Inducing complex matrix multiplication via the 3m and 4m methods

FLAME Working Note #81

Field G. Van Zee
Tyler M. Smith
Department of Computer Science
The University of Texas at Austin
Austin, TX 78712

October 18, 2016

Abstract

In this article, we explore the implementation of complex matrix multiplication. We begin by briefly identifying various challenges associated with the conventional approach, which calls for a carefully-written kernel that implements complex arithmetic at the lowest possible level (i.e., assembly language). We then set out to develop a method of complex matrix multiplication that avoids the need for complex kernels altogether. This constraint promotes code reuse and portability within libraries such as BLAS and BLIS and allows kernel developers to focus their efforts on fewer and simpler kernels. We develop two alternative approaches—one based on the 3M method and one that reflects the classic 4M formulation—each with multiple variants, all of which rely only upon real matrix multiplication kernels. We discuss the performance characteristics of these so-called "induced" methods, and observe that the assembly-level method actually resides along the 4M spectrum of algorithmic variants. Implementations are developed within the BLIS framework, and testing on modern hardware confirms that while the less numerically stable 3M method yields the fastest runtimes, the more stable (and thus widely applicable) 4M method's performance is somewhat limited due to implementation challenges which appear inherent in nature.

1 Introduction

High performance matrix multiplication resides near the bottom of the food chain for scientific and numerical applications. Implementations for matrix multiplication are commonly accessed via the level-3 Basic Linear Algebra Subprograms (BLAS) [7], a subset of a standardized library API that provides routines for computing various types of matrix products, including general matrix-matrix multiply (GEMM). Many well-known projects either implement the BLAS directly [7, 26, 25, 24, 3, 14, 15]), or build upon the matrix multiplication facilities within the BLAS to perform more sophisticated dense linear algebra (DLA) computations [22, 4, 1, 2, 18, 5]. Most BLAS operations are specified for both the real and complex domains. And while multiplication of real matrices is employed by countless applications, the unique properties of complex numbers, such as their ability to encode both the phase and magnitude of a wave, makes complex domain computations vitally important to many sub-fields of science and engineering, including signal processing, quantum mechanics, and control theory.

Previous work has shown that one particular algorithm for performing matrix multiplication is ideal for most problem shapes [10, 11, 23, 19], particularly the rank-k update that frequently appears within matrix factorizations (e.g. Cholesky, LU, and QR), inversions (e.g. triangular and Hermitian-positive definite), and transformations (e.g. block Householder). Previously expressed as three loops around a monolithic kernel [10], the state-of-the-art formulation of this algorithm consists of five loops[1] around a much smaller micro-kernel [24, 23, 19]. Still, these "micro-kernels" must be implemented very carefully, typically by experts who understand various characteristics of the compute hardware (such as the instruction set, instruction latencies, and the various features of the memory hierarchy). Most of these kernels are implemented in low-level languages such as machine-specific assembly language[2], which unfortunately obscure the mathematics being expressed. To make matters worse, very few modern architectures provide machine instructions that directly implement complex multiplication and addition, forcing kernel developers to orchestrate computation on the real and imaginary components manually.[3] This can be a tedious undertaking, even for those well-versed in assembly language. While still requiring skill and knowledge of the hardware, implementing real domain matrix multiplication is a significantly more accessible task.

Much of the scientific computing community's attention remains fixed on matrix multiplication in the real domain (usually double precision). Evidence of this can be seen most prominently in the way the community benchmarks its supercomputers [9, 8, 21]. These codes consist largely of an LU factorization with partial pivoting on a double-precision real dense matrix. Most of the floating-point operations in an LU factorization are performed, at some level of recursion, within GEMM subproblems. Consequently, when other aspects of the LU factorization are implemented properly, the performance of the overall benchmark is largely determined by the performance of the double-precision GEMM (`dgemm`) implementation. Thus, it is no wonder that we prioritize optimizing real matrix multiplication kernels.

1.1 Motivations

We have set the stage for our present work by observing that, relative to complex matrix kernels, real matrix kernels are not only objectively easier to implement on many current architectures, but they are also more ubiquitous[4] and quite likely to be highly optimized.

In this article, our goal is to provide a blueprint for simplifying the implementation of complex matrix multiplication, and to do so in such a way that exploits the foundational presence of real matrix multiplication in scientific computing environments. Specifically, we set out to investigate whether, and to what degree, real matrix kernels can be repurposed towards the implementation of complex matrix multiplication. This is motivated primarily by three concerns:

- **Productivity.** Perhaps most obvious is the desire to minimize the number and complexity of kernels that must be optimized in order to support high performance within matrix multiplication operations. If complex kernels were, in fact, not necesssary when computing complex matrix multiplication, then DLA library developers could focus on a smaller and simpler set of real domain kernels. This reduced set of kernels would be more easily maintained on existing hardware, and allow developers to more rapidly instantiate BLAS-like functionality on new hardware.

- **Portability.** At its face, complex matrix multiplication is fundamentally different from real matrix multiplication. (This is primarily because scalar multiplication differs fundamentally in the complex domain.) Thus, if we can avoid reliance upon complex matrix kernels, it is not because we have avoided the mechanics of the complex domain entirely, but rather because we have encoded the notion of complex matrix product at a different (higher) level. Expressing complex matrix multiplication at a level higher than the lowest-level matrix kernel would allow us to encode the approach portablity within a library such as the BLAS-Like Library Instantiation Software (BLIS) framework [24, 23]. If successful, this would allow developers and other experts to effectively induce complex matrix multiplication "for free" as long as the corresponding real matrix multiplication kernel is present.

[1]This newer approach exposes two additional loops which previously, in the original algorithm, were hidden within the monolithic kernel.

[2]Low-level APIs known as vector intrinsics are sometimes used instead, typically because they fit more naturally within higher-level languages such as C.

[3]A notable exception to this is the IBM Blue Gene/Q architecture, which provides instructions specifically targeting complex numbers.

[4]It is unclear how to apportion credit for this ubiquity between (1) the emphasis on real domain benchmarks, (2) the prevalence of real domain applications, and (3) the relative ease of coding real domain arithmetic in hardware.

- **Performance.** A hypothetical complex matrix multiplication based on a real domain kernel would likely inherit the performance properties of that kernel; that is, optimizing the real kernel would accelerate both real and complex domain matrix multiplication. However, the conventional wisdom among DLA experts suggests that complex matrix multiplication is expected to slightly outperform matrix multiplication in the real domain (perhaps by 3-5%).[5] In our current study, several questions arise on this topic. If real domain kernels can be leveraged towards the implementation of complex matrix multiplication, what kind level of relative performance can be realized? What are the sources of performance degredation, if any? Can these hurdles be overcome? And can we identify a meaningful upper bound on the performance that is achieveable? We hope to move toward answering these questions in the course of our study.

1.2 Discussion Preview

The article is laid out as follows. We begin by briefly enumerating some of the challenges typically encountered when implementing a conventional complex matrix multiplication, wherein complex arithmetic is performed at the lowest possible level. We then consider two alternative approaches, each of which casts complex matrix multiplication entirely in terms of real matrix multiplication: the 4M method, which is based on the fundamental definition of multiplication in the complex domain, and the 3M method, which utilizes an alternative expression of complex multiplication that requires only three products. Each method yields a family of algorithms, several of which are discussed in detail. The methods are implemented using the BLIS framework and then evaluated for performance on two modern architectures. We conclude the article by observing key properties of the algorithms and listing several challenges that must be overcome in order to make further progress in the area.

Before proceeding, we provide a few comments regarding the narrative mechanics of the current article.

- In part because of the article's thorough treatment of details and other practical considerations, recent newcomers to the area may find themselves overwhelmed. Admittedly, our target audience primarily consists of DLA library developers and other experts who are relatively well-acquianted with the process of implementing high-performance software for matrix computation. Some "power users" who tinker with DLA libraries may also find the subject matter of interest.

- Some readers will find the relative ordering of Sections 4 and 5 curious. We wish to reassure these readers in advance that our choice of presenting the 3M method prior to the 4M method is intentional and based on subtleties which relate to the overall presentation of the article.

- While not strictly necessary, we encourage the reader to first read [24] and [19]. The former provides a foundational background into BLIS and the insights which lead to its development while the latter further discusses the framework in the context of many-threaded parallelism. Also, the loop-based analyses in Sections 4.3 and 5.1 were modeled after similar analyses found in the second article. Thus, reading that article first will serve to acquaint the reader with the structure and style of discussion in those sections.

1.3 Contributions

This article makes the following contributions:

- It summarizes some of the issues and challenges associated with the conventional approach to implementing complex matrix multiplication. This provides a basic background to help motivate the study of avoiding, if possible, the encoding of complex arithmetic at the lowest levels of matrix multiplication.

- It provides a detailed discussion of the performance characteristics and relative strengths and weaknesses of two alternative methods of performing complex matrix multiplication—the 3M and the 4M methods—giving special attention to implementation issues such as impact on workspace requirements,

[5]This slight outperformance is discussed and simulated in Appendix A.

packing format, cache behavior, multithreadability, and programming effort within the BLIS framework.

- It promotes code reuse and portability by focusing on solutions which may be cast in terms of real matrix multiplication kernels. This increased leverage has clear implications for developer productivity, as it would allow kernel programmers to focus their efforts on fewer and simpler kernels.[6]
- It builds on the theme of the BLIS framework as a productivity multiplier [24], showing how select complex matrix multiplication solutions 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 serves as a reference guide to select implementations of the aforementioned alternative methods for complex matrix multiplication found within the BLIS framework, which is available to the community under an open source software license.[7]

While the subject matter is relatively fundamental, the contributions presented here stand to improve the accessibility and availability of complex level-3 BLAS implementations for the entire community, especially as new hardware is developed.

1.4 Notation

Throughout this article, we will use relatively well-established notation when describing linear algebra objects. Specifically, we will use uppercase Roman letters (e.g. A, B, and C) to refer to matrices, lowercase Roman letters (e.g. x, y, and z) to refer to vectors, and lowercase Greek letters (e.g. χ, ψ, and ζ) to refer to scalars. Subscripts are used typically to denote sub-matrices within a larger matrix (e.g. $A = \left(A_0 \mid A_1 \mid \cdots \mid A_{n-1} \right)$).

We will 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. Also, at times we will use A^{r+i} as shorthand for $A^r + A^i$ (i.e., the sum of the parts, or "summed" part). Note that while this notation for real, imaginary, and complex matrices encodes information about content and origin, it does not encode how the matrices are actually *stored*. We will explicitly address storage details as implementation issues are discussed.

Also, at times we will 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.[8] These definitions allow for a consistent accounting of complex computation relative to the real domain.

This article also discusses and references several hypothetical GEMM-like functions. Unless otherwise noted, a call to function FUNC that implements $C := C + AB$ will appear as $[\ C\ ] :=$ FUNC(A, B, C). Similarly, the operations $C := A + B$ and $C := A - B$ will appear as $[\ C\ ] :=$ ADDM(A, B) and $[\ C\ ] :=$ SUBM(A, B), respectively.

[6]This has implications for architecture design as well. If complex matrix multiplication can be performed only with real domain kernels, then there is one less reason to provide special (potentially costly) instructions and logic for performing complex arithmetic.

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

[8]Later, we generalize our discussion of flops and memops to hardware architectures that support vector instructions, which implement the execution of multiple flops or memops per instruction, usually in proportion to the length (in units of fundamental elements) of the vector register operands.

2 Review

2.1 High-performance matrix multiplication

In this section, we review the general algorithm for high-performance matrix multiplication on conventional microprocessor architectures. This algorithm was first reported on in [10] and further refined in [24]. Figure 1 illustrates the key features of this algorithm.

As alluded to in Section 1, 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.[9] 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.

The algorithm captured by Figure 1 forms the basis for high-performance implementations of the level-3 operations found in the GotoBLAS [11] library, as well as its immediate successor, OpenBLAS [26]. The BLIS framework, discussed in the next section, also employs this algorithm (or a close variant) for each of its level-3 operations.

2.2 The BLIS Framework

The BLIS framework is a relatively new infrastructure for rapidly instantiating high-performance BLAS and BLAS-like libraries. The framework's level-3 operations correspond closely to the nine level-3 BLAS: general matrix multiply (GEMM), Hermitian and symmetric matrix multiply (HEMM and SYMM), Hermitian and symmetric rank-k update (HERK and SYRK), Hermitian and symmetric rank-$2k$ update (HER2K and SYR2K), triangular matrix multiply (TRMM), and triangular solve with multiple right-hand sides (TRSM) [7, 24]. Generally speaking, each of these operations requires its own carefully optimized (assembly-coded) kernel in order to achieve high performance. While the previous state-of-the-art, as captured in [11], established some reuse and consolidation among similar operations, at least four unique computational kernels (and several packing kernels) were still needed in order to provide full level-3 coverage. A key feature of BLIS is that the framework factors out much of the code—corresponding to the first and second loops around the micro-kernel shown in Figure 1—that resides within these assembly-coded kernels. BLIS expresses this code portably, within the higher-level C99 language, which not only promotes code readability and reuse, but also significantly reduces the complexity of what remains of the inner-most kernel, which now amounts to a single loop over tiny $m_R \times n_R$ rank-1 updates. Remarkably, this code refactorization also reduces the *number* of kernels needed to just one. Optimizing this micro-kernel for a given datatype (domain and precision) facilitates the immediate instantiation of high-performance implementations of *all* level-3 operations of the chosen datatype.[10] Thus, when developing BLAS or BLAS-like operations on new hardware, the BLIS framework serves as a substantial productivity multiplier.

Later in this paper, we will use BLIS as a prototyping environment in which to implement and test various alternative approaches to complex matrix multiplication. We will also discuss the amount of programming

[9]Before proceeding further into the article, the reader may wish to look ahead to the entries labeled "BLIS assembly" in Tables 5 and ?? to gain a sense of typical values for these cache and register blocksizes.

[10]The kernel developer may optionally optimize an additional two triangular solve micro-kernels, which yields further optimization for TRSM. The marginal benefit of optimizing these TRSM micro-kernels depends on various factors, such as the hardware design and problem size being tested, but is typically in the range of 5-20%.

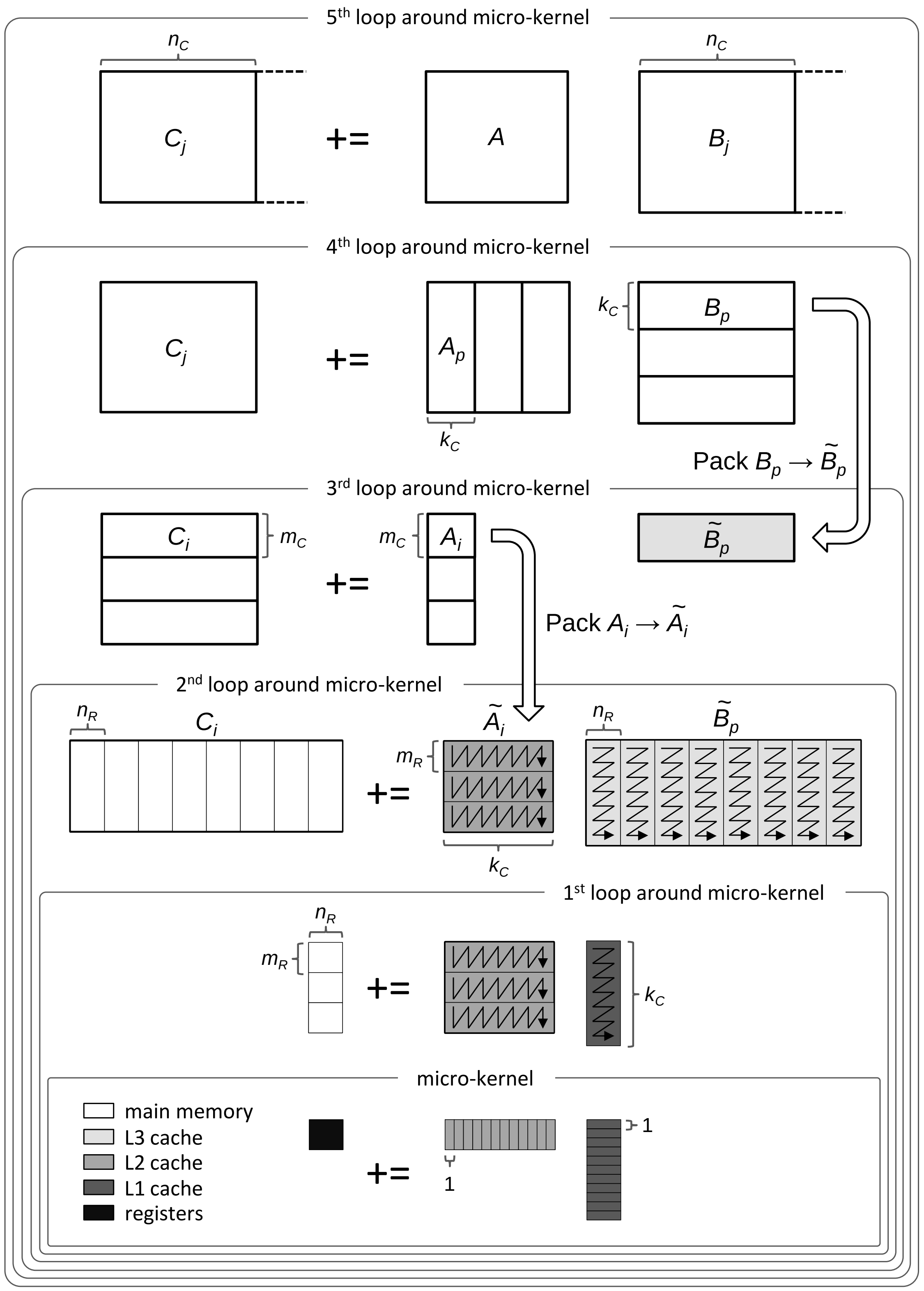

5th loop around micro-kernel
n_C
C_j
+=
A
B_j
4th loop around micro-kernel
C_j
+=
A_p
k_C
B_p
Pack $B_p \rightarrow \tilde{B}_p$
3rd loop around micro-kernel
C_i
m_C
A_i
$\tilde{B}_p$
Pack $A_i \rightarrow \tilde{A}_i$
2nd loop around micro-kernel
n_R
C_i
$\tilde{A}_i$
m_R
$\tilde{B}_p$
1st loop around micro-kernel
micro-kernel
main memory
L3 cache
L2 cache
L1 cache
registers
1

Figure 1: An illustration of the algorithm for computing high-performance matrix multiplication, as expressed within the BLIS framework [24].

effort that would be required to implement a given approach within BLIS.

