Strassen's Algorithm Reloaded

Jianyu Huang*, Tyler M. Smith*†, Greg M. Henry‡, Robert A. van de Geijn*†
*Department of Computer Science and †Institute for Computational Engineering and Sciences,
The University of Texas at Austin, Austin, TX 78712
Email: jianyu,tms,rvdg@cs.utexas.edu
‡Intel Corporation, Hillsboro, OR 97124
Email: greg.henry@intel.com

***Abstract*—We dispel with "street wisdom" regarding the practical implementation of Strassen's algorithm for matrix-matrix multiplication (DGEMM). Conventional wisdom: it is only practical for very large matrices. Our implementation is practical for small matrices. Conventional wisdom: the matrices being multiplied should be relatively square. Our implementation is practical for rank-k updates, where k is relatively small (a shape of importance for libraries like LAPACK). Conventional wisdom: it inherently requires substantial workspace. Our implementation requires no workspace beyond buffers already incorporated into conventional high-performance DGEMM implementations. Conventional wisdom: a Strassen DGEMM interface must pass in workspace. Our implementation requires no such workspace and can be plug-compatible with the standard DGEMM interface. Conventional wisdom: it is hard to demonstrate speedup on multi-core architectures. Our implementation demonstrates speedup over conventional DGEMM even on an Intel® Xeon Phi™ coprocessor[1] utilizing 240 threads. We show how a distributed memory matrix-matrix multiplication also benefits from these advances.**

***Index Terms*—Strassen, numerical algorithm, performance model, matrix multiplication, linear algebra library, BLAS.**

I. Introduction

Strassen's algorithm (STRASSEN) [1] for matrix-matrix multiplication (DGEMM) has fascinated theoreticians and practitioners alike since it was first published, in 1969. That paper demonstrated that multiplication of $n \times n$ matrices can be achieved in less than the $O(n^3)$ arithmetic operations required by a conventional formulation. It has led to many variants that improve upon this result [2], [3], [4], [5] as well as practical implementations [6], [7], [8], [9]. The method can yield a shorter execution time than the best conventional algorithm with a modest degradation in numerical stability [10], [11], [12] by only incorporating a few levels of recursion.

From 30,000 feet the algorithm can be described as shifting computation with submatrices from multiplications to additions, reducing the $O(n^3)$ term at the expense of adding $O(n^2)$ complexity. For current architectures, of greater consequence is the additional memory movements that are incurred when the algorithm is implemented in terms of a conventional DGEMM provided by a high-performance implementation through the Basic Linear Algebra Subprograms (BLAS) [13] interface. A secondary concern has been the extra workspace that is required. This simultaneously limits the size of problem that can be computed and makes it so an implementation is not plug-compatible with the standard calling sequence supported by the BLAS.

An important recent advance in the high-performance implementation of DGEMM is the BLAS-like Library Instantiation Software (BLIS framework) [14], a careful refactoring of the best-known approach to implementing conventional DGEMM introduced by Goto [15]. Of importance to the present paper are the building blocks that BLIS exposes, minor modifications of which support a new approach to implementating STRASSEN. This approach changes data movement between memory layers and can thus mitigate the negative impact of the additional lower order terms incurred by STRASSEN. These building blocks have similarly been exploited to improve upon the performance of, for example, the computation of the K-Nearest Neighbor [16] and Tensor Contraction [17], [18] problem. The result is a family of STRASSEN implementations, members of which attain superior performance depending on the sizes of the matrices.

The resulting family improves upon prior implementations of STRASSEN in a number of surprising ways:

- It can outperform classical DGEMM even for small square matrices.
- It can achieve high performance for rank-k updates (DGEMM with a small "inner matrix size"), a case of DGEMM frequently encountered in the implementation of libraries like LAPACK [19].
- It needs not require additional workspace.
- It can incorporate directly the multi-threading in traditional DGEMM implementations.
- It can be plug-compatible with the standard DGEMM interface supported by the BLAS.
- It can be incorporated into practical distributed memory implementations of DGEMM.

Most of these advances run counter to conventional wisdom and are backed up by theoretical analysis and practical implementation.

II. Standard Matrix-matrix Multiplication

We start by discussing naive computation of matrix-matrix multiplication (DGEMM), how it is supported as a library routine by the Basic Linear Algebra Subprograms (BLAS) [13], how modern implementations block for caches, and how that implementation supports multi-threaded parallelization.

[1]Intel, Xeon, and Intel Xeon Phi are trademarks of Intel Corporation in the U.S. and/or other countries.

A. Computing $C = \alpha AB + C$

Consider $C = \alpha AB + C$, where C, A, and B are $m \times n$, $m \times k$, and $k \times n$ matrices, respectively, and α is a scalar. If the (i, j) entry of C, A, and B are respectively denoted by $c_{i,j}$, $a_{i,j}$, and $b_{i,j}$, then computing $C = \alpha AB + C$ is achieved by

$$c_{i,j} = \alpha \sum_{p=0}^{k-1} a_{i,p} b_{p,j} + c_{i,j},$$

which requires $2mnk$ floating point operations (flops).

B. Level-3 BLAS matrix-matrix multiplication

(General) matrix-matrix multiplication (GEMM) is supported in the level-3 BLAS [13] interface as

```
DGEMM( transa, transb, m, n, k, alpha,
                    A, lda, B, ldb, beta, C, ldc )
```

where we focus on double precision arithmetic and data. This call supports

$$\begin{array}{ll} C = \alpha AB + \beta C, & C = \alpha A^T B + \beta C, \\ C = \alpha AB^T + \beta C, & \text{and } C = \alpha A^T B^T + \beta C \end{array}$$

depending on the choice of `transa` and `transb`. In our discussion we can assume $\beta = 1$ since C can always first be multiplied by that scalar as a preprocessing step, which requires only $O(n^2)$ flops. Also, by internally allowing both a row stride and a column stride for `A`, `B`, and `C` (as the BLIS framework does), transposition can be easily supported by swapping these strides. It suffices then to consider $C = \alpha AB + C$.

C. Computing with submatrices

Important to our discussion is that we partition the matrices and stage the matrix-multiplication as computations with submatrices. For example, let us assume that m, n, and k are all even and partition

$$C = \begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix}, A = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix}, B = \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix},$$

where C_{ij} is $\frac{m}{2} \times \frac{n}{2}$, A_{ij} is $\frac{m}{2} \times \frac{k}{2}$, and B_{ij} is $\frac{k}{2} \times \frac{n}{2}$. Then

$$\begin{array}{rcl} C_{00} &=& \alpha(A_{00}B_{00} + A_{01}B_{10}) + C_{00} \\ C_{01} &=& \alpha(A_{00}B_{01} + A_{01}B_{11}) + C_{01} \\ C_{10} &=& \alpha(A_{10}B_{00} + A_{11}B_{10}) + C_{10} \\ C_{11} &=& \alpha(A_{10}B_{01} + A_{11}B_{11}) + C_{11} \end{array}$$

computes $C = \alpha AB + C$ via eight multiplications and eight additions with submatrices, still requiring approximately $2mnk$ flops.

D. The GotoBLAS algorithm for DGEMM

Figure 1(left) illustrates the way the GOTOBLAS [21] (predecessor of OpenBLAS [22]) approach structures the blocking for three layers of cache (L1, L2, and L3) when computing $C = AB+C$, as implemented in BLIS. For details we suggest the reader consult the papers on the GOTOBLAS DGEMM [15] and BLIS [14]. In that figure, the indicated block sizes m_C, n_C, and k_C are chosen so that submatrices fit in the various caches while m_R and n_R relate to the size of contributions to C that fits in registers. For details on how these are chosen, see [14], [20].

Importantly,

- The row panels B_p that fit in the L3 cache[2] are packed into contiguous memory, yielding $\widetilde{B}_p$.
- Blocks A_i that fit in the L2 cache are packed into buffer $\widetilde{A}_i$.

It is in part this *packing* that we are going to exploit as we implement one or more levels of STRASSEN.

