Implementing high-performance complex matrix multiplication via the 1m method

FIELD G. VAN ZEE, The University of Texas at Austin

In this article, we continue exploring the topic of so-called induced methods for implementing complex matrix multiplication. Previous work investigated two approaches for computing matrix products in the complex domain using only a real matrix multiplication kernel. However, 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 benchmarks. 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. Our derivation also naturally exposes a symmetry that allows the method to perform well when updating either column- or row-stored matrices. Applying the method to two general algorithms for matrix multiplication yields a family of algorithmic variants, each with a unique set of circumstantial affinities. Further analysis suggests 1M will match or exceed the performance of a real matrix multiplication based on the same kernel, especially for certain rank-k updates. Implementations are developed within the BLIS framework, which facilitates their extension to all level-3 operations. Testing on a recent Intel microarchitecture confirms that the 1M method yields performance that is competitive with solutions based on conventionally implemented complex kernels.

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

General Terms: Algorithms, Performance

Additional Key Words and Phrases: linear algebra, DLA, high-performance, complex, matrix, multiplication, micro-kernel, kernel, BLAS, BLIS, 1m, 2m, 3m, 4m, induced

ACM Reference Format:
Field G. Van Zee. 2016. Implementing high-performance complex matrix multiplication via the 1m method *ACM Trans. Math. Softw.* 0, 0, Article 0 (2017), 29 pages.
DOI: http://dx.doi.org/xx.xxxx/xxxxxxx

1. INTRODUCTION

Dense matrix multiplication—the foundation of many dense linear algebra operations—is now ubiquitous within scientific and numerical computing applications. For three decades, libraries that provide the Basic Linear Algebra Subprograms (BLAS) [Dongarra et al. 1990] have exported common interfaces to specific implementations of matrix multiplication and related functionality. However, before a user can employ these highly-tuned functions, a library developer with knowledge of the target hardware must first implement the operations in question.

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).
Authors' addresses: Field G. Van Zee, Department of Computer Sciences and Institute for Computational Engineering and Sciences, The University of Texas at Austin, Austin, TX 78712, field@cs.utexas.edu.
 0098-3500/2017/-ART0 $15.00
DOI: http://dx.doi.org/xx.xxxx/xxxxxxx

Some projects seek to provide ready-made solutions while others focus on streamlining the development process so that third parties may rapidly instantiate dense linear algebra functionality on existing and new hardware [OpenBLAS 2018; Whaley et al. 2000; Van Zee and van de Geijn 2015]. In either case, the basic formula is the same: a developer carefully writes a small computational kernel, usually in assembly (or a low-level, assembly-like) language, which is then inserted into a larger infrastructure of portable code. Within dense linear algebra (DLA) library development circles, it is taken for granted that one matrix multiplication kernel is needed for each floating-point precision and domain to be supported. Thus, to support all four combinations of the single- and double-precision in the real and complex domains, four kernels must be authored to enable a complete set of matrix multiplication implementations.

Many real-world applications, high-performance benchmarks, and pedagogical settings rely only on computation in the real domain. Thus, providing support for real matrix multiplication is understandably a priority. However, DLA—and by proxy, matrix multiplication—in the complex domain is essential for many fields and applications, in part because of complex numbers' unique ability to encode both the phase and magnitude of a wave.

Unfortunately, those who seek to provide support for complex matrix operations must wrestle with additional challenges. Namely, most modern hardware lacks machine instructions for directly computing complex arithmetic on complex numbers. Instead, the library developer, or kernel author, must orchestrate computation on the real and imaginary components explicitly in order to implement multiplication and addition on complex scalars, which in many cases proves to be a more difficult programming task than in the real domain. Furthermore, for maintainers of BLAS, and BLAS-like library frameworks such as BLIS, supporting complex matrix multiplication doubles the number of assembly-coded matrix kernels that must be maintained. In other words, life would be simpler for DLA library developers if complex matrix multiplication was not necessary.

Of course, complex matrix multiplication will always be necessary. But what if complex matrix multiplication *kernels* were found to be unnecessary?[1] To certain actors, particularly DLA library developers, such a finding would carry non-trivial consequence.

Recent work investigates whether (and with what degree of effectiveness) real domain matrix multiplication kernels can be repurposed and leveraged toward the implementation of complex matrix multiplication [Van Zee and Smith 2017]. The authors develop a new class of algorithms that implement these so-called "induced methods" for matrix multiplication 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.[2]

We consider this article a companion and follow-up to this previous work, to which we will often refer [Van Zee and Smith 2017]. For this reason, and for the sake of brevity, we omit much of the typical review of the literature on dense matrix multiplication, which we provide in aforementioned article. We also assume the reader has

[1] Here, it is worth emphasizing that solutions based on complex kernels will always be necessary to some individuals so long as those kernels facilitate superior performance under at least some circumstances. That said, others, such as developers with limited time and resources, will be happy to trade away a modest performance advantage if it greatly increases their productivity. Thus, we acknowledge that precisely defining necessity in this context amounts to a somewhat subjective value judgment, and will vary according to the audience being considered.

[2] In [Van Zee and Smith 2017], 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.

a basic understanding of the background provided by this previous article, either because he or she has already read the piece, or because he or she naturally possesses this familiarity independent of our work. Of course, we will review and summarize a reasonable amount of content specific to that article here, as needed, in order to properly set the stage for our present discussion.

1.1. Contributions

This article makes the following contributions:

— It introduces a new induced method—the 1M method—that relies upon only a single real matrix multiplication. As with the previous article, we introduce a family of algorithms and analyze issues germane to their high-performance implementations, including workspace, packing formats, cache behavior, multithreadability, and programming effort. A detailed review shows how 1M avoids virtually all of the 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 programmers to focus their efforts on fewer and simpler kernels.
— It builds on the theme of the BLIS framework as a productivity multiplier [Van Zee and van de Geijn 2015], 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 multiplication found within the BLIS framework, which is available to the community under an open source software license.[3]

We believe that 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, to say nothing of the potential impact on developer productivity.

1.2. Notation

In this article, we continue the notation established in [Van Zee and Smith 2017]. 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. $A = (A_0 \,|\, A_1 \,|\, \cdots \,|\, A_{n-1})$) or scalars within a larger matrix or vector.

We also 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 themselves 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 en-

[3]The BLIS framework is available under the so-called "new" or "modified" or "3-clause" BSD license.

code how the matrices are actually *stored*. We will explicitly address storage details as implementation issues are discussed.

Also, 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 fundamental elements and the latter to be a load or store operation on a single fundamental element. 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 [Van Zee and van de Geijn 2015; Van Zee et al. 2016; Van Zee and Smith 2017]. 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 also discusses and references 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 [Van Zee and Smith 2017], 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 [Van Zee et al. 2016].
— **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 [Van Zee and van de Geijn 2015]. 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 [Van Zee and Smith 2017] 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 $\chi, \psi, \zeta \in \mathbb{C}$:

$$\begin{aligned} \zeta^r &:= \zeta^r + \chi^r \psi^r - \chi^i \psi^i \\ \zeta^i &:= \zeta^i + \chi^r \psi^i + \chi^i \psi^r \end{aligned} \tag{1}$$

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:

$$\begin{aligned} C^r &:= C^r + A^r B^r - A^i B^i \\ C^i &:= C^i + A^r B^i + A^i B^r \end{aligned} \tag{2}$$

This definition expresses 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:

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

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 [Van Zee and Smith 2017] 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. The blocked algorithm used in that article is shown in Figure 1 (left) and explained in detail in Section 2.1 of [Van Zee and Smith 2017] 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 exceeded the theoretical peak rate of the hardware.[4] 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 [Higham 1992] found that 3M was stable enough for most practical purposes, many applications simply will not be willing to stray from the numerical expectations implicit in conventional matrix multiplication. Thus, going forward, we will turn our attention away from 3M and instead focus on the 4M as the standard reference method against which we will compare.

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.

— 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 fundamental elements of matrices A and

[4]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.

B (or rather, F.E. of the packed copies of these matrices, $\tilde{A}_i$ and $\tilde{B}_p$) are reused[5]. 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 $\tilde{B}_p$ unchanged. The level from which F.E. are reused is determined by the level of the matrix multiplication algorithm to which the 4M method was originally applied. The lower the 4M method is applied, the higher the efficiency of data reuse from and movement through the cache hierarchy.

— 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, with virtually no excess movement due to incidental cache line proximity, and reuses F.E. from the L1 cache. It relies on a relatively simple packing format in which complex micro-panels are stored with real and imaginary F.E. separated into two consecutive real micro-panels, each with identical register blocksize and k dimensions. Algorithm 4M_1A requires negligible workspace (approximately equal to the storage capacity of the vector register set), is well-suited for multithreading, and is minimally disruptive to the encoding within the BLIS framework. And while algorithm 4M_1B—a slightly higher-level application—very narrowly outperforms 4M_1A by trading away the most optimal cache reuse behavior for an unreduced k_C cache blocksize, 4M_1A proves to be more versatile and can be extended relatively easily to all other level-3 operations.

— The conventional approach 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, typically, then optimally encoded via vector instructions).

— The way complex numbers are stored has a significant effect on performance. Interleaved pair-wise storage of real and imaginary values naturally favors implementations that reuse F.E. from vector registers, as is common with conventional implementations.[6] However, this storage is awkward for algorithms based on 4M (and 3M for that matter), largely because it prevents the use of vector instructions for loading and storing F.E. of C[7], as these instructions implicitly must load or store several *consecutive* values in memory. Indeed, 4M_1A suffers from a *quadrupling*[8] of the number of memops on C, in addition to being forced to access these F.E. in a non-contiguous manner. If, however, applications stored complex matrices with real and imaginary parts separated, the penalty paid by 4M (and 3M) would be partially mitigated.

— While observed performance of low-level applications of 4M is decent, far exceeding an unoptimized reference implementation, it not only falls short of a comparable conventional solution based on a complex kernel, it appears to fall 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. This level of performance may be disappointing for some, even if it achieves 90-95% of what is possible with a complex kernel. The authors conclude that its somewhat attenuated performance would relegate 4M, in practice, to serving mostly as a placeholder, to be used when complex kernels have not yet been written, rather than a competitive replacement that makes complex kernels unnecessary.

[5]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, not the reuse of whole (complex) elements that naturally occurs in the execution of the matrix multiplication algorithm in Figure 1 (left).

[6]The advantages of interleaving data in advance of computation via vector instructions is also discussed by the authors of [Dongarra et al. 2017] in the context of performing "batched" level-3 operations.

[7]The traditional pair-wise storage is also awkward for 4M algorithms during the packing of data from A and B, but this effect is not nearly as dramatic.

[8]A factor of two comes from the fact that, as shown in Eq. 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 micro-panels of $\tilde{A}_i$ and $\tilde{B}_p$.