3 Conventional Approach

In this section, we briefly discuss some of the issues one might expect to encounter when attempting to implement high-performance complex matrix multiplication in the conventional manner. Specifically, we touch upon topics germane to implementing complex matrix multiplication via an assembly-coded micro-kernel.

3.1 Potential Challenges

The conventional approach to implementing complex matrix multiplication entails writing a low-level kernel that performs complex arithmetic, and then inserting this kernel into a BLAS-like framework such as BLIS. In order to attain high performance, the kernel developer would typically write this kernel in assembly language. However, since most modern architectures do not natively implement complex domain instructions (i.e., in terms of complex numbers) the developer must express the complex multiplications and additions in terms of suboperations on the complex numbers' real and imaginary components. This complicates the task considerably, and exposes the kernel developer to various hurdles that do not manifest in the case of the real domain micro-kernel.

3.1.1 Programmability

Programming complex matrix multiplication in assembly language, (via instructions on real scalars) can be more difficult than for real matrix multiplication. We provide qualitative evidence of this in Figure 2.

The diagrams in Figure 2 (left) show the flow of execution when implementing a 4×4 rank-1 update for the real domain in a hypothetical assembly language using vector instructions.[11] These pseudo-instructions perform element-wise arithmetic and assume the presence of vector registers capable of holding four scalar elements. Note that these diagrams capture the body of the inner-most loop in Figure 1 (i.e. the loop body of the micro-kernel).

Figure 2 (right) shows two similar diagrams for a 2×2 rank-1 update for the complex domain. From these, we can see that implementing complex arithmetic in assembly language require additional instructions for permuting elements (as well as a special SubtractAdd instruction), and would likely require more careful planning. By contrast, the pseudo-codes implied by Figure 2 (left) are quite simple, and can be described concisely as an outer product (and accumulation) implemented as a series of column-wise axpy operations on vector register operands. Also notice that the code implied by both diagrams in Figure 2 (right) require a reordering of the elements after the loop.

3.1.2 Floating-point latency and register set size

Some floating-point units have an instruction latency that, when combined with a limited number of registers, hobbles the performance of assembly codes that perform rank-1 update. This happens when allocating registers to the loading of (and subsequent computation with) elements of A and B leaves too few registers with which to accumulate sufficiently large (i.e. flop-rich) $m_R \times n_R$ rank-1 products. If the requisite number of registers simply does not exist, performance will be limited.

Furthermore, unless special (uncommon) functionality is present, complex arithmetic sometimes requires extra registers to hold intermediate results, often produced by certain "swizzle" instructions—those that duplicate, shuffle, or permute elements within vector registers, as depicted in Figure 2 (bottom-right).

[11] While the instructions depicted in Figure 2 (left) are not machine-specific, they roughly correspond to those found within the Intel AVX instruction set.

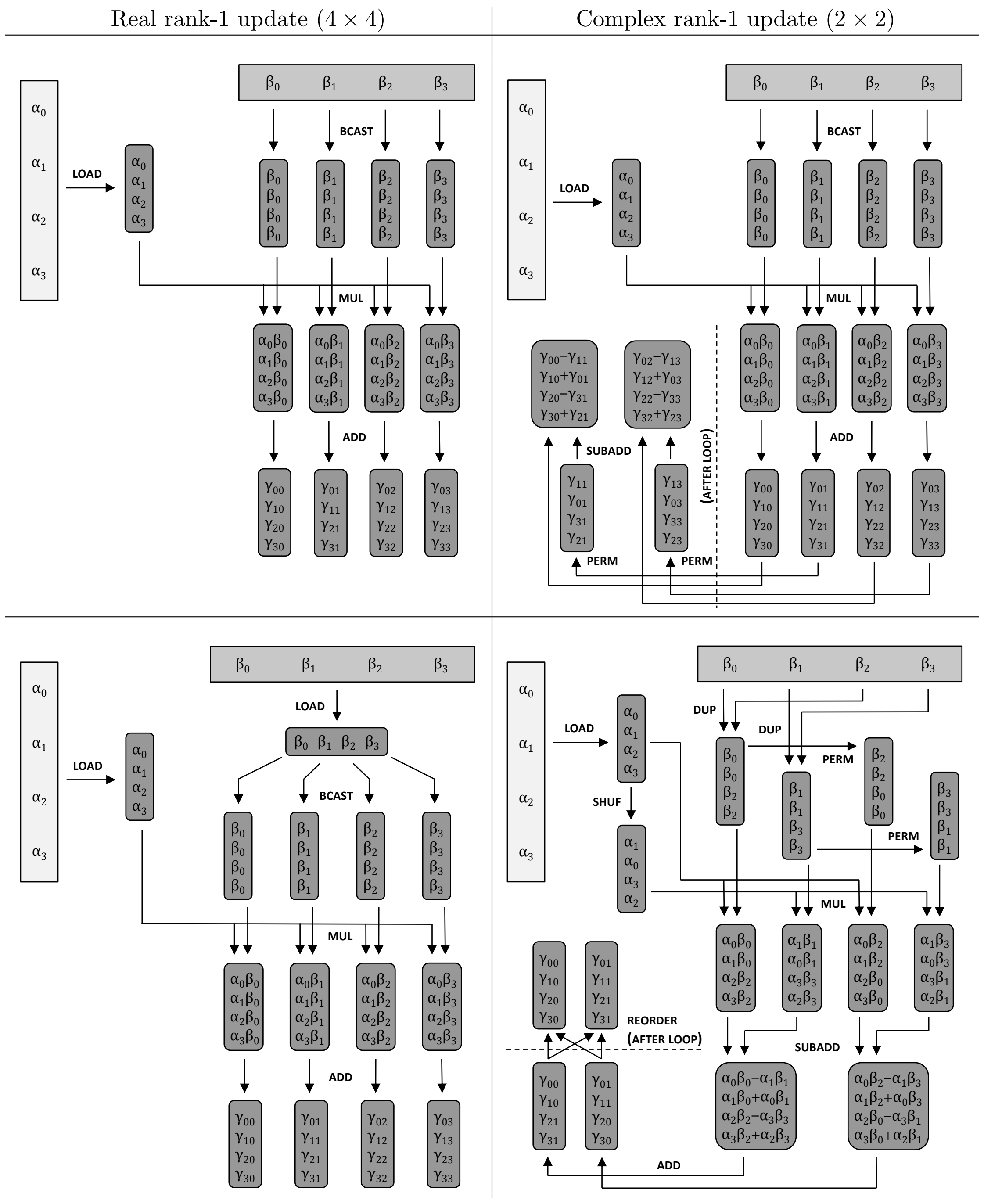


Figure 2: Example flow diagrams of AVX-like instructions operating on four-element vector registers to implement a 4 × 4 rank-1 update in the real domain (left) and a 2 × 2 rank-1 update in the complex domain (right). The assembly code represented by any graph would constitute the loop body of a BLIS micro-kernel. The lighter shaded boxes represent the *p*th column and *p*th row of the packed micro-panels of $\tilde{A}_i$ and $\tilde{B}_p$, which reside in the L2 and L1 caches, respectively, as depicted at the bottom of Figure 1. The darker shaded cells represent vector registers. Since the diagrams on the right do not explicitly label real and imaginary values, we will note here that the micro-column from the current micro-panel of $\tilde{A}_i$ contains the complex values $\begin{pmatrix} \alpha_0 + i\alpha_1 \\ \alpha_2 + i\alpha_3 \end{pmatrix}$ while the micro-row from the current micro-panel of $\tilde{B}_p$ contains $\begin{pmatrix} \beta_0 + i\beta_1 & \beta_2 + i\beta_3 \end{pmatrix}$.

3.1.3 Instruction set

Certain instructions are key to maintaining throughput with complex arithmetic. For example, if the hypothetical architecture depicted in Figure 2 (right) lacked the PERMUTE instruction, complex arithmetic would likely be impossible to implement efficiently with vector instructions.[12] Perhaps more likely is the scenario in which these same instructions exist, but exhibit relatively high execution and/or issue latencies[13], leaving high performance out of reach.

Taken together, these issues, when present, can pose additional challenges when implementing microkernels for complex matrix multiplication. This brings us back to the central question: can we rely on simpler real domain kernels and still achieve acceptable complex domain performance?

4 3m method

We now turn to implementations of complex matrix multiplication that do *not* require complex matrix kernels, and by proxy do *not* require complex arithmetic to be encoded in assembly language. Instead, these methods refactor the computation so that it can be expressed in terms of *only* real matrix multiplication kernels. Going forward, we will refer to such algorithms as performing *induced* complex matrix multiplication.

4.1 Basics

We begin with the fundamental definition of complex scalar multiplication and addition commonly found in algebra textbooks. Let $\chi, \psi, \zeta \in \mathbb{C}$. We can express the complex scalar product update $\zeta := \zeta + \chi\psi$ as:

$$\begin{aligned} \zeta := \zeta + \chi\psi &= \left(\zeta^r + i\zeta^i\right) + \left(\chi^r + i\chi^i\right)\left(\psi^r + i\psi^i\right) \\ &= \zeta^r + \chi^r\psi^r - \chi^i\psi^i + i\left[\zeta^i + \chi^r\psi^i + \chi^i\psi^r\right] \end{aligned} \tag{1}$$

If we translate Eq. 1 into imperative statements in terms of real and imaginary components, we arrive at:

$$\begin{aligned} \zeta^r &:= \zeta^r + \chi^r\psi^r - \chi^i\psi^i \\ \zeta^i &:= \zeta^i + \chi^r\psi^i + \chi^i\psi^r \end{aligned} \tag{2}$$

If ζ^r and ζ^i are computed according to Eq. (2), four scalar multiplications, three additions, and one subtraction are required. Since additions and subtractions are typically comparable in cost, we will henceforth treat and refer to scalar additions and subtractions indistinguishably as "additions."

4.2 Reformulation

Alternative expressions of Eq. (1) may be derived. Let us focus on one[14] in particular:

$$\begin{aligned} \zeta &= \zeta^r + \chi^r\psi^r - \chi^i\psi^i + i\left[\zeta^i + \chi^r\psi^i + \chi^i\psi^r\right] \\ &= \zeta^r + \chi^r\psi^r - \chi^i\psi^i + i\left[\zeta^i + \chi^r\psi^i + \chi^i\psi^r + \left(\chi^r\psi^r - \chi^r\psi^r\right) + \left(\chi^i\psi^i - \chi^i\psi^i\right)\right] \\ &= \zeta^r + \chi^r\psi^r - \chi^i\psi^i + i\left[\zeta^i + \left(\chi^r\psi^r + \chi^r\psi^i + \chi^i\psi^r + \chi^i\psi^i\right) - \chi^r\psi^r - \chi^i\psi^i\right] \\ &= \zeta^r + \chi^r\psi^r - \chi^i\psi^i + i\left[\zeta^i + \left(\chi^r + \chi^i\right)\left(\psi^r + \psi^i\right) - \chi^r\psi^r - \chi^i\psi^i\right] \end{aligned} \tag{3}$$

Textually, Eq. (3) contains five products. However, since two of the products reoccur once each, this equation exposes a way to compute the complex scalar product $\chi\psi$ using only three unique multiplications. Higham refers to this as the "3M" method [13].

[12]We wish to emphasize that instructions shown in Figure 2 (right) do not *uniquely* enable complex arithmetic; other architectures will offer somewhat different sets of instructions, allowing the construction of slightly different assembly-level algorithms.

[13]Henceforth, in the context of non-floating-point instructions, we will sometimes use "latency" to refer to both execution latency as well as the reciprocal throughput (issue latency).

[14]In [13], Higham reports that, according to Knuth [16] this formulation was first proposed by Peter Ungar in 1963.

If we apply 3M to complex matrices $A \in \mathbb{C}^{m \times k}$, $B \in \mathbb{C}^{k \times n}$, and $C \in \mathbb{C}^{m \times n}$, and express using imperative statements, we have:

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

If we introduce workspace matrices $W \in \mathbb{C}^{m \times n}, S_A \in \mathbb{R}^{m \times k}, S_B \in \mathbb{R}^{k \times n}$, we can reuse the $A^r B^r$ and $A^i B^i$ terms once computed, as expressed by:

$$\begin{aligned} W^r &:= A^r B^r, \quad W^i := A^i B^i \\ S_A &:= A^r + A^i, \quad S_B := B^r + B^i \\ C^r &:= C^r + W^r - W^i \\ C^i &:= C^i + S_A S_B - W^r - W^i \end{aligned} \tag{4}$$

Notice that while 3M saves one multiplication, it incurs three extra additions, for a net increase of two arithmetic operations. However, when applied to matrix operands, these extra operations have negligible impact on the overall cost because they constitute lower-order terms: the 3M method's intermediate matrix products collectively require $3 \times 2mnk$ flops while the matrix additions require only $5mn + mk + kn$ flops. Therefore, asymptotically, 3M incurs nearly 25% fewer flops than the $4 \times 2mnk$ flops required of the conventional approach discussed in Section 3.

Note that implementing 3M at the scalar level forfeits this advantage. At the scalar level, there is little or no benefit to reusing the intermediate products $\chi^r \psi^r$ and $\chi^i \psi^i$, for two reasons. First, modern computers tend to favor computations with a balanced number of MULTIPLY and ADD instructions; thus, trading a MULTIPLY for two additional ADD instructions may be counterproductive. Second, even when this is not the case, the relative cost difference typically observed between MULTIPLY and ADD is not enough to merit use of 3M. In general, the 3M method becomes advantageous only when the cost of a multiplication is significantly higher than that of an addition.

A numerical stability analysis of 3M is presented in [13]. Higham finds that 3M, like the closely related Strassen's Algorithm [20], is slightly less stable than conventional approaches based on Eq. 2, but stable enough to merit use in many practical settings. Higham's conclusions are based on the highest level application of 3M, which we refer to as algorithm 3M_H in the next section and beyond. A comprehensive numerical analysis of 3M when applied to other levels of the matrix multiplication algorithm is beyond the scope of this article.

Now that we have expressed 3M method in terms of real and imaginary sub-matrices, we can explore how to use 3M to implement complex matrix multiplication in terms of real matrix multiplication.

4.3 Application

Previous work has focused on applying 3M-like (Strassen-based) approaches at a relatively high level [12, 6], with BLAS serving as the building blocks. But if we return to Figure 1, we can see that the 3M method, as represented by Eq. 4, can potentially be applied around any one of the loops exposed by the BLIS matrix multiplication algorithm.

In this section, we consider various hypothetical approaches that result from applying 3M within each loop depicted in Figure 1, with special attention given to workspace, packing format, cache performance, programmability, and multithreadability. We will also note when an application of 3M exhibits properties that complicate (or preclude) extension to other level-3 operations. Note that in targeting the various loops, we are essentially defining different real domain matrix multiplication *primitives*, varying from the very coarse grain where the primitive is a general-purpose real GEMM, to the very fine grain where the primitive is a real GEMM micro-kernel. For reference, we provide Figure 3, which shows pseudo-code for real general matrix multiplication (RGEMM) implemented in terms of a real micro-kernel (RKERN). This pseudo-code implements the algorithm depicted in Figure 1.

Since we are targeting matrix multiplication in the complex domain, our discussion will assume that m, k, and n represent problem size dimension in units of complex scalars. However, keep in mind that while

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

Figure 3: An abbreviated pseudo-code for implementing the general matrix multiplication algorithm depicted in Figure 1. Here, RKERN calls a real domain GEMM micro-kernel.

```
Algorithm: [ $C$ ] := 3M_H( $A$, $B$, $C$ )
Acquire workspace $S_A, S_B, W$
$S_A$ := ADDM( $A^r$, $A^i$ ) ;   $S_B$ := ADDM( $B^r$, $B^i$ )
$W^r$ := RGEMM( $A^r$, $B^r$, 0 )
$W^i$ := RGEMM( $A^i$, $B^i$, 0 )
$C^i$ := RGEMM( $S_A$, $S_B$, $C^i$ )
$C^r$ := ADDM( $W^r$, $C^r$ ) ;   $C^r$ := SUBM( $W^i$, $C^r$ )
$C^i$ := SUBM( $W^r$, $C^i$ ) ;   $C^i$ := SUBM( $W^i$, $C^i$ )
```

Figure 4: An algorithm for applying the 3M method at the highest level of general matrix multiplication, with the 5th loop around the micro-kernel becoming the real matrix multiplication primitive (shown here as a call to RGEMM).

m_C, k_C, n_C are the optimal blocksizes chosen for RGEMM, those dimensions may also be used to measure dimensions in units of complex scalars, as determined by context.

4.3.1 Outside the algorithm

Perhaps the most obvious place to apply 3M is at the highest level. This would define our real matrix multiplication primitive to be the 5th loop around the micro-kernel, which would encompass the entire matrix multiplication algorithm. We can implement this case by replacing each real matrix product in Eq. 4 with a call to RGEMM. Similarly, addition and subtraction with the real or imaginary submatrices can be handled with ADDM and SUBM. Note that this is equivalent to applying Eq. 4 from within the calling application.

Algorithm 3M_H in Figure 4 provides pseudo-code which shows the 3M method being implemented at this level.

This approach requires workspace matrices $W \in \mathbb{C}^{m\times n}, S_A \in \mathbb{R}^{m\times k}, S_B \in \mathbb{R}^{k\times n}$.[15] Unfortunately, for large input matrices, the amount of workspace required is considerable. And while 3M_H automatically inherits the multithreadedness of RGEMM, ADDM and SUBM may hinder scalability.[16] However, 3M_H is easy

[15] Notice that since Algorithm 3M_H uses workspace matrices W^r and W^i in separate operations, they may be stored separately as two real matrices, rather than as a single complex matrix with pairs of real and imaginary elements stored contiguously.