E. Multi-threaded implementation

BLIS exposes all the illustrated loops, requiring only the micro-kernel to be optimized for a given architecture. In contrast, in the GOTOBLAS implementation the micro-kernel and the first two loops around it form an inner-kernel that is implemented as a unit. As a result, the BLIS implementation exposes five loops (two more than the GOTOBLAS implementation) that can be parallelized, as discussed in [23]. In this work, we mimic the insights from that paper.

III. STRASSEN'S ALGORITHM

In this section, we present the basic idea and practical considerations of STRASSEN, decomposing it into a combination of general operations that can be adapted to the high-performance implementation of a traditional DGEMM.

A. The basic idea

It can be verified that the operations in Figure 2 also compute $C = \alpha AB + C$, requiring only seven multiplications with submatrices. The computational cost is, approximately, reduced from $2mnk$ flops to $(7/8)2mnk$ flops, at the expense of a lower order number of extra additions. Figure 2 describes what we will call *one-level* STRASSEN.

B. Classic Strassen's algorithm

Each of the matrix multiplications that computes an intermediate result M_k can itself be computed with another level of Strassen's algorithm. This can then be repeated recursively.

If originally $m = n = k = 2^d$, where d is an integer, then the cost becomes

$$(7/8)^{\log_2(n)} 2n^3 = n^{\log_2(7/8)} 2n^3 = 2n^{2.807} \text{ flops.}$$

In this discussion, we ignored the increase in the total number of extra additions.

C. Practical considerations

A high-performance implementation of a traditional matrix-matrix multiplication requires careful attention to details related to data movements between memory layers, scheduling of operations, and implementations at a very low level (often in assembly code). Practical implementations recursively perform a few levels of STRASSEN until the matrices become small enough so that a traditional high-performance DGEMM

[2] If an architecture does not have an L3 cache, this panel is still packed to make the data contiguous and to reduce the number of TLB entries used.

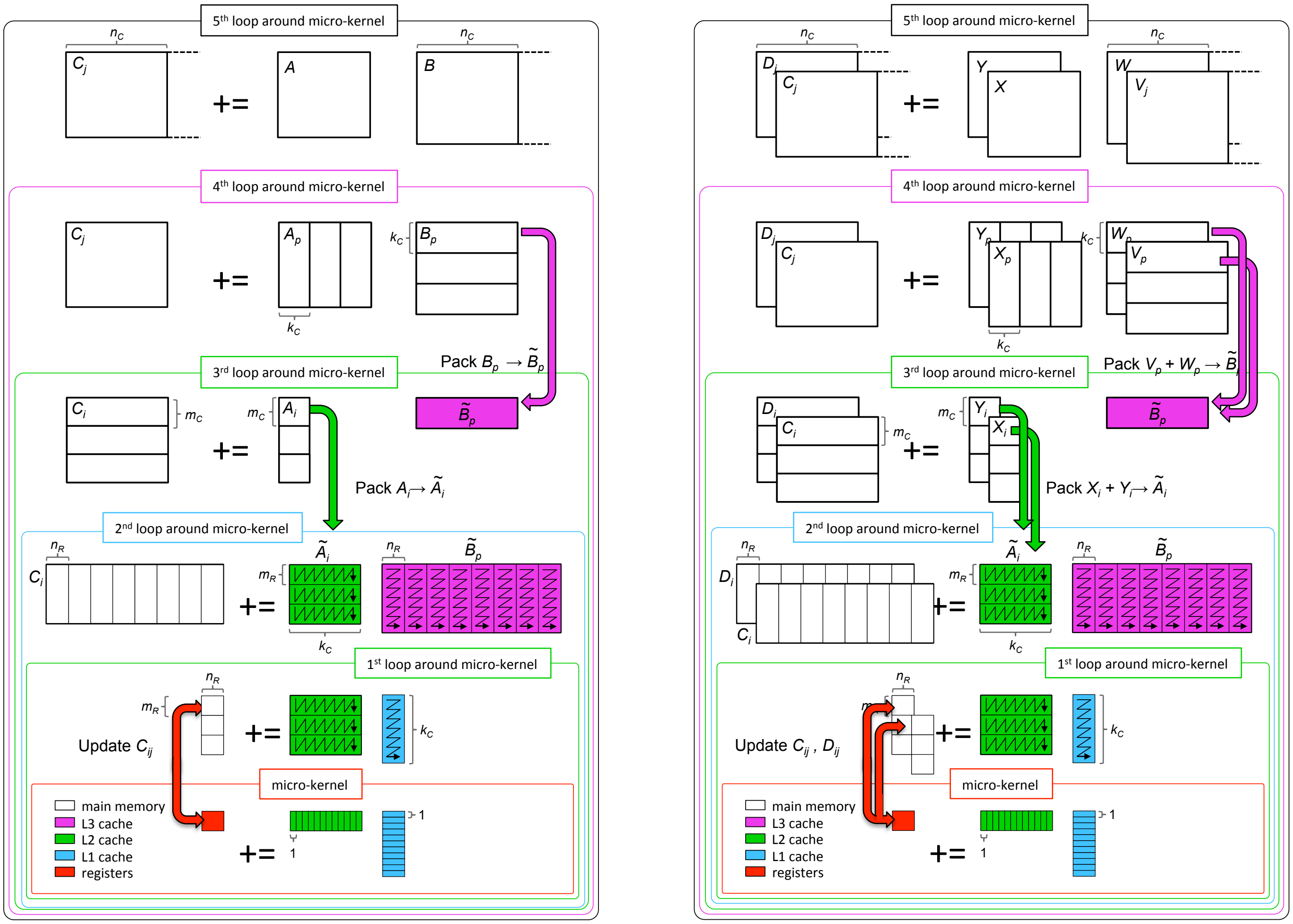

Fig. 1. Left: Illustration (adapted from [20] with permission of the authors) of the BLIS implementation of the GotoBLAS DGEMM algorithm. All computation is cast in terms of a micro-kernel that is highly optimized. Right: modification that implements the representative computation $M = (X+Y)(V+W); C{+}{=} M; D{+}{=} M$ of general operation (1).

$$
\begin{array}{ll}
M_0 = \alpha(A_{00} + A_{11})(B_{00} + B_{11}); & C_{00} {+}{=} M_0; C_{11} {+}{=} M_0; \\
M_1 = \alpha(A_{10} + A_{11})B_{00}; & C_{10} {+}{=} M_1; C_{11} {-}{=} M_1; \\
M_2 = \alpha A_{00}(B_{01} - B_{11}); & C_{01} {+}{=} M_2; C_{11} {+}{=} M_2; \\
M_3 = \alpha A_{11}(B_{10} - B_{00}); & C_{00} {+}{=} M_3; C_{10} {+}{=} M_3; \\
M_4 = \alpha(A_{00} + A_{01})B_{11}; & C_{01} {+}{=} M_4; C_{00} {-}{=} M_4; \\
M_5 = \alpha(A_{10} - A_{00})(B_{00} + B_{01}); & C_{11} {+}{=} M_5; \\
M_6 = \alpha(A_{01} - A_{11})(B_{10} + B_{11}); & C_{00} {+}{=} M_6;
\end{array}
$$

Fig. 2. All operations for one-level STRASSEN. Note that each row is a special case of general operation (1).

is faster. At that point, the recursion stops and a high-performance DGEMM is used for the subproblems. In prior implementations, the switch point is usually as large as 2000 for double precision square matrices on a single core of an x86 CPU [8], [9]. We will see that, for the same architecture, one of our implementations has a switch point as small as 500 (Figure 5).

In an ordinary matrix-matrix multiplication, three matrices must be stored, for a total of $3n^2$ floating point numbers (assuming all matrices are $n \times n$). The most naive implementation of one-level STRASSEN requires an additional seven submatrices of size $\frac{n}{2} \times \frac{n}{2}$ (for M_0 through M_6) and ten matrices of size $\frac{n}{2} \times \frac{n}{2}$ for $A_{00} + A_{11}$, $B_{00} + B_{11}$, etc. A careful ordering of the computation can reduce this to two matrices [24]. We show that the computation can be organized so that no temporary storage beyond that required for a high-performance traditional DGEMM is needed. In addition, it is easy to parallelize for multi-core and many-core architectures with our approach, since we can adopt the same parallel scheme advocated by BLIS.

