Generating Families of Practical Fast Matrix Multiplication Algorithms

Jianyu Huang*†, Leslie Rice*†, Devin A. Matthews†, 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@cs., leslierice@, dmatthews@, rvdg@cs.}utexas.edu

***Abstract*—Matrix multiplication (GEMM) is a core operation to numerous scientific applications. Traditional implementations of Strassen-like fast matrix multiplication (FMM) algorithms often do not perform well except for very large matrix sizes, due to the increased cost of memory movement, which is particularly noticeable for non-square matrices. Such implementations also require considerable workspace and modifications to the standard BLAS interface. We propose a code generator framework to automatically implement a large family of FMM algorithms suitable for multiplications of arbitrary matrix sizes and shapes. By representing FMM with a triple of matrices $[\![U, V, W]\!]$ that capture the linear combinations of submatrices that are formed, we can use the Kronecker product to define a multi-level representation of Strassen-like algorithms. Incorporating the matrix additions that must be performed for Strassen-like algorithms into the inherent packing and micro-kernel operations inside GEMM avoids extra workspace and reduces the cost of memory movement. Adopting the same loop structures as high-performance GEMM implementations allows parallelization of all FMM algorithms with simple but efficient data parallelism without the overhead of task parallelism. We present a simple performance model for general FMM algorithms and compare actual performance of 20+ FMM algorithms to modeled predictions. Our implementations demonstrate a performance benefit over conventional GEMM on single core and multi-core systems. This study shows that Strassen-like fast matrix multiplication can be incorporated into libraries for practical use.**

I. Introduction

Three recent advances have revived interest in the practical implementation of Strassen's algorithm (STRASSEN) and similar Fast Matrix Multiplication (FMM) algorithms. The first [1] is a systematic way in which new FMM algorithms can be identified, building upon conventional calls to the BLAS matrix-matrix multiplication GEMM routine. That work incorporated a code generator, due to the number of algorithms that are identified and the complexity of exploiting subexpressions encountered in the linear combinations of submatrices. Parallelism was achieved through a combination of task parallelism and parallelism within the BLAS. The second [2] was the insight that the BLAS-like Library Instantiation Software (BLIS) [3] framework exposes basic building blocks that allow the linear combinations of submatrices in STRASSEN to be incorporated into the packing and/or computational micro-kernels already existing in the BLIS GEMM implementation. Parallelism in that work mirrored the highly effective data parallelism that is part of BLIS. Finally, the present work also extends insights on how to express multiple levels of STRASSEN in terms of Kronecker products [4] to multi-level FMM algorithms, facilitating a code generator for all methods from [1] (including STRASSEN), in terms of the building blocks created for [2], but allowing different FMM algorithms to be used for each level. Importantly and unique to this work, the code generator also yields performance models that are accurate enough to guide the choice of a FMM implementation as a function of problem size and shape, facilitating the creation of poly-algorithms [5]. Performance results from single core and multi-core shared memory system support the theoretical insights.

We focus on the special case of GEMM given by $C := C + AB$. Extending the ideas to the more general case of GEMM is straightforward.

II. Background

We briefly summarize how the BLIS framework implements GEMM before reviewing recent results [2] on how STRASSEN can exploit insights that underlie this framework.

A. High-performance implementation of standard GEMM

Key to high performance implementations of GEMM is the partitioning of operands in order to (near-)optimally reuse data in the various levels of memory. Figure 1(left) illustrates how BLIS implements the GOTOBLAS [6] approach. Block sizes $\{m_C, n_C, k_C\}$ are chosen so that submatrices fit in the various caches while $\{m_R, n_R\}$ relate to submatrices in registers that contribute to C. These parameters can be analytically determined [7]. To improve data locality, row panels B_p that fit in the L3 cache are "packed" into contiguous memory, yielding $\widetilde{B}_p$. For similar reasons, blocks A_i that fit in the L2 cache are packed into buffer $\widetilde{A}_i$.

B. High-performance implementations of STRASSEN

If one partitions the three operands into quadrants,

$$X = \left(\begin{array}{c|c} X_0 & X_1 \\ \hline X_2 & X_3 \end{array}\right) \quad \text{for } X \in \{A, B, C\} \qquad (1)$$

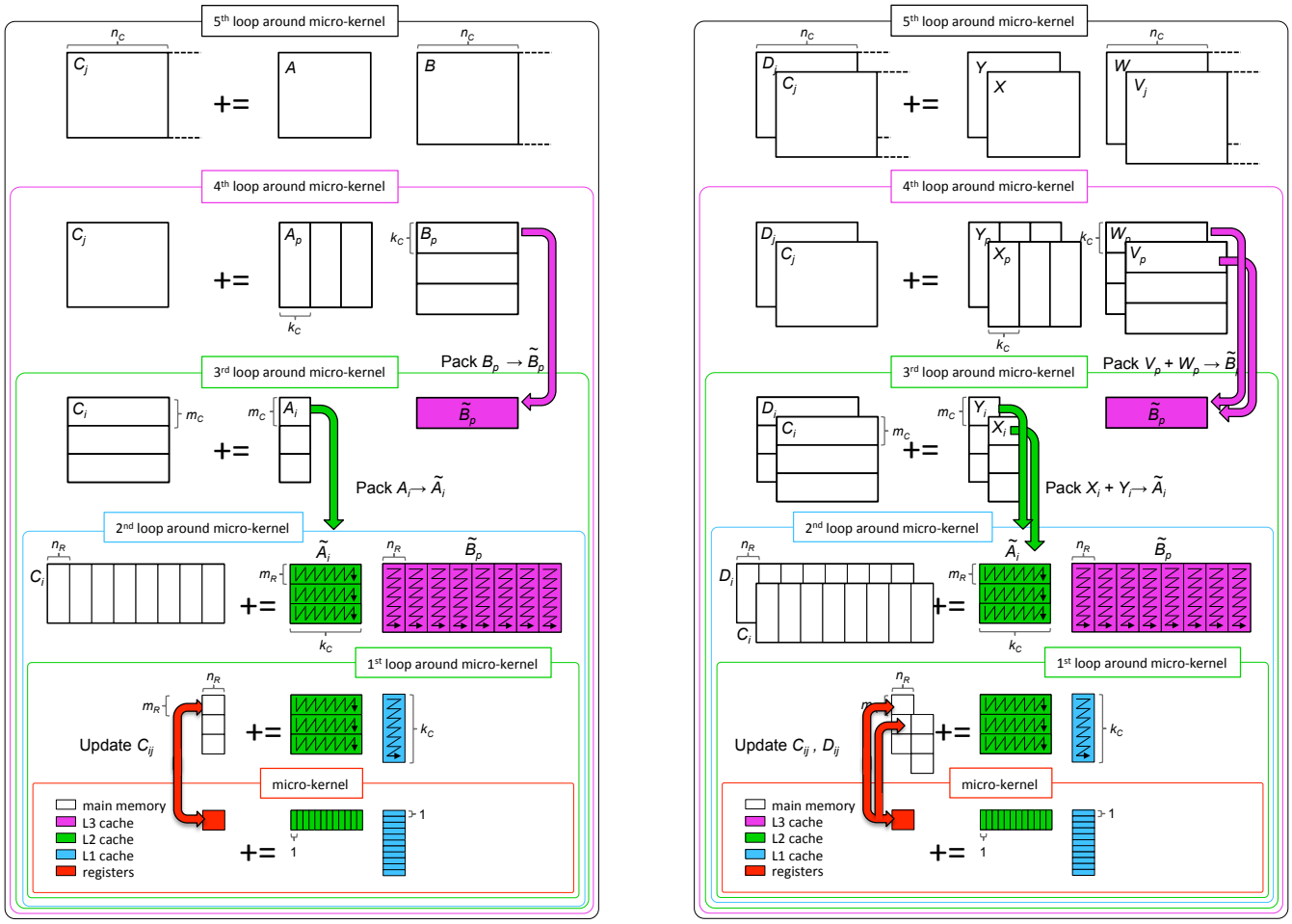

Figure 1. Figure from [2] (used with permission from authors). Left: Illustration of the BLIS implementation of the GOTOBLAS GEMM algorithm. All computation is cast in terms of a micro-kernel that is highly optimized. Right: modification that implements the representative computation $M = (X + Y)(V + W); C \mathrel{+}= M; D \mathrel{+}= M$ of each row of computations in (2). X, Y are submatrices of A; V, W are submatrices of B; C, D are submatrices of the original matrix C; M is the intermediate matrix product. Note that the packing buffers $\widetilde{A}_i$ and $\widetilde{B}_p$ stay in cache.

