how those limitations are overcome by algorithms based on the 1M method.

Number of calls to primitive. The most versatile 4M algorithm, 4M_1A, incurs up to a four-fold increase in function call overhead over a comparable assembly-based implementation. By comparison, 1M algorithms require at most a doubling of microkernel function call overhead, and in certain common cases (e.g., when $\beta \in \mathbb{R}$ and C is row- or column-stored), this overhead can be avoided completely. The 1M method is a clear improvement over 4M due to its one-to-one substitution of the matrix multiplication primitive.

Inefficient reuse of input data from A, B, and C. The most cache-efficient application of 4M is the lowest level algorithm, 4M_1A, which reuses F.E. of A, B, and C from the L1 cache. But, as shown in Table 3.3, both 1M_R and 1M_C variants reuse F.E. of two of the three matrices from registers, with 1M_R_BP reusing F.E. of the third matrix from the L1 cache.

Non-contiguous output to C. Algorithms based on the 4M method must update only the real and then only the imaginary parts of the output matrix, twice each. When C is stored (by rows or columns) in the standard format, with real and imaginary F.E. interleaved, this piecemeal approach prevents the real microkernel from using vector load and store instructions on C during those four updates. The 1M method avoids this issue altogether by packing A and B to formats that allow the real microkernel to update contiguous real and imaginary F.E. of C simultaneously.

Reduction of k_C. Algorithm 4M_1A requires that the real microkernel's preferred k_C blocksize be halved in the complex algorithm in order to maintain proper cache footprints of $\tilde{A}_i$ and $\tilde{B}_p$ as well the footprints of their constituent micropanels.[21] Using these sub-optimally sized micropanels can noticeably hobble the performance of 4M_1A. Looking back at Table 3.1, it may seem like 1M suffers a similar handicap;

[21] Recall that the halving of k_C for 4M_1A was motivated by the desire to keep not just two, but four real micropanels in the L1 cache simultaneously. These correspond to the real and imaginary parts of the current micropanels of $\tilde{A}_i$ and $\tilde{B}_p$.

however, the reason for halving k_C and its effect are both completely different. In the case of 1M, the use of $k_C^z = \frac{1}{2} k_C$ is simply a conversion of units (complex elements to real F.E.) for the purposes of identifying the size of the complex submatrices to be packed that will induce the optimal k_C value from the perspective of the real microkernel, *not* a reduction in the F.E. footprint of the micropanels operated upon by that real microkernel. The ability of 1M to achieve high performance when $k = \frac{1}{2} k_C$ is actually a strength for certain higher-level applications, such as Cholesky, LU, and QR factorizations based on rank-k update. Those operations tend to perform better when the algorithmic blocksize (corresponding to k_C) is as narrow as possible in order to limit the amount of computation in the lower-performing unblocked subproblem.

Framework accommodation. The 1M algorithms are no more disruptive to the BLIS framework than the most accommodating of 4M algorithms, 4M_1A. This is because, like with 4M_1A, almost all of the 1M implementation details are sequestered within the packing routines and the virtual microkernel.

Interference with multithreading. Because the 1M algorithms are implemented entirely within the packing routines and virtual microkernel, they parallelize just as easily as the most thread-friendly of the 4M algorithms, 4M_1A, and entirely avoid the threading difficulties of higher-level 4M algorithms.[22]

Non-applicability to two-operand operations. Certain higher-level applications of 4M are inherently incompatible with two-operand operations because they would overwrite the original contents of the input/output operand even though subsequent stages of computation depend on that original input. 1M avoids this limitation entirely. Like 4M_1A, 1M can easily be applied to two-operand level-3 operations such as TRMM and TRSM.[23]

5.2. Summary. The analysis above suggests that the 1M method solves or avoids most of the performance-degrading weaknesses of 4M and in the remaining cases is no worse off than the best 4M algorithm.

5.3. Limitations of 1m. Although the 1M method avoids most of the weaknesses inherent to the 4M method, a few notable caveats remain.

Non-real values of beta. In the most common cases where $\beta^i = 0$, the 1M implementation may employ the optimization described in Section 3.6.5. However, when $\beta^i \neq 0$, the virtual microkernel must be called. In such cases, 1M yields slightly lower performance due to extra memops.[24]

Algorithmic dependence on I/O preference. If the real domain microkernel is row-preferential (and thus performs row-oriented I/O on C), then the 1M implementation must choose an algorithm based on the 1M_R variant. But (in this scenario), if 1M_C is instead preferred for some reason, then either the underlying microkernel needs to be updated to handle both row- and column-oriented I/O, or a new column-preferential microkernel must be written. A similar caveat holds if the real domain microkernel is column-preferential and the 1M_R variant is preferred.