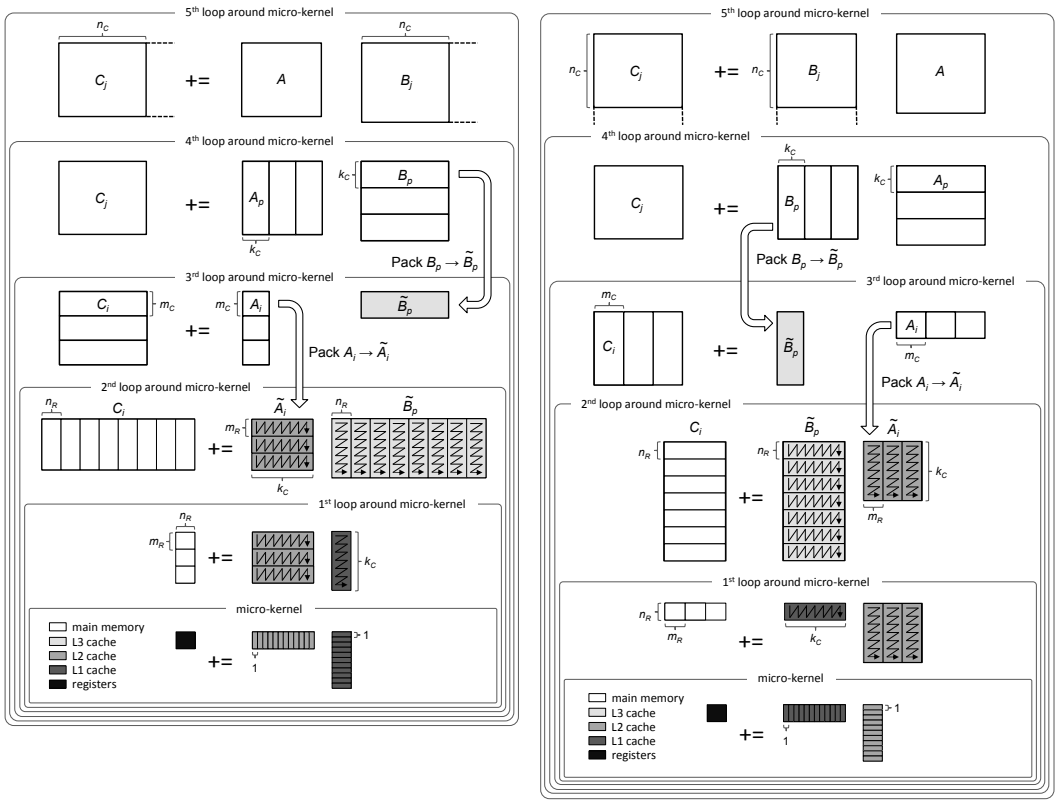

Fig. 1. Left: An illustration of the algorithm for computing high-performance matrix multiplication, taken from [Van Zee and Smith 2017], which expresses computation in terms of a so-called "block-panel" subproblem (macro-kernel). Right: An alternate algorithm that expresses the operation in terms of "panel-block" matrix multiplication.

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 [Goto and van de Geijn 2008] and further refined in [Van Zee and van de Geijn 2015]. Figure 1 (left) 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 micro-kernel 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 micro-kernel 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 micro-kernel loop, an $m_R \times 1$ micro-column vector and a $1 \times n_R$ micro-row vector are loaded from the current

micro-panels 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 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 (left) 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.[9] 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 [Low et al. 2016] present an analytical model for identifying suitable (if not optimal) values for these blocksizes.

Later in this article, we will observe that it may be desirable in some situations to have access to a companion algorithm that casts its largest cache-bound subproblem (computed by the macro-kernel) in terms of panel-block multiplication. This alternative algorithm is depicted in Figure 1 (right).[10] Either algorithm can be used to implement all of the level-3 operations defined by the original BLAS or the BLIS framework, including the most popular and general-purpose operation, general matrix multiplication (GEMM).[11]

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 individually and in separate steps (due to traditional pair-wise storage), and, in the case of 4M_1A, twice as frequently (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. 1, refactored and expressed in terms of matrix and vector notation, where we use $+=$ operator to concisely denote element-wise accumulation:

$$\begin{pmatrix} \zeta^r \\ \zeta^i \end{pmatrix} += \begin{pmatrix} \chi^r & -\chi^i \\ \chi^i & \chi^r \end{pmatrix} \begin{pmatrix} \psi^r \\ \psi^i \end{pmatrix} \tag{3}$$

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 $n = 1$.

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

[9] This terminology describes the shape of the typical problem computed by the macro-kernel, i.e. the second loop around the micro-kernel.

[10] We renamed the matrices (and indices) in the panel-block algorithm in Figure 1 (right) so that B is the left-hand matrix product operand while A appears on the right. This allows $\tilde{A}_i$ and $\tilde{B}_p$ to remain the matrices that reside in the L2 and L3 caches, respectively.

[11] Throughout this document, we will sometimes interchangeably use the term GEMM to refer to matrix multiplication.

Thus, Eq. 3 serves as a packing template that hints at how the data must be stored, which we may generalize. We replace χ, ψ, ζ with α, β, γ so as to more intuitively denote complex elements of matrices A, B, and C, respectively. Also, let us apply the Eq. 3 to the special case of $m = 3$, $n = 4$, and $k = 2$ to better observe the general pattern.

$$\begin{pmatrix} \gamma_{00}^r & \gamma_{01}^r & \gamma_{02}^r & \gamma_{03}^r \\ \gamma_{00}^i & \gamma_{01}^i & \gamma_{02}^i & \gamma_{03}^i \\ \gamma_{10}^r & \gamma_{11}^r & \gamma_{12}^r & \gamma_{13}^r \\ \gamma_{10}^i & \gamma_{11}^i & \gamma_{12}^i & \gamma_{13}^i \\ \gamma_{20}^r & \gamma_{21}^r & \gamma_{22}^r & \gamma_{23}^r \\ \gamma_{20}^i & \gamma_{21}^i & \gamma_{22}^i & \gamma_{23}^i \end{pmatrix} \mathrel{+}= \begin{pmatrix} \alpha_{00}^r & -\alpha_{00}^i & \alpha_{01}^r & -\alpha_{01}^i \\ \alpha_{00}^i & \alpha_{00}^r & \alpha_{01}^i & \alpha_{01}^r \\ \alpha_{10}^r & -\alpha_{10}^i & \alpha_{11}^r & -\alpha_{11}^i \\ \alpha_{10}^i & \alpha_{10}^r & \alpha_{11}^i & \alpha_{11}^r \\ \alpha_{20}^r & -\alpha_{20}^i & \alpha_{21}^r & -\alpha_{21}^i \\ \alpha_{20}^i & \alpha_{20}^r & \alpha_{21}^i & \alpha_{21}^r \end{pmatrix} \begin{pmatrix} \beta_{00}^r & \beta_{01}^r & \beta_{02}^r & \beta_{03}^r \\ \beta_{00}^i & \beta_{01}^i & \beta_{02}^i & \beta_{03}^i \\ \beta_{10}^r & \beta_{11}^r & \beta_{12}^r & \beta_{13}^r \\ \beta_{10}^i & \beta_{11}^i & \beta_{12}^i & \beta_{13}^i \end{pmatrix} \tag{4}$$

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 micro-kernel, and we assume that matrices A and B above represent column-stored and row-stored micro-panels from $\tilde{A}_i$ and $\tilde{B}_p$, respectively, and also that the dimensions are conformal to the register blocksizes of this micro-kernel (i.e., $m = m_R$ and $n = n_R$) then the micro-panels 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 $m_R \times k_C$ micro-panel that the real GEMM micro-kernel expects.
— Similarly, the micro-panels 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 $k_C \times n_R$ micro-panel that the real GEMM micro-kernel expects.

It is easy to see by inspection that the real matrix multiplication implied by Eq. 4 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 and then swapped, with the imaginary component 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).

3.2. Two variants

Notice that implicit in the 1M method suggested by Eq. 4 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 micro-kernel to correctly update the individual real and imaginary F.E. of C with the corresponding F.E. from the matrix product AB.

Suppose, for a moment, that we instead refactored and expressed Eq. 1 as follows:

$$\begin{pmatrix} \zeta^r & \zeta^i \end{pmatrix} \mathrel{+}= \begin{pmatrix} \chi^r & \chi^i \end{pmatrix} \begin{pmatrix} \psi^r & \psi^i \\ -\psi^i & \psi^r \end{pmatrix} \tag{5}$$

This gives us a different template, one that implies different packing formats for matrices A and B. Applying Eq. 5 to the special case of $m = 4$, $n = 3$, and $k = 2$, yields:

$$\begin{pmatrix} \gamma_{00}^r & \gamma_{00}^i & \gamma_{01}^r & \gamma_{01}^i & \gamma_{02}^r & \gamma_{02}^i \\ \gamma_{10}^r & \gamma_{10}^i & \gamma_{11}^r & \gamma_{11}^i & \gamma_{12}^r & \gamma_{12}^i \\ \gamma_{20}^r & \gamma_{20}^i & \gamma_{21}^r & \gamma_{21}^i & \gamma_{22}^r & \gamma_{22}^i \\ \gamma_{30}^r & \gamma_{30}^i & \gamma_{31}^r & \gamma_{31}^i & \gamma_{32}^r & \gamma_{32}^i \end{pmatrix} \mathrel{+}= \begin{pmatrix} \alpha_{00}^r & \alpha_{00}^i & \alpha_{01}^r & \alpha_{01}^i \\ \alpha_{10}^r & \alpha_{10}^i & \alpha_{11}^r & \alpha_{11}^i \\ \alpha_{20}^r & \alpha_{20}^i & \alpha_{21}^r & \alpha_{21}^i \\ \alpha_{30}^r & \alpha_{30}^i & \alpha_{31}^r & \alpha_{31}^i \end{pmatrix} \begin{pmatrix} \beta_{00}^r & \beta_{00}^i & \beta_{01}^r & \beta_{01}^i & \beta_{02}^r & \beta_{02}^i \\ -\beta_{00}^i & \beta_{00}^r & -\beta_{01}^i & \beta_{01}^r & -\beta_{02}^i & \beta_{02}^r \\ \beta_{10}^r & \beta_{10}^i & \beta_{11}^r & \beta_{11}^i & \beta_{12}^r & \beta_{12}^i \\ -\beta_{10}^i & \beta_{10}^r & -\beta_{11}^i & \beta_{11}^r & -\beta_{12}^i & \beta_{12}^r \end{pmatrix} \tag{6}$$

Table I. 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 micro-kernels, 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 $n_R < n_P$.

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

Henceforth, we will refer to the 1M variant exemplified in Eq. 4 as 1M_C, since it is predicated on column storage on the output matrix. Similarly, we will refer to the variant depicted in Eq. 6 as 1M_R, since it assumes the output matrix 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, since the idea is to fool the real GEMM micro-kernel, and the various loops for register and cache blocking around the micro-kernel, 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 (and specifically, which matrix is packed according to the 1E format).

Table I summarizes the complex domain blocksizes prescribed for 1M_C and 1M_R as a function of the real domain values. This is somewhat analogous to the blocksize scaling described in Tables II and III in [Van Zee and Smith 2017]. However, in that article the scaling was optional in the sense that different scaling factors would still work, albeit perhaps with a performance penalty. Here, in the case of Table I, some scaling factors (namely, on m_R or n_R) are required in order for the 1M algorithm to function properly.

Those familiar with the matrix multiplication algorithm implemented by the BLIS framework, as depicted in Figure 1 (left), may be unfamiliar with m_P and n_P, the so-called packing dimensions. These values are, effectively, the leading dimensions of the micro-panels. For most architectures, these values are almost always equal to m_R and n_R, respectively. However, 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 1E.