[16] We assume that either ADDM and SUBM are single-threaded. Even if these operations were multithreaded, their efficiency would be inherently limited. ADDM and SUBM perform only mn flops but incur $3mn$ memory operations, and thus the available

to implement in BLIS. Also, packing functionality and cache blocksizes need not be changed.

4.3.2 5th loop around the micro-kernel

Applying 3M inside the 5th loop around the micro-kernel would result in a similar algorithm to that of 3M_H. The most substantial difference is workspace is reduced somewhat when $n > n_C$, with $W \in \mathbb{C}^{m \times n_C}, S_B \in \mathbb{R}^{k \times n_C}$. We estimate that encoding this approach in BLIS would be somewhat inelegant, because it would require specialization and duplication of code that normally would remain completely general. Also, if parallelism around the n dimension was desired, multithreading may be somewhat complicated by the computation of $S_A := A^r + A^i$, which would either need to be computed redundantly among threads or computed by one thread and then shared. All other properties regarding workspace, packing, and cache blocksizes would remain the same.

We omit showing Algorithm 3M_5 for space reasons.

4.3.3 4th loop around the micro-kernel

Applying 3M within the 4th loop around the micro-kernel would result in a primitive that implements a single rank-k_C update. Here, we assume that the packing of $B_{jp} \rightarrow \tilde{B}_p$ occurs three times, once before each call to the primitive. This approach would further reduce workspace requirements to $W \in \mathbb{C}^{m \times n_C}, S_A \in \mathbb{R}^{m \times k_C}, S_B \in \mathbb{R}^{k_C \times n_C}$. As of this writing, the BLIS framework does not support parallelization of the 4th loop[17], so this approach would not complicate multithreading, nor would the modifications to BLIS be any more (or less) challenging relative to 3M_5. All other properties remain unchanged.

We omit showing Algorithm 3M_4 for space reasons.

4.3.4 3rd loop around the micro-kernel

Targeting the body of the 3rd loop around the micro-kernel would result in a primitive that implements a "block-panel" matrix multiply, which in BLIS is performed by a functional unit of code known as the *macro-kernel.*[18] Here, workspace is reduced again, this time to $W \in \mathbb{C}^{m_C \times n_C}, S_A \in \mathbb{R}^{m_C \times k_C}, S_B \in \mathbb{R}^{k_C \times n_C}$. This reduction is significant because all workspace matrix dimensions are now bounded by constant blocksizes.[19]

As a practical matter, targeting the 3rd loop would require the packing function to be adapted so that $\tilde{B}_p$ is packed to contain $\left(\hat{B}^r \quad \hat{B}^i \quad \hat{B}^{r+i} \right)$, where $\hat{B} = B_{jp}$ and each of the three submatrices are packed in the classic micro-panel format described in [24]. Recall that, while BLIS can separately access the real and imaginary parts of traditionally-stored complex matrices, the purpose of packing is to allow contiguous access of data by the micro-kernel. Since 3M makes use of real and imaginary parts (and their sum) as part of separate sub-operations, this change to the packing facility preserves contiguity at the granularity expected by the micro-kernel. Note that this also gives us an opportunity to hide some of the cost of computing the $S_B = B^{r+i}$ term in Eq. 4, since the constituent elements must flow through the memory hierarchy anyway as part of the packing process. This also means that the workspace required by S_B simply takes the form of larger packing space for $\tilde{B}_p$, rather than matrix storage that must be managed separately.

The packing format of $\tilde{A}_i$ does not change, however, the approach does require that the contents packed into $\tilde{A}_i$ may originate from any one of $\hat{A}^r$, $\hat{A}^i$, and $S_A = \hat{A}^{r+i}$ where $\hat{A} = A_{pi}$. As with $\hat{B}^{r+i}$, $\hat{A}^{r+i}$ can be computed on-the-fly while it is being packed into $\tilde{A}_i$, thus eliminating the need to allocate separate space for S_A.

memory bandwidth of modern multicore systems would be saturated with a relatively few number of threads.

[17]Parallelization within the 4th loop is different than the other four loops because it requires mutual exclusion to support multiple threads updating identical regions of C.

[18]The 2nd loop around the micro-kernel (and the statements over which it iterates) is equivalent to the "inner kernel" (or simply "kernel") defined in GotoBLAS and OpenBLAS.

[19]Presently, BLIS allocates memory statically. This was once performed to ensure that the memory was physically contiguous (rather than merely logically contiguous). We lack enough experimental data to say whether static allocation is still required on modern hardware. If physically contiguous allocation is still effective and advantageous, then bounding 3M workspace by constant blocksizes allows for the use of static memory for that workspace, which may facilitate minor performance benefits on some hardware.

Unfortunately, 3m within the 3rd loop promotes poor use of the L3 cache. The second call to the primitive (to compute $\tilde{A}_i^i \tilde{B}_p^i$) results in much of $\tilde{B}_p^r$ being evicted from the L3 cache, with similar evictions for $\tilde{B}_p^i$ and $\tilde{B}_p^{r+i}$. To compensate, n_C can be reduced by two-thirds so that all three submatrices can coexist within the L3 cache. However, this means that the cost of packing each block $\hat{A}^r$, $\hat{A}^i$, and $\hat{A}^{r+i}$ (i.e. the cost of moving each from main memory into the L2 cache) will be amortized over 66% less computation, since each will be multiplied by a different (now smaller) region of $\tilde{B}_p$. A silver lining in this reduction to n_C is that $\tilde{B}_p$ occupies the same amount of internal workspace as it would in a conventional algorithm, even when the added space for $\hat{B}^{r+i}$ is taken into account.

Another drawback to this approach is that the 3rd loop around the micro-kernel is often targeted for parallelism, especially when each processing core has a private L2 cache [19]. Consequently, the computation of $\hat{B}^{r+i}$ (which occurs *outside* the 3rd loop) may present complicating issues to multithreading similar to those of Algorithm 3m_5, both in terms of performance and programmability.

We omit showing Algorithm 3m_3 for space reasons.

4.3.5 2nd loop around the micro-kernel

Applying 3m within the 2nd loop around the micro-kernel results in a matrix primitive that multiplies an $m_C \times k_C$ block of A by a $k_C \times n_R$ micro-panel of B. This would seem to reduce workspace to $W \in \mathbb{C}^{m_C \times n_R}, S_A \in \mathbb{R}^{m_C \times k_C}, S_B \in \mathbb{R}^{k_C \times n_R}$. However, as with 3m_3, we will show later how blocksize reductions further reduce the workspace required.

For this approach, we need to pack $B_{jp} \rightarrow \tilde{B}_p$ to a special format that is different than described above for 3m_3. Let $B_{jp} = \hat{B}$, so that we may reclaim the subscript for further partitioning. Instead of organizing $\tilde{B}_p$ so that micro-panels of $\hat{B}^r$, $\hat{B}^i$, and $\hat{B}^{r+i}$ are grouped together, this format would extract the real, imaginary, and summed parts of the micro-panels and interleave them as ordered "3-tuples" of real micro-panels:

$$\hat{B} \rightarrow \left(\begin{array}{ccc|ccc|c|ccc} \tilde{B}_0^r & \tilde{B}_0^i & \tilde{B}_0^{r+i} & \tilde{B}_1^r & \tilde{B}_1^i & \tilde{B}_1^{r+i} & \cdots & \tilde{B}_{t-1}^r & \tilde{B}_{t-1}^i & \tilde{B}_{t-1}^{r+i} \end{array} \right)$$

where $t = \lceil n_C / n_R \rceil$ and $\tilde{B}_j$ refers to the jth set of n_R columns (i.e. micro-panel) of $\hat{B}$. Once again, we would need to reduce n_C in order to maintain a similar footprint in the L3 cache.

Because of the nature of our matrix primitive, we would also need to format $\tilde{A}_i$ differently, as $\left(\begin{array}{c} \hat{A}^r \\ \hat{A}^i \\ \hat{A}^{r+i} \end{array} \right)$, where $\hat{A} = A_{pi}$. Note that this format is simply the transposition of the format used on to pack $\tilde{B}_p$ for Algorithm 3m_3. And similarly, we would need to reduce m_C by approximately two-thirds in order for $\tilde{A}_i$ to occupy the same fraction of the L2 cache. However, as before, this means that the cost of moving the current $\tilde{B}_j^r$, $\tilde{B}_j^i$, or $\tilde{B}_j^{r+i}$ micro-panel from the L3 cache into the L1 cache is now amortized over less computation, since each must be multiplied by a different (now smaller) region of $\tilde{A}_i$.

Recall our opening remarks regarding workspace requirements. If $\tilde{A}_i$ and $\tilde{B}_p$ are packed as described above, separate workspaces for S_A and S_B are no longer needed, as those intermediate matrices become integrated into $\tilde{A}_i$ and $\tilde{B}_p$, whose m_C and n_C dimensions, respectively, have been reduced to compensate. We still need workspace $W \in \mathbb{C}^{m_C \times n_R}$, though, in order to compute and reuse the $A^r B^r$ and $A^i B^i$ terms.

The 2nd loop around the micro-kernel is also frequently targeted for parallelism (perhaps even more often than the 3rd loop), especially when the L2 cache is shared among processing cores [19]. Parallelism at this level is also targeted because it yields relatively fine-grain workload balancing, since t is usually quite large.

Now, unlike with Algorithm 3m_3, here the summing of the real and imaginary parts of both A_{pi} and B_{jp} occurs entirely within their respective packing routines. Depending on where parallelism is extracted, this may or may not hinder performance. Parallelizing higher level loops would lead threads to add the real and imaginary sub-matrices of A_i and B_p in parallel. However, parallelizing only within the macro-kernel (2nd or 1st loops) would serialize the computation of the $(A^r + A^i)$ and $(B^r + B^i)$ terms of the 3m method, which may result in poor scaling to many threads.

Applying 3m within the 2nd loop would be moderately disruptive to the existing BLIS code for two reasons. First, it would need to carefully take into account multithreading, especially within the 1st loop

```
**Algorithm**: [ $C$ ] := 3M_1( $A$, $B$, $C$ )
for ( $j = 0 : n-1 : n_C$ )
  Identify $B_j, C_j$ from $B, C$
  for ( $p = 0 : k-1 : k_C$ )
    Identify $A_p, B_{jp}$ from $A, B_j$
    PACK3MINTERLEAVED $B_{jp} \rightarrow \tilde{B}_p$
    for ( $i = 0 : m-1 : m_C$ )
      Identify $A_{pi}, C_{ji}$ from $A_p, C_j$
      PACK3MINTERLEAVED $A_{pi} \rightarrow \tilde{A}_i$
      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 $\tilde{A}_i, C_{jih}$
          $C_{jihl}$ := VK3M_1( $\tilde{A}_{il}$, $\tilde{B}_{ph}$, $C_{jihl}$ )
```

```
[ $C$ ] := VK3M_1( $A$, $B$, $C$ )
Acquire workspace $W$
Expose $A = \begin{pmatrix} A^r \\ A^i \\ A^{r+i} \end{pmatrix}$,
       $B = \begin{pmatrix} B^r & B^i & B^{r+i} \end{pmatrix}$
$W^r$ := RKERN( $A^r$, $B^r$, 0 )
$W^i$ := RKERN( $A^i$, $B^i$, 0 )
$C^i$ := RKERN( $A^{r+i}$, $B^{r+i}$, $C^i$ )
$C^r$ := ADDM( $W^r$, $C^r$ )
$C^r$ := SUBM( $W^i$, $C^r$ )
$C^i$ := SUBM( $W^r$, $C^i$ )
$C^i$ := SUBM( $W^i$, $C^i$ )
```

Figure 5: Left: Pseudo-code for Algorithm 3M_1. Here, PACK3MINTERLEAVED packs the gth micro-panel of $B_{jp} = \hat{B}$ as $\begin{pmatrix} \hat{B}^r_g & \hat{B}^i_g & \hat{B}^{r+i}_g \end{pmatrix}$ and of $A_{pi} = \hat{A}$ as $\begin{pmatrix} \hat{A}^r_g \\ \hat{A}^i_g \\ \hat{A}^{r+i}_g \end{pmatrix}$. Right: Pseudo-code for a virtual micro-kernel used in Algorithm 3M_1. This virtual micro-kernel assumes that the micro-panel arguments A and B were packed with separate real, imaginary, and summed sub-panels.

and with regard to workspace W. Secondly, these modifications would need to be made to all[20] level-3 BLIS macro-kernels.

We omit showing Algorithm 3M_2 for space reasons.

4.3.6 1st loop around the micro-kernel

If we target the first loop around the micro-kernel, our primitive becomes the micro-kernel itself. Here, it makes sense for both $\tilde{A}_i$ and $\tilde{B}_p$ to be packed using the "interleaved" micro-panels approach described for $\tilde{B}_p$ within Algorithm 3M_2. This results in $s = \lceil m_C/m_R \rceil$ triplets of real, imaginary, and summed micro-panels being packed into $\tilde{A}_i$. If packing is performed in this manner, the only workspace required is $W \in \mathbb{C}^{m_R \times n_R}$. This tiny micro-tile of workspace is small enough to be allocated statically (i.e. on the function stack).

Defining the micro-kernel as the primitive comes with other benefits. For example, we can define a "virtual" micro-kernel, written portably in C99, which implements the 3M method (in terms of the real micro-kernel) in a separate function. This allows the details of the 3M method to remain completely abstracted from the existing level-3 macro-kernel code. Thus, Algorithm 3M_1 requires only isolated and relatively modest changes to support within the BLIS framework.

Yet another benefit is that 3M at this level does not hinder multithreading, since the framework is already set up to extract all parallelism *above* the micro-kernel. And since the $W \in \mathbb{C}^{m_R \times n_R}$ workspace can be allocated statically on the function stack of the virtual micro-kernel, no additional workspace issues arise in the context of many threads.

The main drawback of this algorithm is that k_C must be reduced by two-thirds in order to maintain the same the cache footprints of the micro-panels. Unfortunately, this means that the real domain micro-kernel will be called on micro-panels whose k dimensions are, in the best of cases, one-third their optimial value. Hence, 3M_1 updates C three times as frequently as the other 3M algorithms while performing the same number of flops. We expect that this reduction of k_C will cause a small-to-moderate performance penalty, depending on the relative cost of memory operations.

[20]Presently, four macro-kernels are maintained in BLIS, one each to handle the following sets of level-3 operations: (1) GEMM, HEMM, and SYMM; (2) HERK, SYRK, HER2K, and SYR2K; (3) TRMM and TRMM3; and (4) TRSM. All except TRSM perform some kind of matrix multiplication (though TRSM can be expressed partially in terms of matrix multiplication subproblems).

Algorithm 3M_1 is shown in Figure 5 (left), with a separate algorithm for the virtual micro-kernel given in Figure 5 (right).[21]

4.4 Refinements and other implementation details

4.4.1 Handling alpha and beta scalars

We have thus far simplified the general matrix multiplication operation to $C := C + AB$. However, in practice, the operation is implemented as $C := \beta C + \alpha AB$, where $\alpha, \beta \in \mathbb{C}$. Let us use Algorithm 3M_H, in Figure 4, as an example as we now consider how to modify our 3M algorithms to support arbitrary values of α and β.

If β is real, we can pass β into the third call to RGEMM and then replace the ADDM and SUBM operations with calls to an AXPY-like operation: $Y := \beta Y + X$. If instead β is complex, C^r and C^i must be scaled outside of RGEMM and before adding or subtracting by W^r and W^i. Notice that 3M_1 benefits from the fact that it would scale by β within the virtual micro-kernel, which means the operation is automatically done in parallel when multithreading.

When α is real, the scaling may be performed by the primitive. Now, at first glance, scaling by complex α would seem problematic. In this situation, a naive approach might be to scale a copy of A or B. However, this scaling can be folded into the packing of either $\tilde{A}_i$ or the $\tilde{B}_p$. While the scaling that occurs during packing is not, by default, optimized to use vector instructions, we do not see this as much of a problem, for two reasons. First, the limiting factor in packing is the memory operations themselves, which, compared to MULTIPLY and ADD instructions, are quite costly, especially when data is moved from main memory. Second, the overall cost of packing, which is $\mathcal{O}(k(m+n))$, is small relative to the overall cost of floating-point operations, which is $\mathcal{O}(2mnk)$. For these reasons, we would not expect vector instructions to provide much speedup.

Thus, with a little extra logic in the primitive and the packing facility, scaling can be incorporated. This logic is conveniently hidden when the primitive is a virtual micro-kernel.

4.4.2 Alignment of packed matrices

Recall that some of the algorithms for 3M reduce one or more of the cache blocksizes by two-thirds in order to keep the packed matrices' cache footprints similar to what they would be in a classic real domain algorithm, such as expressed in Figure 1 and RGEMM. But since blocksizes m_C, n_C, and k_C are integers, division by three results in a non-integer value, either truncated or rounded up, which will usually not be aligned to a power-of-two memory address. The authors of [17] allude to the importance of keeping the micro-panels aligned. We further speculate that micro-panel alignment may be needed on some systems to facilitate more predictable cache replacement behavior, especially for the L1 cache where it is crucial that the current micro-panel of $\tilde{B}_p$ remain as the 1st loop executes. In these cases, padding may be employed to maintain the desired alignment.

4.4.3 Avoiding workspace

The workspace requirements, in one form or another, cause a myriad of challenges to the 3M algorithms presented in Section 4.3:

- In Algorithms 3M_5 and 3M_4, the workspace itself is unacceptably large. Even Algorithm 3M_3, which bounds workspace by constant blocksizes, requires substantially more workspace than the conventional, assembly-level approach.
- In all algorithms except 3M_1, the potential serialization from ADDM and SUBM subproblems may hinder scalability in many-threaded settings. (Scalability is unaffected, however, if, for applicable algorithms, ADDM and SUBM are fused with the packing stage *and* the packing function is parallelized.)

