The Dissertation Committee for Jianyu Huang
certifies that this is the approved version of the following dissertation:

Practical Fast Matrix Multiplication Algorithms

Committee:

Robert van de Geijn, Supervisor

Don Batory

George Biros

Greg Henry

Keshav Pingali

Practical Fast Matrix Multiplication Algorithms

by

Jianyu Huang

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2018

Dedicated to my family for their love and support.

Acknowledgments

"Ever tried. Ever failed. No matter. Try again. Fail again. Fail better."

— Samuel Beckett

My five-year Ph.D. adventure at UT Austin was an amalgam of frustration, happiness, disappointment, excitement, anxiety, tranquility, and many other emotions. But it was a priceless treasure for me to experience this unforgettable journey, which further teaches me to be brave and persistent enough to overcome any challenges going forward in life. I was especially fortunate to work with incredible people full of inspiration and innovative spirit.

This research has been funded by the following sources: NSF Award CCF-1714091: SHF: Small: Making Strassen's Algorithm Practical, NSF Award ACI-1550493: Collaborative Research: SI2-SSI: Sustaining Innovation in the Linear Algebra Software Stack for Computational Chemistry and other Sciences, NSF Award ACI-1148125: Collaborative Research: SI2-SSI: A Linear Algebra Software Infrastructure for Sustained Innovation in Computational Chemistry and other Sciences, NSF Award CCF-1218483: SHF: Small: Algorithm/Architecture Co-Design for Low Power and High Performance Linear Algebra Fabrics, an Intel Parallel Computing Center grant, a gift from Qualcomm, the Microelectronics and Computer Development Fellowship, and Graduate School Summer Fellowship from UT Austin. The author acknowledges the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources that have contributed to the research results reported within this dissertation. The author also gratefully acknowledges Microway, Inc. for providing access to their Nvidia Tesla GPU-accelerated compute cluster.

First and foremost, I owe my deepest gratitude to my advisor, Prof. Robert van de Geijn. Without his continuous optimism, enthusiasm, encouragement, confidence, and support, it would have been impossible for me to complete this five-year journey. Despite the supervisor-student hierarchy, Robert always respected and trusted me, and allowed me to explore wild research problems as if we were colleagues, while still guiding me in refining new ideas by sharing his experience and insight. He, along with Dr. Maggie Myers, whom I also want to thank deeply, appreciated my efforts regardless of whether my research was a success or failure, treated my mistakes as a learning experience, and supported me and my decisions whenever needed, all of which made me feel as if I were their own child. This kind of relationship made my whole journey much more joyful.

I would also like to thank other members of my thesis committee: Prof. Don Batory, Prof. George Biros, Dr. Greg Henry, and Prof. Keshav Pingali. Their sharp insights and constructive feedback were absolutely invaluable to improving my thesis. Furthermore, Prof. George Biros taught me numerical linear algebra and parallel computing, and Prof. Don Batory taught me automatic software design; both laid the foundation for my research with high-performance computing software. Dr. Greg Henry has collaborated with me in my first successful research project and offered me many pieces of advice. Prof. Don Batory and Prof. Keshav Pingali have served on my committee since my research preparation exams.

I am also deeply grateful to my friends and colleagues in the SHPC group: Field Van Zee, Tyler Smith, Devin Matthews, Chenhan Yu, Devangi

Parikh, Victor Eijkhout, Tze Meng Low, Kyungjoo Kim, Martin Schatz, Bryan Marker, Woody Austin, and Leslie Rice. Field and Tyler exposed me to the details of the BLIS framework, and answered my questions with great patience. Chenhan is exceptionally smart and hard-working. He provided me with his insight and intuition whenever I had a difficult problem. Devin introduced me to tensors and facilitated the idea of applying my research for a higher-demensional space. Devangi led pedagogical efforts towards the education of high-performance computing, which gave my research a fresh perspective. My research is built on the shoulders of giants and the infrastructure developed by all members of the SHPC group.

In addition, I want to thank many other friends and fellow graduate students during the five-year journey: Xiaofan Lu, Zuocheng Ren, Bo Xiong, Xinyu Wang, Yuepeng Wang, Yuanzhong Xu, Yu Feng, Wenzhi Cui, Hongkun Yang, Xue Chen, Jia Chen, Mochamad Asri, Yinan Zhao, Ruohan Gao, Wenguang Mao, Jian He, Kostas Ferles, Zhuode Liu, Ruohan Zhang, Mei Wang, and Zhiting Zhu. My five-year journey has been so colorful and inspiring with the help and support from all of you!

I want to also thank Prof. Wenjun Hu for opening the door to research for me. Wenjun hired me to work as a research intern in Microsoft Research Asia, a world-class research institute in Beijing. She later moved to Yale, and continued to offer me great advice during my early Ph.D. career.

Last but not least, this work would not have been possible without the support of my family. I would especially like to thank my mother, Haiyun Sun, and my father, Chao Huang, for their unconditional love.

Jianyu Huang

The University of Texas at Austin
August 2018

Practical Fast Matrix Multiplication Algorithms

Publication No. _____

Jianyu Huang, Ph.D.
The University of Texas at Austin, 2018

Supervisor: Robert van de Geijn

Matrix multiplication is a core building block for numerous scientific computing and, more recently, machine learning applications. Strassen's algorithm, the original Fast Matrix Multiplication (FMM) algorithm, has long fascinated computer scientists due to its startling property of reducing the number of computations required for multiplying $n \times n$ matrices from $O(n^3)$ to $O(n^{2.807})$. Over the last half century, this has fueled many theoretical improvements such as other variations of Strassen-like FMM algorithms. Previous implementations of these FMM algorithms led to the "street wisdom" that they are only practical for large, relatively square matrices, that they require considerable workspace, and that they are difficult to achieve thread-level parallelism. The thesis of this work dispels these notions by demonstrating significant benefits for small and non-square matrices, requiring no workspace beyond what is already incorporated in high-performance implementations of matrix multiplication, and achieving performance benefits on multi-core, many-core, and distributed memory architectures.

Table of Contents

List of Figures

Chapter 1

Introduction

At the core of dense linear algebra lies matrix multiplication (GEMM), a fundamental primitive of great importance to numerous scientific disciplines such as machine learning [125, 18], numerical linear algebra [63, 39, 2, 43], graph analysis [66], and more. Considerable effort has been made to improve the performance of this task over the last decades.

Strassen's algorithm (STRASSEN) [115], the original Fast Matrix Multiplication (FMM)[1] algorithm, 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 $O(n^3)$ floating point operations required by a conventional matrix multiplication. It has led to many variants (other Strassen-like FMM algorithms) that improve upon this result [129, 12, 100, 104] as well as practical implementations [32, 59, 26, 9]. The method can yield a shorter execution time than the best conventional algorithm with a modest degradation in numerical stability [50, 28, 6] by only incorporating a few levels of the recursion that underlies the method.

[1]Fast matrix multiplication (FMM) algorithms are those matrix multiplication algorithms which perform asympototically fewer arithmetic operations than the classical algorithm. STRASSEN can be found in Section 3.1, and another example of FMM algorithms is given in Appendix C.

1.1 Motivation

From 30,000 feet, STRASSEN and similar Strassen-like FMM algorithms 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 GEMM provided by a high-performance implementation through the Basic Linear Algebra Subprograms (BLAS) [30] 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. These constraints expose the "street wisdom" for implementing STRASSEN and similar Strassen-like FMM algorithms:

- STRASSEN and similar Strasen-like FMM algorithms are only practical for very large matrices.
- For STRASSEN and similar Strassen-like FMM algorithms to be effective, the matrices being multiplied should be relatively square.
- STRASSEN and other Strassen-like FMM algorithms inherently require substantial workspace.
- A STRASSEN or other similar Strassen-like FMM algorithm interface must allow the caller to pass in workspace or allocate substantial workspace internally.

- It is hard to demonstrate speedup on multi-core architectures.

These limitations motivate us to work towards the theory and practice of the fast matrix multiplication algorithms. The goal of this dissertation is to explore the practical implementation of STRASSEN and Strassen-like FMM algorithms on various architectures to overcome these limitations.