[22] This thread-friendly property holds even when the virtual microkernel is bypassed altogether as discussed in Section 3.6.5

[23] As with 4M_1A, 1M support for TRSM requires a separate pair of virtual microkernels that fuse a matrix multiplication with a triangular solve with n_R right-hand sides.

[24] The 4M method suffers lower performance when $\beta^i \neq 0$ for similar reasons.

Higher bandwidth on $\tilde{A}_i$ and $\tilde{B}_p$. Compared to a conventional, assembly-based GEMM, implementations based on the 1M method require twice as much memory bandwidth when reading packed matrices $\tilde{A}_i$ and $\tilde{B}_p$. Microkernels that encode complex arithmetic at the assembly level are able to load real and imaginary F.E. and then reuse those F.E. from registers, thus increasing the microkernel's arithmetic intensity. By contrast, the 1M method's reliance on real domain microkernels means that it must reuse real and imaginary F.E. from some level of cache and thus incur additional memory traffic.[25] The relative benefit of the conventional approach is likely to be most visible when parallelizing GEMM across all cores of a many-core system since that situation tends to saturate memory bandwidth.

5.4. Further discussion. Before concluding, we offer some final thoughts on the 1M method and its place in the larger spectrum of approaches to implementing complex matrix multiplication.

5.4.1. Geometric interpretation. Matrix multiplication is sometimes thought of as a three-dimensional operation with a contraction (accumulation) over the k dimension. This interpretation carries into the complex domain as well. However, when each complex element is viewed in terms of its real and imaginary components, we find that a fourth pseudo-dimension of computation (of fixed size 2) emerges, one which also involves a contraction. The 1M method reorders and duplicates elements of A and B in such a way that exposes and "flattens" this extra dimension of computation. This, combined with the exposed treatment of real and imaginary F.E., causes the resulting floating-point operations to appear indistinguishable from a real domain matrix multiplication with m and k dimensions (for column-stored C) or k and n dimensions (for row-stored C) that are twice as large.

5.4.2. Data reuse: efficiency vs. programmability. Both the conventional approach and 1M move data efficiently through the memory hierarchy.[26] However, once in registers, a conventional complex microkernel reuses those loaded values to perform twice as many flops as 1M. The previous article observes that all 4M algorithms make different variations of the same tradeoff: by forgoing the reuse of F.E. from registers and instead reusing those data from some level of cache, the algorithms avoid the need to explicitly encode complex arithmetic at the assembly level. As it turns out, 1M makes a similar tradeoff, but gives up less while gaining more: it is able to effectively reuse F.E. from two of the three matrix operands from registers while still avoiding the need for a complex microkernel, and it manages to replace that kernel operation with a single real matrix multiplication. And we would argue that increasing programmability and productivity by forfeiting a modest performance advantage is a good trade to make under almost any circumstance.

5.4.3. Storage. The supremacy of the 1M method is closely tied to the interleaved storage of real and imaginary values—specifically, of the output matrix C. If applications instead store complex matrices with their real and imaginary components split into two separate real matrices, the 4M approach (for numerically sensitive settings) as well as low-level applications of 3M (for numerically insensitive settings) may become more appropriate [19, 21].

[25] The 4M method suffers the same "bandwidth penalty" as 1M for the same reason.

[26] This is in contrast to, for example, Algorithm 4M_HW, which the previous article showed makes rather inefficient use of cache lines as they travel through the L3, L2, and L1 caches.

6. Conclusions. We began the article by reviewing the general motivations for induced methods for complex matrix multiplication as well as the specific methods, 3M and 4M, studied in the previous article. Then, we recast complex scalar multiplication (and accumulation) in such a way that revealed a template that could be used to fashion a new induced method, one that casts complex matrix multiplication in terms of a single real matrix product. The key is the application of two new packing formats on the left- and right-hand matrix product operands that allows us to disguise the complex matrix multiplication as a real matrix multiplication with slightly modified input parameters. This 1M method is shown to have two variants, one each favoring row-stored and column-stored output matrices. When implemented in the BLIS framework, competitive performance was observed for 1M algorithms on three modern microarchitectures. Finally, we reviewed the limitations of the 4M method that are overcome by 1M and concluded by discussing a few high-level observations.