[21]As noted in Footnote 15 for Algorithm 3M_H, W^r and W^i may be stored separately as real matrices.

```
Algorithm: [ $C$ ] := 3M_HW( $A$, $B$, $C$ )
$C$ := RGEMM3M_HW( $A$, $B$, $C$, 0 )
$C$ := RGEMM3M_HW( $A$, $B$, $C$, 1 )
$C$ := RGEMM3M_HW( $A$, $B$, $C$, 2 )
```

```
[ $C$ ] := RGEMM3M_HW( $A$, $B$, $C$, $q$ )
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 $A, B_j$
    Select $\hat{B}$ from $\{B^r_{jp}, B^i_{jp}, B^{r+i}_{jp}\}$
      using $q$ and PACK $\hat{B} \rightarrow \tilde{B}_p$
    for ( $i = 0 : m-1 : m_C$ )
      Identify $A_{pi}, C_{ji}$ from $A_p, C_j$
      Select $\hat{A}$ from $\{A^r_{pi}, A^i_{pi}, A^{r+i}_{pi}\}$
        using $q$ and PACK $\hat{A} \rightarrow \tilde{A}_i$
      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 $\tilde{A}_i, C_{jih}$
          $C_{jihl}$ := VK3M_HW( $\tilde{A}_{il}$, $\tilde{B}_{ph}$, $C_{jihl}, q$ )
```

```
[ $C$ ] := VK3M_HW( $A$, $B$, $C$, $q$ )
Acquire workspace $W$
$W$ := RKERN( $A$, $B$, 0 )
if ( $q = 0$ )
  $C^r$ := ADDM( $W$, $C^r$ )
  $C^i$ := SUBM( $W$, $C^i$ )
else if ( $q = 1$ )
  $C^r$ := SUBM( $W$, $C^r$ )
  $C^i$ := SUBM( $W$, $C^i$ )
else if ( $q = 2$ )
  $C^i$ := ADDM( $W$, $C^i$ )
```

Figure 6: An abbreviated pseudo-code for Algorithm 3M_HW (top-left), implemented in terms of RGEMM3M_HW (bottom-left) and virtual micro-kernel VK3M_HW (bottom-right).

- In Algorithms 3M_3, 3M_2, and 3M_1, the use of packing space to store summed terms require awkward blocksize reductions by two-thirds, which may cause alignment issues. Those algorithms also use a smaller values of m_C, n_C, or k_C, which in turn means the cost of data movement through the cache hierarchy is amortized over fewer flops.

At first glance, it may seem that the workspace requirements, and all of the ways they complicate the 3M algorithms, are unavoidable. However, there is an optimization which nearly eliminates workspace altogether.

The most straightforward way to explain this optimization is in the context of 3M_H.[22] We begin with three separate calls to a slightly modified RGEMM, as depicted in Figure 6 (top-left). Here, the first invocation of RGEMM will focus solely on computing the $A^r B^r$ term of the 3M method, as if A^r and B^r were each standalone real matrices. (The second and third invocations of RGEMM will focus on the other two terms of the 3M method.) We also pass in a new value q that encodes which of the three "phases" is being executed. The phase value is first used by the packing facility to determine the appropriate contents of $\tilde{A}_i$ and $\tilde{B}_p$, as shown in Figure 6 (bottom-left). A modified virtual 3M micro-kernel, shown in Figure 6 (bottom-right), then uses the phase information to determine how the current micro-kernel product is accumulated into the current micro-tile. For example, in the case of the the first phase, $\tilde{A}_i$ and $\tilde{B}_p$ are packed with A^r_{pi} B^r_{jp}, respectively. Then, within the modified virtual micro-kernel, the intermediate product $A^r B^r$ is added to the real part of the micro-tile, and also subtracted from the imaginary part. The second and third phases proceed in a similar manner.

This modified 3M_HW algorithm addresses nearly all of the challenges mentioned above:

- Workspace is now reduced to only $\mathbb{C}^{m_R \times n_R}$. This workspace is also used only within the virtual micro-kernel. As a result, it is typically small enough to be statically allocated on the function stack. This

[22]Large workspace requirements can also be eliminated from Algorithms 3M_5 and 3M_4 in a manner similar to that of 3M_H. The workspace optimization can also be applied to 3M_3, 3M_2, and 3M_1. Even though, at these levels, S_A and S_B have already been folded into the packing formats of $\tilde{A}_i$ and $\tilde{B}_p$, these algorithms still benefit from the reduction of workspace needed to hold W^r and W^i.

greatly simplifies workspace allocation, especially in the context of multithreading.

- As with 3M_1, this algorithm benefits from the fact that the ADDM and SUBM components, which reuse the intermediate product W during the first and second phases, are embedded within the virtual micro-kernel VK3M_HW. This means that scalability is not hindered by those accumulation steps. In fact, multithreading is almost completely unaffected.[23]
- Performing the computation in three phases allows each phase to use unreduced (e.g. optimal) values of k_C when packing and computing with micro-panels of A and B. While not as impactful on performance, m_C and n_C may also be used unmodified.

Since changes are limited to the packing facility and the virtual micro-kernel, 3M_HW causes no significant disruption to the level-3 operation infrastructure within BLIS or the underlying multithreading framework. The phase information, passed down to the micro-kernel from outside the 5th loop, can be hidden within existing abstractions for most of its journey. And even though we are only packing data from one phase at a time, it is still possible to scale by $\alpha \in \mathbb{C}$ during packing.[24]

The most significant drawback of the workspace optimization employed by Algorithm 3M_HW is that it does not extend to the two-operand level-3 operations TRMM and TRSM. This stems from the fact that, in these operations, one of the operands involved in the matrix product (matrix B) is being overwritten. Thus, the second and third stages will not have the necessary input values available in order to perform the computation associated with those phases.

It may be of interest to some readers to point out that OpenBLAS (and the GotoBLAS before it) takes a similar approach to implementing `cgemm3m` and `zgemm3m`, except that the workspace optimization is applied to that project's equivalent of 3M_4. Also, since OpenBLAS encodes the entire macro-kernel in assembly code, an entirely new assembly-coded kernel specific to 3M must be written (and maintained) for each target architecture. In contrast, BLIS allows us to construct our 3M_HW algorithm in terms of a portable virtual micro-kernel, which can then leverage existing micro-kernel code. Now, the OpenBLAS approach is slightly more efficient, because those custom kernels can, based on the current phase being executed, write to the real and imaginary parts of each micro-tile *directly* from registers. In the BLIS-based approach, the real micro-kernel writes to a temporary $m_R \times n_R$ workspace, and then the virtual 3M micro-kernel uses the phase value to determine how to update C, which requires an extra $m_R n_R$ memory operations for each call to the virtual micro-kernel. In practice, though, these extra $m_R n_R$ memops come at a relatively minor cost since the workspace micro-tile resides in the L1 cache.

4.4.4 Kernel support for general stride

The BLIS framework exposes separate row and column strides of matrix operands in its native API, and thus tracks both strides internally. The BLIS micro-kernel interface similarly exposes separate row and column strides for matrix C, which means that BLIS micro-kernels are required to support matrices stored with general stride (such as non-contiguous slices of higher dimensional tensors). This feature can be used to specify an "in-place" update to only the real, or only the imaginary part of the micro-tile of C. For example, in VK3M_1, general stride support allows the $A^{r+i}B^{r+i}$ micro-kernel product to be accumulated directly into C^i, without first storing the product to temporary workspace.

However, updating the micro-tile of C in the case of general stride can degrade performance for two reasons. First, and most obviously, when both row and column strides are non-unit, the micro-kernel often cannot employ vector store instructions to output an entire register's contents to contiguous memory.

[23] Scalability within Algorithm 3M_HW is only hindered somewhat when the number of threads is high relative to problem size, because parallelism may only occur *within* the three phases. Threading *across* phases cannot be employed until support is added for multithreading within the 4th loop around the micro-kernel, which will allow multiple threads to update the same region of C in a mutually exclusive manner.

[24] Notice that scaling during packing may be employed by Algorithm 3M_HW, even though any given phase only packs the real, imaginary, or summed values. For example, let as assume that we have decided to apply a scalar $\alpha \in \mathbb{C}$ to matrix A. To accomplish this, during the $q = 0$ phase A^r_{pi} would be packed as $(\alpha^r A^r_{pi} - \alpha^i A^i_{pi}) \rightarrow \tilde{A}_i$. Then, when $q = 1$, A^i_{pi} would be packed as $(\alpha^r A^i_{pi} + \alpha^i A^r_{pi}) \rightarrow \tilde{A}_i$. And finally, when $q = 2$, A^{r+i}_{pi} would be packed as $\left((\alpha^r + \alpha^i)A^r_{pi} + (\alpha^r - \alpha^i)A^i_{pi}\right) \rightarrow \tilde{A}_i$.

Algorithm	F.E. memops required to ...[a]							
	update micro-tiles C^i, W^r, W^i	reuse W^r, W^i	pack $\tilde{A}_i$	form S_A	move $\tilde{A}_i$ from L2 to L1 cache[b]	pack $\tilde{B}_p$	form S_B	move $\tilde{B}_p$ from L3 to L1 cache[c]
3M_H[d]	$4mn\frac{k}{k_C}$	$8mn\left[\frac{k}{k_C}\right]$	$3\cdot 2mk\frac{n}{n_C}$	$3mk\left[\frac{n}{n_C}\right]$	$3\cdot mk\frac{n}{n_R}$	$3\cdot 2kn$	$3kn$	$3\cdot kn\frac{m}{m_C}$
3M_5[d]	$4mn\frac{k}{k_C}$	$8mn\left[\frac{k}{k_C}\right]$	$3\cdot 2mk\frac{n}{n_C}$	$3mk\frac{n}{n_C}$	$3\cdot mk\frac{n}{n_R}$	$3\cdot 2kn$	$3kn$	$3\cdot kn\frac{m}{m_C}$
3M_4	$4mn\frac{k}{k_C}$	$8mn\frac{k}{k_C}$	$3\cdot 2mk\frac{n}{n_C}$	$3mk\frac{n}{n_C}$	$3\cdot mk\frac{n}{n_R}$	$3\cdot 2kn$	$3kn$	$3\cdot kn\frac{m}{m_C}$
3M_3[e]	$4mn\frac{k}{k_C}$	$8mn\frac{k}{k_C}$	$3\cdot 2mk\frac{3n}{n_C}$	$3mk\frac{3n}{n_C}$	$3\cdot mk\frac{n}{n_R}$	$5kn$		$3\cdot kn\frac{m}{m_C}$
3M_2[e]	$4mn\frac{k}{k_C}$	$8mn\frac{k}{k_C}$	$5mk\frac{3n}{n_C}$		$3\cdot mk\frac{n}{n_R}$	$5kn$		$3\cdot kn\frac{3m}{m_C}$
3M_1[e]	$4mn\frac{3k}{k_C}$	$8mn\frac{3k}{k_C}$	$5mk\frac{n}{n_C}$		$3\cdot mk\frac{n}{n_R}$	$5kn$		$3\cdot kn\frac{m}{m_C}$

Table 1: F.E. memops incurred by various 3M algorithms, broken down by stage of computation. Note: We intentionally factor some values as $3 \cdot e$. This unsimplified expression occurs wherever the same memop term e is incurred three times as a result of the three intermediate products present in all 3M algorithms.

[a] We express the number of iterations executed in the 5th, 4th, and 3rd loops as $\frac{n}{n_C}$, $\frac{k}{k_C}$, $\frac{m}{m_C}$. (The total number of iterations executed in the 2nd loop also appears as $\frac{n}{n_R}$.) The precise number of iterations along a dimension x using a cache blocksize x_C would actually be $\lceil \frac{x}{x_C} \rceil$. Simlarly, when blocksize scaling of $\frac{1}{3}$ is required, the precise value $\lceil \frac{x}{\lfloor x_C/3 \rfloor} \rceil$ is expressed as $\frac{3x}{x_C}$. These simplifications allow easier comparison between algorithms while still providing meaningful approximations. special cases of interest where (a) $x = x_C$ (or, if blocksize scaling is required, $x = \frac{1}{3}x_C$), and (b) $x \gg x_C$.

[bc] The cost of moving micro-panels of $\tilde{A}_i$ and $\tilde{B}_p$ from the L2 and L3 caches, respectively, to the L1 cache is incurred entirely within the primitive's real domain micro-kernel, where those memops can be mostly hidden (i.e. executed concurrently with useful flops), especially when prefetching is employed. Therefore, these terms do not substantially contribute to the overall runtime costs.

[d] Iteration terms enclosed in brackets apply only when the workspace optimization is employed.

[e] Merged table cells for these 3M algorithms indicate that the packing of $\tilde{A}_i$ is merged with the formation of S_A, and/or the packing of $\tilde{B}_p$ is merged with the formation of S_B.

Exceptions exist, such as with the "scatter" instructions found in the KNC instruction set.[25] However, while these instructions concisely capture the desired operation, their performance is still penalized at runtime due to the non-contiguity of the elements being updated. Second, storing elements to (or loading elements from) memory individually can be instruction-intensive, requiring one instruction just to extract an element from a vector register and another to output that element to memory.[26] By contrast, row or column storage of C allows multiple elements to be output to memory with a single vector instruction.

Notice that, in the case of VK3M_1, shown in Figure 5 (right), we can speed up the runtime of the real micro-kernel by instead writing the $A^{r+i}B^{r+i}$ product to temporary (contiguous) workspace. However, we must then incur the cost of copying that temporary product to the current micro-tile of C. Thus, it is not clear that the general stride feature always provides a net benefit. Indeed, whether it is advantageous to update the real and imaginary parts of C directly (in-place) probably depends on specific properties of the hardware.

[25] AVX2 contains instructions for gathering data from non-contiguous memory locations to a single register, but lacks instructions for the corresponding reverse (scatter) operation.

[26] Using AVX on Sandy Bridge architectures, 9 instructions are needed to output four double-precision elements from a vector register to non-contiguous memory locations, and 15 instructions are needed to output eight single-precision elements.

Algorithm	Total F.E. memops required (Sum of columns of Table 1)	Scaling req'd to maintain cache footprints			l_{L1}: # of times each cache line is moved into L1 cache (per rank-k_C update).		
		k_C	m_C	n_C	l_{L1}^C	l_{L1}^A	l_{L1}^B
3M_H [a]	$12mn\left(\frac{k}{k_C}\right)+9mk\left(\frac{n}{n_C}+\frac{1}{3}\frac{n}{n_R}\right)+3kn\left(3+\frac{m}{m_C}\right)$	1	1	1	3	3	3
3M_5 [a]	$12mn\left(\frac{k}{k_C}\right)+9mk\left(\frac{n}{n_C}+\frac{1}{3}\frac{n}{n_R}\right)+3kn\left(3+\frac{m}{m_C}\right)$	1	1	1	3	3	3
3M_4	$12mn\left(\frac{k}{k_C}\right)+9mk\left(\frac{n}{n_C}+\frac{1}{3}\frac{n}{n_R}\right)+3kn\left(3+\frac{m}{m_C}\right)$	1	1	1	3	3	3
3M_3	$12mn\left(\frac{k}{k_C}\right)+9mk\left(\frac{3n}{n_C}+\frac{1}{3}\frac{n}{n_R}\right)+3kn\left(\frac{5}{3}+\frac{m}{m_C}\right)$	1	1	1/3	3	3	1
3M_2	$12mn\left(\frac{k}{k_C}\right)+5mk\left(\frac{3n}{n_C}+\frac{3}{5}\frac{n}{n_R}\right)+3kn\left(\frac{5}{3}+\frac{3m}{m_C}\right)$	1	1/3	1/3	3	1	1
3M_1	$12mn\left(\frac{3k}{k_C}\right)+5mk\left(\frac{n}{n_C}+\frac{3}{5}\frac{n}{n_R}\right)+3kn\left(\frac{5}{3}+\frac{m}{m_C}\right)$	1/3	1	1	1	1	1

Table 2: Performance properties of various 3M algorithms.

[a] We assume that the workspace optimization is employed for Algorithms 3M_H and 3M_5.

4.5 Summary

In this section, we have presented a family of algorithms for performing the 3M method to compute a complex matrix multiplication.

Table 1 summarizes the F.E. memops incurred by each 3M algorithm previously discussed.[27] Table 2 shows the total F.E. memop cost approximations shown in Table 1, factored to emphasize their relative differences. The table also shows any cache blocksize scaling required as well as the number of times each cache line of each matrix operand is moved into the L1 cache.

Based on this analysis, we find Algorithms 3M_H and 3M_2 to be of particular interest. Algorithm 3M_H has the lowest F.E. memop cost for very large matrices and among the lowest for rank-k_C updates. 3M_H, even with the workspace optimization, is straightforward to implement in BLIS. Algorithm 3M_2, while requiring a less elegant implementation, maintains a relatively low F.E. memop cost when $k = k_C$ and works for all level-3 operations, including TRMM and TRSM. Both algorithms use unreduced values for k_C, the cache blocksize to which performance is most sensitive. By contrast, we can see the steep cost of updating C inherent in Algorithm 3M_1.

In general, employing the 3M method would be most appropriate when performance is a priority. However, for those many applications that are unwilling or unable to risk any additional numerical instability, the 3M method may be of little utility. Thus, we continue our investigation.

5 4m Method

Notice that we can extend the imperative statements in Eq. 2 to complex matrices while forgoing the reformulation shown in Eq. 3:

$$\begin{aligned} C^r &:= C^r + A^r B^r - A^i B^i \\ C^i &:= C^i + A^r B^i + A^i B^r \end{aligned} \qquad (5)$$

We call this the 4M method because it casts a complex matrix multiplication in terms of four real matrix multiplications.