1.2 Solution

An important recent advance in the high-performance implementation of GEMM is the BLAS-like Library Instantiation Software (BLIS framework) [124], a careful refactoring of the best-known approach to implementing conventional GEMM introduced by Goto [38]. A similar effort made for Nvidia GPUs is CUDA Templates for Linear Algebra Subroutines (CUTLASS) [67]. Of importance to the solution to the basic problem above are the building blocks that BLIS and CUTLASS expose, minor modifications of which support a new approach to implementating STRASSEN and similar Strassen-like FMM algorithms. This approach changes data movement between memory layers and can thus mitigate the negative impact of the additional lower-order terms incurred by STRASSEN and other Strassen-like FMM algorithms. These building blocks have similarly been exploited to improve upon the performance of, for example, the computation of the K-Nearest Neighbor [131, 97] and Tensor Contraction [85, 112] problem. The result is a family of STRASSEN and other Strassen-like FMM implementations, members of which attain superior performance depending on the sizes of the matrices.

The resulting family improves upon prior implementations of STRASSEN and Strassen-like FMM algorithms in a number of surprising ways:

- It can outperform classical GEMM even for small square matrices.
- It can achieve high performance for rank-k updates (GEMM with a small "inner matrix size"), a case of GEMM frequently encountered in the implementation of libraries like LAPACK [2].
- It requires no additional workspace beyond the small buffers that are already incorporated in BLIS and CUTLASS.
- It can incorporate directly the multi-threading in traditional GEMM implementations.
- It can be plug-compatible with the standard GEMM interface supported by the BLAS and cuBLAS.
- It can be incorporated into practical distributed memory implementations of GEMM.

Most of these advances run counter to "street wisdom."

1.3 Contributions

This disseration makes the following contributions:

- **It dispels the conventional notions regarding implementing** STRASSEN **and similar Strassen-like FMM algorithms,** outperforming the conventional matrix multiplication for small and non-square matrices and requiring no additional workspace other than what is already embeded in the high-performance implementations of GEMM.

- **It demonstrates performance benefits of** STRASSEN **on single core, multi-core, many-core, and distributed parallel CPU architectures.** Thread-level parallelism without the extra overhead of task parallelism is attained by adopting the same loop structure and parallel scheme as the high-performance GEMM implementation.

- **It facilitates a code generator to automatically implement families of Strassen-like FMM algorithms,** which expresses the composition of multi-level FMM algorithms as Kronecker products.

- **It builds an accurate performance model,** which can also be generated by the code generator and used for guiding the choice of a FMM implementation as a function of problem size and matrix shape, facilitating the creation of "poly-algorithms" [77] that select the best algorithm without exhaustive empirical search.

- **It provides the first efficient implementation of** STRASSEN **for tensor contraction,** the multi-dimensional generalization of matrix multiplication, without the explicit transposition of data that inherently incur significant memory movement and workspace overhead.

- **It achieves more parallelism and requires less memory with a practical STRASSEN implementation on Nvidia GPUs,** which can be viewed as massively-parallel, throughput-oriented computing engines. The specialized kernel is developed to utilize the memory hierarchy and thread hierarchy, to reduce the additional workspace requirement through reusing the shared memory and the register files, and to exploit the inter-kernel parallelism as well as the intra-kernel parallelism.

1.4 Outline of the dissertation

The rest of this dissertation is organized as follows:

- Chapter 2 discusses the literature related to the high-performance implementations of matrix multiplication algorithms and the theoretical and practical breakthroughs of STRASSEN and similar Strassen-like fast matrix multiplication algorithms.

- Chapter 3 presents a novel practical implementation of STRASSEN, which demonstrates significant speedups for small problem sizes and non-square matrix shapes, avoids additional workspace requirement other than what is already in GEMM, and achieves better performance on various CPU architectures.

- Chapter 4 further develops a code generator framework to automatically implement a large family of fast matrix multiplication algorithms suitable for multiplications of arbitrary matrix sizes and shapes. Performance models

are incorporated into the code generator to guide the choice of a FMM implementation as a function of problems sizes and matrix shapes.

- Chapter 5 extends the novel insights from Chapter 3 into tensor contraction, a higher dimensional generalization of matrix multiplication.
- Chapter 6 describes how to apply STRASSEN on GPUs with less memory and more parallelism.
- Chapter 7 concludes this dissertation and proposes ideas for future research.
- Appendices A and B provide tables of acronyms and symbols used throughout this dissertation.
- Appendix C shows an example of Strassen-like FMM algorithms other than STRASSEN.
- Appendix D derives that the composition of multi-level FMM algorithms can be expressed as Kronecker products.
- Appendix E addresses the numerical stability for STRASSEN and similar Strassen-like FMM algorithms.

Chapter 2

Related Work

In this chapter, we will look at related work, including a brief description of the state-of-the-art implementations and algorithms for matrix multiplication, followed by a brief literature review of STRASSEN and similar Strassen-like FMM algorithms over the last half century.

2.1 High-performance matrix multiplication algorithm

We start by discussing the conventional matrix multiplication (GEMM), how it is supported as a library routine by the Basic Linear Algebra Subprograms (BLAS) [30], how modern implementations block for caches, and how that implementation supports multi-threaded parallelization.

2.1.1 Computing $C = \alpha AB + \beta C$

Consider $C = \alpha AB + \beta C$, where C, A, and B are $m \times n$, $m \times k$, and $k \times n$ matrices, respectively, and α and β are scalars. 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 $C = \alpha AB + \beta C$ is achieved by

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

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

2.1.2 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

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

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

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

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

2.1.3 Basic Linear Algebra Subprograms (BLAS)

The Basic Linear Algebra Subprograms (BLAS) [73, 31, 30] specifies an interface for a set of dense linear algebra (DLA) operations on which higher level linear algebra libraries, such at LAPACK [2] and `libflame` [121], are built. The BLAS operations are divided into three sets: the level-1 BLAS (vector-vector operations), the level-2 BLAS (matrix-vector operations), and the level-3 BLAS (matrix-matrix operations). Examples of BLAS libraries on CPUs include IBM ESSL [60], Intel MKL [61], ATLAS [3], GotoBLAS [40], OpenBLAS [90], and BLIS [14]. Other implementations of BLAS operations

with similar interfaces on GPUs include clBLAS [20], Nvidia cuBLAS [87], and AMD rocBLAS [1].

2.1.4 Level-3 BLAS matrix-matrix multiplication

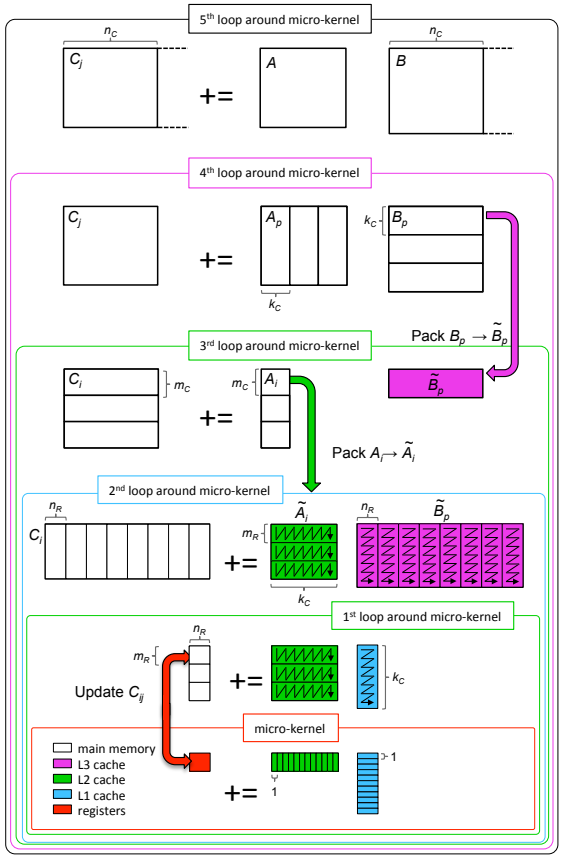
(General) matrix-matrix multiplication (GEMM) is supported in the level-3 BLAS [30] interface as

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

where "`DGEMM`" and "`SGEMM`" are for double precision and single precision, respectively. These calls support

$$C = \alpha AB + \beta C, \quad C = \alpha A^T B + \beta C,$$
$$C = \alpha AB^T + \beta C, \quad \text{and } C = \alpha A^T B^T + \beta C$$

depending on the choices of `transa` and `transb` (indicating whether the matrices A and B should be transposed). The parameters `C`, `A`, and `B` are the arrays used to store the matrices C, A, and B. The parameters `m`, `n`, `k`, `alpha`, and `beta` are coresponding to m, n, k, α, and β in Section 2.1.1. The parameters `lda`, `ldb`, and `ldc` specify the leading dimension of A, B, and C, that is, the the number of elements between successive columns (for column major storage) in memory. In this dissertation, we focus on the special case $\alpha = 1$ and $\beta = 1$ for brevity. 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 $C = AB + C$.