The general case where one or more dimensions are not a convenient multiple of a power of two leads to the need to either pad matrices or to treat a remaining "fringe" carefully [7]. Traditionally, it is necessary to pad m, n, and k to be even. In our approach this can be handled internally by padding $\widetilde{A}_i$ and $\widetilde{B}_p$, and by using tiny ($m_R \times n_R$) buffers for C along the fringes (much like the BLIS framework does).

$M_0 = \alpha(A_{0,0}+A_{2,2}+A_{1,1}+A_{3,3})(B_{0,0}+B_{2,2}+B_{1,1}+B_{3,3});$
$C_{0,0} += M_0;\ C_{1,1} += M_0;\ C_{2,2} += M_0; C_{3,3} += M_0;$
$M_1 = \alpha(A_{1,0}+A_{3,2}+A_{1,1}+A_{3,3})(B_{0,0}+B_{2,2});$
$C_{1,0} += M_1;\ C_{1,1} -= M_1;\ C_{3,2} += M_1; C_{3,3} -= M_1;$
$M_2 = \alpha(A_{0,0}+A_{2,2})(B_{0,1}+B_{2,3}+B_{1,1}+B_{3,3});$
$C_{0,1} += M_2;\ C_{1,1} += M_2;\ C_{2,3} += M_2; C_{3,3} += M_2;$
$M_3 = \alpha(A_{1,1}+A_{3,3})(B_{1,0}+B_{3,2}+B_{0,0}+B_{2,2});$
$C_{0,0} += M_3;\ C_{1,0} += M_3;\ C_{2,2} += M_3; C_{3,2} += M_3;$
$M_4 = \alpha(A_{0,0}+A_{2,2}+A_{0,1}+A_{2,3})(B_{1,1}+B_{3,3});$
$C_{0,0} -= M_4;\ C_{0,1} += M_4;\ C_{2,2} -= M_4; C_{2,3} += M_4;$
$M_5 = \alpha(A_{1,0}+A_{3,2}+A_{0,0}+A_{2,2})(B_{0,0}+B_{2,2}+B_{0,1}+B_{2,3});$
$C_{1,1} += M_5;\ C_{3,3} += M_5;$
$M_6 = \alpha(A_{0,1}+A_{2,3}+A_{1,1}+A_{3,3})(B_{1,0}+B_{3,2}+B_{1,1}+B_{3,3});$
$C_{0,0} += M_6;\ C_{2,2} += M_6;$
$M_7 = \alpha(A_{2,0}+A_{2,2}+A_{3,1}+A_{3,3})(B_{0,0}+B_{1,1});$
$C_{2,0} += M_7;\ C_{3,1} += M_7;\ C_{2,2} -= M_7; C_{3,3} -= M_7;$
$\vdots$
$M_{42} = \alpha(A_{0,2}+A_{2,2}+A_{1,3}+A_{3,3})(B_{2,0}+B_{2,2}+B_{3,1}+B_{3,3});$
$C_{0,0} += M_{42}; C_{1,1} += M_{42};$
$M_{43} = \alpha(A_{1,2}+A_{3,2}+A_{1,3}+A_{3,3})(B_{2,0}+B_{2,2});$
$C_{1,0} += M_{43}; C_{1,1} -= M_{43};$
$M_{44} = \alpha(A_{0,2}+A_{2,2})(B_{2,1}+B_{2,3}+B_{3,1}+B_{3,3});$
$C_{0,1} += M_{44}; C_{1,1} += M_{44};$
$M_{45} = \alpha(A_{1,3}+A_{3,3})(B_{3,0}+B_{3,2}+B_{2,0}+B_{2,2});$
$C_{0,0} += M_{45}; C_{1,0} += M_{45};$
$M_{46} = \alpha(A_{0,2}+A_{2,2}+A_{0,3}+A_{2,3})(B_{3,1}+B_{3,3});$
$C_{0,0} -= M_{46}; C_{0,1} += M_{46};$
$M_{47} = \alpha(A_{1,2}+A_{3,2}+A_{0,2}+A_{2,2})(B_{2,0}+B_{2,2}+B_{2,1}+B_{2,3});$
$C_{1,1} += M_{47};$
$M_{48} = \alpha(A_{0,3}+A_{2,3}+A_{1,3}+A_{3,3})(B_{3,0}+B_{3,2}+B_{3,1}+B_{3,3});$
$C_{0,0} += M_{48};$

Fig. 3. Representative computations for two levels of Strassen.

D. *One-level* STRASSEN *reloaded*

The operations summarized in Figure 2 are all special cases of

$$M = \alpha(X+\delta Y)(V+\epsilon W); \quad C += \gamma_0 M; \quad D += \gamma_1 M; \quad (1)$$

for appropriately chosen $\gamma_0, \gamma_1, \delta, \epsilon \in \{-1, 0, 1\}$. Here, X and Y are submatrices of A, V and W are submatrices of B, and C and D are submatrices of original C.

Let us focus on how to modify the algorithm illustrated in Figure 1(left) in order to accommodate the representative computation

$$M = (X+Y)(V+W); C += M; D += M.$$

As illustrated in Figure 1(right), the key insight is that the additions of matrices $V+W$ can be incorporated in the packing into buffer $\widetilde{B}_p$ and the additions of matrices $X+Y$ in the packing into buffer $\widetilde{A}_i$. Also, when a small block of $(X+Y)(V+W)$ is accumulated in registers it can be added to the appropriate parts of both C and D, multiplied by $\alpha\gamma_0$ and $\alpha\gamma_1$, as needed, inside a modified micro-kernel. This avoids multiple passes over the various matrices, which would otherwise add a considerable overhead from memory movements.

E. *Two-level* STRASSEN *reloaded*

Let

$$C = \left(\begin{array}{cc|cc} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ \hline C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{array}\right), A = \left(\begin{array}{cc|cc} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ \hline A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{array}\right),$$
$$\text{and } B = \left(\begin{array}{cc|cc} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ \hline B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{array}\right),$$

where $C_{i,j}$ is $\frac{m}{4} \times \frac{n}{4}$, $A_{i,p}$ is $\frac{m}{4} \times \frac{k}{4}$, and $B_{p,j}$ is $\frac{k}{4} \times \frac{n}{4}$. Then it can be verified that the computations in Figure 3 compute $C = \alpha AB + C$. The operations found there can be cast as special cases of

$$\begin{aligned} M = \alpha(X_0 + \delta_1 X_1 + \delta_2 X_2 + \delta_3 X_3) \times \\ (V_0 + \epsilon_1 V_1 + \epsilon_2 V_2 + \epsilon_3 V_3); \\ C_0 += \gamma_0 M; C_1 += \gamma_1 M; C_2 += \gamma_2 M; C_3 += \gamma_3 M \end{aligned}$$

by appropriately picking $\gamma_i, \delta_i, \epsilon_i \in \{-1, 0, 1\}$. Importantly, the computation now requires 49 multiplications for submatrices as opposed to 64 for a conventional DGEMM.

To extend the insights from Section III-D so as to integrate two-level STRASSEN into the BLIS DGEMM implementation, we incorporate the addition of up to four submatrices of A and B, the updates of up to four submatrices of C inside the micro-kernel, and the tracking of up to four submatrices in the loops in BLIS.

F. *Additional levels*

A pattern now emerges. The operation needed to integrate k levels of STRASSEN is given by

$$\begin{aligned} & M = \alpha \left(\sum_{s=0}^{l_X-1} \delta_s X_s\right) \left(\sum_{t=0}^{l_V-1} \epsilon_t V_t\right); \\ & C_r += \gamma_r M \quad \text{for } r = 0, \ldots, l_C - 1. \end{aligned} \quad (2)$$

For each number, l, of levels of STRASSEN that are integrated, a table can then be created that captures all the computations to be executed.

IV. IMPLEMENTATION AND ANALYSIS

We now discuss the details of how we adapt the high-performance GOTOBLAS approach to these specialized operations to yield building blocks for a family of STRASSEN implementations. Next, we also give a performance model for comparing members of this family.

A. *Implementations*

We implement a family of algorithms for up to two levels[3] of STRASSEN, building upon the BLIS framework.