then it can be shown that the operations

$$\begin{array}{ll} M_0 = (A_0 + A_3)(B_0 + B_3); & C_0 \mathrel{+}= M_0; C_3 \mathrel{+}= M_0; \\ M_1 = (A_2 + A_3)B_0; & C_2 \mathrel{+}= M_1; C_3 \mathrel{-}= M_1; \\ M_2 = A_0(B_1 - B_3); & C_1 \mathrel{+}= M_2; C_3 \mathrel{+}= M_2; \\ M_3 = A_3(B_2 - B_0); & C_0 \mathrel{+}= M_3; C_2 \mathrel{+}= M_3; \\ M_4 = (A_0 + A_1)B_3; & C_1 \mathrel{+}= M_4; C_0 \mathrel{-}= M_4; \\ M_5 = (A_2 - A_0)(B_0 + B_1); & C_3 \mathrel{+}= M_5; \\ M_6 = (A_1 - A_3)(B_2 + B_3); & C_0 \mathrel{+}= M_6; \end{array} \tag{2}$$

compute $C := AB + C$, but with seven (sub)matrix multiplications, reducing the cost by a factor of $7/8$ (ignoring a lower order number of extra additions). If all matrices are square and of size $n \times n$, classical STRASSEN exploits this recursively, reducing the cost for GEMM to $O(n^{2.807})$.

Only a few levels of the recursion are exploited in practice because the cost of extra additions and extra memory movements quickly offsets the reduction in floating point operations. Also, STRASSEN is known to become more numerically unstable particularly when more than two levels of recursion are employed [8], [9], [10][1].

In [2], captured in Figure 1(right), it was noted that the additions of the submatrices of A and B can be incorporated into the packing buffers $\widetilde{A}_i$ and $\widetilde{B}_p$, avoiding extra memory movements. In addition, once a submatrix that contributes to C is computed in the registers, it can be directly added to the appropriate parts of multiple submatrices of C, thus avoiding the need for temporary matrices M_r, again avoiding extra memory movements. As demonstrated in [2], this makes the method practical for smaller matrices and matrices of special shape (especially rank-k updates, where k is relatively small).

[1]Note that [10] provides techniques for ameliorating the numerical stability issues of fast matrix multiplication algorithms. The dominant computations after applying those amending techniques are still the same as we are targeting in [2] and this paper.

III. Fast Matrix Multiplication Algorithms

We now present the basic idea that underlies families of FMM algorithms and how to generalize one-level formula for multi-level FMM utilizing Kronecker products and recursive block storage indexing.

A. One-level fast matrix multiplication algorithms

In [1], the theory of tensor contractions is used to find a large number of FMM algorithms. In this subsection, we use the output (the resulting algorithms) of their approach.

Generalizing the partitioning for STRASSEN, consider $C := C + AB$, where C, A, and B are $m \times n$, $m \times k$, and $k \times n$ matrices, respectively. [1] defines a $\langle \widetilde{m}, \widetilde{k}, \widetilde{n} \rangle$ algorithm by partitioning

$$C = \left(\begin{array}{c|c|c} C_0 & \cdots & C_{\widetilde{n}-1} \\ \hline \vdots & & \vdots \\ \hline C_{(\widetilde{m}-1)\widetilde{n}} & \cdots & C_{\widetilde{m}\widetilde{n}-1} \end{array}\right), A = \left(\begin{array}{c|c|c} A_0 & \cdots & A_{\widetilde{k}-1} \\ \hline \vdots & & \vdots \\ \hline A_{(\widetilde{m}-1)\widetilde{k}} & \cdots & A_{\widetilde{m}\widetilde{k}-1} \end{array}\right),$$

$$\text{and } B = \left(\begin{array}{c|c|c} B_0 & \cdots & B_{\widetilde{n}-1} \\ \hline \vdots & & \vdots \\ \hline B_{(\widetilde{k}-1)\widetilde{n}} & \cdots & B_{\widetilde{k}\widetilde{n}-1} \end{array}\right)$$

Note that A_i, B_j, and C_p are the submatrices of A, B and C, with a single index in the row major order. Then, $C := C + AB$ is computed by,

for $r = 0, ..., R-1$,

$$\begin{aligned} M_r &:= \left(\sum_{i=0}^{\widetilde{m}\widetilde{k}-1} u_{ir} A_i\right) \times \left(\sum_{j=0}^{\widetilde{k}\widetilde{n}-1} v_{jr} B_j\right); \\ C_p &+= w_{pr} M_r \ (p = 0, ..., \widetilde{m}\widetilde{n}-1) \end{aligned} \tag{3}$$

where $(\times)$ is a matrix multiplication that can be done recursively, u_{ir}, v_{jr}, and w_{pr} are entries of a $(\widetilde{m}\widetilde{k}) \times R$ matrix U, a $(\widetilde{k}\widetilde{n}) \times R$ matrix V, and a $(\widetilde{m}\widetilde{n}) \times R$ matrix W, respectively. Therefore, the classical matrix multiplication which needs $\widetilde{m}\widetilde{k}\widetilde{n}$ submatrix multiplications can be completed with R submatrix multiplications. The set of coefficients that determine the $\langle \widetilde{m}, \widetilde{k}, \widetilde{n} \rangle$ algorithm is denoted as $[\![U, V, W]\!]$.

For example, assuming that m, n, and k are all even, one-level STRASSEN has $\langle 2, 2, 2 \rangle$ partition dimensions and, given the partitioning in (1) and computations in (2),

$$[\![\begin{pmatrix} 1 & 0 & 1 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & -1 \end{pmatrix}, \begin{pmatrix} 1 & 1 & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & -1 & 0 & 1 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 0 & 1 & -1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & -1 & 1 & 0 & 0 & 1 & 0 \end{pmatrix}]\!] \tag{4}$$

specifies $[\![U, V, W]\!]$ for one-level STRASSEN.

Figure 2 summarizes a number of such algorithms that can be found in the literature that we eventually test in Section V. We only consider $2 \leq \widetilde{m}, \widetilde{k}, \widetilde{n} \leq 6$ and don't include arbitrary precision approximate (APA) algorithms [11], due to their questionable numerical stability.

$\langle \widetilde{m}, \widetilde{k}, \widetilde{n} \rangle$	Ref.	$\widetilde{m}\widetilde{k}\widetilde{n}$	R	Speedup (%)				
				Theory	Practical #1		Practical #2	
					Ours	[1]	Ours	[1]
$\langle 2,2,2 \rangle$	[13]	8	7	14.3	11.9	-3.0	13.1	13.1
$\langle 2,3,2 \rangle$	[1]	12	11	9.1	5.5	-13.1	7.7	7.7
$\langle 2,3,4 \rangle$	[1]	24	20	20.0	11.9	-8.0	16.3	17.0
$\langle 2,4,3 \rangle$	[10]	24	20	20.0	4.8	-15.3	14.9	16.6
$\langle 2,5,2 \rangle$	[10]	20	18	11.1	1.5	-23.1	8.6	8.3
$\langle 3,2,2 \rangle$	[10]	12	11	9.1	7.1	-6.6	7.2	7.5
$\langle 3,2,3 \rangle$	[10]	18	15	20.0	14.1	-0.7	17.2	16.8
$\langle 3,2,4 \rangle$	[10]	24	20	20.0	11.9	-1.8	16.1	17.0
$\langle 3,3,2 \rangle$	[10]	18	15	20.0	11.4	-8.1	17.3	16.5
$\langle 3,3,3 \rangle$	[14]	27	23	17.4	8.6	-9.3	14.4	14.7
$\langle 3,3,6 \rangle$	[14]	54	40	35.0	-34.0	-41.6	24.2	20.1
$\langle 3,4,2 \rangle$	[1]	24	20	20.0	4.9	-15.7	16.0	16.8
$\langle 3,4,3 \rangle$	[14]	36	29	24.1	8.4	-12.6	18.1	20.1
$\langle 3,5,3 \rangle$	[14]	45	36	25.0	5.2	-20.6	19.1	18.9
$\langle 3,6,3 \rangle$	[14]	54	40	35.0	-21.6	-64.5	19.5	17.8
$\langle 4,2,2 \rangle$	[10]	16	14	14.3	9.4	-4.7	11.9	12.2
$\langle 4,2,3 \rangle$	[1]	24	20	20.0	12.1	-2.3	15.9	17.3
$\langle 4,2,4 \rangle$	[10]	32	26	23.1	10.4	-2.7	18.4	19.1
$\langle 4,3,2 \rangle$	[10]	24	20	20.0	11.3	-7.8	16.8	15.7
$\langle 4,3,3 \rangle$	[10]	36	29	24.1	8.1	-8.4	19.8	20.0
$\langle 4,4,2 \rangle$	[10]	32	26	23.1	-4.2	-18.4	17.1	18.5
$\langle 5,2,2 \rangle$	[10]	20	18	11.1	7.0	-6.7	8.2	8.5
$\langle 6,3,3 \rangle$	[14]	54	40	35.0	-33.4	-42.2	24.0	20.2