Loop 5 **for** $j_c = 0 : n-1$ **steps of** n_c
$\mathcal{J}_c = j_c : j_c + n_c - 1$
Loop 4 **for** $p_c = 0 : k-1$ **steps of** k_c
$\mathcal{P}_c = p_c : p_c + k_c - 1$
$B(\mathcal{P}_c, \mathcal{J}_c) \rightarrow \widetilde{B}$ // Pack into $\widetilde{B}_p$
Loop 3 **for** $i_c = 0 : m-1$ **steps of** m_c
$\mathcal{I}_c = i_c : i_c + m_c - 1$
$A(\mathcal{I}_c, \mathcal{P}_c) \rightarrow \widetilde{A}$ // Pack into $\widetilde{A}_i$
// `macro-kernel`
Loop 2 **for** $j_r = 0 : n_c - 1$ **steps of** n_r
$\mathcal{J}_r = j_r : j_r + n_r - 1$
Loop 1 **for** $i_r = 0 : m_c - 1$ **steps of** m_r
$\mathcal{I}_r = i_r : i_r + m_r - 1$
// `micro-kernel`
Loop 0 **for** $p_r = 0 : k_c - 1$
$C(\mathcal{I}_r, \mathcal{J}_r) \mathrel{+}= \widetilde{A}(\mathcal{I}_r, p_r)\ \widetilde{B}(p_r, \mathcal{J}_r)$
endfor
endfor
endfor
endfor
endfor

Figure 2.1: Left: illustration of the BLIS implementation of the GotoBLAS algorithm. All computation is cast in terms of a highly optimized micro-kernel. Right: the same algorithm, but expressed as loops. The left figure originally appeared in [123], and the right figure originally appeared in [109].

2.1.5 The GotoBLAS algorithm

A key insight underlying modern high-performance implementations of GEMM is to organize the computations by partitioning the operands into blocks for temporal locality, and to *pack* (copy) such blocks into contiguous buffers that fit into various levels of memory for spatial locality. Figure 2.1 illustrates the GotoBLAS algorithm as implemented in BLIS.

- Cache blocking parameters $\{m_C, n_C, k_C\}$ determine the submatrix sizes of B_p ($k_C \times n_C$) and A_i ($m_C \times k_C$), such that they fit in various caches. During the computation, row panels B_p are contiguously packed into buffer $\widetilde{B}_p$ to fit in the L3 cache.[1] Blocks A_i are similarly packed into buffer $\widetilde{A}_i$ to fit in the L2 cache.

- Register block sizes $\{m_R, n_R\}$ relate to submatrices in registers that contribute to C. In the *micro-kernel* (the inner most loop), a small $m_R \times n_R$ micro-tile of C is updated by a pair of $m_R \times k_C$ and $k_C \times n_R$ micro panels of $\widetilde{A}_i$ and $\widetilde{B}_p$.

The above parameters can be analytically chosen [80].

2.1.6 Hierarchical memory architectures

The modern computer architecture with a hierarchical memory architectures has memory that is laid out as a pyramid, with fast and small memo-

[1] 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.

ries close to the processor, and slow and large memories far away from it [48].

Gunnels *et al.* [44] describes a family of algorithms for matrix multiplication on hierarchical memory architectures. The theoretical insight showed that, depending on the matrix shape of matrix multiplication that is executed at the current level of cache, there are two locally-optimal strategies for the matrix shape of matrix multiplication that must happen at the next highest level of cache. That motivates a tree of locally-optimal decisions, each path from root to leaf of which represents a member of the family of algorithms.

Smith [106] further derived a new family of algorithms for optimizing the I/O cost of matrix multiplication at multiple levels of the memory hierarchy simultaneously. He showed that one can compose two loops per level of cache in order to encounter an optimal subproblem, whose shapes arise from algorithms that attain the theoretical I/O lower bounds for matrix multiplications for a single level of cache, at each layer of the memory hierarchy. With the expansion of the family of algorithms to allow more flexibility and to include GotoBLAS as a part of this family, the Multilevel Optimized Matrix-matrix Multiplication Sandbox (MOMMS) [105] was created to demonstrate the performance improvements of practical algorithms from the family over the state-of-the-art GotoBLAS in terms of I/O cost. Importantly to that work, it showed some members of the family of algorithms can outperform the state-of-the-art when the bandwidth to main memory is low. Future computer architectures are expected to be bandwidth bound, while the conventional implementations of Strassen and similar Strassen-like FMM al-

gorithms reduce the total number of computations at the cost of consuming more bandwidth. That work may be the answer to how to develop STRASSEN and similar Strassen-like FMM algorithms based on matrix multiplication in a low bandwidth scenario. Details of this go beyond this dissertation.

2.1.7 Multi-threaded implementation

BLIS exposes all the illustrated loops in Figure 2.1, 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 (known as a "macro-kernel" in BLIS terminology) 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 [108].

2.1.8 Other approaches

In the early stages, high-performance GEMM typically required hand-written assembly code optimized for a specific processor. To avoid this tedious task, autotuning is proposed where the best parameters are determined by measuring performance empirically when thosed parameters are tuned. The Portable High Performance ANSI C (PHiPAC) [11] project provided guidelines for writing the high-performance GEMM in `C` and introduced the code generator to autogenerate and find the best parameters for a given system. Further, the Automatically Tuned Linear Algebra Software (ATLAS) [126] were built

on these insights and made autogeneration and autotuning of BLAS libraries mainstream. The main contribution of ATLAS was the idea of casting matrix multiplication in terms of a basic unit of computation, which was originally called "on-chip multiply" and is known as "inner-kernel" or "macro-kernel" now. However, as pointed out by Smith [106], the I/O cost of the algorithms used by ATLAS is 50% higher than the tight lower bound in [107]. On the other hand, model-driven optimization can be more effective to select the best parameters with the best performance for a given architecture. Yotov *et al.* [130] showed that one can identify the near-optimal block sizes for GEMM with the analytical model, and thus empirical search is unnecessary. Low [80] further applied the analytical model approach to the more modern GotoBLAS.

Kågström [63] showed that the level-3 BLAS operations can be implemented mostly based on GEMM. This is sometimes referred to as "poorman's BLAS" in the sense that it is only necessary to optimize GEMM, then all level-3 BLAS can be built in terms of this "for free", which brings both modularity and high performance.

2.2 Literature review for FMM algorithms

There is an extensive literature on the theory and practice for Strassen and FMM algorithms. Here we only mention the most influential ones.

2.2.1 Theory

A history for complexity improvements

Until the late 1960's, it was widely accepted that the number of operations for the multiplication of $n \times n$ matrices was essentially $O(n^3)$. In 1969, Strassen [115] demonstrated that it suffices to perform matrix multiplication with only $O(n^{2.8074})$ arithmetic operations, which opened a new era of reseach for fast matrix multiplication algorithms. Figure 2.2 shows the chronological improvement of the exponents ω for the FMM arithmetic complexity $O(n^\omega)$ over the last decades.