The key takeaway from our study of induced methods is that the real and imaginary elements of complex matrices can always be reordered to accommodate the desired fundamental primitives, whether those primitives are defined to be various forms of real matrix multiplication (as is the case for the 4M, 3M, 2M, and 1M methods), or vector instructions (as is the case for microkernels that implement complex arithmetic in assembly code). Indeed, even in the real domain, the classic matrix multiplication algorithm's packing format is simply a reordering of data that targets the fundamental primitive implicit in the microkernel—namely, an $m_R \times n_R$ rank-1 update. The family of induced methods presented here and in the previous article expand upon this basic reordering so that the mathematics of complex arithmetic can be expressed at different levels of the algorithm and of its corresponding implementation, each yielding different benefits, costs, and performance.

Acknowledgements. We thank the Texas Advanced Computing Center for providing access to the the Intel Xeon "Lonestar5" (Haswell) compute node on which the performance data presented in Section 4 were gathered. We also kindly thank Marvell and Oracle Corporation for arranging access to the Marvell ThunderX2 and AMD EPYC (Zen) systems, respectively, on which the performance data presented in Appendix A were gathered. Finally, we thank Devangi Parikh for helpfully gathering the results on the ThunderX2 system.

REFERENCES

[1] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff, *A set of level 3 basic linear algebra subprograms*, ACM Trans. Math. Soft., 16 (1990), pp. 1–17.

[2] G. Frison, D. Kouzoupis, T. Sartor, A. Zanelli, and M. Diehl, *BLASFEO: Basic linear algebra subroutines for embedded optimization*, ACM Trans. Math. Soft., 44 (2018), pp. 42:1–42:30, https://doi.org/10.1145/3210754, http://doi.acm.org/10.1145/3210754.

[3] K. Goto and R. A. van de Geijn, *Anatomy of high-performance matrix multiplication*, ACM Trans. Math. Soft., 34 (2008), pp. 12:1–12:25, https://doi.org/10.1145/1356052.1356053, http://doi.acm.org/10.1145/1356052.1356053.

[4] K. Goto and R. A. van de Geijn, *High-performance implementation of the level-3 BLAS*, ACM Trans. Math. Soft., 35 (2008), pp. 4:1–4:14, https://doi.org/10.1145/1377603.1377607, http://doi.acm.org/10.1145/1377603.1377607.

[5] J. A. Gunnels, G. M. Henry, and R. A. van de Geijn, *A family of high-performance matrix multiplication algorithms*, in Proceedings of the International Conference on Computational Sciences-Part I, ICCS '01, Berlin, Heidelberg, 2001, Springer-Verlag, pp. 51–60, http://dl.acm.org/citation.cfm?id=645455.653765.

[6] N. J. Higham, *Stability of a method for multiplying complex matrices with three real matrix multiplications*, SIAM J. Matrix Anal. App., 13 (1992), pp. 681–687, https://doi.org/10.1137/0613043, https://doi.org/10.1137/0613043.

[7] J. Huang, *Practical fast matrix multiplication algorithms*, (2018). PhD thesis, The University of Texas at Austin.

[8] J. Huang, D. A. Matthews, and R. A. van de Geijn, *Strassen's algorithm for tensor contraction*, SIAM Journal on Scientific Computing, 40 (2018), pp. C305–C326, https://doi.org/10.1137/17M1135578, https://doi.org/10.1137/17M1135578, https://arxiv.org/abs/https://doi.org/10.1137/17M1135578.

[9] J. Huang, L. Rice, D. A. Matthews, and R. A. van de Geijn, *Generating families of practical fast matrix multiplication algorithms*, in 31th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2017), May 2017, pp. 656–667, https://doi.org/10.1109/IPDPS.2017.56.

[10] J. Huang, T. M. Smith, G. M. Henry, and R. A. van de Geijn, *Strassen's algorithm reloaded*, in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '16, Piscataway, NJ, USA, 2016, IEEE Press, pp. 59:1–59:12, http://dl.acm.org/citation.cfm?id=3014904.3014983.