Figure 2. Theoretical and practical speedup for various FMM algorithms. $\widetilde{m}\widetilde{k}\widetilde{n}$ is the number of multiplication for classical matrix multiplication algorithm. R is the number of multiplication for fast matrix multiplication algorithm. Theoretical speedup is the speedup per recursive step. Practical #1 speedup is the speedup for one-level FMM comparing with GEMM when $m = n = 14400, k = 480$ (rank-k updates)[2]. Practical #2 speedup is the speedup for one-level FMM comparing with GEMM when $m = n = 14400, k = 12000$ (approximately square). We report the practical speedup of the best implementation of our generated code (generated GEMM) and the implementations in [1] (linked with Intel MKL) on single core. More details about the experiment setup is described in Section V.

B. Kronecker Product

If X and Y are $m \times n$ and $p \times q$ matrices with (i, j) entries denoted by $x_{i,j}$ and $y_{i,j}$, respectively, then the Kronecker product [12] $X \otimes Y$ is the $mp \times nq$ matrix given by

$$X \otimes Y = \begin{pmatrix} x_{0,0}Y & \cdots & x_{0,n-1}Y \\ \vdots & \ddots & \vdots \\ x_{m-1,0}Y & \cdots & x_{m-1,n-1}Y \end{pmatrix}$$

Thus, entry $(X \otimes Y)_{p(r-1)+v, q(s-1)+w} = x_{r,s} y_{v,w}$.

C. Recursive Block Indexing (Morton-like Ordering)

An example of recursive block storage indexing (Morton-like ordering) [15] is given in Figure 3. In this example,

[2]Note that symmetric rotations (e.g. $\langle 2, 3, 4 \rangle$ vs. $\langle 2, 4, 3 \rangle$) may have different performance. This is determined by the block size k_C and the partition dimension $\widetilde{k}$. If $\widetilde{k}$ is relatively large for rank-k updates, then the problem size $k/\widetilde{k}$ after partition might be smaller than k_C.

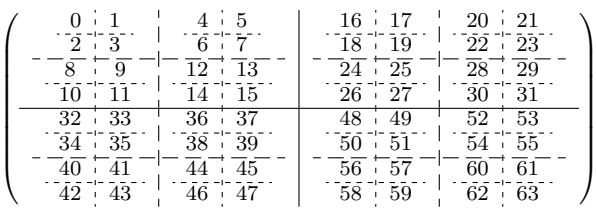

Figure 3. Illustration of recursive block storage indexing (Morton-like ordering) [15] on $m \times k$ matrix A where the partition dimensions $\widetilde{m} = \widetilde{k} = 2$ for three-level recursions.

A undergoes three levels of recursive splitting, and each submatrix of A is indexed in row major form. By indexing A, B, and C in this manner, data locality is maintained when operations are performed on their respective submatrices.

D. Representing two-level FMM with the Kronecker Product

In [4], it is shown that multi-level $\langle 2,2,2\rangle$ STRASSEN can be represented as Kronecker product. In this paper, we extend this insight to multi-level FMM, where each level can use a different choice of $\langle \widetilde{m}, \widetilde{k}, \widetilde{n}\rangle$.

Assume each submatrix of A, B, and C is partitioned with another level of $\langle \widetilde{m'}, \widetilde{k'}, \widetilde{n'}\rangle$ FMM algorithm with the coefficients $[\![U', V', W']\!]$, and A_i, B_j, C_p are the submatrices of A, B and C, with a single index in two-level recursive block storage indexing. Then it can be verified that $C := C + AB$ is computed by,

for $r = 0, ..., R \cdot R' - 1$,

$$M_r := \left(\sum_{i=0}^{\widetilde{m}\widetilde{k}\cdot\widetilde{m}'\widetilde{k}'-1} (U \otimes U')_{i,r} A_i \right) \times \left(\sum_{j=0}^{\widetilde{k}\widetilde{n}\cdot\widetilde{k}'\widetilde{n}'-1} (V \otimes V')_{j,r} B_j \right);$$
$$C_p \mathrel{+}= (W \otimes W')_{p,r} M_r (p = 0, ..., \widetilde{m}\widetilde{n} \cdot \widetilde{m}'\widetilde{n}' - 1)$$

where $\otimes$ represents Kronecker Product. Note that $U \otimes U'$, $V \otimes V'$, and $W \otimes W'$ are $(\widetilde{m}\widetilde{k} \cdot \widetilde{m}'\widetilde{k}') \times (R \cdot R')$, $(\widetilde{k}\widetilde{n} \cdot \widetilde{k}'\widetilde{n}') \times (R \cdot R')$, $(\widetilde{m}\widetilde{n} \cdot \widetilde{m}'\widetilde{n}') \times (R \cdot R')$ matrices, respectively.

The set of coefficients of a two-level $\langle \widetilde{m}, \widetilde{k}, \widetilde{n}\rangle$ and $\langle \widetilde{m'}, \widetilde{k'}, \widetilde{n'}\rangle$ FMM algorithm can be denoted as $[\![U \otimes U', V \otimes V', W \otimes W']\!]$.

For example, the two-level STRASSEN is represented by the coefficients $[\![U \otimes U, V \otimes V, W \otimes W]\!]$ where $[\![U, V, W]\!]$ are the one-level STRASSEN coefficients given in (4).

E. Additional levels of FMM

Comparing one-level and two-level FMM, the same skeleton pattern emerges. The formula for defining L-level FMM is given by,

for $r = 0, ..., \prod_{l=0}^{L-1} R_l - 1$,

$$M_r := \left(\sum_{i=0}^{\prod_{l=0}^{L-1} \widetilde{m}_l\widetilde{k}_l-1} (\bigotimes_{l=0}^{L-1} U_l)_{i,r} A_i \right) \times \left(\sum_{j=0}^{\prod_{l=0}^{L-1} \widetilde{k}_l\widetilde{n}_l-1} (\bigotimes_{l=0}^{L-1} V_l)_{j,r} B_j \right);$$
$$C_p \mathrel{+}= (\bigotimes_{l=0}^{L-1} W_l)_{p,r} M_r (p = 0, ..., \prod_{l=0}^{L-1} \widetilde{m}_l\widetilde{n}_l - 1) \quad (5)$$

The set of coefficients of an L-level $\langle \widetilde{m_l}, \widetilde{k_l}, \widetilde{n_l}\rangle$ $(l = 0, 1, ..., L-1)$ FMM algorithm can be denoted as $[\![\bigotimes_{l=0}^{L-1} U_l, \bigotimes_{l=0}^{L-1} V_l, \bigotimes_{l=0}^{L-1} W_l]\!]$.

IV. IMPLEMENTATION AND ANALYSIS

The last section shows that families of one-level FMM algorithms can be specified by $\langle \widetilde{m}, \widetilde{k}, \widetilde{n}\rangle$ and $[\![U, V, W]\!]$. It also shows how the Kronecker product can be used to generate multi-level FMM algorithms that are iterative rather than recursive. In this section, we discuss a code generator that takes as input $\langle \widetilde{m}, \widetilde{k}, \widetilde{n}\rangle$ and $[\![U, V, W]\!]$ and as output generates implementations that build upon the primitives that combine taking linear combinations of matrices with the packing routines and/or micro-kernels that underlie BLIS. The code generator also provides a model of cost for each implementation that can then be used to choose the best FMM for a matrix of given size and shape. This code generator can thus generate code for arbitrary levels of FMM that can use different FMM choices at each level. In this way, we have generated and compared more than 200 FMM algorithms.