In 1971, Winograd [129] proved that the minimum required number of multiplications for 2×2 and 2×2 submatrix multiplication is seven. This paper further led to a more efficient Winograd variant of Strassen's algorithm, in which the recursive step has 15 instead of 18 submatrix additions. In 1978, Pan [91] constructed an exact algorithm with $O(n^{2.7951})$ arithmetic complexity, by trilinear aggregation and a base case of multiplying 70×70 submatrix multiplications in 143640 operations. In 1979, Bini *et al.* [12] presented a method for multiplying 3×2 and 2×2 submatrices with $O(n^{2.7799})$ complexity, by introducing the notion of arbitrary precision approximate (APA) algorithms and border rank. In 1981, Schönhage [100] further generalized that notion of APA and border rank, proved the asymptotic sum inequality, and obtained $O(n^{2.522})$ complexity based on 3×3 and 3×3 submatrix multiplications. The following year, Coppersmith and Winograd [23] achieved $O(n^{2.4966})$ complexity with similar techniques, the first result beating $O(n^{2.5})$. In 1986, Strassen [116]

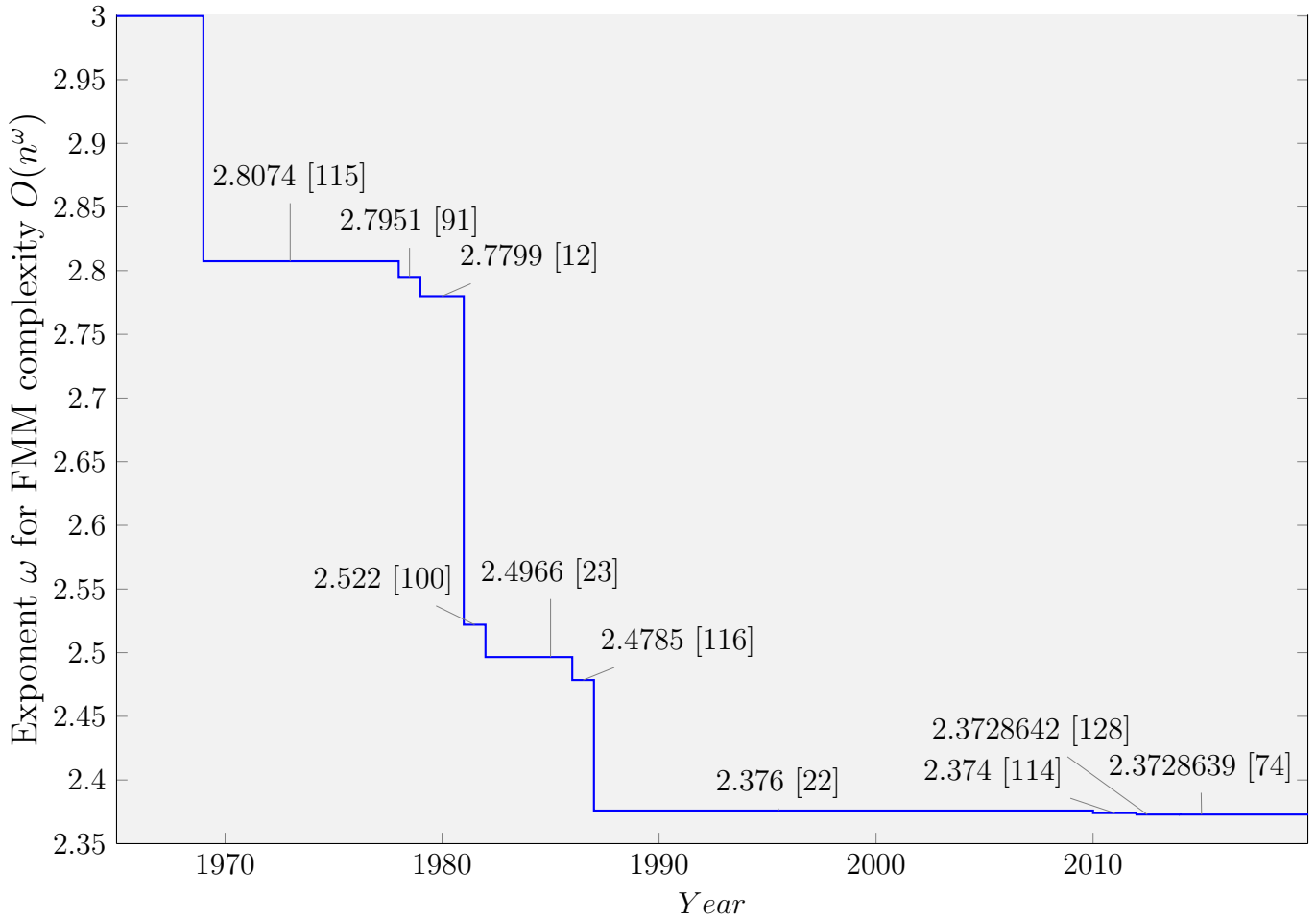


Figure 2.2: Chronological decrease of the exponents ω for the arithmetic compelxity of (fast) matrix multiplication, $O(n^\omega)$. Note that we only cover some important papers.

introduced a new method called the laser method technique with $O(n^{2.4785})$ complexity, which was adopted by Coppersmith and Winograd [22] one year later to push to the famous bound $O(n^{2.376})$. This new bound had been the state of the art for the following twenty years, until finally being advanced by Stothers [114] with the new upper bound $O(n^{2.374})$, in 2010. Two years later, Williams [128] futher improved the bound to $O(n^{2.3728642})$. More recently, in 2014, François [74] achieved the current world champion bound $O(n^{2.3728639})$ by polishing up Williams' methods.

This is just a small sample of the important breakthroughs in this area.

Numerical stability

Bini and Lotti [13] proves the "normwise stability" and provides the first general error bound for Strassen and similar Strassen-like FMM algorithms. Since then, a number of papers [50, 28, 25] have been published to analyze and improve the numerical properties of various FMM algorithms. Even more recently, Ballard *et al.* [6] explored ways to improve the accuracy both thoeretically and empirically. These papers show FMM algorithms are reasonably numerically stable as long as only a few levels of recursions are adopted. More details about the numerical stability of Strassen and similar Strassen-like FMM algorithms can be found in Appendix E.

Feasible algorithms

A number of theoretical research papers mentioned above [12, 100, 23, 116, 23, 114, 128, 74] are of purely theoretical interest and not feasible in practice. On the one hand, these papers only prove the existence of fast APA algorithms with better asymptotic complexity, which are valid only for inputs beyond any practical size [100]. On the other hand, those APA algorithms exhibit serious numerical instability problems [9].

A recent review [33] pointed out that the current complexity record for "moderate" problem size with $n < 1000000$ is $O(n^{2.7734})$, achieved by Pan with the 44×44 submatrix multiplications in [92]. Remarkably, Smirnov [104] proposed systematic approaches to discover feasible algorithms and presented several new algorithms, including an algorithm with complexity $O(n^{2.7743})$. Most recently, Karstadt and Schwartz [65] obtained a faster FMM algorithm with the same base case size and asymptotic complexity as Winograd [129], but with the coefficient reduced from 6 to 5.

2.2.2 Practice

Layering FMM algorithms upon standard GEMM calls

There is a rich history of work that combines a few levels of STRASSEN and related variants with calls to highly optimized implementations of the GEMM that are part of the BLAS: IBM's ESSL library [60] long included such routines as an option and portable libraries were reported in Huss-Lederman *et al.* [59] and Douglas *et al.* [32].

Lower level implementation

A very thorough treatment was given by D'Alberto *et al.* [27], which not only supported the layering upon GEMM but also studied how to optimize below the BLAS interface. However, in these studies, extra temporary matrices are still required.

Sequential

Bailey [5], Douglas *et al.* [32], and Huss-Lederman [59] provided sequential implementations of Strassen's algorithm. Kaporin [64] implemented the algorithms presented in [91, 71], showing that the running time for one-level of such algorithms (e.g., 70×70 submatrix multiplications) was comparable with Strassen's algorithm on a sequential machine.

Parallelization for shared memory

Scheduling tasks (multiplications with submatrices) to multiprocessors and multi-threaded architectures has been widely studied [27, 9, 26, 70]. A fundamental problem is that STRASSEN creates 7^l such tasks, where l equals the number of levels of STRASSEN. This makes it difficult to load balance [9]. Other Strassen-like FMM algorithms suffer similar problems.

Parallelization for distributed memory

The distributed memory implementation of STRASSEN by Luo and Drake [81] based on Cannon's algorithm [17] inspired the more practical im-

plementation by Grayson and van de Geijn [42] that builds upon the SUMMA algorithm for distributed memory matrix-matrix multiplication by van de Geijn and Watts [120]. Ballard [7] and Lipshitz [79] demonstrated a parallel STRASSEN implementation for distributed memory that minimizes communication cost.

Parallelization for GPUs

STRASSEN and Winograd [129] variants are efficiently implemented on a single GPU in [76, 72]. Lai *et al.* [72] supported arbitrary sized matrices by use of dynamic peeling [117] and exploited multi-kernel concurrency to leverage the inter-block and intra-block parallelism in the lowest level of recursion, based on the cuBLAS library [87].

2.3 Summary

We have reviewed the related work on the high-performance implementations of the conventional matrix multiplication and the theoretical and practical breakthroughs of STRASSEN and similar Strassen-like FMM algorithms over the last half century. More details for the work directly related to this dissertation will be given in the later chapters.