[27] Although some 3M algorithms execute additional flops, we do not tally these flops since (a) they are always the same order of magnitude as the memops, and (b) flops generally take (sometimes many) fewer cycles to execute than memops.

```
Algorithm: [ $C$ ] := 4m_h( $A$, $B$, $C$ )
    $C^r$ := rgemm( $A^r$, $B^r$, $C^r$ ) ;   $C^r$ := rgemm( $-A^i$, $B^i$, $C^r$ )
    $C^i$ := rgemm( $A^r$, $B^i$, $C^i$ ) ;   $C^i$ := rgemm( $A^i$, $B^r$, $C^i$ )
```

Figure 7: An algorithm for applying the 4m method at the highest level of general matrix multiplication.

While 4m does not provide any opportunities to skip computation (via reuse of intermediate products), it avoids the potential for numerical instability that is inherent in 3m. Also, unlike the 3m method, 4m employs each real or imaginary submatrix of A and B twice each, which provides the opportunity to (potentially) reuse each of those operands from some level of cache rather than from main memory.

The 4m method requires $8mnk$ flops. We can see by analysis and inspection that 4m can be implemented as a series of four instances of real domain gemm, where each instance executes $2mnk$ flops.

5.1 Application

As with 3m, the 4m method can be applied around any of the loops shown in Figure 1. Furthermore, when applied within the 5th, 4th, 3rd, and 2nd loops around the micro-kernel, several orderings of the real/imaginary matrix product subproblems are possible. A full analysis is beyond the scope of this article, which we leave as an exercise for the reader. We will, however, present a small selection of algorithms—enough to convey the general ideas behind applying 4m.

5.1.1 Outside the algorithm

Like the 3m method, 4m can be applied outside the matrix multiplication algorithm in a straightforward manner by simply implementing Eq. 5 as a series of calls to rgemm on the real and imaginary submatrices of A, B, and C. This is shown in Figure 7 as Algorithm 4m_h.

Notice that the 4m method does not require workspace at any level.

Since Algorithm 4m_h uses rgemm unmodified, no changes to cache blocksizes or packing formats are necessary. Similarly, multithreading *within* any single invocation to rgemm is unaffected, with the only limitation being, as with 3m, that multithreading may not be extracted *across* calls to rgemm unless the 4th loop is multithreaded (which would provide the ability for threads to update the same parts of matrix C in a mutually exclusive manner, as mentioned previously in Footnote 23).

Since rgemm requires no changes, it is quite easy to implement 4m_h within the BLIS framework. However, 4m_h fails to achieve any significant reuse of the real and imaginary parts of A and B. That is, unless A and B are quite small, the fundamental elements that would be reused in a subsequent call to rgemm will have already been evicted from the L1 and L2 caches.

5.1.2 1st loop around the micro-kernel

Algorithm 4m_1a in Figure 8 (left) encodes the result of applying 4m within the 1st loop around the micro-kernel. This application is done in a manner similar to that of Algorithm 3m_1, whereby the loop body of the 1st loop computes all intermediate 4m products. As with 3m_1, the real gemm micro-kernel serves as the primitive. Note that Algorithm 4m_1a inlines the 4m computation directly into the loop rather than abstract those statements into a separate virtual micro-kernel. (This is *only* done to allow textual comparison to a different algorithmic variant, 4m_1b, which we will discuss shortly. A virtual micro-kernel may still be employed for both algorithms.)

As with Algorithm 3m_1, it makes sense to pack both both $\tilde{A}_i$ and $\tilde{B}_p$ with interleaved real and imaginary micro-panels. This format is similar to the one introduced in Section 4.3.5, except here we omit the third micro-panel in the 3-tuple (or third "sub-panel"). This results in $s = \lceil m_C/m_R \rceil$ pairs of micro-panels packed in $\tilde{A}_i$ and $t = \lceil n_C/n_R \rceil$ pairs of micro-panels packed in $\tilde{B}_p$.

Targeting this loop carries some benefits similar to those of 3m_1. For example, the invocations of the micro-kernel primitive may be hidden within a virtual 4m micro-kernel, which abstracts the 4m details

```
Algorithm: [ $C$ ] := 4M_1A( $A$, $B$, $C$ )

for ( $j = 0 : n-1 : n_C$ )
  Identify $B_j, C_j$ from $B, C$
  for ( $p = 0 : k-1 : k_C$ )
    Identify $A_p, B_{jp}$ from $A, B_j$
    PACK4MINTERLEAVED $B_{jp} \rightarrow \tilde{B}_p$
    for ( $i = 0 : m-1 : m_C$ )
      Identify $A_{pi}, C_{ji}$ from $A_p, C_j$
      PACK4MINTERLEAVED $A_{pi} \rightarrow \tilde{A}_i$
      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 $\tilde{A}_i, C_{jih}$
          Let $\tilde{A}_{il} = \hat{A}, \tilde{B}_{ph} = \hat{B}, C_{jihl} = \hat{C}$
          Expose $\hat{A} = \begin{pmatrix} \hat{A}^r \\ \hat{A}^i \end{pmatrix}, \hat{B} = \begin{pmatrix} \hat{B}^r & \hat{B}^i \end{pmatrix}$
          $\hat{C}^r$ := RKERN( $\hat{A}^r$, $\hat{B}^r$, $\hat{C}^r$ )
          $\hat{C}^r$ := RKERN( $-\hat{A}^i$, $\hat{B}^i$, $\hat{C}^r$ )
          $\hat{C}^i$ := RKERN( $\hat{A}^r$, $\hat{B}^i$, $\hat{C}^i$ )
          $\hat{C}^i$ := RKERN( $\hat{A}^i$, $\hat{B}^r$, $\hat{C}^i$ )
```

```
Algorithm: [ $C$ ] := 4M_1B( $A$, $B$, $C$ )

for ( $j = 0 : n-1 : n_C$ )
  Identify $B_j, C_j$ from $B, C$
  for ( $p = 0 : k-1 : k_C$ )
    Identify $A_p, B_{jp}$ from $A, B_j$
    PACK4MINTERLEAVED $B_{jp} \rightarrow \tilde{B}_p$
    for ( $i = 0 : m-1 : m_C$ )
      Identify $A_{pi}, C_{ji}$ from $A_p, C_j$
      PACK4MINTERLEAVED $A_{pi} \rightarrow \tilde{A}_i$
      for ( $h = 0 : n_C-1 : n_R$ )
        Identify $\tilde{B}_{ph}, C_{jih}$ from $\tilde{B}_p, C_{ji}$
        Let $\tilde{B}_{ph} = \hat{B}, C_{jih} = \hat{C}$
        Expose $\hat{B} = \begin{pmatrix} \hat{B}^r & \hat{B}^i \end{pmatrix}$
        for ( $l = 0 : m_C-1 : m_R$ )
          Identify $\hat{A}_l, \hat{C}_l$ from $\tilde{A}_i = \hat{A}, \hat{C}$
          Expose $\hat{A}_l = \begin{pmatrix} \hat{A}^r_l \\ \hat{A}^i_l \end{pmatrix}$
          $\hat{C}^r_l$ := RKERN( $\hat{A}^r_l$, $\hat{B}^r$, $\hat{C}^r_l$ )
          $\hat{C}^i_l$ := RKERN( $\hat{A}^i_l$, $\hat{B}^r$, $\hat{C}^i_l$ )
        for ( $l = 0 : m_C-1 : m_R$ )
          Identify $\hat{A}_l, \hat{C}_l$ from $\tilde{A}_i = \hat{A}, \hat{C}$
          Expose $\hat{A}_l = \begin{pmatrix} \hat{A}^r_l \\ \hat{A}^i_l \end{pmatrix}$
          $\hat{C}^r_l$ := RKERN( $-\hat{A}^i_l$, $\hat{B}^i$, $\hat{C}^r_l$ )
          $\hat{C}^i_l$ := RKERN( $\hat{A}^r_l$, $\hat{B}^i$, $\hat{C}^i_l$ )
```

Figure 8: Pseudo-codes for two variations of applying 4M to the 1st loop around the micro-kernel. Here, PACK4MINTERLEAVED packs the gth micro-panel of $B_{jp} = \hat{B}$ as $\begin{pmatrix} \hat{B}^r_g & \hat{B}^i_g \end{pmatrix}$ and of $A_{pi} = \hat{A}$ as $\begin{pmatrix} \hat{A}^r_g \\ \hat{A}^i_g \end{pmatrix}$.

from the macro-kernel code. Similarly, since all parallelism is extracted outside the virtual micro-kernel, multithreading is unaffected. And with only the packing and virtual micro-kernel functions in need of modification, this method fits nicely into the existing BLIS framework. Finally, unlike 4M_H, 4M_1A is able to efficiently reuse the micro-panels' real and imaginary sub-panels from the L1 cache.

A significant drawback to this algorithm, however, is that k_C must be reduced by half in order to maintain the same L1 cache footprint for micro-panels of $\tilde{A}_i$ and $\tilde{B}_p$, as well as the same L2 and L3 cache footprints for $\tilde{A}_i$ and $\tilde{B}_p$, respectively. This means that, for sufficiently large k, 4M_1A requires twice as many memops to update C compared to 4M_H, which already executes memops on C twice as often as a conventional algorithm built on an assembly-coded complex micro-kernel. We expect this reduction of k_C to cause a noticeable performance penalty.

Algorithm 4M_1B, shown in Figure 8 (right), reorders the 4M computation within the 1st loop. The idea here is that instead of computing with the real and imaginary sub-panels of each micro-panel of $\tilde{A}_i$ before moving to the next iteration of the 2nd loop, all computation associated with the hth real sub-panel of $\tilde{B}_p$ is performed before moving on to the hth imaginary sub-panel. Only *then* does the 2nd loop iterate. In other words, we have applied a loop *fission* transformation, splitting the 1st loop into two loops, one for the real sub-panel of the current micro-panel of $\tilde{B}_p$, and one for the imaginary sub-panel. This reordering forfeits reuse of the real and imaginary sub-panels of $\tilde{A}_i$ from the L1 cache, and instead reuses them from the L2 cache. However, since this algorithm does not oscillate back and forth between accessing the real and imaginary sub-panels of the current micro-panel of $\tilde{B}_p$, those sub-panels' k_C dimension need not be reduced by half.[28] This means that, for large enough values of k, Algorithm 4M_1B updates C half as often as 4M_1A. Unfortunately, Algorithm 4M_1A cannot be be applied to TRSM, though it may be used for TRMM.

Notice that we could further fissure the two instances of the 1st loop in Algorithm 4M_1B into four instances. This would call for a different packing format on $\tilde{A}_i$ similar to the one employed by Algorithm 3M_2, where $A_{pi} = \hat{A}$ is packed as $\begin{pmatrix} \hat{A}^r \\ \hat{A}^i \end{pmatrix} \rightarrow \tilde{A}_i$. Also, notice that this hypothetical algorithm 4M_1C (which we omit for space reasons) is equivalent to what would be 4M_2A, since it would essentially expose four instances of the primitive naturally found in the 2nd loop around the micro-kernel.

5.2 Refinements and other implementation details

5.2.1 Handling alpha and beta scalars

Generally speaking, arbitrary scalars α and β in 4M-based algorithms may be handled in a manner similar to that of 3M.

Note that if α is real, the negation on the A^iB^i term of Eq. 5 may be implemented by negating the α scalar, which is passed into each call to the matrix multiplication primitive. If α is complex, then the scalar must be applied to a copy of A or B, which may be done during packing. In that case, $\alpha = -1$ can then be passed into the primitive to negate the A^iB^i term.

Unfortunately, since 4M does not require workspace, there is no natural place to apply a complex-valued α scalar in algorithms that implement 4M above the level at which packing takes place. This includes Algorithm 4M_H as well as hypothetical algorithms that target the 5th loop (i.e. 4M_5A and 4M_5B). Our workaround to this is to employ the same algorithmic structure prescribed by the 3M workspace optimization discussed in Section 4.4.3, which would transform, for example, Algorithm 4M_H into an Algorithm 4M_HW capable of handling $\alpha \in \mathbb{C}$. The goal here is not be to avoid workspace needed by the basic (scalar-omitting) algorithms—since those algorithms have no such requirement to begin with—but rather to avoid introducing workspace for the sole purpose of supporting complex α scalars. This allows A or B to be scaled as described for $q = 0$ and $q = 1$ in Footnote 24.

If β is real, it can be applied during the first updates to C^r and C^i (for example, when A^rB^r and A^rB^i, respectively, are computed). If β is complex, C must be scaled separately, before accumulating the 4M products. Given that these additional memops must be incurred, it makes sense to use $m_R \times n_R$ workspace, which facilitates the use of vector instructions for up to half of the memops on the micro-tile.

5.2.2 Alignment of packed matrices

Recall that 4M_1 requires that k_C be reduced to maintain micro-panel L1 cache footprints. Since the reduction factor is $\frac{1}{2}$, and since no extra data is being packed, as in the case of the summed sub-panels for the interleaved 3M packing format, alignment of micro-panels does not pose an issue.

5.2.3 Kernel support for general stride

As mentioned previously, 4M does not, strictly speaking, require any workspace (except in some cases of applying a complex α scalar at high levels). However depending on the hardware, workspace may improve performance of the virtual micro-kernel. As discussed in Section 4.4.4, the micro-kernel *can* separately access the real and imaginary parts of an output matrix C that stores those values in pairs, as shown in Figure 8. However, doing so typically incurs a performance penalty. Introducing a small $m_R \times n_R$ workspace would allow the real micro-kernel to employ vector instructions to accumulate the intermediate results. However, as before with 3M, the temporary workspace must then be written back to C. If this technique is applied to any algorithm above the level targeted by 4M_1A, the number of F.E. memops increases two-fold.[29] These

[28]While Algorithm 4M_1B avoids the need to reduce k_C, it does require that m_C and n_C be halved in order to maintain then footprints of $\tilde{A}_i$ and $\tilde{B}_p$ within the L2 and L3 caches, respectively. However, overall performance tends to not be as sensitive to these cache blocksizes as it is to k_C.

[29]This two-fold increase in memops, also mentioned in Footnote [a] of Table 3, can be optimized down to a 50% increase for 4M_1A if the real micro-kernel skips the loading of the C micro-tile elements when the micro-kernel's β parameter is zero. This

Algorithm	F.E. memops required to ...					Scaling req'd to maintain cache footprints		
	update micro-tiles[a] 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	k_C	m_C	n_C
4M_H 4M_5[AB] 4M_4[AB] 4M_3B	$8mn\frac{k}{k_C}$	$8mk\frac{n}{n_C}$	$4mk\frac{n}{n_R}$	$8kn$	$4kn\frac{m}{m_C}$	1	1	1[c]
4M_3A	$8mn\frac{k}{k_C}$	$8mk\frac{2n}{n_C}$	$4mk\frac{n}{n_R}$	$8kn$	$4kn\frac{m}{m_C}$	1	1[d]	1/2
4M_2[AB] 4M_1B	$8mn\frac{k}{k_C}$	$8mk\frac{2n}{n_C}$	$4mk\frac{n}{n_R}$	$8kn$	$4kn\frac{2m}{m_C}$	1	1/2	1/2
4M_1A	$8mn\frac{2k}{k_C}$	$8mk\frac{n}{n_C}$	$4mk\frac{n}{n_R}$	$8kn$	$4kn\frac{m}{m_C}$	1/2	1	1
assembly [b]	$4mn\frac{k}{k_C}$	$4mk\frac{n}{n_C}$	$2mk\frac{n}{n_R}$	$4kn$	$2kn\frac{m}{m_C}$	–	–	–

Table 3: F.E. memops incurred by various 4M algorithms, broken down by stage of computation, with required blocksize scaling.

[a] As described in Section 5.2.1, $m_R \times n_R$ workspace becomes mandatory 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.

[b] An assembly-coded complex implementation may call for different cache blocksize values than in the real domain. However, this is not particularly remarkable, and so we omit the complex-to-real cache blocksize ratios for the assembly case.

[c] In the case of 4M_4A, this unreduced value of n_C assumes the A^rB^r and A^iB^r subproblems are computed consecutively, so that B^r is fully used before packing B^i (to then compute A^iB^i and A^rB^i).

[d] This unreduced value of m_C assumes the A^rB^r and A^rB^i subproblems are computed consecutively, so that A^r is fully used before packing A^i (to then compute A^iB^i and A^iB^r).

additional memory operations may negate some (or all) of the benefit of using contiguous workspace in the first place.

5.3 Summary

In this section, we have presented a family of algorithms for performing the 4M method for complex matrix multiplication.

Table 3 tallies the total number of F.E. memops required by various 4M-based algorithms (including several that were not explicitly presented or discussed). Similarly, Table 4 summarizes the main performance properties.

This analysis confirms that 4M_1A suffers from a higher memop cost of updating C than its siblings (due to a reduced k_C), but reuses F.E. from all matrix operands from the L1 cache, and moves matrix cache lines into the L1 cache only once. Its modifications are also limited to the packing function and virtual micro-kernels, making it easy to implement. Algorithm 4M_1B uses an unreduced k_C blocksize and

is because the 4M_1A virtual micro-kernel has the unique opportunity to update C^r and C^i only after *both* of their micro-panel products have been accumulated to each of W^r and W^i:
$W^r := A^rB^r$; $W^r := W^r - A^iB^i$;
$W^i := A^rB^i$; $W^i := W^r + A^iB^r$;
$C := C + W$

Algorithm	Total F.E. memops required (Sum of columns of Table 3)	Level from which F.E. of matrix X are reused, and l_{L1}: # of times each cache line is moved into the L1 cache (per rank-k_C update).					
		C	l^C_{L1}	A	l^A_{L1}	B	l^B_{L1}
4M_H 4M_5[AB]	$8mn\left(\frac{k}{k_C}\right)+4mk\left(\frac{2n}{n_C}+\frac{n}{n_R}\right)+2kn\left(4+\frac{2m}{m_C}\right)$	MEM	4	MEM	4	MEM	4
4M_4B [a]		MEM	4	MEM	4	MEM	2
4M_4A 4M_3B		MEM	4	MEM	4	L3	2
4M_3A	$8mn\left(\frac{k}{k_C}\right)+4mk\left(\frac{4n}{n_C}+\frac{n}{n_R}\right)+2kn\left(4+\frac{2m}{m_C}\right)$	MEM	4	L2	2	L3	1
4M_2B	$8mn\left(\frac{k}{k_C}\right)+4mk\left(\frac{4n}{n_C}+\frac{n}{n_R}\right)+2kn\left(4+\frac{4m}{m_C}\right)$	MEM	4	L2	1	L3	1
4M_2A		L2	4	L2	1	L1	1
4M_1B		L2	2[b]	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[b]	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

Table 4: Performance properties of various 4M algorithms, including the assembly-based approach.