3.4. Algorithms

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

Fig. 2. Abbreviated pseudo-codes for implementing the general matrix multiplication algorithms depicted in Figure 1. Here, RKERN calls a real domain GEMM micro-kernel. Note that the only difference between the algorithms is that the loop bounds on the 5th and 3rd loops around the micro-kernel are swapped. Also, RMMPB labels the matrix product operands differently, with B referring to the left-hand matrix and A referring to the right-hand matrix.

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, we provide pseudo-code, targeting the real domain, for the two algorithms depicted in Figure 1. These algorithms are shown as RMMBP and RMMPB in Figure 2.

Notice that in the context of RMMPB, the 5th and 3rd loops around the micro-kernel (RKERN) iterate over different dimensions, relative to those loops in RMMBP, but the blocksizes used in those loops are the same. Thus, n_C is used to partition in the m dimension while m_C is used to partition in the n dimension. The authors of [Van Zee and van de Geijn 2015] initially studied matrix multiplication only in the context of an algorithm based on block-panel subproblems, which led them to name the blocksizes after the dimensions along which they partitioned. When generalized to support the panel-block algorithm, these blocksizes refer not to a specific dimension but rather to a specific level of cache being targeted. That is, m_C targets the L2 cache while n_C blocks for the L3 cache. However, for historical purposes, we will continue to use the original names throughout this article. We also choose to label the matrix product operands differently in the panel-block setting; the left-hand matrix is named B while the right-hand matrix is named A.[12]

3.4.2. 1m-specific algorithm. When applied to the block-panel algorithm depicted in Figure 1 (left), 1M_C and 1M_R yield nearly identical algorithms whose differences can be encoded within a few conditional statements within key parts of the high and low levels of code. We will refer to these specific algorithms as 1M_C_BP and 1M_R_BP, respectively. Figure 3 shows a hybrid algorithm that encompasses both, supporting row- and column-stored matrices C.

[12]It is important to note that the renaming in panel-block algorithms such as RMMPB does *not* necessarily represent a swapping of the operands, as would happen if the entire operation were transposed to $C^T += B^T A^T$, nor does it represent a change to the operation's interface. RMMBP and RMMPB simply use different names for the left- and right-hand matrices within the *algorithm descriptions*. This renaming allows us to reference the L2-bound block and L3-bound panel as $\tilde{A}_i$ and $\tilde{B}_p$, respectively, in both algorithms.

Algorithm: [C] := 1M_[CR]_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_{use}) $= 1$
if (COLSTORE) CS(C_{use}) $\times= 2$
else RS(C_{use}) $\times= 2$
N(A) $\times= 2$; M(B) $\times= 2$
C_{use} := RKERN(A, B, C_{in})
if (USEW)
  $C := W$

Fig. 3. 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 (left). Here, PACK1E and PACK1R pack matrices into the 1E and 1R formats, respectively. Right: Pseudo-code for a virtual micro-kernel used by all 1M algorithms.

As with the 3M and 4M algorithms in [Van Zee and Smith 2017], we have separated the so-called virtual micro-kernel into a separate function, shown in Figure 3 (right). This 1M-specific virtual micro-kernel, VK1M, largely consists of a call to the real domain micro-kernel RKERN, with some added special case handling and book-keeping needed to properly induce complex matrix multiplication. Some of the details of the virtual micro-kernel will be addressed later.

Note that 1M_C and 1M_R can also be applied to the panel-block algorithm depicted in Figure 1 (right), yielding 1M_C_PB and 1M_R_PB. We omit pseudo-code for these algorithms for the sake of brevity.

3.5. Performance properties and algorithmic pairs

Table II tallies the total number of F.E. memops required by the block-panel and panel-block algorithms of both variants of 1M (1M_C and 1M_R). 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 [Van Zee and Smith 2017]. Note that Algorithms 4M_H, 4M_1B, 4M_1A, and the assembly implementation employ a block-panel algorithm, and therefore their memop counts more closely resemble those of 1M_C_BP and 1M_R_BP.

Notice that 1M_C_BP (and 1M_R_BP) incur additional memops relative to a conventional assembly-based solution.

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 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

Table II. F.E. memops incurred by various algorithms, broken down by stage of computation.

Algorithm[a]	F.E. memops required to ...[b]				
	update micro-tiles[c] 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_PB		$6kn\frac{m}{n_C}$	$4kn\frac{m}{n_R}$	$4mk$	$2mk\frac{2n}{m_C}$
1M_R_BP		$4mk\frac{n}{n_C}$	$2mk\frac{2n}{n_R}$	$6kn$	$4kn\frac{m}{m_C}$
1M_C_PB		$4kn\frac{m}{n_C}$	$2kn\frac{2m}{n_R}$	$6mk$	$4mk\frac{n}{m_C}$

[a] Algorithms 4M_H, 4M_1B, 4M_1A, and the assembly implementation employ a block-panel algorithm, and therefore their memop counts more closely resemble those of 1M_C_BP and 1M_R_BP.

[b] 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 $\frac{n}{n_R}$ (with m and n swapped for panel-block algorithms 1M_C_PB and 1M_R_PB). 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.

[c] As described in Section 3.7.1, $m_R \times n_R$ 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.

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 micro-kernels with different micro-tile shapes (i.e., different values of m_R and n_R), those micro-kernels' differing performance properties will likely cause the performance signatures of 1M_C_BP and 1M_R_BP to deviate further.

Notice that the only difference between the rows for 1M_C_BP and 1M_R_PB is the swapping of the m and n dimensions. This stems from the fact that the block-panel and panel-block algorithms iterate over different dimensions in the 5th and 3rd loops, but are otherwise identical. Indeed, for large enough problems, we expect 1M_R_BP and 1M_C_PB to have the same performance properties because in each algorithm the L2-bound packed matrix $\tilde{A}_i$ is formatted with 1R and the L3-bound $\tilde{B}_p$ is formatted with 1E. These algorithms' similarities sometimes make it convenient to refer to them simultaneously as an algorithmic pair. The 1M_R_BP and 1M_C_PB algorithms form a second such pair, as they share properties that are distinct from the first.

Table III summarizes Table II and also lists the level of the memory hierarchy from which each matrix operand is reused as well as a measure of memory movement efficiency. The information listed for 4M and assembly algorithms is reproduced from Table IV of the previous article.

Notice that transposing the entire operation, and using 1M_R_PB to compute $C^T \mathrel{+}= B^T A^T$ would cause the algorithm's performance properties listed in Table III to align indistinguishably with that of using 1M_C_BP without transposition. A similar symmetry would be observed with 1M_C_PB and 1M_R_BP.

Table III. Performance properties of various algorithms.

Algorithm	Total F.E. memops required (Sum of columns of Table II)	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_{L1}^C	A	l_{L1}^A	B	l_{L1}^B
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_PB	$4mn\left(\frac{2k}{k_C}\right)+2kn\left(\frac{3m}{n_C}+\frac{2m}{n_R}\right)+2mk\left(2+\frac{2n}{m_C}\right)$	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
1M_C_PB	$4mn\left(\frac{2k}{k_C}\right)+2kn\left(\frac{2m}{n_C}+\frac{2m}{n_R}\right)+2mk\left(3+\frac{2n}{m_C}\right)$	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-sensitive details

3.6.1. Micro-kernel storage preference. Within the BLIS framework, micro-kernels are registered with a property that describes their output *preference*—that is, the semantic row or column orientation of the vector registers that load and store micro-tiles of C.[13] Whenever possible, the BLIS framework will perform logical transpositions[14] so that the apparent storage of C matches the preference property of the micro-kernel being used. This guarantees that the micro-kernel 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 rather important for constructing a correctly-functioning implementation. Specifically, the micro-kernel's storage preference determines whether the 1M_C or 1M_R algorithm is prescribed. Generally speaking, a 1M_C algorithmic variant must employ a micro-kernel that prefers to access C by columns, while a 1M_R algorithmic variant must use a micro-kernel that prefers to access C by rows. An exception (or at least a caveat) to this rule is discussed in Section 3.6.3.

3.6.2. The panel-block algorithm. Now that we have established two algorithmic pairs—1M_C_BP/1M_R_PB and 1M_R_BP/1M_C_PB—the question becomes, why choose one algorithm over another within the same pair? It turns out that, at least in the context of 1M, we suspect that the reasons apply to limited scenarios and that the two algorithms are otherwise interchangeable.

[13] Even though micro-kernels are always registered as having a row or column preference for the purposes of accessing C, these kernels still support matrices stored with general stride (e.g. tensors). However, matrix multiplication via 1M should never need to exercise the real micro-kernel's built-in support for general stride. This topic is discussed in Section 3.7.1.
[14] This amounts to a swapping of the row and column strides, and a swapping of the m and n dimensions.

— **Small m or n dimensions.** One reason to consider the panel-block algorithm is that it is somewhat more forgiving of problems where the m dimension is relatively small. To understand this phenomenon, let us instead begin by explaining why the block-panel algorithm can tolerate small n somewhat better than small m.
Notice that, according to Figure 1 (left), micro-panels of $\tilde{B}_p$ typically reside in the L3 cache. Those micro-panels are therefore reused from the L3 cache as they are multiplied against different blocks $\tilde{A}_i$. That is, they are reused from the L3 cache across iterations of the 3rd loop. However, if n is very small, such that $\tilde{B}_p$ consists of only a few micro-panels, then $\tilde{B}_p$ may not fall back to the L3 cache, but rather linger in the portion of the L2 cache that is not occupied by $\tilde{A}_i$. This tends to result in a small performance improvement.
Now, let us consider what happens if m is very small, such that $\tilde{A}_i$ consists of only a few micro-panels. Note that the algorithm is already predisposed to keeping those micro-panels in the L2 cache, and the L1 cache is usually too small to hold even one additional micro-panel of $\tilde{A}_i$. Thus, there exists no cache benefit to small values of m for block-panel algorithms.
The corresponding small dimension affinity for the panel-block algorithm is reversed. That algorithm tolerates small m somewhat better for the same reason that the block-panel algorithm tolerates small n dimensions—that is, because m is the dimension along which the n_C blocksize is applied, which determines the number of micro-panels in $\tilde{B}_p$.
— **Odd register blocksizes.** Usually, m_R and n_R are both even numbers. However, sometimes one or the other is odd.[15] In this case, the variant of 1M is pre-determined. That is, if n_R is odd, then only 1M_C may be used. This stems from the fact that, as shown in Table I, the complex register blocksize n_R^z used during packing of 1M_R must be equal to $\frac{1}{2}n_R$. Obviously, $n_R^z = \frac{1}{2}n_R$ cannot be computed as a whole number if n_R is odd. Similarly, if m_R is odd, then only 1M_R may be used.

Figure 4 illustrates distinguishing details of the four 1M algorithms at the level of the macro-kernel, while Table IV summarizes key properties of each 1M algorithm.

3.6.3. Implementing the panel-block algorithm. At first glance, it may seem that implementing the panel-block algorithms 1M_C_PB and 1M_R_PB would require a significant effort. After all, panel-block algorithms partition the matrix dimensions in a different order, result in a different macro-kernel, and produce different micro-kernel subproblems that multiply $n_R \times k_C$ micro-panels of $\tilde{B}_p$ by $k_C \times m_R$ micro-panels of $\tilde{A}_i$. In fact, this would seem to call for a different micro-kernel altogether.