A. Code generation

Our code generator generates various implementations of FMM, based on the coefficient representation $[\![U, V, W]\!]$, levels of recursion, and packing routine/micro-kernel incorporation specifications.

There are two stages for our code generator: generating the skeleton framework, and generating the typical operations given in (3).

Generating the skeleton framework: During this stage, the code generator

- Computes the Kronecker Product of the coefficient matrices $[\![U_l, V_l, W_l]\!]$ in each level l to get the new coefficients $[\![\bigotimes_{l=0}^{L-1} U_l, \bigotimes_{l=0}^{L-1} V_l, \bigotimes_{l=0}^{L-1} W_l]\!]$.
- Generates the matrix partition code by conceptual recursive block storage indexing with $\langle \widetilde{m_l}, \widetilde{k_l}, \widetilde{n_l}\rangle$ partition dimensions for each level.
- For the general cases where one or more dimensions are not multiples of corresponding $\prod_{l=0}^{L-1} \widetilde{m_l}$, $\prod_{l=0}^{L-1} \widetilde{k_l}$, $\prod_{l=0}^{L-1} \widetilde{n_l}$, it generates dynamic peeling [16] code to handle the remaining "fringes" after invoking FMM, which requires no additional memory.

τ_a	Time (in sec.) of one arithmetic (floating point), operation, reciprocal of theoretical peak GFLOPS.
τ_b	(Bandwidth) Amortized time (in sec.) of 8 Bytes contigous data movement from DRAM to cache.
T	Total execution time (in sec.).
T_a	Time for arithmetic operations (in sec.).
T_m	Time for memory operations (in sec.).
$T_a^{\times}$	T_a for submatrix multiplications.
T_a^{A+}, T_a^{B+}, T_a^{C+}	T_a for extra submatrix additions.
$T_m^{A\times}$, $T_m^{B\times}$	T_m for reading submatrices in packing routines (Fig. 1).
$T_m^{\widetilde{A}\times}$,$T_m^{\widetilde{B}\times}$	T_m for writing submatrices in packing routines (Fig. 1).
$T_m^{C\times}$	T_m for reading *and* writing submatrices in micro-kernel (Fig. 1).
T_m^{A+}, T_m^{B+}, T_m^{C+}	T_m for reading *or* writing submatrices, related to the temporary buffer as part of **Naive FMM** and **AB FMM**.
$nnz(X)$	non-zero entry number in matrix or vector X.

Figure 4. Notation table for performance model.

Generating the typical operations: To generate the code for the typical operations in (3), the code generator

- Generates packing routines (written in C), that sum a list of submatrices of A integrated into the packing routine, yielding $\widetilde{A}_i$, and similarly sum a list of submatrices of B integrated into the packing routine, yielding $\widetilde{B}_p$, extending what is illustrated in Figure 1.
- Assembles a specialized micro-kernel comprised of a hand-coded GEMM kernel and automatically generated updates to multiple submatrices of C.

Further variations: In [2], a number of variations on the theme illustrated in Figure 1 (right) are discussed:

- **Naive FMM**: A classical implementation with temporary buffers for storing the sum of A, B, and the intermediate matrix product M_r.
- **AB FMM**: The packing routines incorporate the summation of submatrices of A, B into the packing of buffers $\widetilde{A}_i$ and $\widetilde{B}_p$ but explicit temporary buffers for matrices M_r are used.
- **ABC FMM**: **AB FMM**, but with a specialized micro-kernel that incorporates addition of M_r to multiple submatrices of C.

Incorporating the generation of these variations into the code generator yields over 200 FMM implementations.

B. Performance model

In [2], a performance model was given to estimate the execution time T for the one-level/two-level ABC, AB, and Naive variations of $\langle 2,2,2\rangle$ STRASSEN. In this subsection, we generalize that performance model to predict the execution time T for the various FMM implementations generated by our code generator. Theoretical estimation helps us better understand the computation and memory footprint of different FMM implementations, and allows us to avoid exhaustive empirical search when searching for the best implementation for different problem sizes and shapes. Most importantly, our code generator can embed our performance model to guide the selection of a FMM implementation as a function of problem size and shape, with the input $\langle \widetilde{m}_l, \widetilde{k}_l, \widetilde{n}_l\rangle$ and $[\![U_l, V_l, W_l]\!]$ specifications on each level l. These performance models are themselves automatically generated.

①	*Effective* GFLOPS $= 2\cdot m\cdot n\cdot k/T\cdot 10^{-9}$
②	$T = T_a + T_m$
③	$T_a = N_a^{\times}\cdot T_a^{\times} + N_a^{A+}\cdot T_a^{A+} + N_a^{B+}\cdot T_a^{B+} + N_a^{C+}\cdot T_a^{C+}$
④	$T_m = N_m^{A\times}\cdot T_m^{A\times} + N_m^{B\times}\cdot T_m^{B\times} + N_m^{C\times}\cdot T_m^{C\times} + N_m^{A+}\cdot T_m^{A+} + N_m^{B+}\cdot T_m^{B+} + N_m^{C+}\cdot T_m^{C+}$

	type	τ	GEMM	L-level
$T_a^{\times}$	-	τ_a	$2mnk$	$2\frac{m}{\widetilde{M_L}}\frac{n}{\widetilde{N_L}}\frac{k}{\widetilde{K_L}}$
T_a^{A+}	-	τ_a	-	$2\frac{m}{\widetilde{M_L}}\frac{k}{\widetilde{K_L}}$
T_a^{B+}	-	τ_a	-	$2\frac{k}{\widetilde{K_L}}\frac{n}{\widetilde{N_L}}$
T_a^{C+}	-	τ_a	-	$2\frac{m}{\widetilde{M_L}}\frac{n}{\widetilde{N_L}}$
$T_m^{A\times}$	r	τ_b	$mk\lceil\frac{n}{n_c}\rceil$	$\frac{m}{\widetilde{M_L}}\frac{k}{\widetilde{K_L}}\lceil\frac{n/\widetilde{N_L}}{n_c}\rceil$
$T_m^{\widetilde{A}\times}$	w	τ_b	$mk\lceil\frac{n}{n_c}\rceil$	$\frac{m}{\widetilde{M_L}}\frac{k}{\widetilde{K_L}}\lceil\frac{n/\widetilde{N_L}}{n_c}\rceil$
$T_m^{B\times}$	r	τ_b	nk	$\frac{n}{\widetilde{N_L}}\frac{k}{\widetilde{K_L}}$
$T_m^{\widetilde{B}\times}$	w	τ_b	nk	$\frac{n}{\widetilde{N_L}}\frac{k}{\widetilde{K_L}}$
$T_m^{C\times}$	r/w	τ_b	$2\lambda mn\lceil\frac{k}{k_c}\rceil$	$2\lambda\frac{m}{\widetilde{M_L}}\frac{n}{\widetilde{N_L}}\lceil\frac{k/\widetilde{K_L}}{k_c}\rceil$
T_m^{A+}	r/w	τ_b	mk	$\frac{m}{\widetilde{M_L}}\frac{k}{\widetilde{K_L}}$
T_m^{B+}	r/w	τ_b	nk	$\frac{n}{\widetilde{N_L}}\frac{k}{\widetilde{K_L}}$
T_m^{C+}	r/w	τ_b	mn	$\frac{m}{\widetilde{M_L}}\frac{n}{\widetilde{N_L}}$

	GEMM	L-level		
		ABC	AB	Naive
$N_a^{\times}$	1	R_L	R_L	R_L
N_a^{A+}	-	$nnz(\bigotimes U)$-R_L	$nnz(\bigotimes U)$-R_L	$nnz(\bigotimes U)$-R_L
N_a^{B+}	-	$nnz(\bigotimes V)$-R_L	$nnz(\bigotimes V)$-R_L	$nnz(\bigotimes V)$-R_L
N_a^{C+}	-	$nnz(\bigotimes W)$	$nnz(\bigotimes W)$	$nnz(\bigotimes W)$
$N_m^{A\times}$	1	$nnz(\bigotimes U)$	$nnz(\bigotimes U)$	R_L
$N_m^{\widetilde{A}\times}$	-	-	-	-
$N_m^{B\times}$	1	$nnz(\bigotimes V)$	$nnz(\bigotimes V)$	R_L
$N_m^{\widetilde{B}\times}$	-	-	-	-
$N_m^{C\times}$	1	$nnz(\bigotimes W)$	R_L	R_L
N_m^{A+}	-	-	-	$nnz(\bigotimes U)$+R_L
N_m^{B+}	-	-	-	$nnz(\bigotimes V)$+R_L
N_m^{C+}	-	-	$3nnz(\bigotimes W)$	$3nnz(\bigotimes W)$