[a] In 4M_4B, we assume that the 4th loop is fissured into two loops, the first of which computes $A^r_p B^r_{jp}$ and $A^i_p B^r_{jp}$ for all iterations p, and the second of which computes $A^r_p B^i_{jp}$ and $A^i_p B^i_{jp}$.

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

therefore requires fewer memops to update C, but exhibits slightly less efficient use of data from $\tilde{A}_i$. It also requires changes to the macro-kernels, making it somewhat less elegant to support. We would expect all other algorithms to perform, to varying degrees, more poorly than 4M_1B. Also, 4M_1A is the only 4M algorithm that works for TRSM.

While 3M executes many fewer flops, the 4M method should be used instead when numerical stability must remain undiminished—a common requirement in high-performance computing (HPC) applications. Yet even these more general-purpose 4M-based algorithms have weaknesses which must be thoughtfully considered.

6 Performance

In this section we present performance of various implementations of the 3M and 4M methods along with conventional implementations based on assembly-coded micro-kernels.

6.1 General results

6.1.1 Platform and implementation details

Results presented in this section were gathered on a single Dell Zeus C8220z compute node consisting of two eight-core Intel Xeon E5-2680 processors featuring the "Sandy Bridge" microarchitecture. Each core, clocked at 3.1 GHz, provides a peak performance of 24.8 GFLOPS in double precision and 49.6 GFLOPS in

Precision/Domain	Implementation	m_R	n_R	m_C	k_C	n_C
single real	BLIS assembly	8	8	128	384	2048
single complex	BLIS 3M_HW	8	8	128	384	2048
	BLIS 3M_1	8	8	128	384/3	2048
	BLIS 4M_HW	8	8	128	384	2048
	BLIS 4M_1B	8	8	128/2	384	2048/2
	BLIS 4M_1A	8	8	128	384/2	2048
	BLIS assembly	8	4	96	256	2048
	OpenBLAS	8	4	512	256	16240
double real	BLIS assembly	8	4	96	256	2048
double complex	BLIS 3M_HW	8	4	96	256	2048
	BLIS 3M_1	8	4	96	256/3	2048
	BLIS 4M_HW	8	4	96	256	2048
	BLIS 4M_1B	8	4	96/2	256	2048/2
	BLIS 4M_1A	8	4	96	256/2	2048
	BLIS assembly	4	4	64	192	2048
	OpenBLAS	4	4	256	192	10384

Table 5: Register and cache blocksizes used by the various implementations of matrix multiplication, as configured for an Intel Xeon E5-2680 "Sandy Bridge" processor.

single precision.[30] Each socket has a 20MB 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 CentOS 6.3 operating system running the Linux 2.6.32 (x86_64) kernel. Source code was compiled by the GNU C compiler (`gcc`), version 4.7.1. The version of BLIS used in all tests was 0.1.6-51.[31]

Subsequent performance graphs will include results from experimental BLIS implementations of Algorithms 3M_HW, 3M_3, 3M_2, 3M_1, 4M_HW, 4M_1A, and 4M_1B as described in Sections 4 and 5. Also included are results based on conventional, assembly-based micro-kernels written by hand for the Sandy Bridge microarchitecture via GNU extended inline assembly syntax.

All experiments were performed on randomized, column-stored matrices with GEMM scalars held constant: $\alpha = -1$ and $\beta = 1$. In all performance graphs, each data point represents the best of three trials. For 3M results, we report the effective rate of computation—in other words, the rate of flops that would have been achieved if complex arithmetic were being performed in the conventional manner (or via 4M).

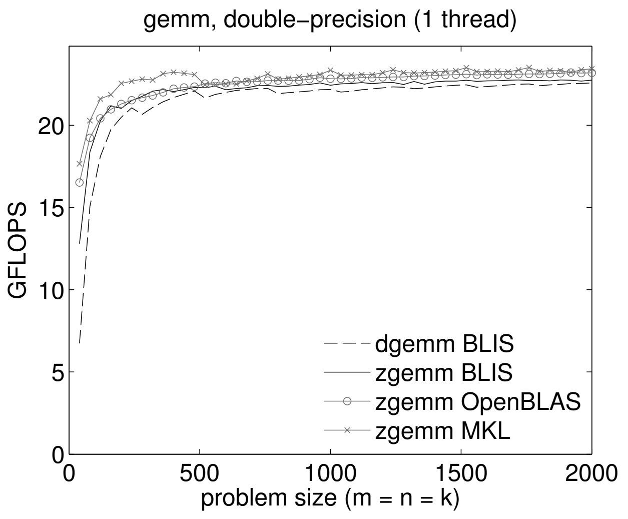
Blocksizes for each of the implementations tested are provided in Table 5. For reference, we also provide the blocksizes for single-precision and double-precision real domain matrix multiplication, as well as those used by complex GEMM implementations in OpenBLAS 0.2.12.

6.1.2 Comparing to other implementations

Before proceeding with the main body of our results, we wish to emphasize that our goal is not simply to compare the performance of every 3M and 4M implementation with that of existing BLAS solutions. Rather, our goal is to study various algorithms for the 3M and 4M induced methods by comparing performance

[30]This system uses Intel's TurboBoost 2.0 dynamic frequency throttling technology. Upon inquiry, system administrators informed us that the processors are well cooled and thus typically run at the maximum allowable frequency of 3.1GHz. This was confirmed by running test drivers through the `perf stat` performance monitoring command.

[31]This version of BLIS may, with high probability, be uniquely identified by the first 10 digits of its `git` "commit" (SHA1 hash) number: c84286d5ce.



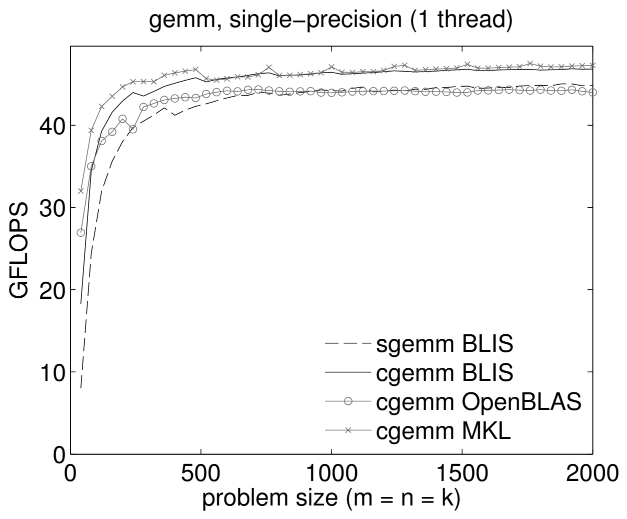


Figure 9: Single-threaded performance of various conventional (i.e., assembly-coded) implementations of double-precision (left) and single-precision (right) complex GEMM on a single core of an Intel Xeon E5-2680 "Sandy Bridge" processor. The theoretical peak performance coincides with the top of the graphs.

primarily within families. Nonetheless, we acknowledge the importance of providing some way to judge the general level of performance of these methods against that of other libraries, and therefore have included Figure 9. This figure serves two purposes. First, it allows the reader to directly compare the conventional implementations of complex matrix multiplication (`zgemm` and `cgemm`) available in BLIS with those provided by OpenBLAS and MKL. The graphs clearly show that BLIS's conventional complex implementations are competitive. Secondly, it introduces data that will recur in subsequent graphs, thus allowing readers to indirectly compare the performance of any induced method implementation with that of OpenBLAS or MKL. Our aim here is to facilitate comparision (albeit indirectly) while simultaneously keeping the focus on comparing different implementations within the same framework (BLIS) and also keeping the graphs uncluttered and readable.

6.1.3 Sequential results

Figure 10 reports performance results for various implementations of double- and single-precision complex matrix multiplication on a single core of the Sandy Bridge processor. The x-axes of the graphs denote the problem size and the y-axes show observed floating-point performance in units of 10^9 flops (gigaflops) per second. For the top graphs, the theoretical peak performance coincides is represented by the dotted line, while in the bottom graphs, the peak coincides with the top of the graphs. 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 5. We justify focusing 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 product frequently occurs within high-performance implementations of more sophisticated DLA operations such as Cholesky, LU, and QR factorizations, and (3) it is the foundation for matrix multiplications where all three dimensions are large.[32]

As expected, implementations based on the 3M method yield effective performance that exceeds the theoretical peak of the processor core. Also, we see that 3M_HW outperforms 3M_1. Recall that this outperformance is expected since 3M_1 must use a reduced value for k_C. This leads to fewer flops being performed per update of C.

In the case of 4M, the situation is reversed: 4M_1A outperforms 4M_HW, albeit modestly. We attribute

[32]The authors of [10] 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 one depicted in Figure 1. This topic deserves special treatment, and thus is beyond the scope of the present article.

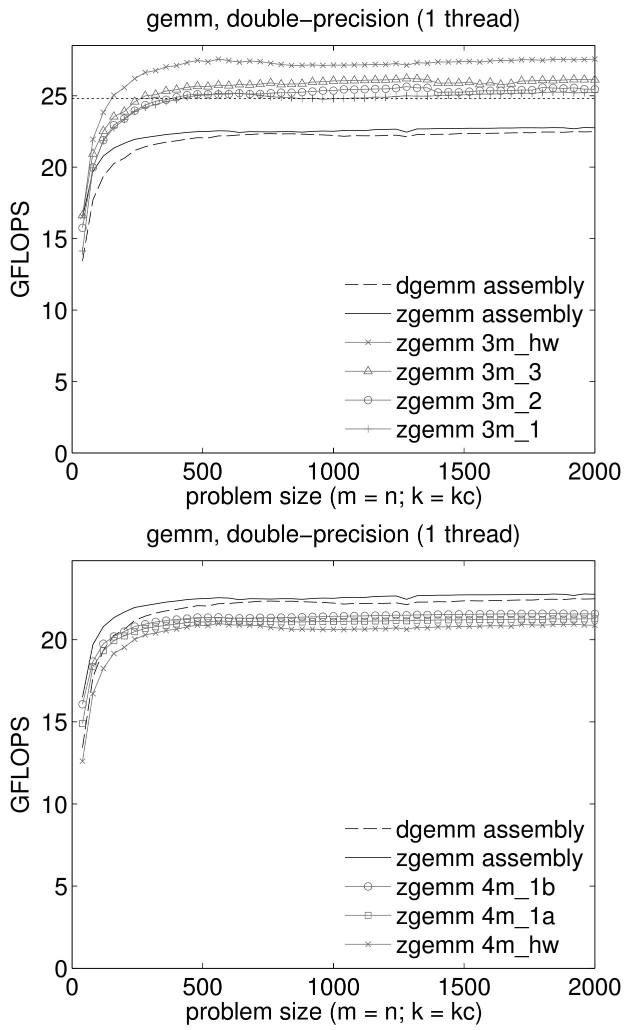


Figure 10: Single-threaded performance of various implementations of double-precision (left) and single-precision (right) complex GEMM on a single core of an Intel Xeon E5-2680 "Sandy Bridge" processor. The theoretical peak performance (in terms of effective GFLOPS) coincides with the dotted line in the top graphs and as the top of the graph in the bottom graphs.

this to the fact that the computation in 4M_HW is organized in four phases, where each phase sweeps through the entire matrices to compute one of the four products in Eq. 5, which leaves very little opportunity for data to be reused from the L1 or L2 caches. This poor reuse of data, particularly when updating F.E. of C, is enough to offset the main benefit of 4M_HW, which is that the full value of k_C can be used. By contrast, 4M_1A efficiently moves data into the L1 cache, where it is then reused towards each of the four 4M products. This efficient data reuse from the L1 cache is enough to outweigh the penalty incurred from a halved k_C. Also, the data for 4M_1B show that using an unreduced k_C while reusing micro-panels of $\tilde{A}_i$ from the L2 cache (instead of the L1 cache) results in a slight net benefit relative to 4M_1A.

We can see from Figure 10 that, while lower level applications of 4M tend to yield higher performance, the pattern is reversed for 3M. Recall that, unlike 4M, 3M inherently does not reuse any of its *input operands*, only its *intermediate products*. In fact, neither 3M_1 nor 3M_HW reuse input data beyond what is prescribed by the overall matrix multiplication algorithm (depicted in Figure 1). Ultimately, 3M_1 suffers because it reorders and interleaves the three phases of 3M_HW down to the micro-panel level. This reordering carries no apparent benefit (beyond compatibility with TRMM and TRSM), but results in lower performance because of the necessarily reduced k_C.

The graphs in Figure 10 also report performance for double- and single-precision real domain GEMM. These curves provide a reference against which we may judge the overall success of the 4M (and 3M) implementations. On the Intel Xeon, all 4M implementations fall within 5-10% of their real domain "benchmarks" (i.e., a comparable implementation of real GEMM using the same matrix primitive or kernel).

6.1.4 Multithreaded results

Figure 11 shows double- and single-precision performance using 16 threads, with one thread bound to each core of the processor. Performance is presented in units of gigaflops per core to provide easy visual assessment of scalability. For all implementations, we employed 2-way parallelism within the 5th loop and 8-way parallelism within the 3rd loop, for a total of 16 threads. This parallelization scheme was chosen as follows [19]. The two sockets of the Xeon E5-2680 each have an L3 cache that is shared among those sockets' cores. This encourages two-way parallelism in the 5th loop, which, for large enough n, 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 eight cores of either socket can pack separate blocks $\tilde{A}_i$ into their private L2 caches. Thus, when 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.

Similar to the single-threaded case, we find that 4M_HW underperforms 4M_1A and 4M_1B, except this time by a larger margin. We conducted a separate empirical study[33] to investigate the proportion of 4M_HW performance degredation attributable to (1) the reading of F.E. of A and B and (2) the updating of F.E. of C. We found that most of the 4M_HW penalty visible in Figure 11 is due to the inefficient updating of individual F.E. of C. Because of the structure of 4M_HW, which executes in four phases, the F.E. of C are almost never reused from any level of cache. Similarly, the study confirmed that updating C (and not the reading of A and B during packing) is also the main culprit behind the gap between the 4M_HW performance shortfall in the single-threaded case. This makes sense, since 4M_1A and 4M_1B reuse micro-tiles of C from the L1 and L2 caches, respectively. We suspect that the larger gap present in multithreaded execution is due to the system's memory bandwidth becoming saturated as all 16 threads vie to update their independent micro-tiles nearly simultaneously.

The double-precision results for 4M_1A and 4M_1B once again fall within 10% of the benchmark real domain implementation. Surprisingly, in the single-precision case, the 4M_1A and 4M_1B implementations track closely with the performance of their real GEMM benchmark.

In general, multithreaded 3M implementations perform somewhat more poorly relative to the single-threaded case. This is likely due to the additional memops constituting a higher percentage of overall runtime cost (relative to 4M), which more quickly leads to memory bandwidth saturation.

[33]The details of this study extend beyond the scope of this article.

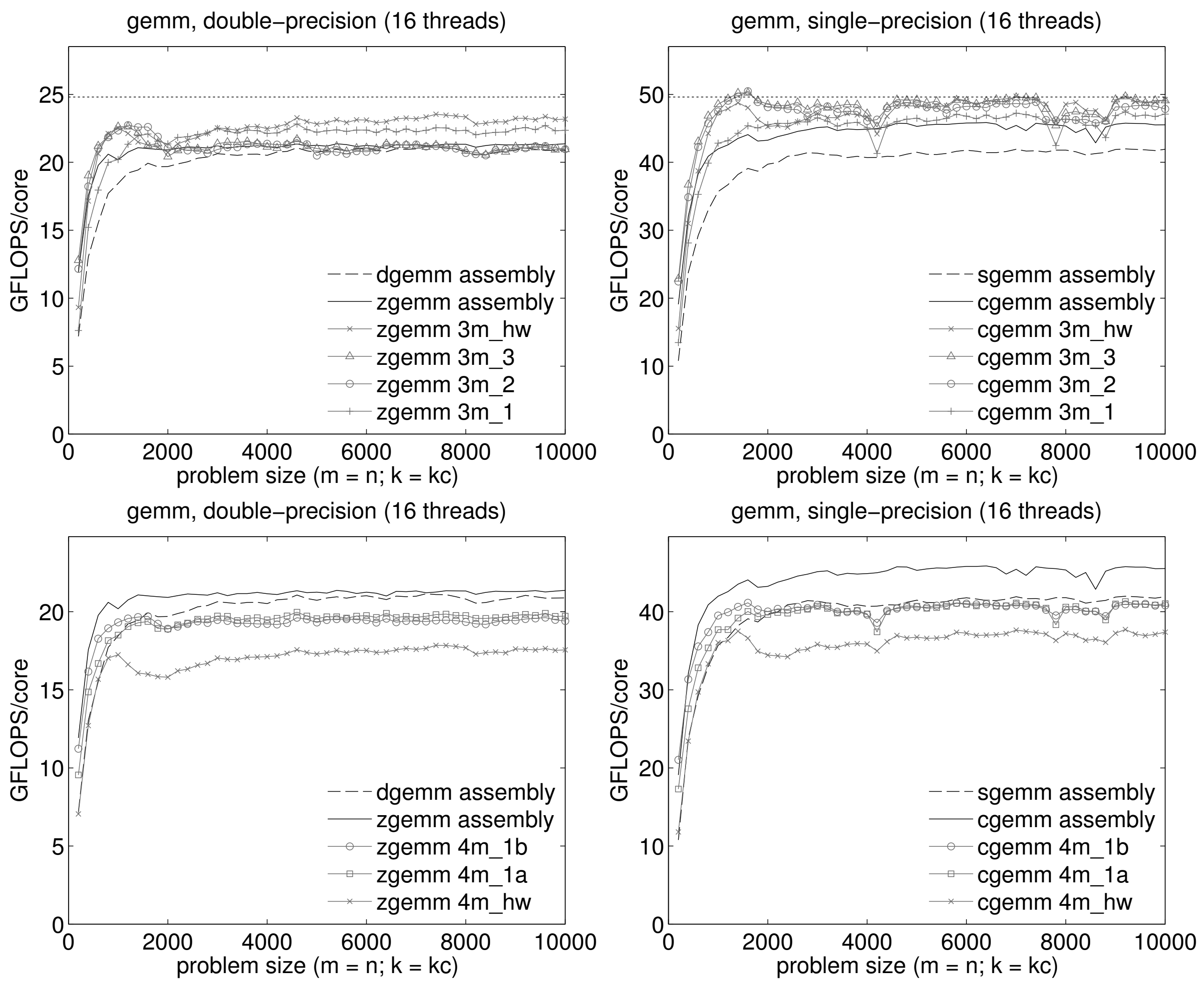


Figure 11: Multithreaded performance of various implementations of double-precision (left) and single-precision (right) complex GEMM on two Intel Xeon E5-2680 "Sandy Bridge" processors, each with eight cores. All data points reflect the use of 16 threads. The theoretical peak performance (in terms of effective GFLOPS/core) coincides with the dotted line in the top graphs and as the top of the graph in the bottom graphs.