As it turns out, the panel-block algorithm can be implemented by recycling existing code and leveraging the similarities within algorithmic pairs. Let us consider the pair 1M_C_BP and 1M_R_PB. Note that the micro-kernel for 1M_C_BP as well as higher-level code (including the block-panel macro-kernel) already exist within the BLIS framework. The higher-level code for 1M_R_PB can be implemented in BLIS by building an alternate control tree structure that steps through the loop partitioning functions in

[15]The author has never observed or heard of an optimal or near-optimal case where both register blocksizes were odd. This can be attributed to the fact that the register blocksizes capture the orientation and layout of the vector registers used to accumulate the micro-kernel's matrix product. Since vector register lengths are (so far) universally powers of two, they are also even numbers, and we cannot imagine any benefits to forgoing use of some vector register elements (except perhaps at edge cases). In fact, the analytical model proposed in [Low et al. 2016] explicitly constrains the space of possible register allocations to those where vector registers are fully populated. For these reasons, we anticipate that all reasonable register allocations will have at least one even register blocksize.

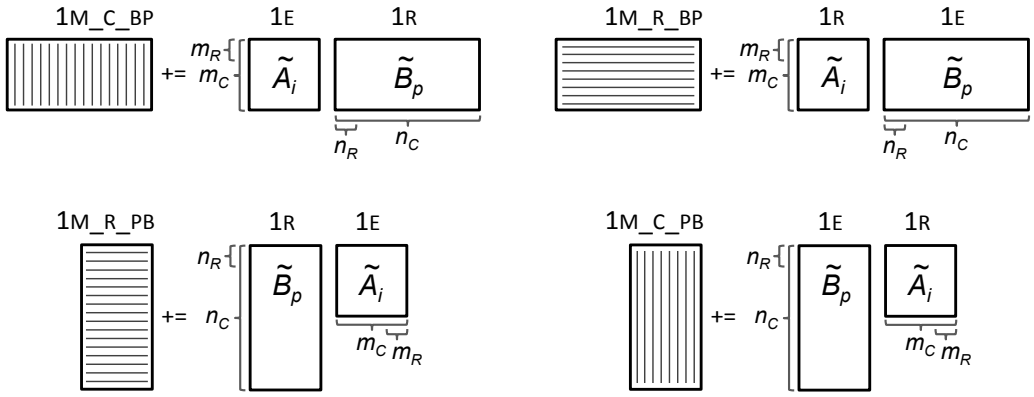

Fig. 4. Illustrations that depict key details of the four 1M algorithms at the level of the macro-kernel, including the 1E/1R packing formats of $\tilde{A}_i$ and $\tilde{B}_p$, whether each packed matrix is a left- or right-hand operand to the matrix product, and the corresponding relationship between the m and n dimensions and the cache and register blocksizes m_C, m_R, n_C, and n_R. In each diagram, the storage of the output matrix prescribed by the 1M_C/1M_R variant is shown as vertical (column-stored) or horizontal (row-stored) lines. The algorithms represented by the diagrams on the left will exhibit similar performance properties when the matrices are large. A similar relationship holds between the pair of algorithms on the right.

Table IV. Summary of key properties of 1M algorithms.

Algorithm	Packing format / Level from which F.E. are reused		Requires μ-kernel with preference for ...	Tolerates μ-kernel with odd ...	Small dimension affinity
	A	B			
1M_C_BP	1E / L2	1R / REG	columns	n_R	n
1M_R_PB[a]	1E / L2	1R / REG	rows[b]	n_R	m
1M_R_BP	1R / REG	1E / L1	rows	m_R	n
1M_C_PB[a]	1R / REG	1E / L1	columns[b]	m_R	m

[a] Recall that panel-block algorithms 1M_R_PB and 1M_C_PB compute $C \mathrel{+}= BA$, with matrix A being multiplied by matrix B from the right.
[b] Note that while 1M_R_PB requires a row-preferential micro-kernel in principle, it does not require one in practice. Instead, we can support 1M_R_PB by simply applying a logical transposition to the entire operation, which allows us to recast the problem in terms of a column-preferential micro-kernel (and 1M_C_BP macro-kernel), as discussed in Section 3.6.3. A similar observation can be made of recycling a row-preferential micro-kernel (and 1M_R_BP macro-kernel) when implementing 1M_C_PB.

a different order.[16] Furthermore, it is possible to implement the panel-block macro-kernel needed by 1M_R_PB in terms of the existing block-panel macro-kernel used by 1M_C_BP. Upon calling the panel-block macro-kernel function, we simply apply a log-

[16] A control tree is a data structure used internally by BLIS to encode the basic properties of level-3 algorithms. It specifies, for example, which dimension to partition, the blocksize to use in that partitioning, and a reference to the next node in the tree, which is processed in the body of the next loop. In general, control trees allow a library developer to factor out certain parameters from the code local to any given loop. These codes become generic and, when stored as separate functions, can then be composed together at runtime into arbitrary compound algorithms.

ical transposition of the three matrix operands while also swapping $\tilde{B}_p$ and $\tilde{A}_i$. This transformation orients C and the micro-panels of the packed matrices into a format and sequence expected by 1M_C_BP's macro-kernel and its column-preferential micro-kernel.

In a similar manner, 1M_C_PB can be implemented by recycling the code and row-preferential micro-kernel used by 1M_R_BP.

Note that in Section 4 (specifically in Figures 5 and 6), we will report performance results in which we refer to panel-block algorithms 1M_C_PB and 1M_R_PB as employing row-preferring and column-preferring micro-kernels, respectively. This reversal comes from the fact that we implement each panel-block macro-kernel in terms of its block-panel counterpart, which, in our present discussion, uses a micro-kernel of opposite preference.

3.7. Algorithm-agnostic details

3.7.1. 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 micro-kernel 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) β has a non-zero imaginary component. If any of these situations apply, then the 1M virtual micro-kernel will need to use workspace. This corresponds to the setting of USEW in VK1M. The idea is simply that RKERN will be called to compute the micro-panel product and store it to the workspace W. Subsequently, the result in W can be accumulated back to C.

Cases (1) and (2), while supported, actually never occur in practice because BLIS will perform (at a high level within the framework) a logical transposition whenever necessary. The net effect is that the storage of C will always appear to match the preference of the micro-kernel.

Case (3) is needed because the real micro-kernel is programmed to support the updating of *real* matrices stored with general stride, which cannot be spoofed to match the updating of *complex* matrices stored with general stride. The reason is even when stored with general stride, complex matrices store real and imaginary F.E. in contiguous pairs. There is no way to coax this pattern of data access from a real domain micro-kernel. Thus, general stride support must be implemented outside RKERN, within VK1M.

Case (4) is needed because real domain micro-kernels are not semantically equipped to scale C by complex scalars β (that is, β such that $\beta^i \neq 0$).

3.7.2. 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 Algorithms 1M_C_BP and 1M_R_BP in Figure 3 as an example of how to support arbitrary values of α and β.

If no workspace is needed (because none of the four situations described in Section 3.7.1 apply), we can simply pass β^r into the RKERN call. However, if β is complex, or (regardless of whether β is real or complex) if any of the other three workspace cases apply, 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 almost always incurs no additional costs (since many micro-kernels multiply their intermediate AB product by α unconditionally). Scaling by α with non-zero imaginary components can 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 since packing is a memory-bound operation.

3.7.3. Multithreading. As with Algorithm 4M_1A in the previous article, 1M_C_BP and 1M_R_BP parallelize in a straightforward manner for multicore and many-core environments. Because those algorithms encode the 1M method entirely within the packing functions and the virtual micro-kernel, 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 the corresponding real domain matrix multiplication function, RMMBP.

A similar analysis holds for panel-block algorithmic variants 1M_C_PB and 1M_R_PB.

3.7.4. Bypassing the virtual micro-kernel. Because the 1M virtual micro-kernel serves as a wrapper to the real domain micro-kernel, it would seem at first glance that there exists the potential for additional overhead in 1M algorithms, particularly from the extra function calls. However, there are a few things to consider.

First, we should consider that a conventional solution would implement matrix multiplication using a complex micro-kernel, which sometimes has a smaller micro-tile footprint (i.e., fewer F.E.). But, a complex micro-kernel that updates fewer F.E. would need to be called more times per rank-k_C update in order to fully update the output matrix. Thus, the function call overhead incurred by 1M algorithms may already be at or near parity with that of a conventional implementation.

Secondly, even if the complex micro-kernel updates the same number of F.E. as its real domain counterpart, there exists a simple optimization that can be applied as long as $\beta \in \mathbb{R}$ and C is either row- or column-stored (but not general-stored). If these conditions are met and detected by the implementation, the real domain *macro*-kernel can be called with modified parameters to induce the equivalent complex domain subproblem. This simple optimization avoids all overhead introduced by the virtual micro-kernel, including (but not limited to) the cost incurred by additional function calls.

Finally, we suspect that, even if this optimization cannot be applied, the slowdown that results from the additional overhead may not necessarily be prohibitive.

3.8. 2m

The previous article noted that because of its expression in terms of real and imaginary matrices, 4M would perform more favorably on complex matrices that stored real and imaginary values separately, as two real matrices. This would not benefit the accesses on A and B since those matrices must almost always be packed to achieve high performance and can easily be separated during that process. However, it would benefit the accesses on C. Referring back to Eq. 4, we can see that storing the real and imaginary F.E. of C separately is equivalent to a permutation of the rows of C. In order to keep the computation expressed unchanged, this permutation would need to be applied to matrix A as well, since they both share an m dimension. Applying such a permutation to Eq. 4 yields:

$$\left(\begin{array}{cccc} \gamma_{00}^r & \gamma_{01}^r & \gamma_{02}^r & \gamma_{03}^r \\ \gamma_{10}^r & \gamma_{11}^r & \gamma_{12}^r & \gamma_{13}^r \\ \gamma_{20}^r & \gamma_{21}^r & \gamma_{22}^r & \gamma_{23}^r \\ \hline \gamma_{00}^i & \gamma_{01}^i & \gamma_{02}^i & \gamma_{03}^i \\ \gamma_{10}^i & \gamma_{11}^i & \gamma_{12}^i & \gamma_{13}^i \\ \gamma_{20}^i & \gamma_{21}^i & \gamma_{22}^i & \gamma_{23}^i \end{array}\right) \mathrel{+}= \left(\begin{array}{cccc} \alpha_{00}^r & -\alpha_{00}^i & \alpha_{01}^r & -\alpha_{01}^i \\ \alpha_{10}^r & -\alpha_{10}^i & \alpha_{11}^r & -\alpha_{11}^i \\ \alpha_{20}^r & -\alpha_{20}^i & \alpha_{21}^r & -\alpha_{21}^i \\ \hline \alpha_{00}^i & \alpha_{00}^r & \alpha_{01}^i & \alpha_{01}^r \\ \alpha_{10}^i & \alpha_{10}^r & \alpha_{11}^i & \alpha_{11}^r \\ \alpha_{20}^i & \alpha_{20}^r & \alpha_{21}^i & \alpha_{21}^r \end{array}\right) \left(\begin{array}{cccc} \beta_{00}^r & \beta_{01}^r & \beta_{02}^r & \beta_{03}^r \\ \beta_{00}^i & \beta_{01}^i & \beta_{02}^i & \beta_{03}^i \\ \beta_{10}^r & \beta_{11}^r & \beta_{12}^r & \beta_{13}^r \\ \beta_{10}^i & \beta_{11}^i & \beta_{12}^i & \beta_{13}^i \end{array}\right) \tag{7}$$