[11] J. Huang, C. D. Yu, and R. A. van de Geijn, *Implementing Strassen's algorithm with CUTLASS on NVIDIA Volta GPUs*, FLAME Working Note #88, TR-18-08, The University of Texas at Austin, Department of Computer Science, 2018, https://apps.cs.utexas.edu/apps/sites/default/files/tech_reports/GPUStrassen.pdf.

[12] Intel, *Math Kernel Library*. https://software.intel.com/en-us/mkl, 2019.

[13] Intel Corporation, *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, no. 248966-033, June 2016.

[14] Intel Corporation, *Intel® Xeon® Processor E5 v3 Product Family: Processor Specification Update*, no. 330785-010US, September 2016.

[15] T. M. Low, F. D. Igual, T. M. Smith, and E. S. Quintana-Ortí, *Analytical modeling is enough for high-performance BLIS*, ACM Trans. Math. Soft., 43 (2016), pp. 12:1–12:18, https://doi.org/10.1145/2925987, http://doi.acm.org/10.1145/2925987.

[16] *OpenBLAS*. http://xianyi.github.com/OpenBLAS/, 2019.

[17] D. T. Popovici, F. Franchetti, and T. M. Low, *Mixed data layout kernels for vectorized complex arithmetic*, in 2017 IEEE High Performance Extreme Computing Conference (HPEC), Sep. 2017, pp. 1–7, https://doi.org/10.1109/HPEC.2017.8091024.

[18] T. M. Smith, R. A. van de Geijn, M. Smelyanskiy, J. R. Hammond, and F. G. Van Zee, *Anatomy of high-performance many-threaded matrix multiplication*, in Proceedings of the 28th IEEE International Parallel & Distributed Processing Symposium (IPDPS), IPDPS '14, Washington, DC, USA, 2014, IEEE Computer Society, pp. 1049–1059, https://doi.org/10.1109/IPDPS.2014.110, https://doi.org/10.1109/IPDPS.2014.110.

[19] F. G. Van Zee, *Inducing complex matrix multiplication via the 1m method*, FLAME Working Note #85 TR-17-03, The University of Texas at Austin, Department of Computer Sciences, February 2017.

[20] F. G. Van Zee, T. Smith, F. D. Igual, M. Smelyanskiy, X. Zhang, M. Kistler, V. Austel, J. Gunnels, T. M. Low, B. Marker, L. Killough, and R. A. van de Geijn, *The BLIS framework: Experiments in portability*, ACM Trans. Math. Soft., 42 (2016), pp. 12:1–12:19, http://doi.acm.org/10.1145/2755561.

[21] F. G. Van Zee and T. M. Smith, *Implementing high-performance complex matrix multiplication via the 3m and 4m methods*, ACM Trans. Math. Soft., 44 (2017), pp. 7:1–7:36.

[22] F. G. Van Zee and R. A. van de Geijn, *BLIS: A framework for rapidly instantiating BLAS functionality*, ACM Trans. Math. Soft., 41 (2015), pp. 14:1–14:33, http://doi.acm.org/10.1145/2764454.

[23] R. C. Whaley, A. Petitet, and J. J. Dongarra, *Automated empirical optimization of software and the ATLAS project*, Parallel Computing, 27 (2001), pp. 3–35, https://doi.org/https://doi.org/10.1016/S0167-8191(00)00087-9, http://www.sciencedirect.com/science/article/pii/S0167819100000879. New Trends in High Performance Computing.

[24] C. D. Yu, J. Huang, W. Austin, B. Xiao, and G. Biros, *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, New York, NY, USA, 2015, ACM, pp. 7:1–7:12, https://doi.org/10.1145/2807591.2807601, http://doi.acm.org/10.1145/2807591.2807601.

Appendix A. Additional Performance Results.

In this section we present performance results for implementations of 1M method on two additional types of hardware. The primary purpose of gathering these results was to confirm 1M performance on additional architectures beyond the Intel Haswell system reported on in the main article.

A.1. Marvell ThunderX2.

In this section, we report the performance of the 1M method on the Marvell ThunderX2, a high-performance ARMv8 microarchitecture.

A.1.1. Platform and implementation details.