7 Further Discussion

7.1 Unifying Observations

In light of our theoretical analyses and empirical results, we now offer a series of observations which, taken together, begin to condense and unify some of the findings of this article.

7.1.1 Variance in memops explains relative performance

All 3M and 4M algorithms execute the same number of flops[34] as the other algorithms within their respective families, and so each algorithm's performance relative to others in its family depends on the number (and circumstance, which we will comment upon shortly) of memops executed. Tables 1 and 3 reveal that the level at which 3M or 4M is applied determines the total number of F.E. memops required. In the case of 3M, the loop level determines the F.E. memop count in two ways. First, the targeted loop determines the packing strategy, which sometimes requires blocksize scaling in order to avoid over-occupying the various levels of cache. Second, the loop being targeted for 3M determines whether the computation of S_A and S_B can be fused with the packing of $\tilde{A}_i$ and $\tilde{B}_p$. Blocksize scaling causes the total number of memops to increase, while fusing the computation of S_A and S_B with packing causes the total number of memops to decrease. In the case of 4M, only the former effect is applicable. This, combined with a less drastic blocksize scaling factor, results in less variation of F.E. memops across 4M algorithms.

7.1.2 A spectrum of algorithms

Algorithm 4M_HW moves each fundamental element of A, B, and C from main memory to the L1 cache a total of four times each—twice for the 4M method, and twice due to incidental spatial (i.e. cache line) proximity—per rank-k_C update. And of the reusable elements that are packed to $\tilde{A}_i$ and $\tilde{B}_p$, none are ever reused *across* calls to RGEMM. Thus, 4M_HW makes poor use of each cache line that is moved through the cache hierarchy.

Algorithm 4M_1 moves each fundamental element of A, B, and C from main memory to the L1 cache only once per rank-k_C update. Furthermore, only elements that are needed are moved; there is virtually no excess data movement due to spacial proximity. And as packed micro-panels of $\tilde{A}_i$ and $\tilde{B}_p$ are moved into the L1 cache, they are reused from the L1 cache. Thus, 4M_1A makes very efficient use of each cache line that is moved through the cache hierarchy.

We can observe, then, that 4M-based algorithms exist along a spectrum with the following trade-off properties: (1) The higher the 4M method is applied, the lower the efficiency of data movement through the caches, and the lower the 4M method is applied, the higher the efficiency of data movement[35]; (2) Higher applications can use larger primitives, while lower applications must use smaller primitives. Larger primitives tend to require fewer modifications to the underlying matrix multiplication framework because they hide details (particularly with regards to packing and multithreading) that surface when using smaller primitives.[36]

Notice that data efficiency in the context of 3M is somewhat different. 3M inherently does not reuse any input data beyond that which is already prescribed by the matrix multiplication algorithm. And we suspect that the total number of F.E. memops required, which exhibits greater variation within the 3M family, generally has a greater effect on performance than the degree of redundant cache line movements. For example, our empirical results suggest that, at least in the (important) case of-k_C update, the excessive number of F.E. memops incurred by 3M_1 outweighs the combined benefit of accessing operands A, B, and C only once each and reusing micro-panels of $\tilde{A}_i$ and $\tilde{B}_p$ from the L1 cache.

[34]In this hypothetical accounting of flops, we assume that the workspace optimization is employed for 3M, and thus the flops involved in the formation of S_A and reuse of W^r, W^i is constant across all 3M algorithms listed in Table 1.

[35]Here, a "higher" or "lower" level application of 4M (or 3M) refers to the targeted loop's relative position within the overall matrix multiplication algorithm, as depicted in Figure 1.

[36]We demonstrated, however, that when the level at which 4M is applied falls just outside the micro-kernel, most code changes can remain hidden (within the virtual micro-kernels) and thus those algorithms represent a programming "sweet spot" within the BLIS framework, even if targeting this level leads to sub-optimal performance.

7.1.3 The conventional approach: a special case

The conventional approach discussed in Section 3 corresponds to the special case whereby 4M is applied at the scalar level, within a single rank-1 update of the micro-kernel's loop. This special case resides at the lowest end of the spectrum of 4M algorithms. Here, efficiency is maximized, as real and imaginary elements are reused only once they have been moved into registers, where they are computed upon by an extremely small primitive: a single vector instruction. The cost of this efficiency, though, is that some form of swizzling must be performed at the sub-vector level, sometimes resulting in the programming difficulties suggested by Figure 2 (right). Note that in the cases of higher level applications (e.g. 4M_1A and higher), swizzling is somewhat implicit, and is analogous to the referencing of different real or imaginary sub-regions of A and B when calling the primitive defined for that level. Of course, this implicit swizzling is not entirely "free" since it often requires the data to be rearranged in advance. However, it can be mostly free if packing must occur anyway, as is the case with several mid-to-lower level applications of 4M, and whatever overhead is incurred in the modified packing operation is amortized over plenty of floating-point operations.

7.1.4 The impact of storage

Another contributing factor to the performance of complex matrix multiplication, one might argue, is actually in the *storage* of complex numbers. The traditional, interleaved pair-wise storage of real and imaginary values increases the potential efficiency of data movement from main memory to registers, and also unlocks the potential to reuse that data from registers, but the price of realizing those efficiencies is assembly-level swizzling. If complex numbers were stored in such a way that did not favor complex arithmetic being performed in registers, the alternative methods based on real primitives would be more obvious. Also, the task of expressing (programming) complex micro-kernels simplifies greatly, as demonstrated by algorithm 4M_1A, if we simply forgo pursuit of the best-case scenario of reusing F.E. from registers, and instead settle for reuse from the L1 cache. Granted, with this trade-off comes lower performance. But evidence suggests that the primary cause behind the performance drag on 4M algorithms is not reusing F.E. from a more distant level of cache, but rather the doubling or quadrupling of memops on C, combined with the non-contiguous nature of those data accesses (which also precludes use of most vector load and store instructions).[37] If there existed an induced method that did not incur such high costs when updating C, that induced might perform competitively with conventional implementations.[38] Indeed, 4M-based algorithms would be able to update C much more efficiently if complex matrices were stored with real and imaginary parts divided into separate (contiguous row- or column-stored) matrices.[39] This separated storage format would allow the underlying real micro-kernel to always use vector load and store instructions when accessing the output matrix, thus (largely) avoiding the performance penalty incurred when accessing only real or only imaginary values. However, this storage scheme is relatively unconventional within the HPC community, while the traditional interleaved format is quite entrenched. Therefore, it would be unreasonable to expect applications to adapt.

Given these realities, we are left to conclude that, if there exists a better solution, it will likely not be found through the 4M method.

[37]This observation makes sense when one considers that doubling or quadupling the memop cost on the input operands (or their packed equivalents) represents a lower order term, $mk_C + k_C n$, compared to the potentially much larger impact from increasing memops on the $m \times n$ output matrix C.

[38]We hypothesize that the upper bound of measured performance (in flops per second) of the overall complex matrix multiplication via induced methods will be equal to the performance that *would* have been observed of an real domain matrix multiplication (based on the same kernel primitive) if its total memop cost were reduced by half. Appendix A provides theoretical evidence that this upper bound can be approximated by the performance of a conventional complex implementation.

[39]The 3M method has a similar affinity towards separate storage of real and imaginary parts. Except since the goal of 3M is to reuse intermediate terms, extra memory operations (outside of the micro-kernel) are unavoidable, even if they are performed efficiently via vector instructions. By contrast, separating complex matrices into real and imaginary matrices would benefit 4M algorithms by avoiding workspace-related memops *and* ensuring that the remaining memops on C can execute via vector instructions.

7.2 Progress and the (challenging) path forward

The 4M method comes with concrete benefits. It allows developers to focus on fewer and simpler micro-kernels—reducing coding and maintenance burdens both because there is less code, and also because that remaining code tends to be more easily expressed in assembly language. And it builds on existing kernels, allowing it to be encoded portably within a BLAS-like framework such as BLIS, where it extends to any level-3 operation with only modest effort. Our experience thus far suggests that 4M can induce "placeholder" implementations that deliver acceptable (even if short of optimal) performance, which may still be of value in situations where time, resources, and/or expertise is limited.

Unfortunately, the 4M family of induced algorithms also collectively exhibit real limitations which appear inherent in nature. Any future work in this area will be forced to confront these challenges, or find novel ways of circumventing them altogether:

- **Number of calls to primitive.** Instantiating 4M algorithms can result in close to a four-fold increase in function call overhead relative to a conventional assembly-based code, or an eight-fold increase when virtual micro-kernels are employed to hide implementation details. The small amount of logic present in virtual micro-kernels may also contribute noticeable overhead. These extra costs become an issue when they cannot be amortized across a sufficient number of floating-point operations, either because the costs themselves are large or because the L1 cache size limits k_C, and thus the number of cycles worth of computation that can be performed by a single micro-kernel call.

- **Inefficient reuse of input data from A, B, and C.** Higher and mid-level applications of 4M inefficiently move data from the input matrices through the memory hierarchy, only to let those elements be evicted from cache before they are reused. Only the lowest level algorithms reuse real and imaginary F.E. once they have been moved into cache.

- **Non-contiguous output to C.** 4M-based algorithms must, at some level, update only the real and then only the imaginary parts of the output matrix, twice each, which tends to incur a non-trivial increase in runtime cost. Long-term trends in this area are not favorable to 4M: First, as vector register lengths are extended with newer architectures and instruction sets, the relative cost of updating individual real or imaginary values will only grow; Second, the cost of memory access relative to the cost of floating-point computation will likely continue its upward trajectory, further magnifying the cost of repeatedly accessing the micro-tile of C.

- **Framework accommodation.** Many 4M algorithms would require significant and disruptive changes to BLIS's underlying code. This effect would likely hold even for other frameworks, because those changes to the framework stem from changes to the matrix multiplication algorithm itself. Ideally, this algorithmic specialization would be avoided.

- **Reduction of k_C.** Some algorithms, such as 4M_1A, may require reducing the k_C blocksize to maintain cache footprints of packed matrices $\tilde{A}_i$ and $\tilde{B}_p$ as well as individual micro-panels within those matrices. This causes the real domain micro-kernel to be called on micro-panels with k dimensions that are, at best, half their optimal size. Consequently, fewer flops are executed per update of C, which means a larger fraction of total runtime is spent accessing elements of the output matrix, thus eroding performance.

- **Interference with multithreading.** Some applications of 4M, particularly those to high- and mid-level loops, complicate (or prevent) the ability to express an ideal partitioning of work and data for many-threaded parallelism, as with Algorithm 4M_HW.

- **Non-applicability to two-operand operations.** Higher level applications, such as 4M_HW, cannot be used to implement TRMM or TRSM without first making workspace-expensive, performance-degrading copies of the input/output operand B.

These issues conspire to limit (to varying degrees) the achievable performance of 4M-based algorithm—a shortfall some may find unacceptable.

8 Conclusions

We began the article with a brief overview of the conventional approach to implementing matrix multiplication in the complex domain. Then, we investigated the 3M method and provided a detailed analysis of the method's applicability to each loop within a commonly accepted algorithm for matrix multiplication, exposing several variants in the process. The more general-purpose 4M method was also discussed, accompanied by an abbreviated analysis of a select set of possible algorithms. Decent performance was observed on a modern Intel system. Finally, we identified key familial properties of the 3M and 4M algorithms, and summarized their benefits and limitations.

We conclude that while the 3M method can yield effective performance that exceeds the peak of the machine, some may be wary of its distinct numerical properties. The 4M method does not forfeit any numerical advantage over the conventional approach, but often lags slightly behind its real domain benchmarks. Given that conventionally-implemented complex matrix multiplication typically slightly exceeds that of its real domain counterpart, this lag may be troublesome to many. Still, 4M provides a family of backstop solutions—a lower bound on attainable complex performance in the absence of a complex kernel. Movement toward the upper bound, characterized by the memop-adjusted flop rate of an optimized real domain matrix multiplication, should be the focus of future work.

Acknowledgements

This research was partially sponsored by grants from the National Science Foundation (Awards CCF-1320112 and ACI-1148125/1340293). 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).

We kindly thank Tze Meng Low for participating in early performance analysis. We also thank the Texas Advanced Computing Center for providing access to the the Intel Xeon "Stampede" compute cluster on which Sandy Bridge performance data was gathered.[40]

References

[1] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. volume 180, 2009.

[2] Emmanuel Agullo, Henricus Bouwmeester, Jack Dongarra, Jakub Kurzak, Julien Langou, and Lee Rosenberg. Towards an efficient tile matrix inversion of symmetric positive definite matrices on multicore architectures. June 2010.

[3] AMD. AMD Core Math Library. `http://developer.amd.com/tools/cpu/acml/pages/default.aspx`, 2012.

[4] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, Jack J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. *LAPACK Users' guide (third ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999.

[5] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker. ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. In *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 120–127. IEEE Comput. Soc. Press, 1992.

[6] James W. Demmel and Nicholas J. Higham. Stability of block algorithms with fast level-3 BLAS. *ACM Trans. Math. Soft.*, 18(3):274–291, September 1992.

[40]The Stampede system was funded by the National Science Foundation via Award ACI-1134872.

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

[8] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. The LINPACK benchmark: Past, present, and future. concurrency and computation: Practice and experience. *Concurrency and Computation: Practice and Experience*, 15(9):803–820, 2003.

[9] J.J. Dongarra, J.R. Bunch, C.B. Moler, and G.W. Stewart. *LINPACK Users' Guide.* Society for Industrial and Applied Mathematics, Philadelphia, 1979.

[10] Kazushige Goto and Robert van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Soft.*, 34(3):12:1–12:25, May 2008.

[11] Kazushige Goto and Robert van de Geijn. High-performance implementation of the level-3 BLAS. *ACM Trans. Math. Soft.*, 35(1):1–14, July 2008.

[12] Nicholas J. Higham. Exploiting fast matrix multiplication within the level 3 BLAS. *ACM Trans. Math. Soft.*, 16(4):352–368, December 1990.

[13] Nicholas J. Higham. Stability of a method for multiplying complex matrices with three real matrix multiplications. *SIAM J. Matrix Anal. App.*, 13(3):681–687, July 1992.

[14] IBM. Engineering and Scientific Subroutine Library. `http://www.ibm.com/systems/software/essl/`, 2012.

[15] Intel. Math Kernel Library. `http://developer.intel.com/software/products/mkl/`, 2014.

[16] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms, Second Edition.* Addison-Wesley, Reading, MA, USA, 1981.

[17] Tze Meng Low, Francisco D. Igual, Tyler M. Smith, and Enrique S. Quintana-Ortí. Analytical modeling is enough for high-performance BLIS. *ACM Trans. Math. Soft.*, 43(2):12:1–12:18, August 2016.

[18] Jack Poulson, Bryan Marker, Robert A. van de Geijn, Jeff R. Hammond, and Nichols A. Romero. Elemental: A new framework for distributed memory dense matrix computations. *ACM Trans. Math. Softw.*, 39(2):13:1–13:24, February 2013.

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

[20] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, 1969.

[21] Top 500 supercomputer sites. `http://www.top500.org/`, 2015.

[22] Field G. Van Zee, Ernie Chan, Robert van de Geijn, Enrique S. Quintana-Ortí, and Gregorio Quintana-Ortí. The libflame library for dense matrix computations. *IEEE Computation in Science & Engineering*, 11(6):56–62, 2009.

[23] Field G. Van Zee, Tyler Smith, Francisco D. Igual, Mikhail Smelyanskiy, Xianyi Zhang, Michael Kistler, Vernon Austel, John Gunnels, Tze Meng Low, Bryan Marker, Lee Killough, and Robert A. van de Geijn. The BLIS framework: Experiments in portability. *ACM Trans. Math. Soft.*, 42(2):12:1–12:19, June 2016.

[24] Field G. Van Zee and Robert A. van de Geijn. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Trans. Math. Soft.*, 41(3):14:1–14:33, June 2015.

[25] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of SC'98*, 1998.

[26] Zhang Xianyi, Wang Qian, and Zhang Yunquan. Model-driven level 3 BLAS performance optimization on Loongson 3A processor. In *IEEE 18th International Conference on Parallel and Distributed Systems (ICPADS)*, December 2012.