If we also permute even-indexed columns of A to be consecutive with one another and grouped together while odd-numbered columns are permuted to immediately follow

them, and permute rows of B accordingly, we have:

$$\left(\begin{array}{cccc} \gamma^r_{00} & \gamma^r_{01} & \gamma^r_{02} & \gamma^r_{03} \\ \gamma^r_{10} & \gamma^r_{11} & \gamma^r_{12} & \gamma^r_{13} \\ \gamma^r_{20} & \gamma^r_{21} & \gamma^r_{22} & \gamma^r_{23} \\ \hline \gamma^i_{00} & \gamma^i_{01} & \gamma^i_{02} & \gamma^i_{03} \\ \gamma^i_{10} & \gamma^i_{11} & \gamma^i_{12} & \gamma^i_{13} \\ \gamma^i_{20} & \gamma^i_{21} & \gamma^i_{22} & \gamma^i_{23} \end{array}\right) \mathrel{+}= \left(\begin{array}{cc|cc} \alpha^r_{00} & \alpha^r_{01} & -\alpha^i_{00} & -\alpha^i_{01} \\ \alpha^r_{10} & \alpha^r_{11} & -\alpha^i_{10} & -\alpha^i_{11} \\ \alpha^r_{20} & \alpha^r_{21} & -\alpha^i_{20} & -\alpha^i_{21} \\ \hline \alpha^i_{00} & \alpha^i_{01} & \alpha^r_{00} & \alpha^r_{01} \\ \alpha^i_{10} & \alpha^i_{11} & \alpha^r_{10} & \alpha^r_{11} \\ \alpha^i_{20} & \alpha^i_{21} & \alpha^r_{20} & \alpha^r_{21} \end{array}\right) \left(\begin{array}{cccc} \beta^r_{00} & \beta^r_{01} & \beta^r_{02} & \beta^r_{03} \\ \beta^r_{10} & \beta^r_{11} & \beta^r_{12} & \beta^r_{13} \\ \hline \beta^i_{00} & \beta^i_{01} & \beta^i_{02} & \beta^i_{03} \\ \beta^i_{10} & \beta^i_{11} & \beta^i_{12} & \beta^i_{13} \end{array}\right)$$

Or, more generally:

$$\left(\frac{C^r}{C^i}\right) \mathrel{+}= \left(\frac{A^r \,|\, -A^i}{A^i \,|\, A^r}\right) \left(\frac{B^r}{B^i}\right) \tag{8}$$

This matrix multiplication can be computed via just *two* calls to a real domain matrix multiplication primitive:

$$C^r \mathrel{+}= \left(\, A^r \,|\, -A^i \,\right) \left(\frac{B^r}{B^i}\right), \;\; C^i \mathrel{+}= \left(\, A^i \,|\, A^r \,\right) \left(\frac{B^r}{B^i}\right)$$

provided that $\left(\, A^r \,|\, -A^i \,\right)$, $\left(\, A^i \,|\, A^r \,\right)$, and $\left(\frac{B^r}{B^i}\right)$ can each be stored so that they can be referenced as single matrices. We call this the 2M method and refer to the specific instance derived from Eq. 8 as 2M_C.

Similarly, applying permutations to the n and k dimensions to Eq. 6 yields:

$$\left(\, C^r \,|\, C^i \,\right) \mathrel{+}= \left(\, A^r \,|\, A^i \,\right) \left(\frac{B^r \,|\, B^i}{-B^i \,|\, B^r}\right) \tag{9}$$

which can also be broken down in two instances of real matrix multiplication:

$$C^r \mathrel{+}= \left(\, A^r \,|\, A^i \,\right) \left(\frac{B^r}{-B^i}\right), \;\; C^i \mathrel{+}= \left(\, A^r \,|\, A^i \,\right) \left(\frac{B^i}{B^r}\right)$$

which corresponds to 2M_R.

We can now make a few observations about 2M:

— Eqs. 8 and 9 are identical to Eqs. 3 and 5, respectively, except that scalars are replaced with matrices.
— The permutation along the k dimension is actually unnecessary, and so the formatting captured by Eq. 7 also falls under 2M. This permutation (or lack thereof) only changes the order in which intermediate terms are accumulated.
— The storage of C^r and C^i in both Eqs. 8 and 9 is unspecified and does not depend on which matrix, A or B, was originally formatted with 1E (prior to permutation). Either C^r or C^i may be stored by rows, columns, or a more exotic storage scheme, and their storage formats need not even be identical. Thus, neither 2M_C nor 2M_R implies the storage of C; rather, they only imply how the A and B matrices are formatted and stored—that is, which one contains duplicated (and negated) F.E..
— The 2M method can be applied at an arbitrary level of matrix multiplication. For example, if we assume from Eq. 8 that input matrices $\left(\, A^r \,|\, -A^i \,\right)$, $\left(\, A^i \,|\, A^r \,\right)$, and $\left(\frac{B^r}{B^i}\right)$ are each stored as micro-panels (column-stored, column-stored, and row-stored, respectively), then this application of 2M prescribes that C is stored by contiguous $m_R \times n_R$ micro-tiles. If, instead, the aforementioned input matrices were generally

Table V. Register and cache blocksizes used by various 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	$144/2$	$256/2$	2040
	BLIS 1M_R	6	$16/2$	144	$256/2$	$2040/2$
	BLIS assembly	3	8	144	256	2040
	OpenBLAS	8	2	384	192	$?^a$
double complex	BLIS 1M_C	$8/2$	6	$72/2$	$256/2$	2040
	BLIS 1M_R	6	$8/2$	72	$256/2$	$2040/2$
	BLIS assembly	3	4	72	256	2040
	OpenBLAS	4	2	256	128	$?^a$

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 micro-kernels when performing real matrix multiplication.
[a] We were unable to confidently determine the n_C^z blocksizes used by OpenBLAS for the Haswell architecture.

large, then C would need to be stored in whatever manner is appropriate for the real matrix multiplication primitive.

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 4M algorithms as well as 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[17], provides a single-core peak performance of 51.2 gigaflops (GFLOPS) in double precision and 102.4 GFLOPS in single precision.[18] 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 SuSE 11 operating system running the Linux 3.0.101 (x86_64) kernel. Source code was compiled by the GNU C compiler (`gcc`) version 5.2.0.[19] The version of BLIS used in these tests was not officially released at the time of this writing, and was adapted from version 0.2.1-85.[20]

Algorithms 1M_C_BP, 1M_C_PB, 1M_R_BP, and 1M_R_PB were implemented in the BLIS framework, as described in Sections 3.4 through 3.7. We also refer to results based on existing conventional, assembly-based micro-kernels written by hand for the Haswell microarchitecture via GNU extended inline assembly syntax.

All experiments were performed on randomized, column-stored matrices with GEMM scalars held constant: $\alpha = 2$ and $\beta = 1$. In all performance graphs, each data point represents the best of three trials.

[17]This system uses Intel's Turbo Boost 2.0 dynamic frequency throttling technology. According to [Intel Corporation 2016b], 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.
[18]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.
[19]The following optimization flags were used during compilation: `-O3 -mavx2 -mfma -mfpmath=sse -march=core-avx2`.
[20]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: `1c732d3ddc`.

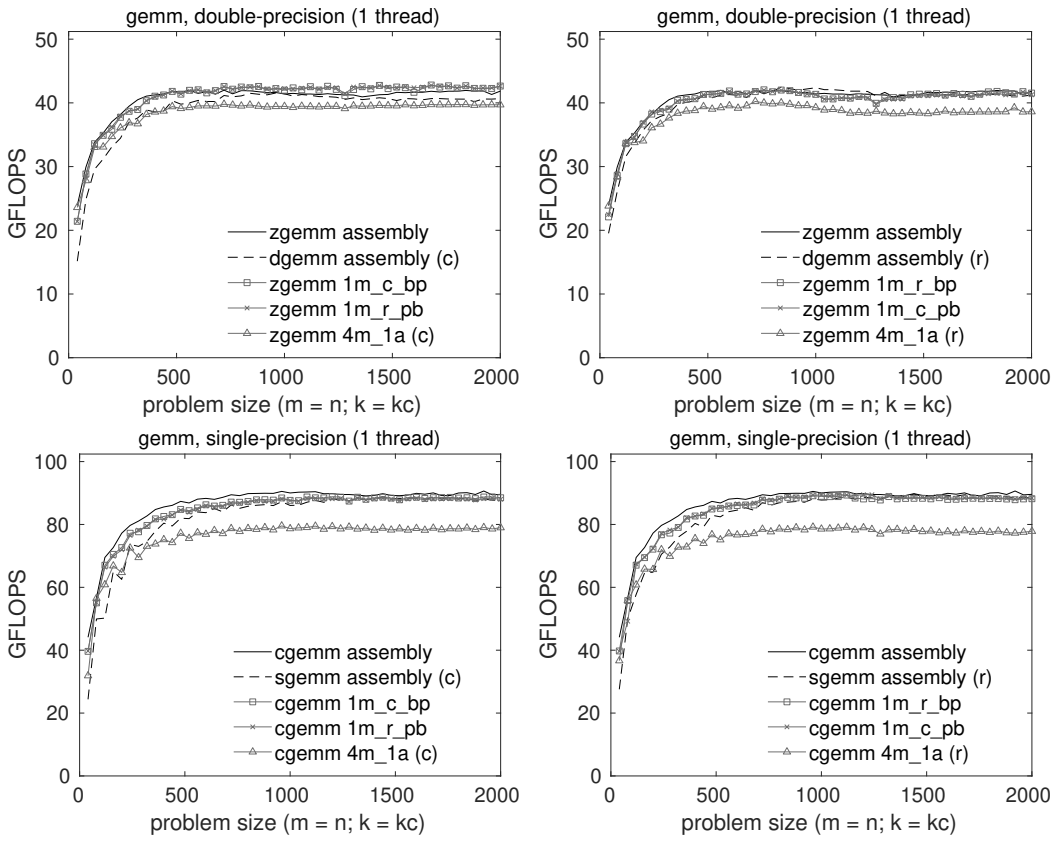

Fig. 5. Single-threaded performance of various implementations of double-precision (top) and single-precision (bottom) complex GEMM on a single core of an Intel Xeon E5-2690 v3 "Haswell" processor. The left and right graphs differ in which 1M implementations they report, with the left graphs reporting 1M_C_BP and 1M_R_PB (which employ column-preferring micro-kernels) and the right graphs reporting 1M_R_BP and 1M_C_PB (which employ row-preferring micro-kernels). The graphs contain one reference curve that are identical between left and right graphs—an assembly-coded complex GEMM found in BLIS—as well as two additional reference curves corresponding to implementations of the assembly-coded real GEMM and the 4M_1A implementation found in BLIS. For consistency with the 1M curves, these latter two reference implementations differ from left to right graphs in the output preference of their underlying micro-kernel, 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.

Blocksizes for each of the implementations tested are provided in Table V. For reference, we also provide the blocksizes for single-precision and double-precision real domain matrix multiplication, as well as those used by complex GEMM implementations in OpenBLAS 0.2.19.