[3]We can support three or more levels of STRASSEN, by modifying the packing routines and the micro-kernel to incorporate more summands. However, the crossover point for the three-level Strassen to outperform all one/two-level STRASSEN implementations is very large (∼ 10000 for square matrices). There are also concerns regarding to numerical stability issues with many levels of recursions. So we don't go beyond two levels in this paper.

Building blocks: The BLIS framework provides three primitives for composing DGEMM: a routine for packing B_p into $\widetilde{B}_p$, a routine for packing A_i into $\widetilde{A}_i$, and a micro-kernel for updating an $m_R \times n_R$ submatrix of C. The first two are typically written in `C` while the last one is typically written in (inlined) assembly code.

To implement a typical operation given in (2),

- the routine for packing B_p is modified to integrate the addition of multiple matrices V_t into packed buffer $\widetilde{B}_p$;
- the routine for packing A_i is modified to integrate the addition of multiple matrices X_s into packed buffer $\widetilde{A}_i$; and
- the micro-kernel is modified to integrate the addition of the result to multiple submatrices.

Variations on a theme: The members of our family of STRASSEN implementations differ by how many levels of STRASSEN they incorporate and which of the above described modified primitives they use:

- **Naive Strassen**: A traditional implementation with temporary buffers.
- **AB Strassen**: Integrates the addition of matrices into the packing of buffers $\widetilde{A}_i$ and $\widetilde{B}_p$ but creates explicit temporary buffers for matrices M.
- **ABC Strassen**: Integrates the addition of matrices into the packing of buffers $\widetilde{A}_i$ and $\widetilde{B}_p$ *and* the addition of the result of the micro-kernel computation to multiple submatrices of C. For small problem size k this version has the advantage over **AB Strassen** that the temporary matrix M is not moved in and out of memory multiple times. The disadvantage is that for large k the submatrices of C to which contributions are added are moved in and out of memory multiple times instead.

B. Performance Model

In order to compare the performance of the traditional BLAS DGEMM routine and the various implementations of STRASSEN, we define the *effective* GFLOPS metric for $m \times k \times n$ matrix multiplication, similar to [9], [25], [26]:

$$\textit{effective}\ \text{GFLOPS} = \frac{2 \cdot m \cdot n \cdot k}{\text{time (in seconds)}} \cdot 10^{-9}. \quad (3)$$

We next derive a model to predict the execution time T and the *effective* GFLOPS of the traditional BLAS DGEMM and the various implementations of STRASSEN. Theoretical predictions allow us to compare and contrast different implementation decisions, help with performance debugging, and (if sufficiently accurate) can be used to choose the right member of the family of implementations as a function of the number of threads used and/or problem size.

Assumption: Our performance model assumes that the underlying architecture has a modern memory hierarchy with fast caches and relatively slow main memory (DRAM). We assume the latency for accessing the fast caches can be ignored (either because it can be overlapped with computation or because it can be amortized over sufficient computation) while the latency of loading from main memory is exposed. For memory store operations, our model assumes that a lazy write-back policy guarantees the time for storing into fast caches can be hidden. The slow memory operations for BLAS DGEMM and the various implementation of STRASSEN consist of three parts: (1) memory packing in (adapted) DGEMM routine; (2) reading/writing the submatrices of C in (adapted) DGEMM routine; and (3) reading/writing of the temporary buffer that are part of **Naive Strassen** and **AB Strassen**, outside (adapted) DGEMM routine. Based on these assumptions, the execution time is dominated by the arithmetic operations and the slow memory operations.

	type	τ	DGEMM	one-level	two-level
$T_a^{\times}$	-	τ_a	$2mnk$	$7 \times 2\frac{m}{2}\frac{n}{2}\frac{k}{2}$	$49 \times 2\frac{m}{4}\frac{n}{4}\frac{k}{4}$
T_a^{A+}	-	τ_a	-	$5 \times 2\frac{m}{2}\frac{k}{2}$	$95 \times 2\frac{m}{4}\frac{k}{4}$
T_a^{B+}	-	τ_a	-	$5 \times 2\frac{k}{2}\frac{n}{2}$	$95 \times 2\frac{k}{4}\frac{n}{4}$
T_a^{C+}	-	τ_a	-	$12 \times 2\frac{m}{2}\frac{n}{2}$	$154 \times 2\frac{m}{4}\frac{n}{4}$
$T_m^{A\times}$	`r`	τ_b	$mk\lceil\frac{n}{n_c}\rceil$	$\frac{m}{2}\frac{k}{2}\lceil\frac{n/2}{n_c}\rceil$	$\frac{m}{4}\frac{k}{4}\lceil\frac{n/4}{n_c}\rceil$
$T_m^{\widetilde{A}\times}$	`w`	τ_b	$mk\lceil\frac{n}{n_c}\rceil$	$\frac{m}{2}\frac{k}{2}\lceil\frac{n/2}{n_c}\rceil$	$\frac{m}{4}\frac{k}{4}\lceil\frac{n/4}{n_c}\rceil$
$T_m^{B\times}$	`r`	τ_b	nk	$\frac{n}{2}\frac{k}{2}$	$\frac{n}{4}\frac{k}{4}$
$T_m^{\widetilde{B}\times}$	`w`	τ_b	nk	$\frac{n}{2}\frac{k}{2}$	$\frac{n}{4}\frac{k}{4}$
$T_m^{C\times}$ (*)	`r/w`	τ_b	$2mn\lceil\frac{k}{k_c}\rceil$	$2\frac{m}{2}\frac{n}{2}\lceil\frac{k/2}{k_c}\rceil$	$2\frac{m}{4}\frac{n}{4}\lceil\frac{k/4}{k_c}\rceil$
T_m^{A+}	`r/w`	τ_b	mk	$\frac{m}{2}\frac{k}{2}$	$\frac{m}{4}\frac{k}{4}$
T_m^{B+}	`r/w`	τ_b	nk	$\frac{n}{2}\frac{k}{2}$	$\frac{n}{4}\frac{k}{4}$
T_m^{C+}	`r/w`	τ_b	mn	$\frac{m}{2}\frac{n}{2}$	$\frac{m}{4}\frac{n}{4}$

		$N_m^{A\times}$	$N_m^{B\times}$	$N_m^{C\times}$	N_m^{A+}	N_m^{B+}	N_m^{C+}
DGEMM		1	1	1	-	-	-
one-level	ABC	12	12	12	-	-	-
	AB	12	12	7	-	-	36
	Naive	7	7	7	19	19	36
two-level	ABC	194	194	154	-	-	-
	AB	194	194	49	-	-	462
	Naive	49	49	49	293	293	462

Fig. 4. The top table shows theoretical run time breakdown analysis of BLAS DGEMM and various implementations of STRASSEN. The time shown in the first column for DGEMM, one-level STRASSEN, two-level STRASSEN can be computed separately by multiplying the parameter in τ column with the number in the corresponding entries. Due to the software prefetching effects, the row marked with $(*)$ needs to be multiplied by an additional parameter $\lambda \in [0.5, 1]$, which denotes the prefetching efficiency. λ is adjusted to match BLIS DGEMM performance. The bottom table shows the coefficient N_m^X mapping table for computing T_m in the performance model.

Notation: Parameter τ_a denotes the time (in seconds) of one arithmetic (floating point) operation, i.e., the reciprocal of the theoretical peak GFLOPS of the system. Parameter τ_b (bandwidth, memory operation) denotes the amortized time (in seconds) of one unit (one double precision floating point number, or eight bytes) of contiguous data movement from DRAM to cache. In practice,

$$\tau_b = \frac{8\text{(Bytes)}}{\text{bandwidth (in GBytes/s)}} \cdot 10^{-9}.$$

For single core, we need to further multiply it by the number of channels.