Chapter 3

A Practical Strassen's Algorithm

Let us now focus on the practical implementation of STRASSEN, the original Fast Matrix Multiplication algorithm. The orthodox implementation of STRASSEN entails "street wisdom" that it is only practical for large and relatively square matrices, that it needs substantial workspace, that it suffers the overhead of extra memory movement that is incurred, and that it is laborious to achieve thread-level parallelism. In this chapter, we counter conventional wisdom, demostrating significant performance improvements for small and non-square matrices, requiring no workspace beyond what is already incorporated in the high-performance implementation of GEMM, and achieving speedup on single core, multi-core, many-core, and distributed memory architectures.

This chapter is based on the conference paper [56] with minor modifications: "Jianyu Huang, Tyler M. Smith, Greg M. Henry, and Robert A. van de Geijn. Strassen's algorithm reloaded. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 16)*, pages 59:1-59:12. IEEE Press, 2016." I am the main contributor in charge of problem formulation, algorithm development, performance analysis, and experimental validations.

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

Figure 3.1: All operations for one-level STRASSEN. Note that each row is a special case of general operation (3.2).

3.1 Strassen's algorithm reloaded

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 GEMM.

3.1.1 The basic idea

If one partitions the three operands into quadrants,

$$
X = \left(\begin{array}{c|c} X_{00} & X_{01} \\ \hline X_{10} & X_{11} \end{array} \right) \quad \text{for } X \in \{A, B, C\} \tag{3.1}
$$

then it can be verified that the operations in Figure 3.1 also compute $C = AB + C$, 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 3.1 describes what we will call *one-level* STRASSEN.

3.1.2 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

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

In this discussion, we ignored the increase in the total number of extra additions, which contributes a lower-order term.

3.1.3 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 GEMM is faster. At that point, the recursion stops and a high-performance GEMM is used for the subproblems. In prior implementations, the crossover point is usually as large as 2000 for double precision square matrices on a single core of an x86 CPU [26, 9]. We will see that, for the same architecture, one of our implementations has a crossover point as small as 500 (Figure 3.8).

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 [15]. We show that the computation can be organized so that no temporary storage beyond that required for a high-performance traditional GEMM 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.

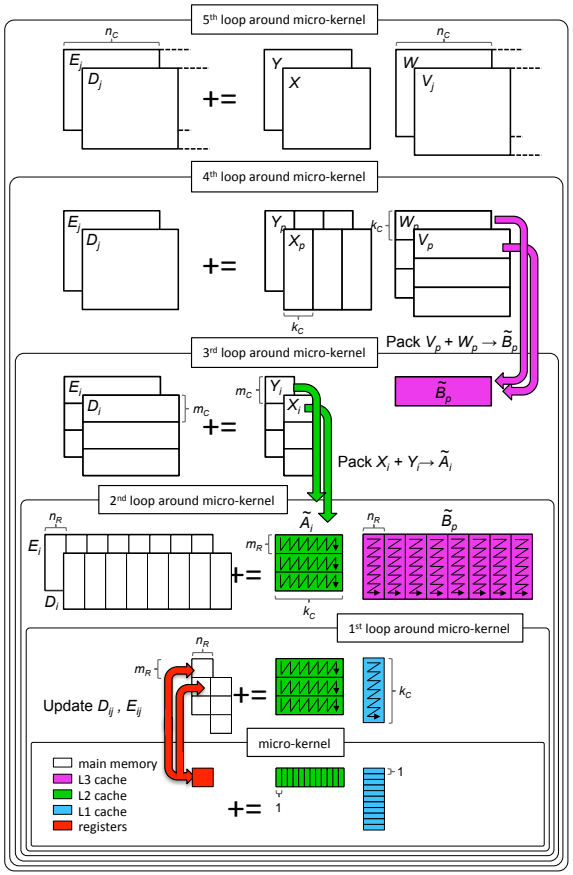
The general case where one or more dimensions are not a convenient multiple of a power of two leads to the need to either pad matrices or to treat a remaining "fringe" carefully [59]. Traditionally, it is necessary to pad m, n, and k to be even. In our approach this can be handled internally by padding $\widetilde{A}_i$ 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, see Section 2.1.5).

3.1.4 One-level STRASSEN reloaded

The operations summarized in Figure 3.1 are all special cases of

$$M = (X + \delta Y)(V + \epsilon W); \quad D \mathrel{+}= \gamma_0 M; \quad E \mathrel{+}= \gamma_1 M; \tag{3.2}$$

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 D and E are submatrices of C.



Loop 5 **for** $j_c = 0 : n-1$ **steps of** n_c
$\mathcal{J}_c = j_c : j_c + n_c - 1$
Loop 4 **for** $p_c = 0 : k-1$ **steps of** k_c
$\mathcal{P}_c = p_c : p_c + k_c - 1$
$V(\mathcal{P}_c, \mathcal{J}_c) + \epsilon W(\mathcal{P}_c, \mathcal{J}_c) \rightarrow \widetilde{B}_p$ // Pack into $\widetilde{B}_p$
Loop 3 **for** $i_c = 0 : m-1$ **steps of** m_c
$\mathcal{I}_c = i_c : i_c + m_c - 1$
$X(\mathcal{I}_c, \mathcal{P}_c) + \delta Y(\mathcal{I}_c, \mathcal{P}_c) \rightarrow \widetilde{A}_i$ // Pack into $\widetilde{A}_i$
// `macro-kernel`
Loop 2 **for** $j_r = 0 : n_c - 1$ **steps of** n_r
$\mathcal{J}_r = j_r : j_r + n_r - 1$
Loop 1 **for** $i_r = 0 : m_c - 1$ **steps of** m_r
$\mathcal{I}_r = i_r : i_r + m_r - 1$
// `micro-kernel`
Loop 0 **for** $p_r = 0 : p_c - 1$ **steps of** 1
$M_r(\mathcal{I}_r, \mathcal{J}_r) \mathrel{+}= \widetilde{A}_i(\mathcal{I}_r, p_r)\ \widetilde{B}_p(p_r, \mathcal{J}_r)$
endfor
$D(\mathcal{I}_r + i_c, \mathcal{J}_r + j_c) \mathrel{+}= \gamma_0 M_r(\mathcal{I}_r, \mathcal{J}_r)$
$E(\mathcal{I}_r + i_c, \mathcal{J}_r + j_c) \mathrel{+}= \gamma_1 M_r(\mathcal{I}_r, \mathcal{J}_r)$
endfor
endfor
endfor
endfor

Figure 3.2: Left: modification of Figure 2.1 that implements the representative computation $M = (X + Y)(V + W)$; $D \mathrel{+}= M$; $E \mathrel{+}= M$ of general operation (3.2). X, Y are submatrices of A; V, W are submatrices of B; D, E are submatrices of C; M is the intermediate matrix product. Note that the packing buffers $\widetilde{A}_i$ and $\widetilde{B}_p$ stay in cache. Right: the same algorithm, but expressed as loops ($\gamma_0 = \gamma_1 = \delta = \epsilon = 1$).