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 5 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, $m = n$ was bound to the problem size while the k dimension

was fixed to the corresponding value of k_C, as listed in Table V. As in the previous article, we focus on this problem shape—rank-k_C update—because: (1) it will yield near-optimal performance for all of our implementations tested, (2) this type of matrix multiplication frequently appears within high-performance implementations of higher-level DLA operations such as Cholesky, LU, and QR factorizations, and (3) it is the foundation for matrix multiplications where all three dimensions are large.[21]

Results for 1M_C_BP and 1M_R_PB (which use column-preferring micro-kernels) appear on the left while those of 1M_R_BP and 1M_C_PB (which use row-preferring micro-kernels) appear on the right. We group the 1M algorithms in this manner so that we can visually confirm the similarities (or differences) in performance within algorithm pairs.

Each graph in Figure 5 contains three reference implementations. One of those reference implementations—BLIS's high-performance complex GEMM based on conventional assembly-coded complex kernels ("zgemm assembly")—is identical between the top-left and top-right graphs, which report double-precision results. Similarly, single-precision results from BLIS ("cgemm assembly") are duplicated between the bottom-left and bottom-right graphs. We also include curves for two other reference implementations found in BLIS: high-performance real domain GEMM and, as representative of the 4M method, an implementation of Algorithm 4M_1A. For these two reference codes, we report results based on column-preferential micro-kernels on the left and those of row-preferential micro-kernels on the right in order to provide consistency with the 1M results. (We indicate the preference of the underlying micro-kernel of those algorithms with a "(c)" or "(r)" in the graph legends.[22])

As predicted in Section 3.6.2, we find that the performance signatures of the algorithms *within* each of the aforementioned 1M algorithm pairs are virtually indistinguishable. That is, 1M_C_BP tracks closely with 1M_R_PB, and 1M_R_BP with 1M_C_PB. This behavior holds for both single- and double-precision. The performance signatures *between* pairs, however, differs 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 pair would be superior over the other, or if there would be a material difference at all. It seems that, at least in the sequential case, the difference between the 1M pairs' performance signatures are minimal, though the difference is somewhat more noticeable in double-precision. This difference is almost certainly due to the different performance characteristics of the row- and column-preferential micro-kernels. We find evidence of this in the 4M_1A results as well as those of the real GEMM implementation, which are also affected by the change in micro-kernel output preference.

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

Somewhat surprisingly, the 1M implementations match or exceed the performance of their real domain benchmarks (the dotted lines in each graph), and are very competitive with assembly-coded complex GEMM regardless of the algorithm employed.

[21]The authors of [Goto and van de Geijn 2008] propose a taxonomy that includes other shape scenarios besides rank-k_C update and large quasi-square multiplication. Some of these other types of matrix product may favor algorithms that are different from the ones depicted in Figure 1. This topic deserves special treatment, and thus is beyond the scope of the present article.

[22]Though it is not indicated in the graph legends, we chose to always compare against an assembly-coded complex GEMM based on a row-preferential micro-kernel because it exhibited higher performance than its column-preferential counterpart on the test hardware. The likely reason for this outperformance relates to the way m_R and n_R affect the bandwidth needed from the L2 and L1 caches, and is briefly discussed in Section 4.3.

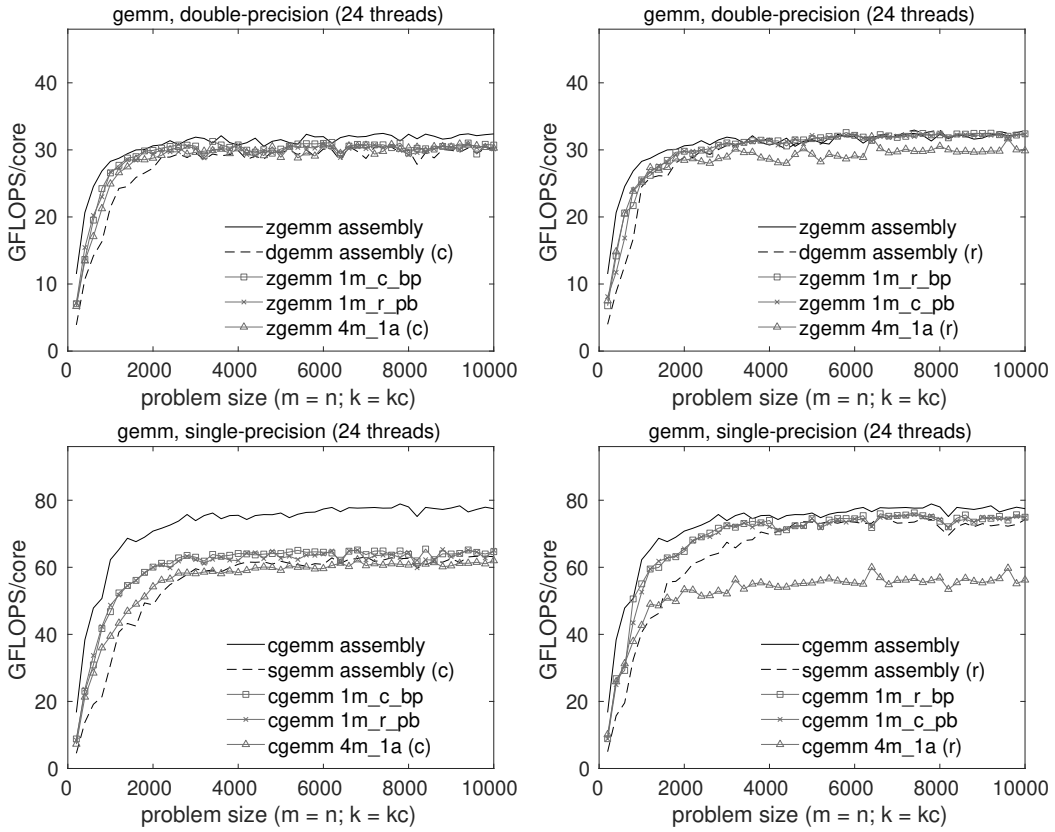

Fig. 6. Multithreaded performance of various implementations of double-precision (top) and single-precision (bottom) complex GEMM on two Intel Xeon E5-2690 v3 "Haswell" processors, each with 12 cores. All data points reflect the use of 24 threads. The left and right graphs differ in which 1M implementations they report, with the left graphs reporting 1M_C_BP and 1M_R_PB (which employ column-preferring micro-kernels) and the right graphs reporting 1M_R_BP and 1M_C_PB (which employ row-preferring micro-kernels). The graphs contain one reference curve that are identical between left and right graphs—an assembly-coded complex GEMM found in BLIS—as well as two additional reference curves corresponding to implementations of the assembly-coded real GEMM and the 4M_1A implementation found in BLIS. For consistency with the 1M curves, these latter two reference implementations differ from left to right graphs in the output preference of their underlying micro-kernel, 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.

4.3. Multithreaded results

Figure 6 shows double- and single-precision performance using 24 threads, with one thread bound to each 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 2-way parallelism within the 5th loop and 12-way parallelism within the 3rd loop for a total of 24 threads. This parallelization scheme was chosen in a manner consistent with that of the previous article[23] using a strategy set forth in [Smith et al. 2014].

[23]The two sockets of the Xeon E5-2690 v3 each have an L3 cache that is shared among those sockets' cores. This encourages two-way parallelism in the 5th loop, which produces two panels $\tilde{B}_p$ to be packed and used simultaneously on completely independent parts of matrix C. Furthermore, by also parallelizing the 3rd loop, each of the 12 cores of either socket pack separate blocks $\tilde{A}_i$ into their private L2 caches. Thus, when

Like in the single-threaded case, the performance of 1M algorithms closely track one another within pairs, albeit with somewhat more jitter. However, here we find a marked difference in performance between pairs. Specifically, the 1M_R_BP and 1M_C_PB implementations (those based on the row-preferring micro-kernel) outperform those of 1M_C_BP and 1M_R_PB (based on the column-preferring micro-kernel), with the difference more pronounced in single-precision. We suspect this is rooted not in the algorithms, per se, but in the differing micro-kernel implementations used by each pair. The 1M_R_BP/1M_C_PB algorithms are implemented with a real micro-kernel that is 6×16 and 6×8 in the single- and double-precision cases, respectively, while 1M_C_BP/1M_R_PB use 16×6 and 8×6 micro-kernels for single- and double-precision implementations, respectively. The observed difference in performance between the 1M pairs is likely attributable to the fact that the micro-kernels' different values for m_R and n_R place different bandwidth requirements when reading F.E. from the caches (primarily L1 and L2). Specifically, larger values of m_R place a heavier burden on loading elements from the L2 cache, which is usually disadvantageous since that cache resides further from the processor core. By contrast, a micro-kernel 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 processor, and relies less heavily on loading elements from the L2 cache.

Even in the worst case (for 1M_C_BP/1M_R_PB), the 1M implementations match or exceed their real domain counterparts. And when the 1M_R_BP/1M_C_PB algorithm pair is employed, performance is competitive with that of the conventional implementations based on complex assembly-coded micro-kernels, particularly in double-precision.

The 1M algorithms based on row-preferential micro-kernels, 1M_R_BP/1M_C_PB, outperform 4M_1A, especially in single-precision where the margin is quite wide. The 1M algorithms based on column-preferential micro-kernels, 1M_C_BP/1M_R_PB, perform more poorly and in fact outpace 4M_1A by only a negligible amount in the double precision case. We suspect that 4M_1A is more resilient to the lower-performing column-preferential micro-kernel by virtue of the fact that the algorithm's virtual micro-kernel 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) [Intel Corporation 2016a].

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, we agree that some basic comparison is appropriate and thus have included Figure **??** (left). These graphs are similar to those in Figure 5, except that only one variant from the better-performing 1M algorithmic pair is included. We also include results for complex GEMM implementations provided by OpenBLAS 0.2.19 and Intel MKL 11.3.[24]

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

These graphs show that BLIS's assembly and 1M implementations are highly competitive with (and in many cases outperform) OpenBLAS, while usually falling short of Intel's highly optimized MKL library.

---

each core executes the 2nd loop (i.e., the macro-kernel), it multiplies its local block $\tilde{A}_i$ by the row-panel $\tilde{B}_p$ that is shared among all cores on the socket.

[24]Note that while we did not know the value of k_C used by MKL's implementations, we settled on a fixed value of $k = 192$ after empirically determining that this value would yield near-optimal performance.