Figure 5. The top table shows the equations for computing the execution time T and *Effective* GFLOPS in our performance model. The middle table shows the various components of arithmetic and memory operations for BLAS GEMM and various implementations of FMM. The time shown in the first column for GEMM and L-level FMM can be computed separately by multiplying the parameter in τ column with the arithmetic/memory operation number in the corresponding entries. The bottom table shows the coefficient N_a^X/N_m^X mapping table for computing T_a/T_m in the performance model. Here $\widetilde{M_L} = \prod_{l=0}^{L-1}\widetilde{m_l}$, $\widetilde{K_L} = \prod_{l=0}^{L-1}\widetilde{k_l}$, $\widetilde{N_L} = \prod_{l=0}^{L-1}\widetilde{n_l}$, $\bigotimes U = \bigotimes_{l=0}^{L-1} U_l$, $\bigotimes V = \bigotimes_{l=0}^{L-1} V_l$, $\bigotimes W = \bigotimes_{l=0}^{L-1} W_l$, $R_L = \prod_{l=0}^{L-1} R_l$.

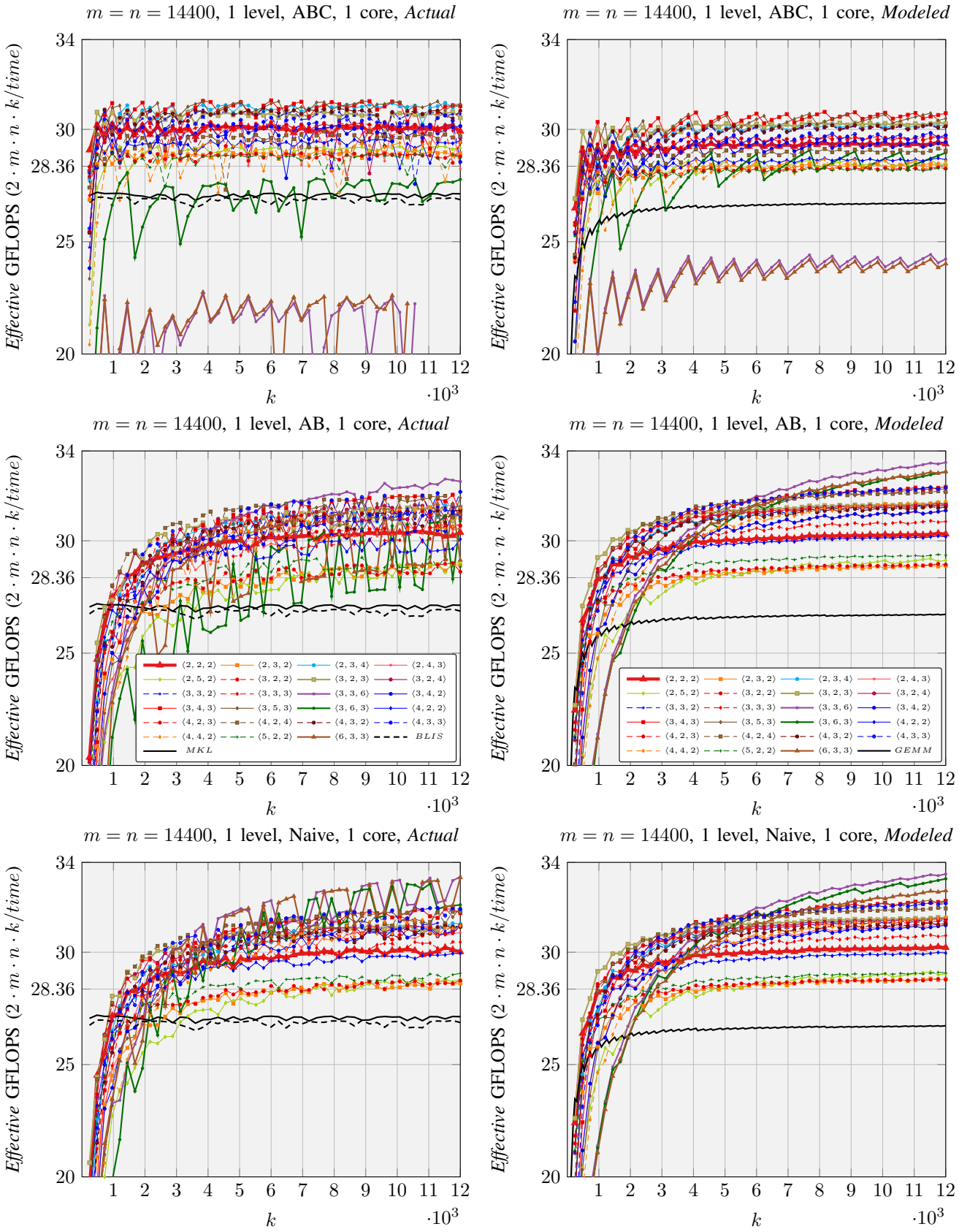

Figure 6. Performance of generated one-level ABC, AB, Naive FMM implementations on single core when m=n=14400, k varies. Left column: actual performance; Right column: modeled performance. Top row: one-level, ABC; Middle row: one-level, AB; Bottom row: one-level, Naive.

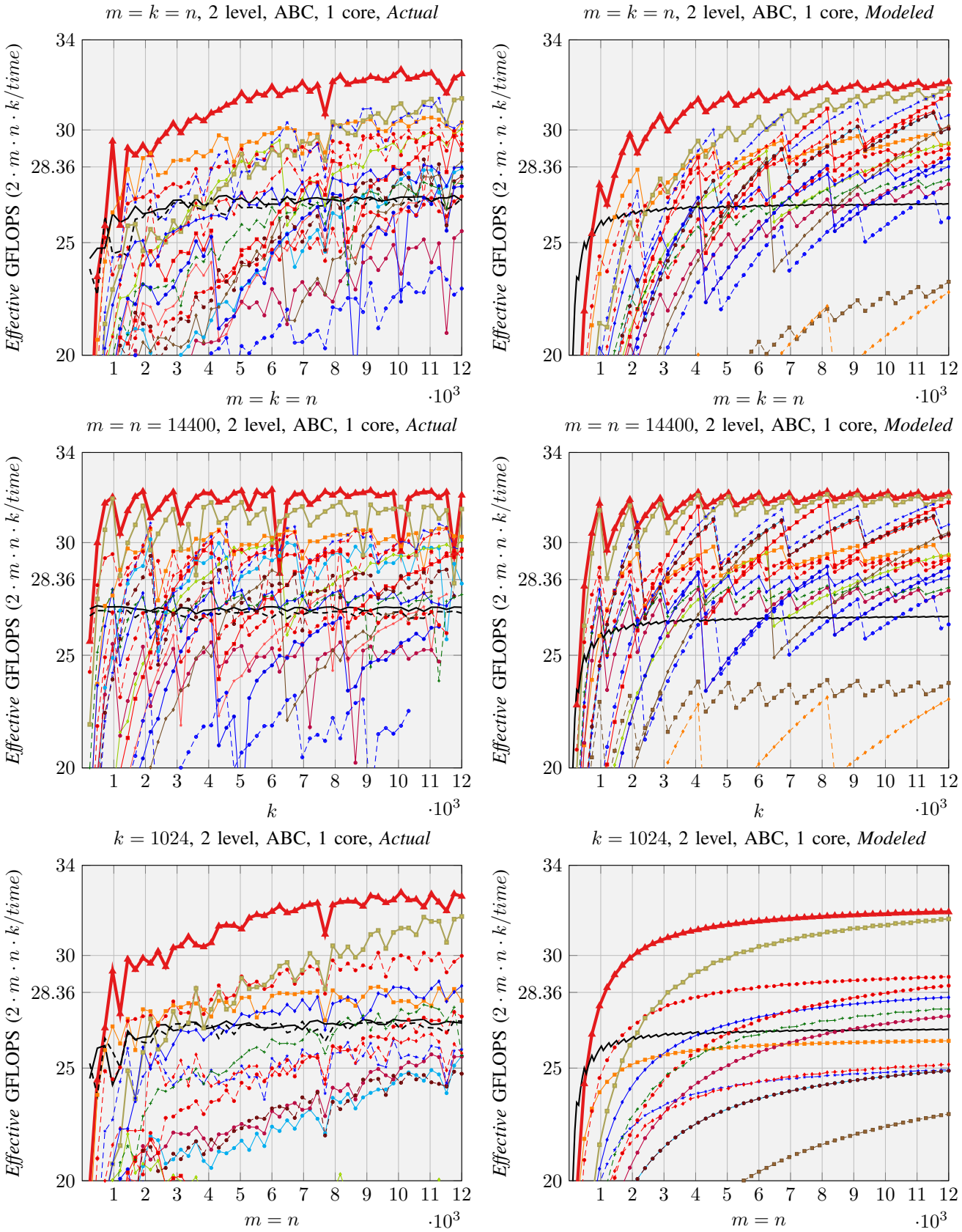

Figure 7. Performance of generated two-level ABC FMM implementations on single core when m=k=n; m=n=14400, k varies; k=1024, m=n vary. Left column: actual performance; Right column: modeled performance. Top row: m=k=n; Middle row: m=n=14400, k varies; Bottom row: k=1024, m=n vary.