Results presented in this section were gathered on a single compute node consisting of two 28-core Marvell ThunderX2 CN9975 processors.[27] Each core, 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 socket has a 32MB L3 cache that is shared among cores, and each core has a private 256KB L2 cache and 32KB L1 (data) cache. Performance experiments were gathered under the Ubuntu 16.04 operating system running the Linux 4.15.0 kernel. Source code was compiled by the GNU C compiler (`gcc`) version 7.3.0.[28] The version of BLIS used in these tests was version 0.5.0-1.[29]

In this section, we show 1M results for only Algorithm 1M_C_BP. Unlike the results shown in the main article, we did not develop conventional assembly-based microkernels and thus cannot compare against a complex domain solution based on those kernels. For further comparison, we measured performance for the complex GEMM implementations found in OpenBLAS[30] and ARMPL 18.4.0.

All other parameters, such as values of α and β, and the number of trials performed for each problem size, as well as graphing conventions, such as scaling of the y-axis, remain identical to those of the main article.

A.1.2. Analysis.

Figure A.1 contains single-threaded (left) and multithreaded (right) performance of single-precision (top) and double-precision (bottom) complex GEMM implementations. In addition to the 1M_C_BP implementation within BLIS, we also show the corresponding real domain GEMM implementation and the `cgemm` or `zgemm` found in OpenBLAS and ARMPL. For all BLIS implementations, we employed 4-way parallelism within the 5th loop and 14-way parallelism within the 3rd loop for a total of 56 threads.

In Figure A.1 (top-left), single-precision 1M and its corresponding real domain benchmark track each other closely in the single-threaded configurations tested, as we would have expected. Somewhat surprisingly, the vendor library, ARMPL, does not appear to scale well at 56 threads, as shown in Figure A.1 (top-right). Also somewhat surprisingly, OpenBLAS performance is consistently low, even for sequential execution. This suggests that while parallelism may be well-configured, their kernel is likely underperforming.

[27] While four-way symmetric multithreading is 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.

[28] The following optimization flags were used during compilation of BLIS and its test drivers: `-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`.

[29] This version of BLIS may be uniquely identified, with high probability, by the first 10 digits of its `git` "commit" (SHA1 hash) number: e90e7f309b.

[30] This version of OpenBLAS may be uniquely identified, with high probability, by the first 10 digits of its `git` commit number: `52d3f7af50`.