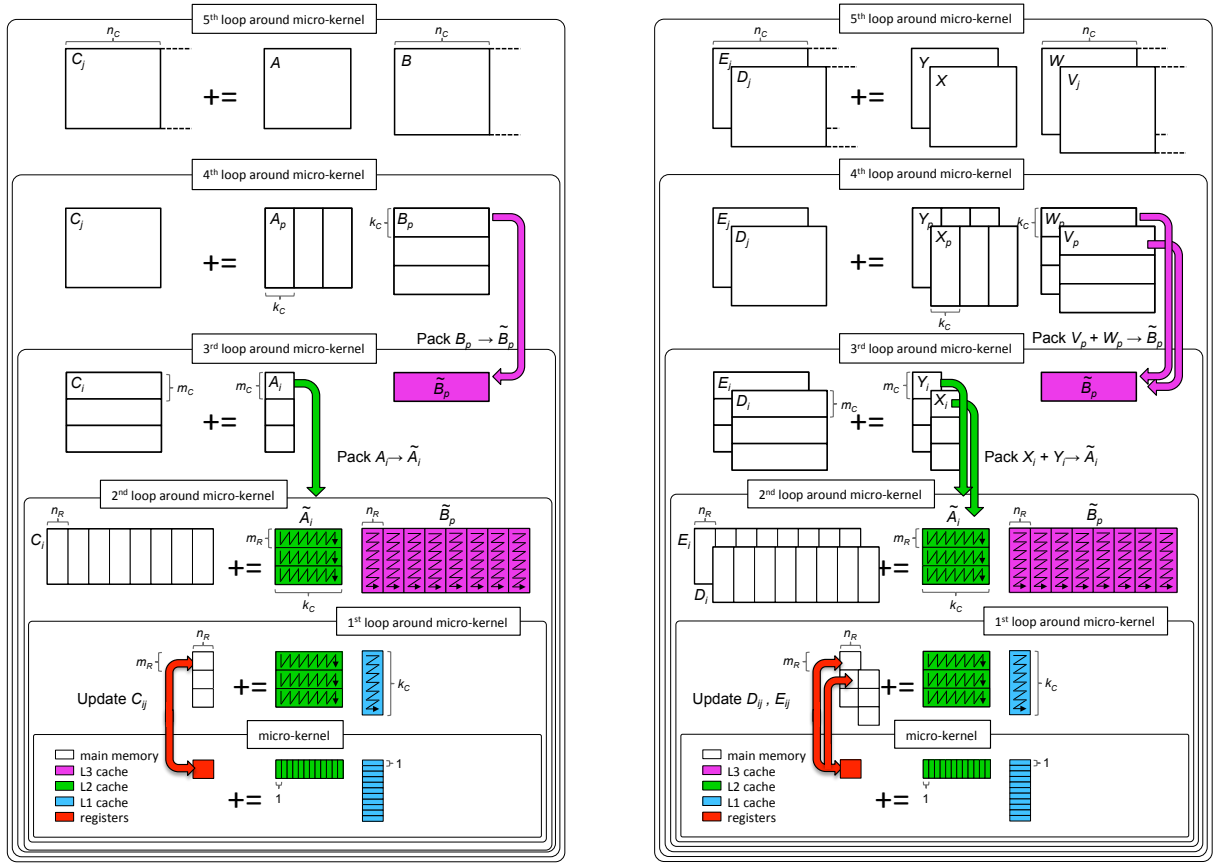


Figure 3.3: A side-by-side comparison of the BLIS implementation of the GotoBLASalgorithm and our modifications for implementing the representive computation $M = (X + Y)(V + W); D \mathrel{+}= M; E \mathrel{+}= M$. Left: Figure 2.1; Right: Figure 3.2.

Let us focus on how to modify the algorithm illustrated in Figure 2.1 in order to accommodate the representative computation

$$M = (X + Y)(V + W); D \mathrel{+}= M; E \mathrel{+}= M.$$

As illustrated in Figure 3.2 and Figure 3.3, 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 D and E, multiplied by γ_0 and γ_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.

3.1.5 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 3.4 compute $C = AB + C$. The operations found there can be cast as special cases of