Assumption: We have similar architecture assumptions as in [2]. Basically we assume that the architecture has two layers of modern memory hierarchy: fast caches and relatively slow main memory (DRAM). For read operations, the latency for accessing cache can be ignored, while the latency for accessing the main memory is counted; For write operations, we assume a lazy write-back policy such that the time for writing into fast caches can be hidden. Based on these assumptions, the memory operations for GEMM and various implementations of FMM are decomposed into three parts:

- memory packing shown in Figure 1.
- reading/writing the submatrices of C in Figure 1.
- reading/writing of the temporary buffer that are parts of **Naive FMM/AB FMM**.

Notation: Notation is summarized in Figure 4.

The total execution time, T, is dominated by arithmetic time T_a and memory time T_m (② in Figure 5).

Arithmetic operations: T_a is decomposed into submatrix multiplications ($T_a^{\times}$) and submatrix additions (T_a^{A+}, T_a^{B+}, T_a^{C+}) (③ in Figure 5). T_a^{X+} has a coefficient 2 because under the hood the matrix additions are cast into FMA operations. The corresponding coefficients N_a^X are tabulated in Figure 5. For instance, $N_a^{A+} = nnz(\bigotimes U) - R_L$ for L-level FMM, because computing $\sum((\bigotimes U)_{i,r}A_i)$ in (5) involves $\sum_{r=0}^{R_L-1}(nnz((\bigotimes U)_{:,r}) - 1) = nnz(\bigotimes U) - R_L$ submatrix additions. Note that $X_{:,r}$ denotes the rth column of X.

Memory operations: T_m is a function of the submatrix sizes $\{m/\widetilde{M_L}, k/\widetilde{K_L}, n/\widetilde{N_L}\}$, and the block sizes $\{m_C, k_C, n_C\}$ in Figure 1(right), because the memory operation can repeat multiple times according to which loop they reside in. T_m is broken down into several components, as shown in ④ in Figure 5. Each memory operation term is characterized in Figure 5 by its read/write type and the amount of memory in units of 64-bit double precision elements. Note that $T_m^{\widetilde{A}\times}$,$T_m^{\widetilde{B}\times}$ are omitted in ④ because of the assumption of lazy write-back policy with fast caches. Due to the software prefetching effects, $T_m^{C\times}=2\lambda\frac{m}{\widetilde{M_L}}\frac{n}{\widetilde{N_L}}\lceil\frac{k/\widetilde{K_L}}{k_c}\rceil\tau_b$ has an additional parameter $\lambda \in [0.5, 1]$, which denotes the prefetching efficiency. λ is adapted to match GEMM performance. Note that this is a ceiling function proportional to k, because rank-k updates for accumulating submatrices of C recur $\lceil\frac{k/\widetilde{K_L}}{k_c}\rceil$ times in 4th loop in Figure 1. The corresponding coefficients N_m^X are tabulated in Figure 5. For example, for **Naive FMM** and **AB FMM**, computing $C_p+=(\bigotimes W)_{p,r}M_r(p=0,...)$ in (5) involves 2 read and 1 write related to temporary buffer in slow memory. Therefore, $N_m^{C\times} = 3nnz(\bigotimes W)$.

C. Discussion

We can make estimations about the run time performance of the various FMM implementations generated by our code generator, based on the analysis shown in Figure 5. We

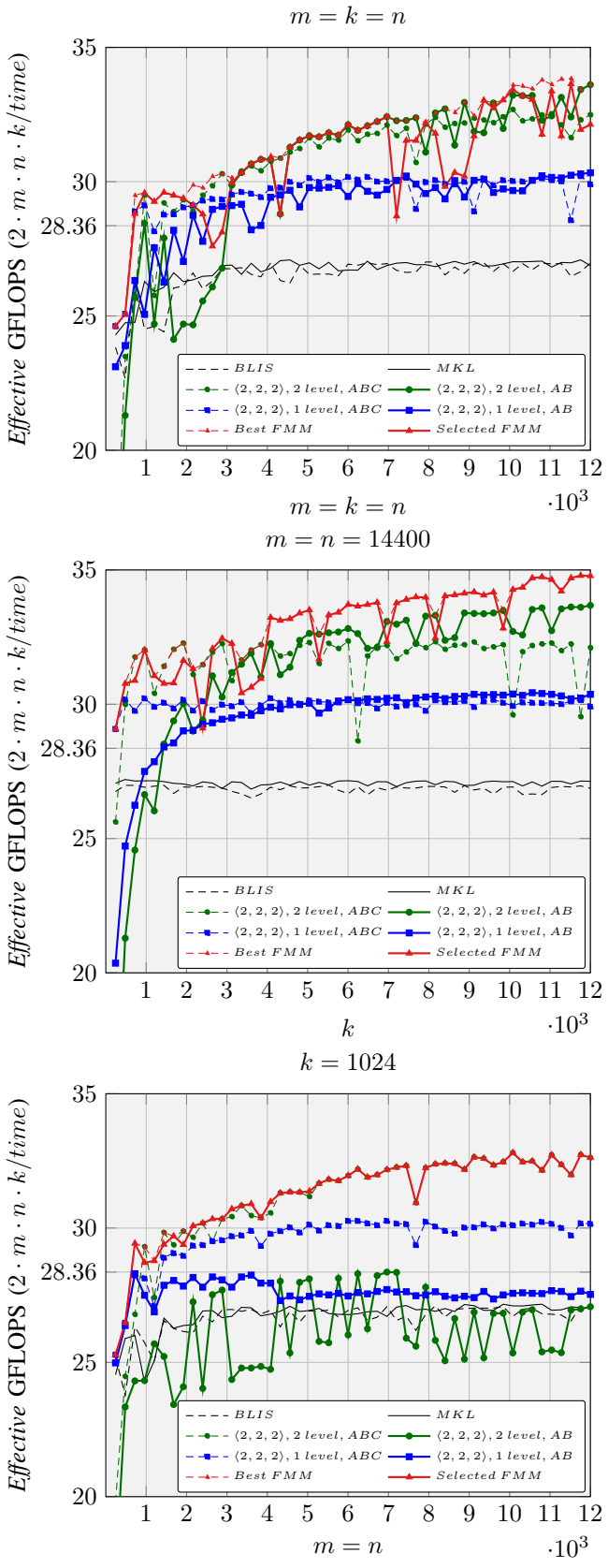

Figure 8. Selecting FMM implementation with performance model.

use *Effective* GFLOPS (defined in ① in Figure 5) as the metric to compare the performance of these various FMM implementations, similar to [1], [17], [18]. The architecture-dependent parameters for the model are given in Section V. We demonstrate the performance of two representative groups of experiments in Figures 6 and 7.