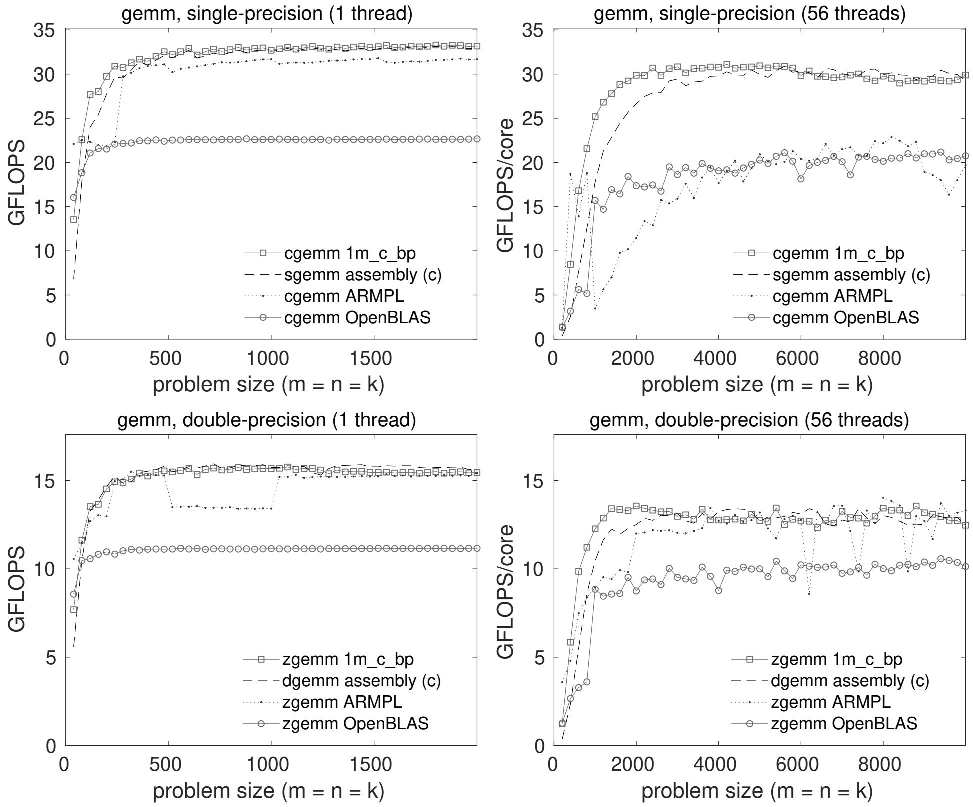

Fig. A.1. *Single-threaded (left) and multithreaded (right) performance of various implementations of single-precision (top) and double-precision (bottom) complex* GEMM *on a single core (left) or 56 cores (right) of a Marvell ThunderX2 CN9975 processor. All multithreaded data points reflect the use of 56 threads. The real domain* GEMM *implementation from BLIS uses a column-preferential microkernel, as indicated the a "(c)" in the legends. (The* 1M_C_BP *implementation uses the same column-preferential microkernel as the real domain* GEMM *implementation.) The theoretical peak performance coincides with the top of each graph.*

Figure A.1 (bottom) tells a similar story of performance among double-precision implementations, except that all BLIS implementations are, for reasons not immediately obvious, somewhat less efficient relative to peak performance than their single-precision counterparts. ARMPL performance is more competitive for both one and 56 threads, though the single-core graph exposes evidence of a "crossover point" strategy gone awry. ARMPL also seems to exhibit large swings in performance for certain large, multithreaded problem sizes. Once again, OpenBLAS performance is much lower, but consistently so.

In summary, BLIS's 1M implementation performs extremely well on the Marvell CN9975 when computing in single precision. Performance and scalability in double precision, while not quite as impressive, is still highly competitive, especially when compared to OpenBLAS and the ARM Performance Library.

A.2. AMD Zen. In this section, we report the performance of the 1M method on the AMD Zen microarchitecture.

A.2.1. Platform and implementation details. Results presented in this section were gathered on a single compute node consisting of two 32-core AMD EPYC 7551 (Zen) processors.[31] Each core runs at a clock rate of 3.0 GHz when using a single core and 2.55 GHz when utilizing all cores simultaneously. The former clock rate yields a single-core peak performance of 24.0 GFLOPS in double precision and 48.0 GFLOPS in single precision, and the latter clock rate yields a multicore peak performance of 20.4 GFLOPS/core and 40.8 GFLOPS/core for single- and double-precision computation, respectively. Each socket has a 64MB of L3 cache (distributed as 8MB for each four-core complex) that is shared among cores, and each core has a private 512KB L2 cache and 32KB L1 (data) cache. Performance experiments were gathered under the Ubuntu 18.04 operating system running the Linux 4.15.0 kernel. Source code was compiled by the GNU C compiler (`gcc`) version 7.4.0.[32] The version of BLIS used in these tests was version 0.6.0-1266.[33]

In this section, we show 1M results for only Algorithm 1M_R_BP. For reference, we also measured performance for the complex GEMM implementations found in OpenBLAS 0.3.7 and Intel MKL 2020 (initial release).

All other parameters, such as values of α and β, and the number of trials performed for each problem size, as well as graphing conventions, such as scaling of the y-axis, remain identical to those of the main article.

A.2.2. Analysis. Figure A.2 contains single-threaded (left) and multithreaded (right) performance of single-precision (top) and double-precision (bottom) complex GEMM implementations. In addition to the 1M_R_BP implementation within BLIS, we also show the corresponding real and complex domain GEMM implementations based on conventional assembly-coded kernels. We also show the `cgemm` or `zgemm` found in OpenBLAS and MKL. For all BLIS implementations, we employed 2-way parallelism within the 5th loop, 8-way parallelism within the 3rd loop, and 4-way parallelism within the 2nd loop for a total of 64 threads.

In Figure A.2 (top-left), all implementations track closely together except for MKL.[34] We see a similar pattern for single-threaded double precision in Figure A.2 (bottom-left).

In Figure A.2 (top-right) and (bottom-right), we see multithreaded performance when utilizing all 64 cores of the AMD EPYC system. The relative performance of 1M_R_BP is consistent with the results seen previously on Haswell. That is, the 1M method facilitates performance that meets or exceeds the performance of an optimized real domain implementation of GEMM (i.e., one that uses the same microkernels as 1M), but falls slightly short of the performance of a conventional assembly-coded complex domain GEMM. Once again, MKL performance suffers noticeably on AMD hardware.[35] OpenBLAS lags somewhat behind the BLIS-based implementations, but

[31] While two-way symmetric multithreading is available on this hardware, a maximum of one logical core per physical core was utilized during our tests.