The total execution time (in seconds), T, is broken down into the time for arithmetic operations, T_a, and memory

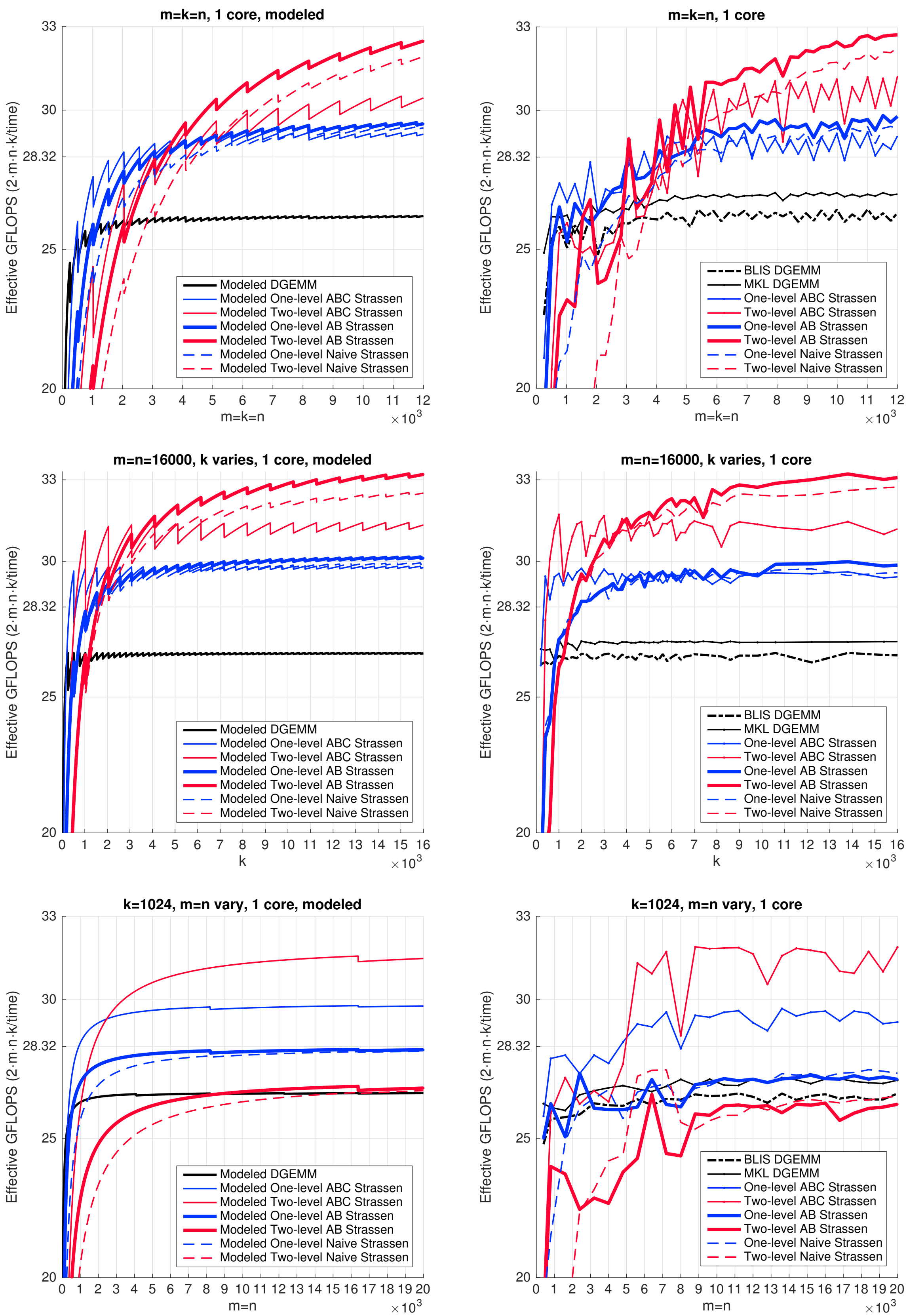

Fig. 5. Performance of the various implementations on an Intel® Xeon® E5 2680 v2 (Ivybridge) processor (*single core*). Left: modeled performance. Right: actual performance. The range of the y-axis does not start at 0 to make the graphs more readable and 28.32 marks theoretical peak performance for this architecture.

operations:

$$T = T_a + T_m. \quad (4)$$

Arithmetic Operations: We break down T_a into separate terms:

$$T_a = T_a^{\times} + T_a^{A+} + T_a^{B+} + T_a^{C+}, \quad (5)$$

where $T_a^{\times}$ is the arithmetic time for submatrix multiplication, and T_a^{A+}, T_a^{B+}, T_a^{C+} denote the arithmetic time of extra additions with submatrices of A, B, C, respectively. For DGEMM since there are no extra additions, $T_a = 2mnk \cdot \tau_a$. For one-level STRASSEN, T_a is comprised of 7 submatrix multiplications, 5 extra additions with submatrices of A, 5 extra additions with submatrices of B, and 12 extra additions with submatrices of C. Therefore, $T_a = (1.75mnk + 2.5mk + 2.5kn + 6mn) \cdot \tau_a$. Note that the matrix addition actually involves 2 floating point operations for each entry because they are cast as `FMA` instructions. Similar analyses can be applied to compute T_a of a two-level STRASSEN implementation. A full analysis is summarized in Figure 4.

Memory Operations: The total data movement overhead is determined by both the original matrix sizes m, n, k, and block sizes m_C, n_C, k_C in our implementation Figure 1(right). We characterize each memory operation term in Figure 4 by its read/write type and the amount of memory (one unit=double precision floating number size=eight bytes) involved in the movement. We decompose T_m into

$$T_m = N_m^{A\times} \cdot T_m^{A\times} + N_m^{B\times} \cdot T_m^{B\times} + N_m^{C\times} \cdot T_m^{C\times} + N_m^{A+} \cdot T_m^{A+} + N_m^{B+} \cdot T_m^{B+} + N_m^{C+} \cdot T_m^{C+}, \quad (6)$$

where $T_m^{A\times}$, $T_m^{B\times}$ are the data movement time for reading from submatrices of A, B, respectively, for memory packing in (adapted) DGEMM routine; $T_m^{C\times}$ is the data movement time for loading *and* storing submatrices of C in (adapted) DGEMM routine; T_m^{A+}, T_m^{B+}, T_m^{C+} are the data movement time for loading *or* storing submatrices of A, B, C, respectively, related to the temporary buffer as part of **Naive Strassen** and **AB Strassen**, outside (adapted) DGEMM routine; the N_m^Xs denote the corresponding coefficients, which are also tabulated in Figure 4.

All write operations ($T_m^{\widetilde{A}\times}$, $T_m^{\widetilde{B}\times}$ for storing submatrices of A, B, respectively, into packing buffers) are omitted because our assumption of lazy write-back policy with fast caches. Notice that memory operations can recur multiple times depending on the loop in which they reside. For instance, for two-level STRASSEN, $T_m^{C\times} = 2\lceil \frac{k/4}{k_c} \rceil \frac{m}{4} \frac{n}{4} \tau_b$ denotes the cost of reading and writing the $\frac{m}{4} \times \frac{n}{4}$ submatrices of C as intermediate result inside the micro-kernel. This is a step function proportional to k, because submatrices of C are used to accumulate the rank-k update in the 5th loop in Figure 1(right).

C. Discussion

From the analysis summarized in Figure 4 we can make predications about the relative performance of the various implementations. It helps to also view the predictions as graphs, which we give in Figure 5, using parameters that capture the architecture described in Section V-A.

- Asymptotically, the two-level STRASSEN implementations outperform corresponding one-level STRASSEN implementations, which in turn outperform the traditional DGEMM implementation.
- The graph for $m = k = n$, 1 core, shows that for smaller square matrices, **ABC Strassen** outperforms **AB Strassen**, but for larger square matrices this trend reverses. This holds for both one-level and two-level STRASSEN. The reason is that, for small k, **ABC Strassen** reduced the number of times the temporary matrix M needs to be brought in from memory to be added to submatrices of C. For large k, it increases the number of times the elements of those submatrices of C themselves are moved in and out of memory.
- The graph for $m = n = 16000$, k varies, 1 core, is particularly interesting: it shows that for k equal to the appropriate multiple of k_C ($k = 2k_C$ for one-level and $k = 4k_C$ for two-level) **ABC Strassen** performs dramatically better than the other implementations, as expected.

The bottom line: depending on the problem size, a different implementation may have its advantages.

V. PERFORMANCE EXPERIMENTS

We give details on the performance experiments for our implementations. The current version of STRASSEN DGEMM is designed for the Intel® Sandy-Bridge/Ivy-Bridge processor and Intel® Xeon Phi™ coprocessor (MIC Architecture, KNC). In addition, we incorporate our implementations in a distributed memory DGEMM.