- Contrary to what was observed in [2], **Naive FMM** may perform better than **ABC FMM** and **AB FMM** for relatively large problem size. For example, in Figure 6, $\langle 3,6,3\rangle$ (with the maximum theoretical speedup among all FMMs we test, Figure 2) has better **Naive FMM** performance than **ABC FMM** and **AB FMM**. This is because the total number of times for packing in $\langle 3,6,3\rangle$ is very large ($N_m^{A\times} = nnz(\bigotimes U)$, $N_m^{B\times} = nnz(\bigotimes V)$). This magnifies the overhead for packing with **AB FMM/ABC FMM**.
- Contrary to what was observed in [1], for rank-k updates (middle column, right column, Figure 7), $\langle 2,2,2\rangle$ still performs the best with **ABC FMM** implementations ([1] observe some other shapes, e.g. $\langle 4,2,4\rangle$, tend to have higher performance). This is because their implementations are similar to **Naive FMM**, with the overhead for forming the M_r matrices explicitly.
- Figure 6 shows that for small problem size, when k is small, **ABC FMM** performs best; when k is large, **AB FMM/Naive FMM** perform better. That can be quantitatively explained by comparing the coefficients of N_m^X in the bottom table in Figure 5.
- The graph for $m = n = 14400$, k varies, ABC, 1core (left column, Figure 6; middle column, Figure 7) shows that for k equal to the appropriate multiple of k_C ($k = \prod_{l=0}^{L-1} k_l \times k_C$), **ABC FMM** achieves the best performance.

D. Apply performance model to code generator

For actual performance, even the best implementation has some unexpected drops, due to the "fringes" which are caused by the problem sizes not being divisible by partition dimesions $\widetilde{m}$, $\widetilde{k}$, $\widetilde{n}$. This is not captured by our performance model. Therefore, given the specific problem size and shape, we choose the best two implementations predicted by our performance model as the top two candidate implementations, and then measure the performance in practice to pick the best one.

In Figure 8 we show the performance results on single core by selecting the generated FMM implementation with the guide of performance model, when m=k=n; m=n= 14400, k varies; and k=1024, m=n vary.

Overall this experiment shows that the performance model is accurate enough in terms of relative performance between various FMM implementations to guide the choice of a FMM implementation, with the problem sizes and shapes as the inputs. That will reduce the potential overhead of exhaustive empirical search.

V. Performance Experiments

We present performance evaluations for various generated FMM implementations.

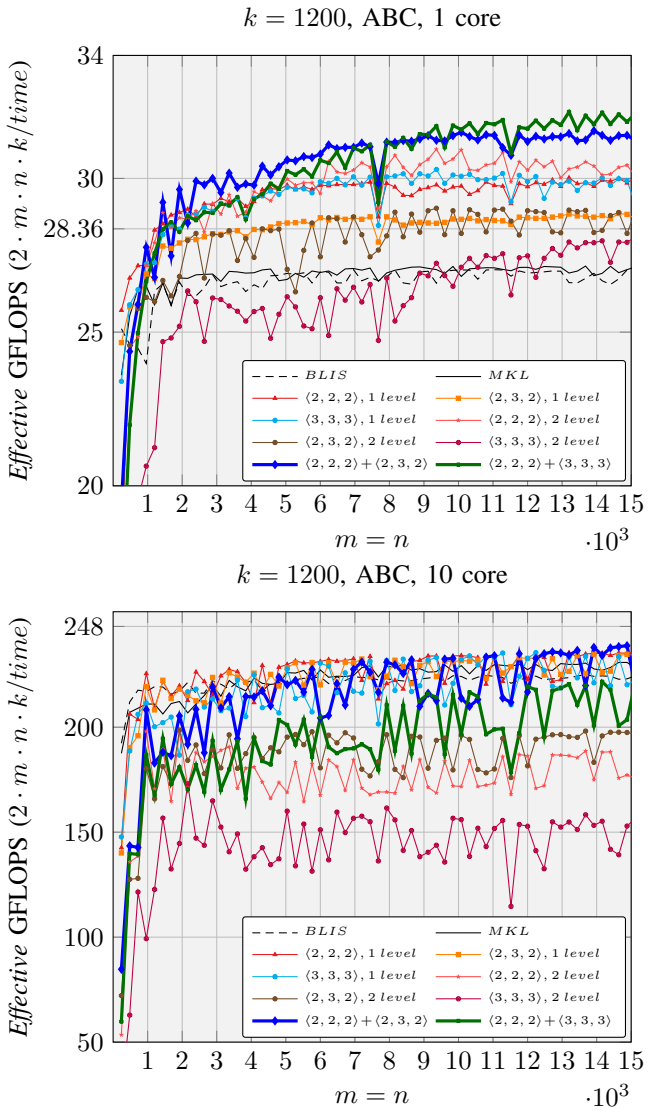

Figure 9. Benefit of hybrid partitions over other partitions.

A. Implementation and architecture information

The FMM implementations generated by our code generator are written in `C`, utilizing `SSE2` and `AVX` assembly, compiled with the Intel `C` compiler version 15.0.3 with optimization flag `-O3 -mavx`.

We compare against our generated DGEMM (based on the packing routines and micro-kernel borrowed from BLIS, marked as BLIS in the performance figures) as well as Intel MKL's DGEMM [19] (marked as MKL in the performance figures).

We measure performance on a dual-socket Intel Xeon E5-2680 v2 (Ivy Bridge, 10 cores/socket) processor with 12.8

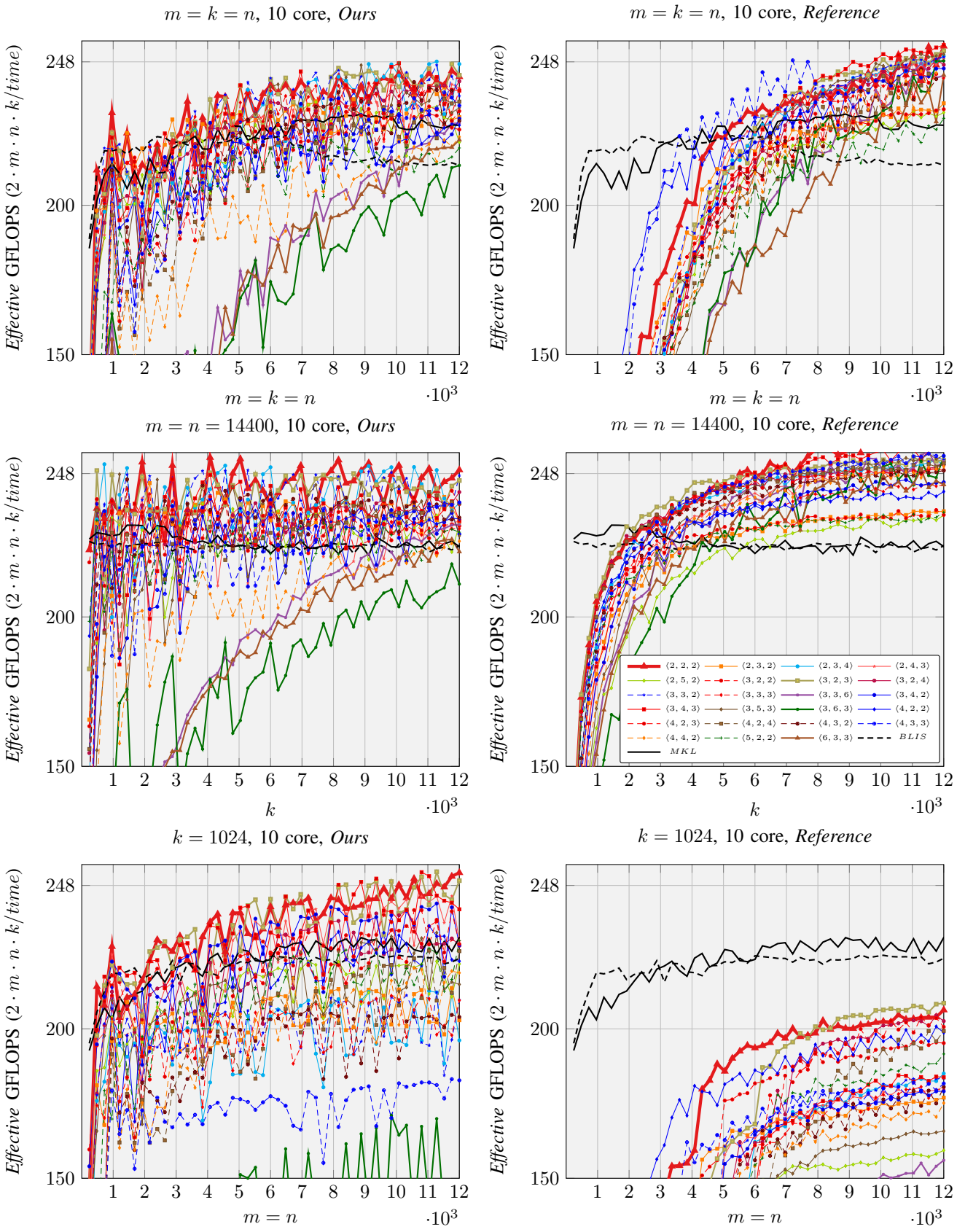

Figure 10. Performance of the best implementation of our generated FMM code and reference implementations [1] on one socket (10 core). Top row: our implementations; Bottom row: reference implementations from [1] (linked with Intel MKL). Left column: m=k=n; Middle column: m=n=14400, k varies; Right column: k=1024, m=n vary.