[32] The following optimization flags were used during compilation of BLIS and its test drivers: `-O3 -march=znver1`. Furthermore, all test drivers were run via `numactl -i all`.

[33] This version of BLIS may be uniquely identified, with high probability, by the first 10 digits of its `git` "commit" (SHA1 hash) number: f391b3e2e7.

[34] We hypothesize that as MKL parses the results of the CPUID instruction, it detects an unexpected CPU vendor (AMD instead of Intel) and therefore selects a "fallback" (safe but low-performing) kernel. If this is the case, then the fix would be trivial, which suggests that MKL's underperformance on AMD hardware is deliberate.

[35] In order to keep the legends in Figure A.2 readable, the curves for MKL in were clipped beyond the first 20 data points. In all four graphs, the omitted data points depict a plateauing of the curve that is consistent with the data that is shown.

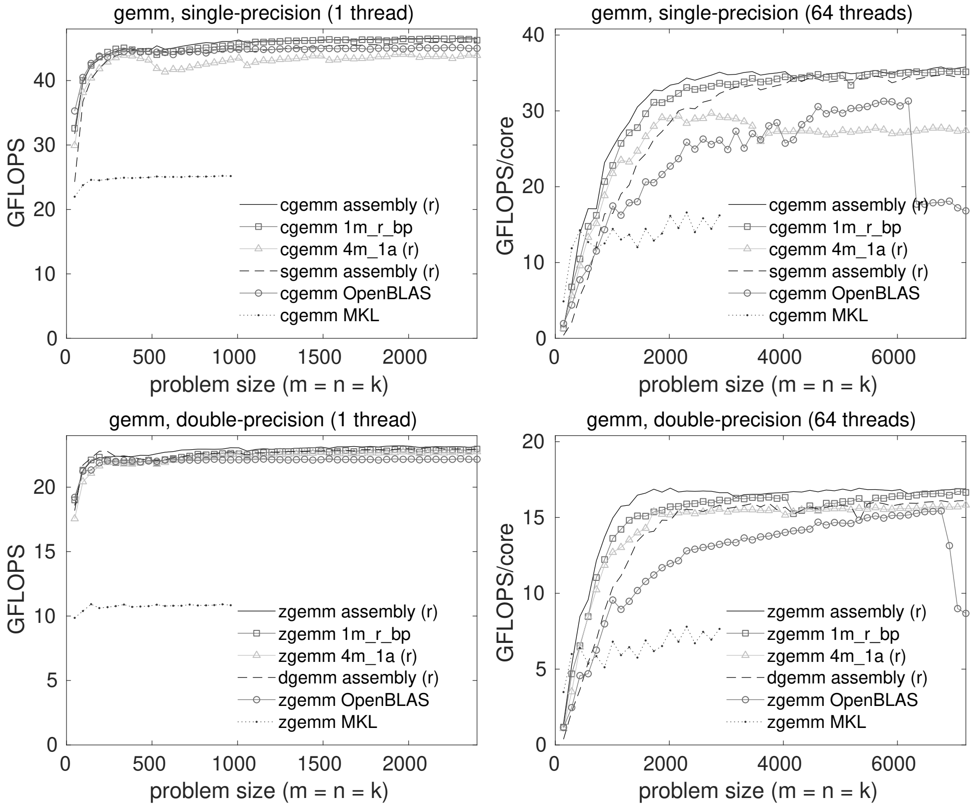

FIG. A.2. *Single-threaded (left) and multithreaded (right) performance of various implementations of single-precision (top) and double-precision (bottom) complex* GEMM *on a single core (left) or 64 cores (right) of an AMD EPYC 7551 (Zen) processor. All multithreaded data points reflect the use of 64 threads. The real and complex domain* GEMM *implementations from BLIS use row-preferential microkernels, as indicated the a "(r)" in the legends. (The* 1M_R_BP *implementation uses the same row-preferential microkernel as the real domain* GEMM *implementation.) The theoretical peak performance coincides with the top of each graph.*

performance unexpectedly drops for very large problem sizes. This behavior was reproducible, though the exact problem size at which the drop-off occurred shifted across repeated experiments.

In summary, BLIS's 1M implementation performs very well on the AMD EPYC 7551 when computing in single and double precision, exceeding the performance of both OpenBLAS and MKL. Scalability (relative to theoretical peak) is also quite good in both precisions considering the challenges that NUMA-based architectures sometimes pose to parallelization efforts.