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

$M_0 = (A_{0,0}+A_{2,2}+A_{1,1}+A_{3,3})(B_{0,0}+B_{2,2}+B_{1,1}+B_{3,3});$	$C_{0,0}\mathrel{+}= M_0;\ C_{1,1}\mathrel{+}= M_0;\ C_{2,2}\mathrel{+}= M_0;\ C_{3,3}\mathrel{+}= M_0;$
$M_1 = (A_{1,0}+A_{3,2}+A_{1,1}+A_{3,3})(B_{0,0}+B_{2,2});$	$C_{1,0}\mathrel{+}= M_1;\ C_{1,1}\mathrel{-}= M_1;\ C_{3,2}\mathrel{+}= M_1;\ C_{3,3}\mathrel{-}= M_1;$
$M_2 = (A_{0,0}+A_{2,2})(B_{0,1}+B_{2,3}+B_{1,1}+B_{3,3});$	$C_{0,1}\mathrel{+}= M_2;\ C_{1,1}\mathrel{+}= M_2;\ C_{2,3}\mathrel{+}= M_2;\ C_{3,3}\mathrel{+}= M_2;$
$M_3 = (A_{1,1}+A_{3,3})(B_{1,0}+B_{3,2}+B_{0,0}+B_{2,2});$	$C_{0,0}\mathrel{+}= M_3;\ C_{1,0}\mathrel{+}= M_3;\ C_{2,2}\mathrel{+}= M_3;\ C_{3,2}\mathrel{+}= M_3;$
$M_4 = (A_{0,0}+A_{2,2}+A_{0,1}+A_{2,3})(B_{1,1}+B_{3,3});$	$C_{0,0}\mathrel{-}= M_4;\ C_{0,1}\mathrel{+}= M_4;\ C_{2,2}\mathrel{-}= M_4;\ C_{2,3}\mathrel{+}= M_4;$
$M_5 = (A_{1,0}+A_{3,2}+A_{0,0}+A_{2,2})(B_{0,0}+B_{2,2}+B_{0,1}+B_{2,3});$	$C_{1,1}\mathrel{+}= M_5;\ C_{3,3}\mathrel{+}= M_5;$
$M_6 = (A_{0,1}+A_{2,3}+A_{1,1}+A_{3,3})(B_{1,0}+B_{3,2}+B_{1,1}+B_{3,3});$	$C_{0,0}\mathrel{+}= M_6;\ C_{2,2}\mathrel{+}= M_6;$
$M_7 = (A_{2,0}+A_{2,2}+A_{3,1}+A_{3,3})(B_{0,0}+B_{1,1});$	$C_{2,0}\mathrel{+}= M_7;\ C_{3,1}\mathrel{+}= M_7;\ C_{2,2}\mathrel{-}= M_7;\ C_{3,3}\mathrel{-}= M_7;$
$M_8 = (A_{3,0}+A_{3,2}+A_{3,1}+A_{3,3})(B_{0,0});$	$C_{3,0}\mathrel{+}= M_8;\ C_{3,1}\mathrel{-}= M_8;\ C_{3,2}\mathrel{-}= M_8;\ C_{3,3}\mathrel{+}= M_8;$
$M_9 = (A_{2,0}+A_{2,2})(B_{0,1}+B_{1,1});$	$C_{2,1}\mathrel{+}= M_9;\ C_{3,1}\mathrel{+}= M_9;\ C_{2,3}\mathrel{-}= M_9;\ C_{3,3}\mathrel{-}= M_9;$
$M_{10} = (A_{3,1}+A_{3,3})(B_{1,0}+B_{0,0});$	$C_{2,0}\mathrel{+}= M_{10};C_{3,0}\mathrel{+}= M_{10};C_{2,2}\mathrel{-}= M_{10};C_{3,2}\mathrel{-}= M_{10};$
$M_{11} = (A_{2,0}+A_{2,2}+A_{2,1}+A_{2,3})(B_{1,1});$	$C_{2,0}\mathrel{-}= M_{11};C_{2,1}\mathrel{+}= M_{11};C_{2,2}\mathrel{+}= M_{11};C_{2,3}\mathrel{-}= M_{11};$
$M_{12} = (A_{3,0}+A_{3,2}+A_{2,0}+A_{2,2})(B_{0,0}+B_{0,1});$	$C_{3,1}\mathrel{+}= M_{12};C_{3,3}\mathrel{-}= M_{12};$
$M_{13} = (A_{2,1}+A_{2,3}+A_{3,1}+A_{3,3})(B_{1,0}+B_{1,1});$	$C_{2,0}\mathrel{+}= M_{13};C_{2,2}\mathrel{-}= M_{13};$
$M_{14} = (A_{0,0}+A_{1,1})(B_{0,2}+B_{2,2}+B_{1,3}+B_{3,3});$	$C_{0,2}\mathrel{+}= M_{14};C_{1,3}\mathrel{+}= M_{14};C_{2,2}\mathrel{+}= M_{14};C_{3,3}\mathrel{+}= M_{14};$
$M_{15} = (A_{1,0}+A_{1,1})(B_{0,2}+B_{2,2});$	$C_{1,2}\mathrel{+}= M_{15};C_{1,3}\mathrel{-}= M_{15};C_{3,2}\mathrel{+}= M_{15};C_{3,3}\mathrel{-}= M_{15};$
$M_{16} = (A_{0,0})(B_{0,3}+B_{2,3}+B_{1,3}+B_{3,3});$	$C_{0,3}\mathrel{+}= M_{16};C_{1,3}\mathrel{+}= M_{16};C_{2,3}\mathrel{+}= M_{16};C_{3,3}\mathrel{+}= M_{16};$
$M_{17} = (A_{1,1})(B_{1,2}+B_{3,2}+B_{0,2}+B_{2,2});$	$C_{0,2}\mathrel{+}= M_{17};C_{1,2}\mathrel{+}= M_{17};C_{2,2}\mathrel{+}= M_{17};C_{3,2}\mathrel{+}= M_{17};$
$M_{18} = (A_{0,0}+A_{0,1})(B_{1,3}+B_{3,3});$	$C_{0,2}\mathrel{-}= M_{18};C_{0,3}\mathrel{+}= M_{18};C_{2,2}\mathrel{-}= M_{18};C_{2,3}\mathrel{+}= M_{18};$
$M_{19} = (A_{1,0}+A_{0,0})(B_{0,2}+B_{2,2}+B_{0,3}+B_{2,3});$	$C_{1,3}\mathrel{+}= M_{19};C_{3,3}\mathrel{+}= M_{19};$
$M_{20} = (A_{0,1}+A_{1,1})(B_{1,2}+B_{3,2}+B_{1,3}+B_{3,3});$	$C_{0,2}\mathrel{+}= M_{20};C_{2,2}\mathrel{+}= M_{20};$
$M_{21} = (A_{2,2}+A_{3,3})(B_{2,0}+B_{0,0}+B_{3,1}+B_{1,1});$	$C_{0,0}\mathrel{+}= M_{21};C_{1,1}\mathrel{+}= M_{21};C_{2,0}\mathrel{+}= M_{21};C_{3,1}\mathrel{+}= M_{21};$
$M_{22} = (A_{3,2}+A_{3,3})(B_{2,0}+B_{0,0});$	$C_{1,0}\mathrel{+}= M_{22};C_{1,1}\mathrel{-}= M_{22};C_{3,0}\mathrel{+}= M_{22};C_{3,1}\mathrel{-}= M_{22};$
$M_{23} = (A_{2,2})(B_{2,1}+B_{0,1}+B_{3,1}+B_{1,1});$	$C_{0,1}\mathrel{+}= M_{23};C_{1,1}\mathrel{+}= M_{23};C_{2,1}\mathrel{+}= M_{23};C_{3,1}\mathrel{+}= M_{23};$
$M_{24} = (A_{3,3})(B_{3,0}+B_{1,0}+B_{2,0}+B_{0,0});$	$C_{0,0}\mathrel{+}= M_{24};C_{1,0}\mathrel{+}= M_{24};C_{2,0}\mathrel{+}= M_{24};C_{3,0}\mathrel{+}= M_{24};$
$M_{25} = (A_{2,2}+A_{2,3})(B_{3,1}+B_{1,1});$	$C_{0,0}\mathrel{-}= M_{25};C_{0,1}\mathrel{+}= M_{25};C_{2,0}\mathrel{-}= M_{25};C_{2,1}\mathrel{+}= M_{25};$
$M_{26} = (A_{3,2}+A_{2,2})(B_{2,0}+B_{0,0}+B_{2,1}+B_{0,1});$	$C_{1,1}\mathrel{+}= M_{26};C_{3,1}\mathrel{+}= M_{26};$
$M_{27} = (A_{2,3}+A_{3,3})(B_{3,0}+B_{1,0}+B_{3,1}+B_{1,1});$	$C_{0,0}\mathrel{+}= M_{27};C_{2,0}\mathrel{+}= M_{27};$
$M_{28} = (A_{0,0}+A_{0,2}+A_{1,1}+A_{1,3})(B_{2,2}+B_{3,3});$	$C_{0,0}\mathrel{-}= M_{28};C_{1,1}\mathrel{-}= M_{28};C_{0,2}\mathrel{+}= M_{28};C_{1,3}\mathrel{+}= M_{28};$
$M_{29} = (A_{1,0}+A_{1,2}+A_{1,1}+A_{1,3})(B_{2,2});$	$C_{1,0}\mathrel{-}= M_{29};C_{1,1}\mathrel{+}= M_{29};C_{1,2}\mathrel{+}= M_{29};C_{1,3}\mathrel{-}= M_{29};$
$M_{30} = (A_{0,0}+A_{0,2})(B_{2,3}+B_{3,3});$	$C_{0,1}\mathrel{-}= M_{30};C_{1,1}\mathrel{-}= M_{30};C_{0,3}\mathrel{+}= M_{30};C_{1,3}\mathrel{+}= M_{30};$
$M_{31} = (A_{1,1}+A_{1,3})(B_{3,2}+B_{2,2});$	$C_{0,0}\mathrel{-}= M_{31};C_{1,0}\mathrel{-}= M_{31};C_{0,2}\mathrel{+}= M_{31};C_{1,2}\mathrel{+}= M_{31};$
$M_{32} = (A_{0,0}+A_{0,2}+A_{0,1}+A_{0,3})(B_{3,3});$	$C_{0,0}\mathrel{+}= M_{32};C_{0,1}\mathrel{-}= M_{32};C_{0,2}\mathrel{-}= M_{32};C_{0,3}\mathrel{+}= M_{32};$
$M_{33} = (A_{1,0}+A_{1,2}+A_{0,0}+A_{0,2})(B_{2,2}+B_{2,3});$	$C_{1,1}\mathrel{-}= M_{33};C_{1,3}\mathrel{+}= M_{33};$
$M_{34} = (A_{0,1}+A_{0,3}+A_{1,1}+A_{1,3})(B_{3,2}+B_{3,3});$	$C_{0,0}\mathrel{-}= M_{34};C_{0,2}\mathrel{+}= M_{34};$
$M_{35} = (A_{2,0}+A_{0,0}+A_{3,1}+A_{1,1})(B_{0,0}+B_{0,2}+B_{1,1}+B_{1,3});$	$C_{2,2}\mathrel{+}= M_{35};C_{3,3}\mathrel{+}= M_{35};$
$M_{36} = (A_{3,0}+A_{1,0}+A_{3,1}+A_{1,1})(B_{0,0}+B_{0,2});$	$C_{3,2}\mathrel{+}= M_{36};C_{3,3}\mathrel{-}= M_{36};$
$M_{37} = (A_{2,0}+A_{0,0})(B_{0,1}+B_{0,3}+B_{1,1}+B_{1,3});$	$C_{2,3}\mathrel{+}= M_{37};C_{3,3}\mathrel{+}= M_{37};$
$M_{38} = (A_{3,1}+A_{1,1})(B_{1,0}+B_{1,2}+B_{0,0}+B_{0,2});$	$C_{2,2}\mathrel{+}= M_{38};C_{3,2}\mathrel{+}= M_{38};$
$M_{39} = (A_{2,0}+A_{0,0}+A_{2,1}+A_{0,1})(B_{1,1}+B_{1,3});$	$C_{2,2}\mathrel{-}= M_{39};C_{2,3}\mathrel{+}= M_{39};$
$M_{40} = (A_{3,0}+A_{1,0}+A_{2,0}+A_{0,0})(B_{0,0}+B_{0,2}+B_{0,1}+B_{0,3});$	$C_{3,3}\mathrel{+}= M_{40};$
$M_{41} = (A_{2,1}+A_{0,1}+A_{3,1}+A_{1,1})(B_{1,0}+B_{1,2}+B_{1,1}+B_{1,3});$	$C_{2,2}\mathrel{+}= M_{41};$
$M_{42} = (A_{0,2}+A_{2,2}+A_{1,3}+A_{3,3})(B_{2,0}+B_{2,2}+B_{3,1}+B_{3,3});$	$C_{0,0}\mathrel{+}= M_{42};C_{1,1}\mathrel{+}= M_{42};$
$M_{43} = (A_{1,2}+A_{3,2}+A_{1,3}+A_{3,3})(B_{2,0}+B_{2,2});$	$C_{1,0}\mathrel{+}= M_{43};C_{1,1}\mathrel{-}= M_{43};$
$M_{44} = (A_{0,2}+A_{2,2})(B_{2,1}+B_{2,3}+B_{3,1}+B_{3,3});$	$C_{0,1}\mathrel{+}= M_{44};C_{1,1}\mathrel{+}= M_{44};$
$M_{45} = (A_{1,3}+A_{3,3})(B_{3,0}+B_{3,2}+B_{2,0}+B_{2,2});$	$C_{0,0}\mathrel{+}= M_{45};C_{1,0}\mathrel{+}= M_{45};$
$M_{46} = (A_{0,2}+A_{2,2}+A_{0,3}+A_{2,3})(B_{3,1}+B_{3,3});$	$C_{0,0}\mathrel{-}= M_{46};C_{0,1}\mathrel{+}= M_{46};$
$M_{47} = (A_{1,2}+A_{3,2}+A_{0,2}+A_{2,2})(B_{2,0}+B_{2,2}+B_{2,1}+B_{2,3});$	$C_{1,1}\mathrel{+}= M_{47};$
$M_{48} = (A_{0,3}+A_{2,3}+A_{1,3}+A_{3,3})(B_{3,0}+B_{3,2}+B_{3,1}+B_{3,3});$	$C_{0,0}\mathrel{+}= M_{48};$

Figure 3.4: Computations for two-level Strassen.