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 ten 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 disable hyper-threading explicitly and use `KMP_AFFINITY=compact`.

The blocking parameters, $n_R = 4$, $m_R = 8$, $k_C = 256$, $n_C = 4096$ and $m_C = 96$, are consistent with parameters used for the standard BLIS DGEMM implementation for this architecture. This makes the size of the packing buffer $\widetilde{A}_i$ 192 KB and $\widetilde{B}_p$ 8192 KB, which then fit the L2 cache and L3 cache, respectively.

Parallelization is implemented mirroring that described in [20], using OpenMP directives that parallelize the third loop around the micro-kernel in Figure 1.

B. Benefit of hybrid partitions

First, we demonstrate the benefit of using different FMM algorithms for each level.

We report the performance of different combinations of one-level/two-level $\langle 2,2,2\rangle$, $\langle 2,3,2\rangle$, and $\langle 3,3,3\rangle$ in Figure 9, when k is fixed to 1200 and $m = n$ vary. As suggested and illustrated in Section IV-C, **ABC FMM** performs best for rank-k updates, which is why we only show the **ABC FMM** performance.

Overall the hybrid partitions $\langle 2,2,2\rangle$ + $\langle 2,3,2\rangle$ and $\langle 2,2,2\rangle$ + $\langle 3,3,3\rangle$ achieve the best performance. This is because 1200 is close to $2\times 3\times k_C$, meaning that the hybrid partitions of 2 and 3 on the k dimension are more favorable. This is consistent with what the performance model predicts. Performance benefits are less for 10 cores due to bandwidth limitations, although performance of hybrid partitions still beats two-level homogeneous partitions.

This experiment shows the benefit of hybrid partitions, facilitated by the Kronecker product representation.

C. Sequential and parallel performance

Results when using a single core are presented in Figures 2, 6, and 7. Our generated **ABC FMM** implementation outperforms **AB FMM/Naive FMM** and reference implementations from [1] for rank-k updates (when k is small). For very large square matrices, our generated **AB FMM** or **Naive FMM** can achieve competitive performance with reference implementations [1] that is linked with Intel MKL. These experiments support the validity of our model.

Figure 10 reports performance results for ten cores within the same socket. Memory bandwidth contention impacts the performance of various FMM when using many cores. Nonetheless we still observe the speedup of FMM over GEMM. For smaller matrices and special shapes such as rank-k updates, our generated implementations achieve better performance than reference implementations [1].

VI. Conclusion

We have discussed a code generator framework that can automatically implement families of FMM algorithms for Strassen-like fast matrix multiplication algorithms. This code generator expresses the composition of multi-level FMM algorithms as Kronecker products. It incorporates the matrix summations that must be performed for FMM into the inherent packing and micro-kernel operations inside GEMM, avoiding extra workspace requirement and reducing the overhead of memory movement. Importantly, it generates an accurate performance model to guide the selection of a FMM implementation as a function of problem size and shape, facilitating the creation of poly-algorithms that select the best algorithm for a problem size. Comparing with state-of-the-art results, we observe a significant performance improvement for smaller matrices and special matrix multiplication shapes such as rank-k updates, without the need for exhaustive empirical search.

There are a number of avenues for future work:

- Task parallelism and various parallel schemes are proposed in the recent literature [21], [1]. We need to pursue how our techniques compare to these and how to combine these with our advances. It may be possible to utilize our performance model for task scheduling.
- Finding the new FMM algorithms by searching the coefficient matrix $[\![U,V,W]\!]$ is an NP-hard problem [22]. It may be possible to prune branches with the performance model as the cost function during the search process.
- In [2], it is shown that Intel Xeon Phi coprocessor (KNC) can benefit from ABC variation of STRASSEN. It may be possible to get performance benefit by porting our code generator to generate variations of FMM implementations for many-core architecture such as second-generation Intel Xeon Phi coprocessor (KNL).
- The asymptotic communication lower bound for Strassen's algorithm and matrix multiplication has been characterized in [23], [24], [25], [26]. It may be possible to apply our performance model to constrain the coefficients of the cubic and quadratic terms and get more precise lower bound for specific architectures.

Acknowledgments

This work was sponsored in part by the National Science Foundation under grant number ACI-1550493, by Intel Corporation through an Intel Parallel Computing Center grant, and by a gift from Qualcomm Foundation. Access to the Maverick supercomputers administered by TACC is gratefully acknowledged. Devin A. Matthews is an Arnold O. Beckman Postdoctoral Fellow. We thank Austin Benson and Grey Ballard for sharing their open source code and helpful discussions, the anonymous reviewers for their constructive comments, and the rest of the SHPC team (`http://shpc.ices.utexas.edu`) for their supports.

REFERENCES

[1] 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 (PPoPP 2015)*. ACM, 2015, pp. 42–53.

[2] J. Huang, T. M. Smith, G. M. Henry, and R. A. van de Geijn, "Strassen's algorithm reloaded," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 16)*. IEEE, 2016, pp. 59:1–59:12.

[3] 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.

[4] C.-H. Huang, J. R. Johnson, and R. W. Johnson, "A tensor product formulation of Strassen's matrix multiplication algorithm," *Applied Mathematics Letters*, vol. 3, no. 3, pp. 67–71, 1990.

[5] J. Li, A. Skjellum, and R. D. Falgout, "A poly-algorithm for parallel dense matrix multiplication on two-dimensional process grid topologies," *Concurrency: Practice and Experience*, vol. 9, no. 5, pp. 345–389, May 1997.

[6] 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.

[7] T. M. Low, F. D. Igual, T. M. Smith, and E. S. Quintana-Orti, "Analytical modeling is enough for high-performance BLIS," *ACM Trans. Math. Softw.*, vol. 43, no. 2, pp. 12:1–12:18, August 2016.

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

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

[10] G. Ballard, A. R. Benson, A. Druinsky, B. Lipshitz, and O. Schwartz, "Improving the numerical stability of fast matrix multiplication," *SIAM Journal on Matrix Analysis and Applications*, vol. 37, no. 4, pp. 1382–1418, 2016.

[11] 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.

[12] A. Graham, "Kronecker products and matrix calculus: With applications." *John Wiley & Sons, Inc., 605 3rd Ave., New York, NY 10158*, 1982.

[13] V. Strassen, "Gaussian elimination is not optimal," *Numerische Mathematik*, vol. 13, no. 4, pp. 354–356, August 1969.

[14] 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.

[15] E. Elmroth, F. Gustavson, I. Jonsson, and B. Kågström, "Recursive blocked algorithms and hybrid data structures for dense matrix library software," *SIAM review*, vol. 46, no. 1, pp. 3–45, 2004.

[16] M. Thottethodi, S. Chatterjee, and A. R. Lebeck, "Tuning Strassen's matrix multiplication for memory efficiency," in *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing (SC 98)*. IEEE, 1998, pp. 1–14.

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

[18] 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 (SC 12)*. IEEE, 2012, pp. 101:1–101:11.

[19] "Intel MKL," https://software.intel.com/en-us/intel-mkl.

[20] 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.

[21] 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.

[22] D. E. Knuth, *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[23] D. Irony, S. Toledo, and A. Tiskin, "Communication lower bounds for distributed-memory matrix multiplication," *Journal of Parallel and Distributed Computing*, vol. 64, no. 9, pp. 1017–1026, 2004.

[24] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz, "Communication-optimal parallel algorithm for Strassen's matrix multiplication," in *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 12)*. ACM, 2012, pp. 193–204.

[25] J. Scott, O. Holtz, and O. Schwartz, "Matrix multiplication I/O-complexity by path routing," in *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 15)*. ACM, 2015, pp. 35–45.

[26] G. Bilardi and L. De Stefani, "The I/O complexity of Strassen's matrix multiplication with recomputation," *arXiv preprint arXiv:1605.02224*, 2016.