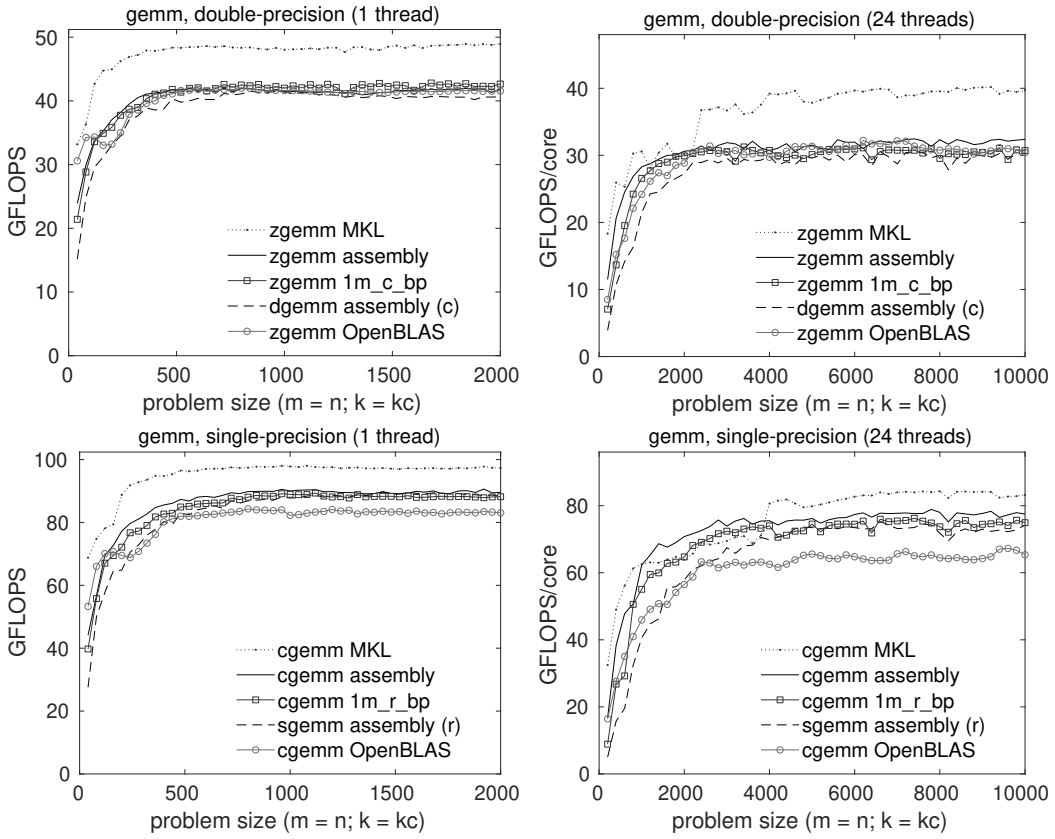

Fig. 7. Single-threaded (left) and multithreaded (right) performance of various implementations of double-precision (top) and single-precision (bottom) complex GEMM on a single core (left) or 12 cores (right) of an Intel Xeon E5-2690 v3 "Haswell" processor. All multithreaded data points reflect the use of 24 threads. The 1M curves represents the best induced method variant from Figures 5 and 6. For MKL results, $k = 192$ was used after empirically determining that this was near-optimal for its underlying GEMM algorithm. The theoretical peak performance coincides with the top of each graph.

4.5. Additional results

Additional performance results were gathered on a 56-core Marvell ThunderX2 compute server. For brevity, we omit these results from the main body of the article. However, we make these results available via Electronic Appendix A. These 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 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 that, collectively, prevent the approach from becoming both a feasible and competitive alternative to matrix multiplication via conventional assembly-based kernels. 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 micro-kernel 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 clearly an improvement over 4M due to its reliance on a single invocation of the matrix multiplication primitive.
— **Inefficient reuse of input date 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 III, both 1M_R and 1M_C variants reuse F.E. of two of the three matrices from registers, and with the inclusion of the panel-block algorithm, two of the four 1M algorithms (one from each variant) reuse F.E. of the third matrix from the L1 cache. This would seem to be a significant improvement, though observed performance improvement over 4M_1A will depend on properties of the hardware (i.e., the L1 cache performance)
— **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. Since C is typically stored (by rows or columns) with real and imaginary F.E. interleaved, this piecemeal approach prevents the real micro-kernel 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 micro-kernel to update contiguous real and imaginary F.E. of C simultaneously within a single invocation. We suspect that this is, perhaps, the largest contributor to 1M's performance superiority over 4M.
— **Reduction of k_C.** Algorithm 4M_1A requires that the real micro-kernel'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 micro-panels.[25] Using such sub-optimally sized micro-panels can noticeably hobble the performance of 4M_1A. Looking back at Table I, it may seem like 1M suffers a similar handicap, 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 micro-kernel, *not* a reduction in the F.E. footprint of the micro-panels operated upon by that real micro-kernel. Indeed, the ability of 1M to achieve high performance when $k = \frac{1}{2}k_C$ is a strength in the context of 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, and much less disruptive than the remaining algorithms. This is because, like with 4M_1A, almost all of the 1M implementation details are sequestered within the packing routines and the virtual micro-kernel.
— **Interference with multithreading.** Because the 1M algorithms are implemented entirely within the packing facility and virtual micro-kernel, they parallelize just as

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

easily as the most thread-friendly of the 4M algorithms, 4M_1A, and entirely avoid the threading difficulties of higher-level 4M algorithms.[26]

— **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.[27]

This analysis 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. Thus, its observed performance superiority was predictable.

5.2. Further discussion

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

5.2.1. 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—a form of swizzling applied when the data is packed—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 long.

5.2.2. Data reuse: efficiency vs. programmability. Both the conventional approach and 1M move data efficiently through the memory hierarchy.[28] However, once in registers, a conventional complex micro-kernel 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 micro-kernel, 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.

5.2.3. Micro-kernel bandwidth. Because 1M algorithms do not *explicitly* reuse F.E. of $\tilde{A}_i$ and $\tilde{B}_p$ from registers, they require higher memory bandwidth during the micro-kernel

[26]This thread-friendly property holds even when the virtual micro-kernel is bypassed altogether as discussed in Section 3.7.4

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

[28]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.

computation than a conventional assembly-based solution.[29] Specifically, increased bandwidth is utilized when reading from the copy of the matrix that is packed into the 1E format, which resides in either the L2 or L1 cache, depending on which algorithm is being employed.[30] In practice, this potential weakness of 1M is not a concern. Yes, in principle, one could design an architecture with sufficiently low memory bandwidth from L2 or L1 cache that a conventional complex matrix multiplication achieves high performance while a 1M-based implementation struggles. However, this would imply a corresponding performance shortfall in the underlying real domain matrix kernel. Given the ubiquity and importance of real matrix multiplication in the scientific community, hardware vendors have great incentive to design architectures that allow `sgemm` and `dgemm` to achieve high performance. Thus, we would expect that 1M will remain a viable alternative for the foreseeable future.

5.2.4. Storage. The supremacy of the 1M method is closely tied to the interleaved, pairwise storage of real and imaginary values—specifically, of the output matrix C. If users and applications decide to start storing complex matrices as two real matrices (traditionally-stored, by rows or columns), one each for real and imaginary components, the 2M approach (for numerically sensitive settings) as well as low-level applications of 3M (for numerically insensitive settings) become more appropriate.

5.2.5. Non-conventional architectures. When considering the applicability of the 1M method, we implicitly assume that the packing kernels are separate units (functions), allowing the details to be abstracted away into a portable framework such as BLIS. Unfortunately, this separation of packing and computational code is not universal across all types of hardware. On some architectures, such a graphical processing units (GPUs), the packing code must be interleaved and fused with the computational kernel code in order to achieve high performance. In these environments, using the 1M method no longer makes sense, as it would likely require just as much work to implement a conventional matrix multiplication based on complex domain kernels.

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, depending on whether the output matrix is stored by rows or columns. We introduced an alternative "panel-block" algorithm that, when combined with the original block-panel algorithm, gives rise to a total of four 1M algorithms, and discussed the similarities and differences in the performance properties of each. We also considered alternative packing formats, including

[29]While this bandwidth distinction actually holds for all induced methods, different algorithms will place differing degrees of bandwidth pressure on the memory hierarchy, depending on the level(s) of cache from which it reuses F.E. of A and B.

[30]The bandwidth from the 1R-formatted matrix is unaffected since it only reorders (rather than duplicates) its F.E.. And since 1M uses half the k_C that is optimal in the real domain (and therefore would perform roughly twice as many rank-k_C updates), the bandwidth required for accessing F.E. of the output matrix may also be higher, though the precise level of increase will depend on the value of k_C^z that would be used by a comparable assembly-based implementation. However, bandwidth requirements on C are already quite low, so we would not expect this difference to measurably impact performance.

a 2M method that would be applicable to more exotic storage arrangements that separate the real and imaginary components. When implemented in the BLIS framework, competitive performance was observed for 1M algorithms on a recent Intel microarchitecture. These tests provided empirical evidence of predicted performance similarities within algorithmic pairs and also confirmed that performance can differ between pairs. 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 micro-kernels 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 micro-kernel—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.

ACKNOWLEDGMENTS

We kindly thank Devangi Parikh for gathering the ThunderX2 performance results presented in Appendix A. We also thank the Texas Advanced Computing Center for providing access to the the Intel Xeon "Lonestar5" compute cluster on which the performance data presented in Section 4 was gathered.

REFERENCES

DONGARRA, J., HAMMARLING, S., HIGHAM, N. J., RELTON, S. D., VALERO-LARA, P., AND ZOUNON, M. 2017. The design and performance of batched BLAS on modern high-performance computing systems. *Procedia Computer Science 108*, 495–504.

DONGARRA, J. J., DU CROZ, J., HAMMARLING, S., AND DUFF, I. 1990. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft. 16,* 1, 1–17.

GOTO, K. AND VAN DE GEIJN, R. 2008. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Soft. 34,* 3, 12:1–12:25.

HIGHAM, N. J. 1992. Stability of a method for multiplying complex matrices with three real matrix multiplications. *SIAM J. Matrix Anal. App. 13,* 3, 681–687.

INTEL CORPORATION. 2016a. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Number 248966-033.

INTEL CORPORATION. 2016b. *Intel® Xeon® Processor E5 v3 Product Family: Processor Specification Update*. Number 330785-010US.

LOW, T. M., IGUAL, F. D., SMITH, T. M., AND QUINTANA-ORTÍ, E. S. 2016. Analytical modeling is enough for high-performance BLIS. *ACM Trans. Math. Soft. 43,* 2, 12:1–12:18.

OpenBLAS 2018. http://xianyi.github.com/OpenBLAS/.

SMITH, T. M., VAN DE GEIJN, R. A., SMELYANSKIY, M., HAMMOND, J. R., AND VAN ZEE, F. G. 2014. Anatomy of high-performance many-threaded matrix multiplication. In *Proceedings of the 28th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*.

VAN ZEE, F. G., SMITH, T., IGUAL, F. D., SMELYANSKIY, M., ZHANG, X., KISTLER, M., AUSTEL, V., GUNNELS, J., LOW, T. M., MARKER, B., KILLOUGH, L., AND VAN DE GEIJN, R. A. 2016. The BLIS framework: Experiments in portability. *ACM Trans. Math. Soft. 42,* 2, 12:1–12:19.

VAN ZEE, F. G. AND SMITH, T. M. 2017. Implementing high-performance complex matrix multiplication via the 3m and 4m methods. *ACM Trans. Math. Soft. 44,* 1, 7:1–7:36.

VAN ZEE, F. G. AND VAN DE GEIJN, R. A. 2015. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Trans. Math. Soft. 41,* 3, 14:1–14:33.

WHALEY, R. C., PETITET, A., AND DONGARRA, J. J. 2000. Automated empirical optimization of software and the ATLAS project. *Parallel Computing 27*, 2001.