A. Single node experiments

Implementation: The implementations are in `C`, utilizing `SSE2` and `AVX` intrinsics and assembly, compiled with the Intel® `C` compiler version 15.0.3 with optimization flag `-O3`. In addition, we compare against the standard BLIS implementation (Version 0.1.8) from which our implementations are derived as well as Intel® MKL's DGEMM (Version 11.2.3) [27].

Target architecture: We measure the CPU performance results on the Maverick system at the Texas Advanced Computing Center (TACC). Each node of that system consists of a dual-socket (10 cores/socket) Intel® Xeon® E5-2680 v2 (Ivy Bridge) processors with 12.8 GB/core of memory (Peak Bandwidth: 59.7 GB/s with four channels) and a three-level cache: 32 KB L1 data cache, 256 KB L2 cache and 25.6 MB L3 cache. The stable CPU clockrate is 3.54 GHz when a single core is utilized (28.32 GFLOPS peak, marked in the graphs) and 3.10 GHz when five or more cores are in use (24.8 GLOPS/core peak). To set thread affinity and to ensure the computation and the memory allocation all reside on the same socket, we use `KMP_AFFINITY=compact`.

We choose the parameters $n_R = 4$, $m_R = 8$, $k_C = 256$, $n_C = 4096$ and $m_C = 96$. This makes the size of the

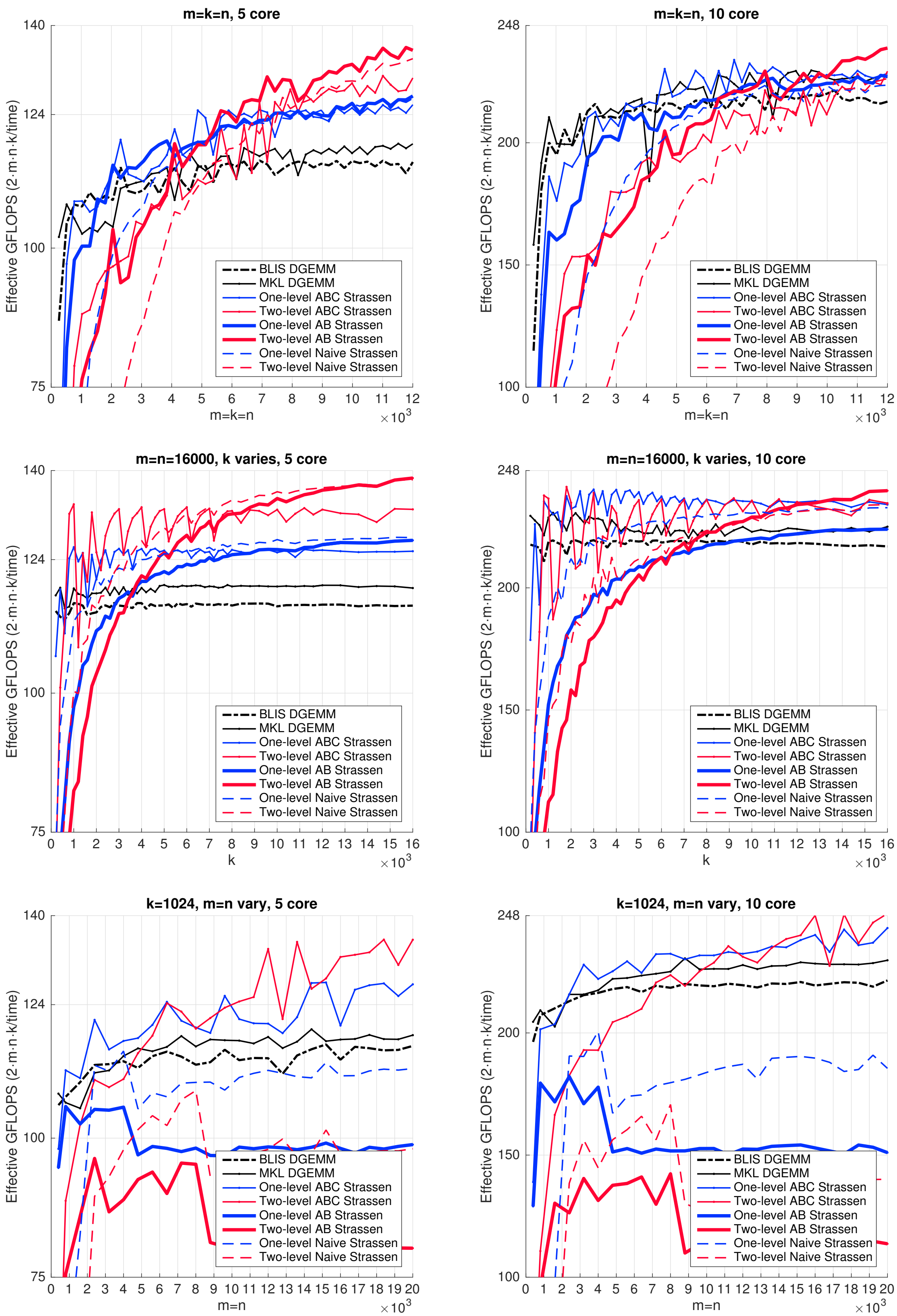

Fig. 6. Performance of the various implementations on an Intel® Xeon® E5 2680 v2 (Ivybridge) processor (*five and ten cores*). Left: 5 core. Right: 10 core. The range of the y-axis does not start at 0 to make the graphs more readable.

packing buffer $\widetilde{A}_i$ 192 KB and $\widetilde{B}_p$ 8192 KB, which then fit the L2 cache and L3 cache, respectively. These parameters are consistent with parameters used for the standard BLIS DGEMM implementation for this architecture.

Each socket consists of 10 cores, allowing us to also perform multi-threaded experiments. Parallelization is implemented mirroring that described in [23], using OpenMP directives that parallelize the 3^{rd} loop around the micro-kernel in Figure 1.

Results: Results when using single core are presented in Figure 5 (right column). As expected, eventually two-level **AB Strassen** performs best, handily beating conventional DGEMM. The exception is the case where k is fixed to equal $1024 = 4 \times k_C$, which is the natural blocking size for a two-level STRASSEN based on our ideas. For those experiments **ABC Strassen** wins out, as expected. These experiments help validate our model.

Figure 6 reports results for five and ten cores, all within the same socket. We do not report results for twenty cores (two sockets), since this results in a substantial performance reduction for all our implementations, including the standard BLIS DGEMM, relative to the MKL DGEMM. This exposes a performance bug[4] in BLIS that has been reported.

When using many cores, memory bandwidth contention affects the performance of the various STRASSEN implementations, reducing the benefits relative to a standard DGEMM implementation.

B. Many-core experiments

To examine whether the techniques scale to a large number of cores, we port our implementation of one-level **ABC Strassen** to the Intel® Xeon Phi™ coprocessor.

Implementation: The implementations of **ABC Strassen** are in `C` and `AVX512` intrinsics and assembly, compiled with the Intel® `C` compiler version 15.0.2 with optimization flag `-mmic -O3`. The BLIS and **ABC Strassen** both parallelize the 2^{nd} and 3^{rd} loop around the micro-kernel, as described for BLIS in [23].

Target architecture: We run the Xeon Phi performance experiments on the SE10P Coprocessor incorporated into nodes of the Stampede system at TACC. This coprocessor has a peak performance of 1056 GFLOPS (for 60 cores with 240 threads used by BLIS) and 8 GB of GDDR5 DRAM with a peak bandwidth of 352 GB/s. It has 512 KB L2 cache, but no L3 cache.

We choose the parameters $n_R = 8$, $m_R = 30$, $k_C = 240$, $n_C = 14400$ and $m_C = 120$. This makes the size of the packing buffer $\widetilde{A}_i$ 225 KB and $\widetilde{B}_p$ 27000 KB, which fits L2 cache and main memory separately (no L3 cache on Xeon Phi). These choices are consistent with those used by BLIS for this architecture.