A Complex domain performance: insights from a simple model

Many DLA software developers expect matrix multiplication in the complex domain to slightly exceed a comparable computation in the real domain. This phenomenon can be seen more clearly by constructing a simple model for the aggregate flop rates of real and complex domain matrix multiplication (GEMM). Let us define a function R for the rate of computation, in flops per second, of matrix multiplication in the real domain:

$$R\left(m,k,n,k_C^{\mathbb{R}},n_C^{\mathbb{R}}\right) = \frac{2mnk}{2mnk \cdot t_{flop} + 2\left(mn\left\lceil\frac{k}{k_C^{\mathbb{R}}}\right\rceil + mk\left\lceil\frac{n}{n_C^{\mathbb{R}}}\right\rceil + kn\right) \cdot t_{memop}} \tag{6}$$

where m, k, and n are the dimensions of the matrix operands, $k_C^{\mathbb{R}}$ and $n_C^{\mathbb{R}}$ are the real domain cache blocksizes used, t_{flop} is the time, in seconds, needed to execute a single flop (assuming a full floating-point instruction pipeline with no stalls), and t_{memop} is the time needed to execute a single memop.[41] Now let us define a similar function C for the rate of computation of a conventionally implemented complex domain matrix multiplication:

$$\begin{aligned} C\left(m,k,n,k_C^{\mathbb{C}},n_C^{\mathbb{C}}\right) &= \frac{8mnk}{8mnk \cdot t_{flop} + 4\left(mn\left\lceil\frac{k}{k_C^{\mathbb{C}}}\right\rceil + mk\left\lceil\frac{n}{n_C^{\mathbb{C}}}\right\rceil + kn\right) \cdot t_{memop}} \\ &= \frac{2mnk}{2mnk \cdot t_{flop} + \left(mn\left\lceil\frac{k}{k_C^{\mathbb{C}}}\right\rceil + mk\left\lceil\frac{n}{n_C^{\mathbb{C}}}\right\rceil + kn\right) \cdot t_{memop}} \end{aligned} \tag{7}$$

where $k_C^{\mathbb{C}}$ and $n_C^{\mathbb{C}}$ are the complex domain cache blocksizes. Notice that, unlike in Table 3, we preserve the ceiling function around the blocksize quotient terms. Also, for simplicity, both models ignore various effects that would be present in an actual implementation, including function/loop overhead as well as variation in memop cost due to prefetching.

It is now easy to see that if we hold m, n, and k constant, with $n_C^{\mathbb{C}} = n_C^{\mathbb{R}}$ and $k_C^{\mathbb{C}} = k_C^{\mathbb{R}}$, $R < C$. That is, multiplying an $m \times k$ matrix by a $k \times n$ matrix in the complex domain will execute flops at a higher rate than in the real domain, assuming identical cache blocksizes. This is due to the lower relative impact of matrix dimensions on the memop cost term in the denominator of Eq. 7.

We suspect that a modified Eq. 6, one with its memop cost term halved, yields an upper bound for the performance of complex induced methods (which we abbreiviate as UBIM). Notice that this is equivalent to Eq. 7 when $k_C^{\mathbb{C}} = k_C^{\mathbb{R}}$. In plain English, this conjecture simply states that an induced method's performance cannot exceed the rate of computation observed when complex matrix multiplication benefits from the favorable flop-to-memop ratio inherently found in conventional (assembly-based) implementations.[42]

We can illustrate the output of Eq. 6 and 7 for various problem sizes by selecting (somewhat arbitrarily) values for the model parameters: $t_{flop} = 1.0\times10^{-10}$ and $t_{memop} = 3.0\times10^{-9}$. These values simulate hardware with a peak performance of 10 GFLOPS, where a memop is 30 times more expensive than a corresponding flop. For this discussion, we assume that these values pertain to double-precision computation, though the model is agnostic to the simulated hardware's floating-point precision.

Figure 12 (left) shows simulated performance of rank-k_C update for real GEMM ("dgemm") as well as complex GEMM ("zgemm") under three different values of $k_C^{\mathbb{C}}$. The graph also includes the UBIM, as described previously, that corresponds to what the performance of the real domain GEMM would be if its memop cost were halved. For our simulation, we chose values of k_C that were equal to those used for the Intel Xeon E5-2680 Sandy Bridge system described in Section 6: $k_C^d = 256$ and $k_C^z = 192$ (for double-precision real

[41]For simplicity, we ignore the time spent moving elements of $\tilde{A}_i$ from the L2 to L1 cache as well as time spent moving elements of $\tilde{B}_p$ from the L3 to L1 cache. In our experience, these data movements can be partially (if not mostly) overlapped with useful computation.

[42]This favorable ratio is a direct reflection of the efficient reuse of F.E. from registers, whereby each complex rank-1 update effectively computes two real rank-1 updates using the same input operands.

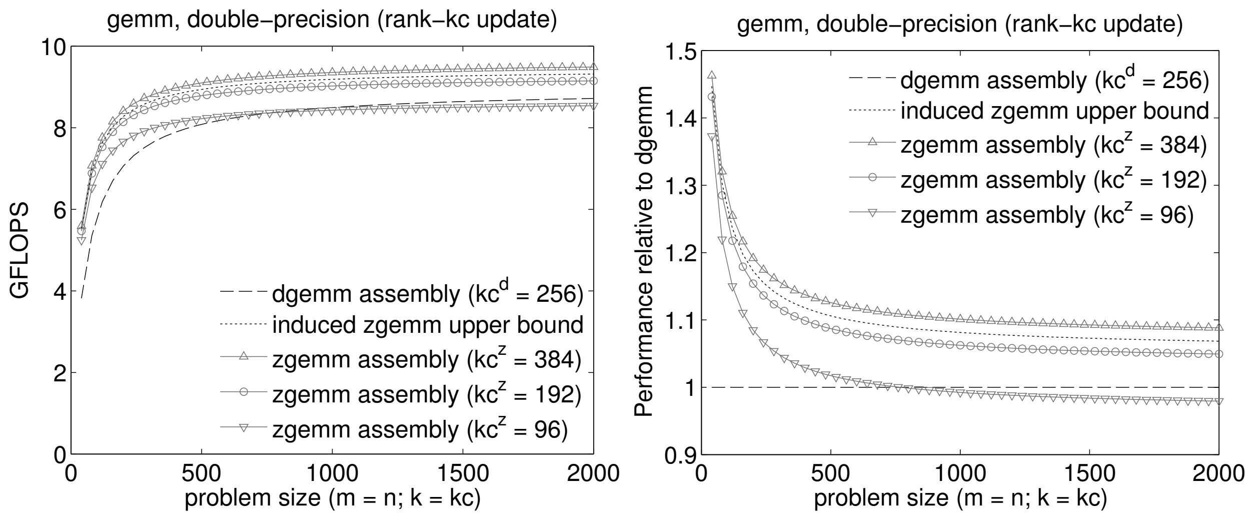


Figure 12: Simulated performance of real GEMM and the proposed theoretical upper bound for complex GEMM via induced methods when k is bound to k_C. The value of k_C^d used here is the same double-precision real value as used on the Intel Xeon E5-2680 "Sandy Bridge" featured in Section 6. Results are also shown for conventional complex GEMM for three different values of k_C^z, including the value $k_C^z = 192$ used for Sandy Bridge shown in Table 5.

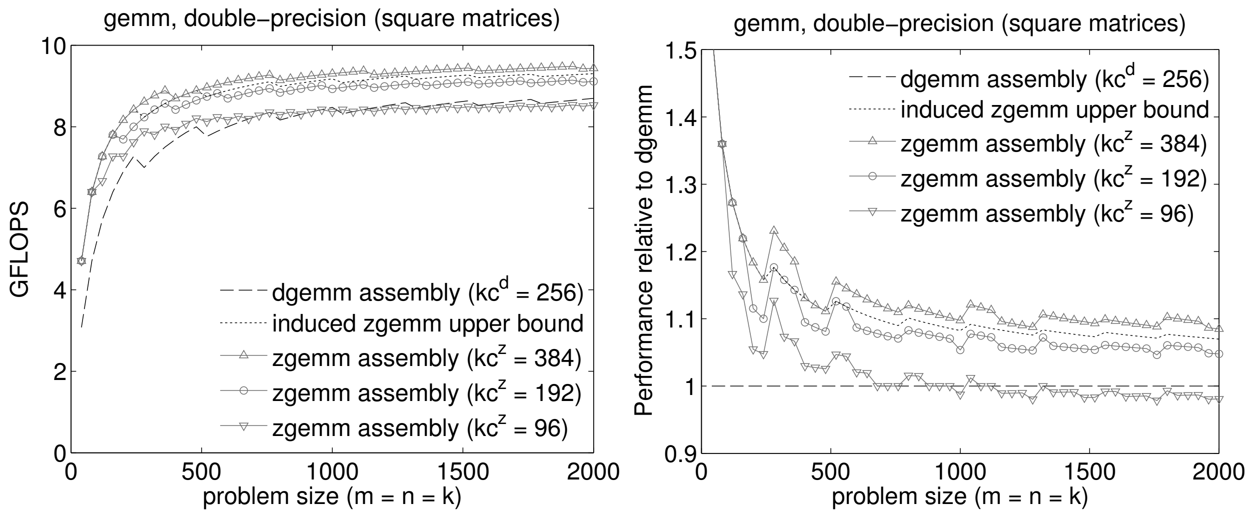


Figure 13: Simulated performance of real GEMM and the proposed theoretical upper bound for complex GEMM via induced methods when matrix operands are square. The value of k_C^d used here is the same double-precision real value as used on the Intel Xeon E5-2680 "Sandy Bridge" featured in Section 6. Results are also shown for conventional complex GEMM for three different values of k_C^z, including the value $k_C^z = 192$ used for Sandy Bridge shown in Table 5.

and complex GEMM, respectively).[43] Results for two other values of k_C^z are also shown. Figure 12 (right) normalizes the results shown in Figure 12 (left) relative to the simulated real GEMM to better show relative performance. Figure 13 shows simulated performance under the same conditions as Figure 12, except that k is allowed to grow with the problem size.

We can now make several observations:

- When performance is measured only as a percentage of theoretical peak, it can be shown that it is only the *ratio* between t_{memop} and t_{flop} that determines performance, rather than their absolute magnitudes. Any values that result in the same ratio t_{memop}/t_{flop} would yield the same simulated performance signatures, provided the graphs were scaled so that the maximum y-axis value shown was $1/t_{flop}$.

- Qualitatively, we can see that the outperformance of complex GEMM (relative to real GEMM) depends heavily on the cache blocksize $k_C^{\mathbb{C}}$. As $k_C^{\mathbb{C}}$ is lowered, the performance of conventional complex GEMM begins to fall, because a falling k_C, where $k_C = k \ll m = n$ causes the number of flops to fall much faster than the number of memops. In fact, Figures 12 and 13 show that for small enough values of $k_C^{\mathbb{C}}$, the outperformance of complex GEMM actually disappears altogether.

- In Figures 12 and 13, the curves labeled "induced zgemm upper bound" show the UBIM. While not plotted explicitly in Figure 12, the simulated performance of conventional complex GEMM coincides exactly with the UBIM when both the conventional complex GEMM and real domain GEMM employ the same cache blocksizes.[44] (Here, it is actually more important that the values of k_C specifically be equal, since n_C is usually quite large and tends to have a relatively small impact on overall performance.)

- The outperformance of conventional complex GEMM depicted in Figures 12 and 13 can be quantified in terms of Eq. 6 and 7 as:

$$\frac{C\left(m,k,n,k_C^{\mathbb{C}},n_C^{\mathbb{C}}\right)}{R\left(m,k,n,k_C^{\mathbb{R}},n_C^{\mathbb{R}}\right)} = \frac{1 + \left(k^{-1}\left\lceil \frac{k}{k_C^{\mathbb{R}}} \right\rceil + n^{-1}\left\lceil \frac{n}{n_C^{\mathbb{R}}} \right\rceil + m^{-1}\right) \cdot \frac{t_{memop}}{t_{flop}}}{1 + \frac{1}{2}\left(k^{-1}\left\lceil \frac{k}{k_C^{\mathbb{R}}} \right\rceil + n^{-1}\left\lceil \frac{n}{n_C^{\mathbb{R}}} \right\rceil + m^{-1}\right) \cdot \frac{t_{memop}}{t_{flop}}} \qquad (8)$$

  For rank-k_C update scenarios (i.e., $k = k_C$) with large enough m and n dimensions (and large $n_C^{\mathbb{R}}$), the m^{-1} and n^{-1} terms become relatively negligible and Eq. 8 approaches

$$\frac{C\left(m,k_C^{\mathbb{C}},n,k_C^{\mathbb{C}},n_C^{\mathbb{C}}\right)}{R\left(m,k_C^{\mathbb{R}},n,k_C^{\mathbb{R}},n_C^{\mathbb{R}}\right)} \approx \frac{1 + \left(\frac{1}{k_C^{\mathbb{R}}}\right) \cdot \frac{t_{memop}}{t_{flop}}}{1 + \frac{1}{2}\left(\frac{1}{k_C^{\mathbb{C}}}\right) \cdot \frac{t_{memop}}{t_{flop}}} \qquad (9)$$

  If we take a step further and evaluate C using $k_C^{\mathbb{R}}$ (which represents an induced method using an unreduced $k_C^{\mathbb{R}}$) and $n_C^{\mathbb{R}}$, Eq. 9 yields an estimate of the UBIM relative to its corresponding real domain GEMM:

$$\frac{C\left(m,k_C^{\mathbb{R}},n,k_C^{\mathbb{R}},n_C^{\mathbb{R}}\right)}{R\left(m,k_C^{\mathbb{R}},n,k_C^{\mathbb{R}},n_C^{\mathbb{R}}\right)} \approx \frac{1 + \left(\frac{1}{k_C^{\mathbb{R}}}\right) \cdot \frac{t_{memop}}{t_{flop}}}{1 + \frac{1}{2}\left(\frac{1}{k_C^{\mathbb{R}}}\right) \cdot \frac{t_{memop}}{t_{flop}}} \qquad (10)$$

  Remarkably, if we ignore machine parameters t_{memop} and t_{flop}, which are constant for any particular hardware, this approximation of the UBIM depends entirely on $k_C^{\mathbb{R}}$.

[43] These specific values carry no significance in our model, and were chosen because they will be familiar to readers of the article.

[44] In practice, optimized assembly-based implementations tend to use cache blocksizes such that $k_C^{\mathbb{C}} \leq k_C^{\mathbb{R}}$.

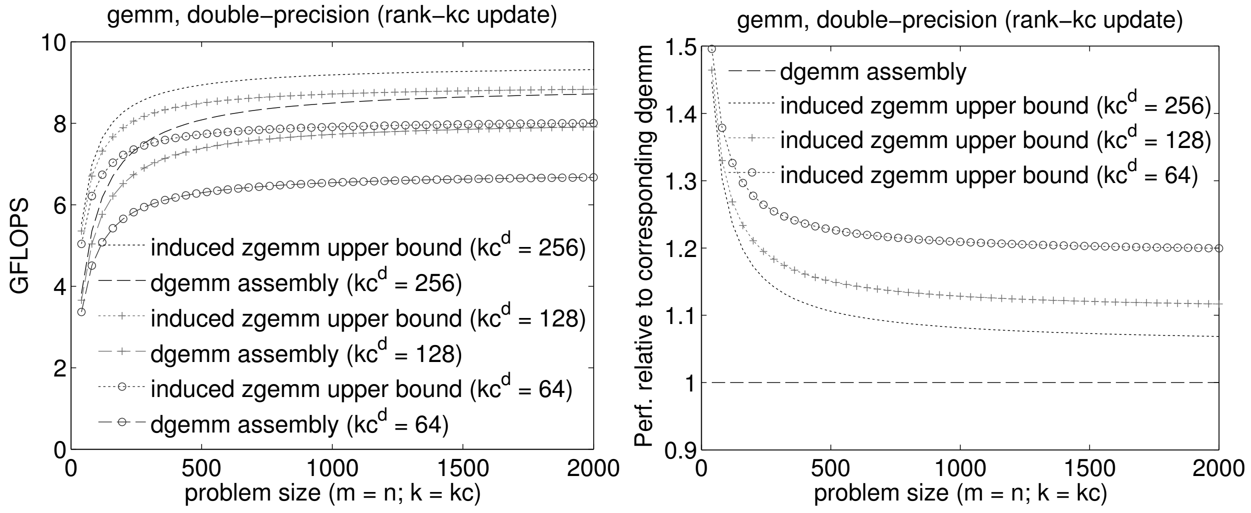


Figure 14: Simulated performance of real GEMM and the proposed theoretical upper bound for complex GEMM via induced methods when k is bound to k_C for three different values of k_C.

- Figure 14 (left) reproduces the same UBIM and its associated real GEMM benchmark from Figure 12 (left) where $k_C^{\mathbb{R}} = 256$, but also superimposes curves for $k_C^{\mathbb{R}} = 128$ and $k_C^{\mathbb{R}} = 64$. Normalized values appear in Figure 14 (right). Notice that these curves plot exact values corresponding to the approximation given by Eq. 10. By inspecting Figure 14, we can see qualitatively how the gap between the UBIM and its corresponding real domain GEMM depends on the cache blocksize $k_C^{\mathbb{R}}$, and how both relate to the simulated peak performance.
- The model would perform poorly when simulating matrix multiplication via the 4M method. Those algorithms' performance signatures are sensitive to variation in the circumstance (i.e., caching) of memops, whereas a straightforward extension of Eq. 7, for the purposes of 4M, would account only for variation in the number of memops. At first glance, it would appear that properly simulating impact of the cache hierarchy would require a significant degree of added sophistication to the model.

While characterizing the UBIM is satisfying from a theoretical perspective, it seems doubtful that there exist any induced methods capable of approaching the bound. This intuition is motivated by the fact that the UBIM represents an *absolute* maximum: it assumes the induced method uses unreduced real domain blocksizes $n_C^{\mathbb{R}}$ and (most importantly) $k_C^{\mathbb{R}}$, *and* incurs no redundant memops or cache line movements on A, B, or C—a seemingly impossible feat given the implementation difficulties encountered thus far with the 4M method. The UBIM may still prove to be a useful threshold against which to measure and compare induced method performance, at least theoretically, provided that either (a) the induced methods in question can be modeled with sufficient precision, or (b) the UBIM can be estimated from measured flop and memop cost components of actual implementation run times. Nevertheless, the emperical results reported in Section 6 lead us to think that future efforts may face a significant challenge in merely matching (let alone exceeding) the performance of the real GEMM benchmark itself.