Received month 0000; revised month 0000; accepted month 0000

Online Appendix to: Implementing high-performance complex matrix multiplication via the 1m method

FIELD G. VAN ZEE, The University of Texas at Austin

A. ADDITIONAL PERFORMANCE RESULTS

In this section we present performance results for implementations of 1M method on a recent ARM-based architecture. We explore scalability for various numbers of threads while also comparing against the corresponding implementation present in OpenBLAS as well as the ARM Performance Library (ARMPL).

A.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.[31] 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.[32] The version of BLIS used in these tests was version 0.5.0-1.[33]

In this section, we show results for only 1M Algorithms 1M_C_BP, omitting results for the other three algorithms. We chose to omit 1M_R_BP because we did not develop a row-preferential micro-kernel for this architecture. (Recall that 1M_R variant algorithms require a micro-kernel that reads and writes elements of C in a row orientation.) Furthermore, we chose to omit the panel-block algorithms given that our previous results seemed to confirm our hypothesis that the panel-block approach is indistinguishable from the classic block-panel algorithms when the problem size is sufficiently large. And unlike the results shown in Section 4, we did not develop conventional assembly-based micro-kernels with which to compare. For reference, we also measured performance for the complex GEMM implementations found in OpenBLAS[34] 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 Section 4.

[31] 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 "virtual" core per physical core and schedules threads accordingly.

[32] The following optimization flags were used during compilation: `-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`.

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

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

 0098-3500/2017/-ART0 $15.00
DOI: http://dx.doi.org/xx.xxxx/xxxxxxx

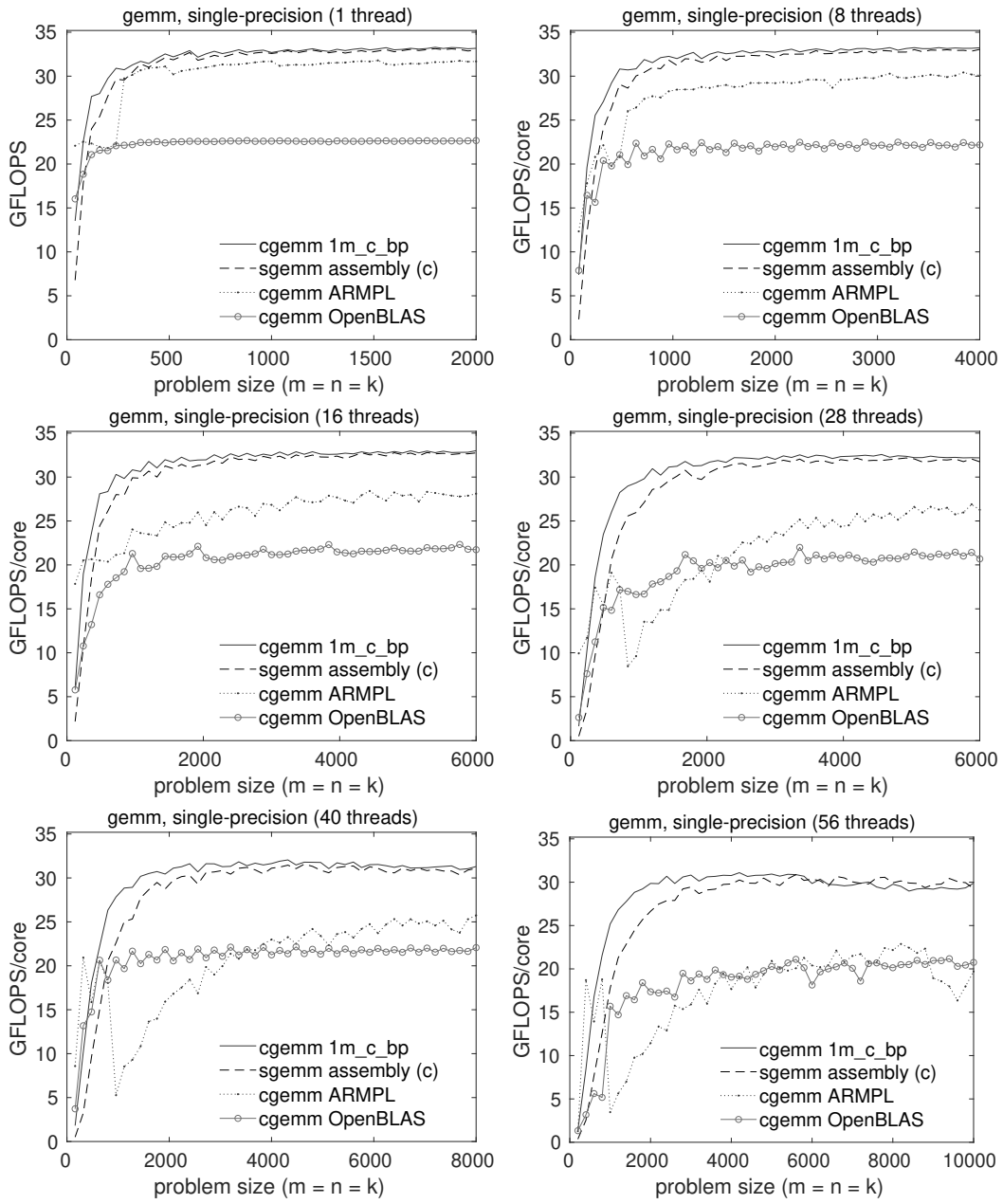

Fig. 8. Performance of various implementations of single-precision complex GEMM using various numbers of threads of a Marvell ThunderX2 CN9975 processor. The real domain GEMM implementation from BLIS uses a column-preferential microkernel, as indicated the a "(c)" in the legends. (The 1M_C_BP implementation uses the same column-preferential microkernel.) The theoretical peak performance coincides with the top of each graph.

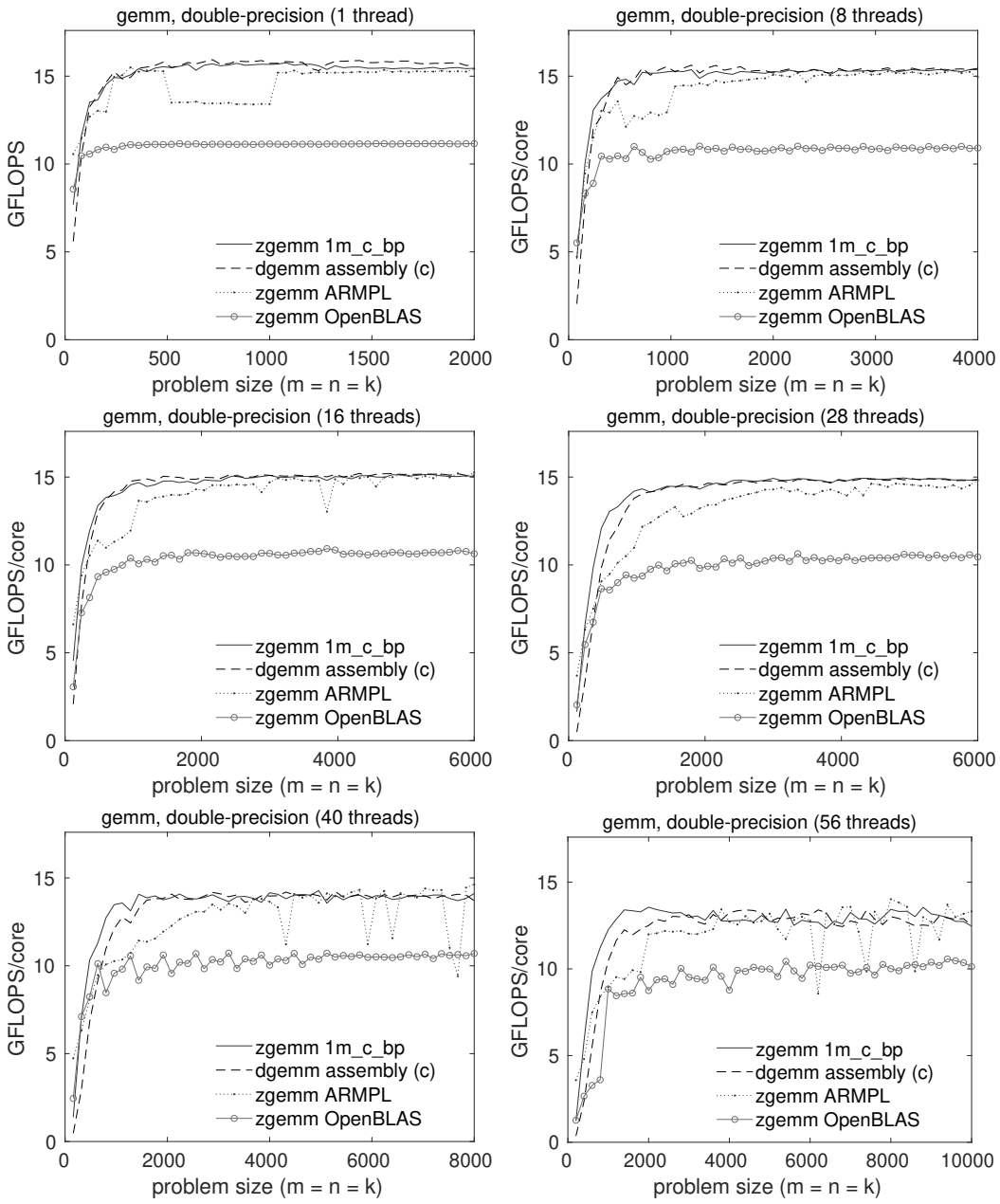

Fig. 9. Performance of various implementations of double-precision complex GEMM using various numbers of threads of a Marvell ThunderX2 CN9975 processor. The real domain GEMM implementation from BLIS uses a column-preferential microkernel, as indicated the a "(c)" in the legends. (The 1M_C_BP implementation uses the same column-preferential microkernel.) The theoretical peak performance coincides with the top of each graph.

A.2. Analysis

Figures 8 and 9 each contains six graphs reporting performance of single-precision and double-precision complex GEMM implementations, respectively, using various numbers of threads. 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. Here, all matrix dimensions are bound to the problem size, shown along the x-axis. The y-axis shows performance in GFLOPS per core and the top of each graph corresponds to the theoretical peak performance (per core).

The primary purpose of gathering the results shown in Figures 8 and 9 was to confirm 1M performance on an non-x86_64 architecture. However, we also wanted to explore whether and to what degree multithreaded efficiency was sustained when utilizing fewer than the maximum number of physical cores. Thus, while the top-left and bottom-right graphs report single-threaded performance and performance with 56 threads, respectively, the graphs in between report performance when utilizing 8, 16, 28, and 40 threads.

In Figure 8, single-precision 1M and its corresponding real domain benchmark track each other closely for all multithreaded configurations tested, as we would have expected. Somewhat surprisingly, the vendor library, ARMPL, does not appear to scale well beyond 8 threads. Furthermore, graphs for 28, 40, and 56 threads suggests the library may be engaging in an unsuccessful attempt to apply two different implementations on either side of a particular problem size (in this case, around 960). 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.

Figure 9 tells a similar story of performance among double-precision implementations, except that all BLIS implementations are, for reasons not immediately obvious, somewhat less efficient than their single-precision counterparts. ARMPL performance is more competitive across all numbers of threads, though the single-core graph exposes additional evidence of a “crossover point” strategy gone awry. ARMPL also seems to exhibit large swings in performance for certain large problem sizes when using high degrees of multithreading. 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.