Results: As illustrated in Figure 7, relative to the BLIS DGEMM implementation, the one-level **ABC Strassen** shows a nontrivial improvement for a rank-k update with a fixed (large)

[4] In the distributed memory experiments, we overcome this NUMA issue by running one MPI process per socket to demonstrate the performance results on both sockets (20 cores) of each node.

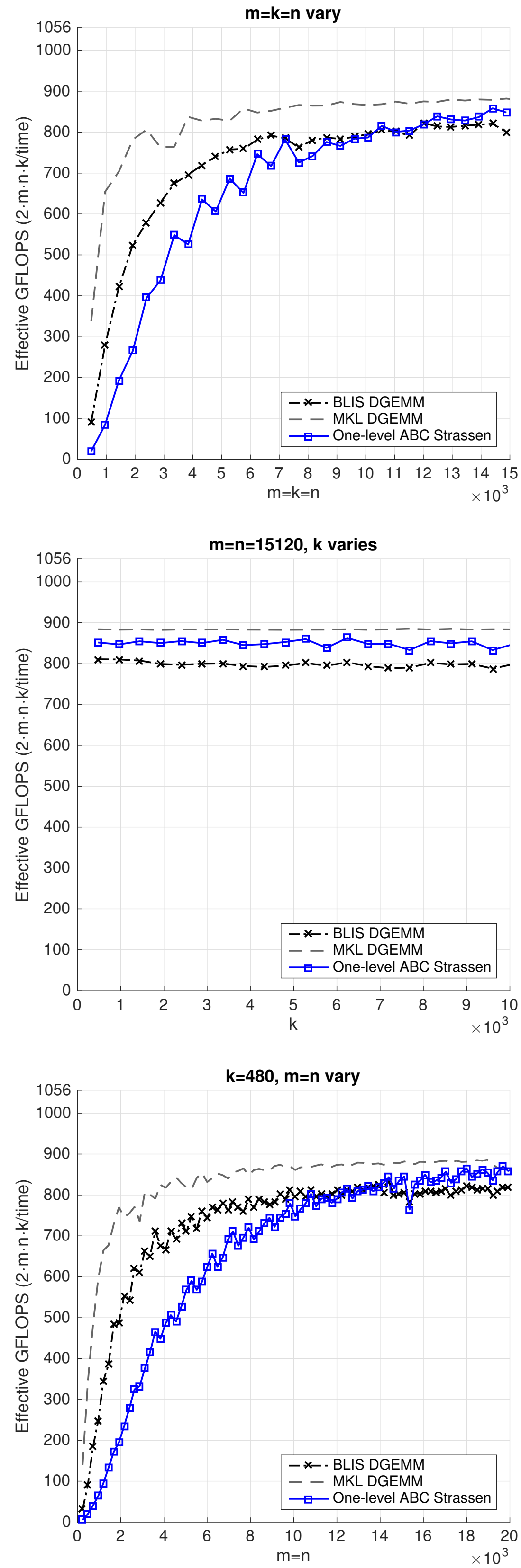

Fig. 7. Performance of one-level **ABC Strassen**, BLIS, and MKL, on an Intel® Xeon Phi™ (KNC) coprocessor (for *60 cores with 240 threads*). The performance results are sampled such that k is a multiple of $480 = 2 \times k_C$.

matrix C (the graph for $m = n = 15120$, k varies). While the BLIS implementation on which our implementation of **ABC Strassen** is based used to be highly competitive with MKL's DGEMM (as reported in [23]), recent improvements in that library demonstrate that the BLIS implementation needs an update. We do not think there is a fundamental reason why our observations cannot be used to similarly accelerate MKL's DGEMM.

C. Distributed memory experiments

Finally, we demonstrate how the **ABC Strassen** implementation can be used to accelerate a distributed memory implementation of DGEMM.

Implementation: We implement the Scalable Universal Matrix Multiplication Algorithm (SUMMA) [28] with MPI. This algorithm distributes the algorithm to a mesh of MPI processes using a 2D block cyclic distribution. The multiplication is broken down into a sequence of rank-b updates,

$$C \;:=\; AB + C = \left(A_0 \mid \cdots \mid A_{K-1} \right) \begin{pmatrix} B_0 \\ \hline \vdots \\ \hline B_{K-1} \end{pmatrix} + C$$

$$= A_0 B_0 + \cdots + A_{K-1} B_{K-1} + C$$

where each A_p consists of (approximately) b columns and each B_p consists of (approximately) b rows. For each rank-b update A_p is broadcast within rows of the mesh and B_p is broadcast within columns of the mesh, after which locally a rank-b update with the arriving submatrices is performed to update the local block of C.

Target architecture: The distributed memory experiments are performed on the same machine described in Section V-A, using the `mvapich2` version 2.1 [29] implementation of MPI. Each node has two sockets, and each socket has ten cores.

Results: Figure 8 reports weak scalability on up to 32 nodes (64 sockets, 640 cores). For these experiments we choose the MPI mesh of processes to be square, with one MPI process per socket, and attained thread parallelism among the ten cores in a socket within BLIS, MKL, or any of our STRASSEN implementations.

It is well-known that the SUMMA algorithm is weakly scalable in the sense that efficiency essentially remains constant if the local memory dedicated to matrices A, B, C, and temporary buffers is kept constant. For this reason, the local problem size is fixed to equal $m = k = n = 16000$ so that the global problem becomes $m = k = n = 16000 \times N$ when an $N \times N$ mesh of sockets (MPI processes) is utilized. As expected, the graph shows that the SUMMA algorithm is weakly scalable regardless of which local DGEMM algorithm is used. The local computation within the SUMMA algorithm matches the shape for which **ABC Strassen** is a natural choice when the rank-k updates are performed with $b = 1024$. For this reason, the one-level and two-level **ABC Strassen** implementations achieve the best performance.

What this experiment shows is that the benefit of using our STRASSEN implementations can be easily transferred to other algorithms that are rich in large rank-k updates.

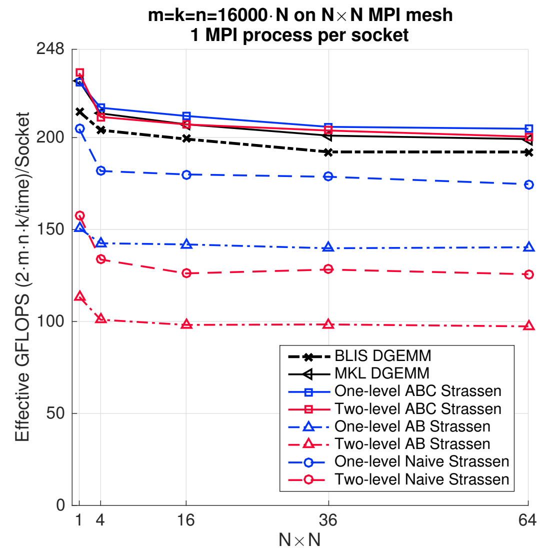

Fig. 8. Performance of the various implementations on distributed memory (weak scalability).

VI. CONCLUSION

We have presented novel insights into the implementations of STRASSEN that greatly reduce overhead that was inherent in previous formulations and had been assumed to be insurmountable. These insights have yielded a family of algorithms that outperform conventional high-performance implementations of DGEMM as well as naive implementations. We develop a model that predicts the run time of the various implementations. Components that are part of the BLIS framework for implementing BLAS-like libraries are modified to facilitate implementation. Implementations and performance experiments are presented that verify the performance model and demonstrate performance benefits for single-core, multi-core, many-core, and distributed memory parallel implementations. Together, this advances more than 45 years of research into the theory and practice of Strassen-like algorithms.

Our analysis shows that the **ABC Strassen** implementation fulfills our claim that STRASSEN can outperform classical DGEMM for small matrices and small k while requiring no temporary buffers beyond those already internal to high-performance DGEMM implementations. The **AB Strassen** implementation becomes competitive once k is larger. It only requires a $\frac{m}{2^L} \times \frac{n}{2^L}$ temporary matrix for an L-level STRASSEN.

A number of avenues for further research and development naturally present themselves.

- The GotoBLAS approach for DGEMM is also the basis for high-performance implementations of all level-3 BLAS [30] and the BLIS framework has been used to implement these with the same micro-kernel and modifications of the packing routines that support DGEMM. This presents the possibility of creating Strassen-like algorithms for some or all level-3 BLAS.

- Only **ABC Strassen** has been implemented for the Intel® Xeon Phi™ (KNC) coprocessor. While this demonstrates that parallelism on many-core architectures can be effectively exploited, a more complete study needs to be pursued. Also, the performance improvements in MKL for that architecture need to be duplicated in BLIS and/or the techniques incorporated into the MKL library.
- Most of the blocked algorithms that underlie LAPACK and ScaLAPACK [31] cast computation in terms of rank-k updates. It needs to be investigated how the **ABC Strassen** implementation can be used to accelerate these libraries.
- Distributed memory implementations of Strassen's algorithms have been proposed that incorporate several levels of Strassen before calling a parallel SUMMA or other distributed memory parallel DGEMM implementation [25]. On the one hand, the performance of our approach that incorporates STRASSEN in the local DGEMM needs to be compared to these implementations. On the other hand, it may be possible to add a local STRASSEN DGEMM into these parallel implementations. Alternatively, the required packing may be incorporated into the communication of the data.
- A number of recent papers have proposed multi-threaded parallel implementations that compute multiple submatrices M_i in parallel [8]. Even more recently, new practical Strassen-like algorithms have been proposed together with their multi-threaded implementations [9]. How our techniques compare to these and whether they can be combined needs to be pursued. It may also be possible to use our cost model to help better schedule this kind of task parallelism.

These represent only a small sample of new possible directions.

ACKNOWLEDGMENTS

This work was sponsored in part by the National Science Foundation grants ACI-1148125/1340293, CCF-1218483; by Intel Corporation through an Intel® Parallel Computing Center. Access to the Maverick and Stampede supercomputers administered by TACC is gratefully acknowledged. We thank Field Van Zee, Chenhan Yu, Devin Matthews, and the rest of the SHPC team (`http://shpc.ices.utexas.edu`) for their supports.

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.

REFERENCES

[1] V. Strassen, "Gaussian elimination is not optimal," *Numer. Math.*, vol. 13, pp. 354–356, 1969.

[2] S. Winograd, "On multiplication of 2×2 matrices," *Linear algebra and its applications*, vol. 4, no. 4, pp. 381–388, 1971.

[3] D. Bini, M. Capovani, F. Romani, and G. Lotti, "O $(n^{2.7799})$ complexity for $n \times n$ approximate matrix multiplication," *Information processing letters*, vol. 8, no. 5, pp. 234–235, 1979.

[4] A. Schönhage, "Partial and total matrix multiplication," *SIAM Journal on Computing*, vol. 10, no. 3, pp. 434–455, 1981.

[5] A. V. Smirnov, "The bilinear complexity and practical algorithms for matrix multiplication," *Computational Mathematics and Mathematical Physics*, vol. 53, no. 12, pp. 1781–1795, 2013. [Online]. Available: http://dx.doi.org/10.1134/S0965542513120129

[6] C. Douglas, M. Heroux, G. Slishman, and R. Smith, "GEMMW - a portable level 3 BLAS Winograd variant of Strassen's matrix-matrix multiplication algorithm," *J. Computational Physics*, pp. 1–10, 1994.

[7] S. Huss-Lederman, E. M. Jacobson, A. Tsao, T. Turnbull, and J. R. Johnson, "Implementation of Strassen's algorithm for matrix multiplication," in *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '96. Washington, DC, USA: IEEE Computer Society, 1996. [Online]. Available: http://dx.doi.org/10.1145/369028.369096

[8] P. D'Alberto, M. Bodrato, and A. Nicolau, "Exploiting parallelism in matrix-computation kernels for symmetric multiprocessor systems: Matrix-multiplication and matrix-addition algorithm optimizations by software pipelining and threads allocation," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 2:1–2:30, December 2011. [Online]. Available: http://doi.acm.org/10.1145/2049662.2049664

[9] A. R. Benson and G. Ballard, "A framework for practical parallel fast matrix multiplication," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 2015. New York, NY, USA: ACM, 2015, pp. 42–53.

[10] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed. Philadelphia, PA, USA: SIAM, 2002.

[11] J. Demmel, I. Dumitriu, O. Holtz, and R. Kleinberg, "Fast matrix multiplication is stable," *Numerische Mathematik*, vol. 106, no. 2, pp. 199–224, 2007.

[12] G. Ballard, A. R. Benson, A. Druinsky, B. Lipshitz, and O. Schwartz, "Improving the numerical stability of fast matrix multiplication algorithms," *CoRR*, vol. abs/1507.00687, 2015. [Online]. Available: http://arxiv.org/abs/1507.00687

[13] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff, "A set of level 3 basic linear algebra subprograms," *ACM Trans. Math. Soft.*, vol. 16, no. 1, pp. 1–17, March 1990.

[14] F. G. Van Zee and R. A. van de Geijn, "BLIS: A framework for rapidly instantiating BLAS functionality," *ACM Trans. Math. Soft.*, vol. 41, no. 3, pp. 14:1–14:33, June 2015. [Online]. Available: http://doi.acm.org/10.1145/2764454

[15] K. Goto and R. A. van de Geijn, "Anatomy of a high-performance matrix multiplication," *ACM Trans. Math. Soft.*, vol. 34, no. 3, p. 12, May 2008, article 12, 25 pages.

[16] C. D. Yu, J. Huang, W. Austin, B. Xiao, and G. Biros, "Performance optimization for the K-Nearest Neighbors kernel on x86 architectures," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: ACM, 2015, pp. 7:1–7:12.

[17] D. A. Matthews, "High-performance tensor contraction without BLAS," *CoRR*, vol. abs/1607.00291, 2016. [Online]. Available: http://arxiv.org/abs/1607.00291

[18] P. Springer and P. Bientinesi, "Design of a high-performance gemm-like tensor-tensor multiplication," *CoRR*, vol. abs/1607.00145, 2016. [Online]. Available: http://arxiv.org/abs/1607.00145

[19] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. J. Dongarra, J. D. Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen, *LAPACK Users' Guide (Third Ed.)*. Philadelphia, PA, USA: SIAM, 1999.

[20] F. G. Van Zee and T. M. Smith, "Implementing high-performance complex matrix ymultiplication," *ACM Transactions on Mathematical Software*, 2016, submitted.

[21] "GOTOBLAS," https://www.tacc.utexas.edu/research-development/tacc-software/gotoblas2.

[22] "OpenBLAS, an optimized BLAS library," http://www.openblas.net.

[23] T. M. Smith, R. A. van de Geijn, M. Smelyanskiy, J. R. Hammond, and F. G. Van Zee, "Anatomy of high-performance many-threaded matrix multiplication," in *28th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2014)*, 2014.

[24] B. Boyer, J.-G. Dumas, C. Pernet, and W. Zhou, "Memory efficient scheduling of Strassen-Winograd's matrix multiplication algorithm," in *Proceedings of the 2009 International Symposium on Symbolic and Algebraic Computation*, ser. ISSAC '09. New York, NY, USA: ACM, 2009, pp. 55–62.

[25] B. Grayson and R. van de Geijn, "A high performance parallel Strassen implementation," *Parallel Processing Letters*, vol. 6, no. 1, pp. 3–12, 1996.

[26] B. Lipshitz, G. Ballard, J. Demmel, and O. Schwartz, "Communication-avoiding parallel Strassen: Implementation and performance," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 101:1–101:11.

[27] "Intel Math Kernel Library," https://software.intel.com/en-us/intel-mkl.

[28] R. van de Geijn and J. Watts, "SUMMA: Scalable universal matrix multiplication algorithm," *Concurrency: Practice and Experience*, vol. 9, no. 4, pp. 255–274, April 1997.

[29] W. Huang, G. Santhanaraman, H. W. Jin, Q. Gao, and D. K. Panda, "Design of high performance MVAPICH2: MPI2 over infiniband," in *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, vol. 1, May 2006, pp. 43–48.

[30] K. Goto and R. van de Geijn, "High-performance implementation of the level-3 BLAS," *ACM Trans. Math. Soft.*, vol. 35, no. 1, pp. 1–14, 2008.

[31] 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*. IEEE Comput. Soc. Press, 1992, pp. 120–127.