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 GEMM.

To extend the insights from Section 3.1.4 so as to integrate two-level STRASSEN into the BLIS GEMM 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.

3.1.6 Additional levels

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

$$M = \left(\sum_{s=0}^{l_X-1} \delta_s X_s\right)\left(\sum_{t=0}^{l_V-1} \epsilon_t V_t\right); C_r \mathrel{+}= \gamma_r M \quad \text{for } r = 0, \ldots, l_C - 1. \quad (3.3)$$

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.

3.2 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.

3.2.1 Implementations

We implemented a family of algorithms for up to two levels[1] of STRASSEN for double precision arithmetic and data, building upon the BLIS framework.

Building blocks

The BLIS framework provides three primitives for composing GEMM: 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[2].

To implement a typical operation given in (3.3),

- 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.

[1] 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 ($\sim$ 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 chapter.

[2] The inlined assembly code is expressed in GNU extended assembly syntax with `asm` and written to utilize `SIMD` vector instructions such as `SSE2` and `AVX`.

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 shape is also called rank-k update), 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.

Different choices lead to a family of STRASSEN implementations.

3.2.2 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, 42, 79]:

$$\textit{effective}\ \text{GFLOPS} = \frac{2 \cdot m \cdot n \cdot k}{\text{time (in seconds)}} \cdot 10^{-9}. \tag{3.4}$$

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:

- memory packing in (adapted) DGEMM routine;
- reading/writing the submatrices of C in (adapted) DGEMM routine; and

τ_a	Time (in seconds) of one arithmetic (floating point) operation.
τ_b	(Bandwidth) Amortized time (in seconds) of 8 Bytes contiguous data movement from DRAM to cache.
T	Total execution time (in seconds).
T_a	The total time for arithmetic operations (in seconds).
T_m	Time for memory operations (in seconds).
$T_a^{\times}$	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 (Figure 3.3).
$T_m^{\tilde{A}_\times}$,$T_m^{\tilde{B}_\times}$	T_m for writing submatrices in packing routines (Figure 3.3).
$T_m^{C_\times}$	T_m for reading *and* writing submatrices in micro-kernel (Figure 3.3).
$T_m^{A_+}$, $T_m^{B_+}$, $T_m^{C_+}$	T_m for reading *or* writing submatrices, related to the temporary buffer as part of **Naive Strassen** and **AB Strassen**.
N_a^X/N_m^X	Coefficient for the corresponding T_a^X/T_m^X.

Figure 3.5: Notation table for STRASSEN performance model.

- 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.

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.

	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	-	$12 \times 2\frac{m}{2}\frac{n}{2}$	$144 \times 2\frac{m}{4}\frac{n}{4}$
$T_m^{A\times}$	`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	$\frac{m}{2}\frac{n}{2}$	$\frac{m}{4}\frac{n}{4}$

Figure 3.6: 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 total execution time (in seconds), T, is broken down into the time for arithmetic operations, T_a, and memory operations:

$$T = T_a + T_m. \tag{3.5}$$

The notations used in our model are summarized in Figure 3.5.

		$N_m^{A\times}$	$N_m^{B\times}$	$N_m^{C\times}$	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	144	-	-	-
	AB	194	194	49	-	-	432
	Naive	49	49	49	293	293	432

Figure 3.7: The coefficient N_m^X mapping table for computing T_m in the performance model.

Arithmetic Operations

We break down T_a into separate terms:

$$T_a = T_a^{\times} + T_a^{A+} + T_a^{B+} + T_a^{C+}, \tag{3.6}$$

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 Figures 3.6

and 3.7.

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 3.2. We characterize each memory operation term in Figure 3.6 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

$$T_m = N_m^{A\times} \cdot T_m^{A\times} + N_m^{B\times} \cdot T_m^{B\times} + N_m^{C\times} \cdot T_m^{C\times} + N_m^{A+} \cdot T_m^{A+} + N_m^{B+} \cdot T_m^{B+} + N_m^{C+} \cdot T_m^{C+}, \quad (3.7)$$

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 3.7.

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 3.2.

3.2.3 Discussion

From the analysis summarized in Figures 3.6 and 3.7 we can make predictions about the relative performance of the various implementations. It helps to also view the predictions as graphs, which we give in Figure 3.8, using parameters that capture the architecture described in Section 3.3.1.

- 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.

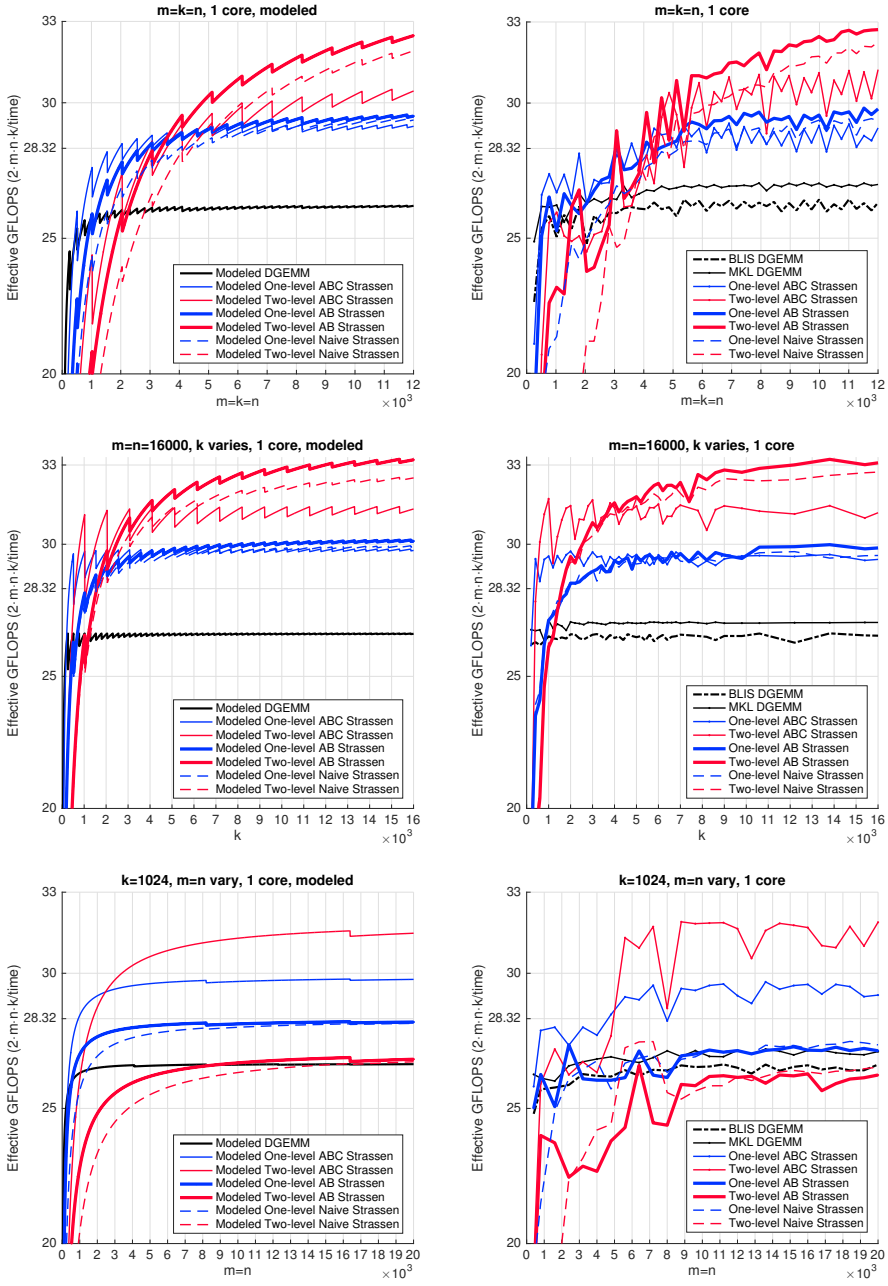


Figure 3.8: Performance of the various implementations on an Intel Xeon E5 2680 v2 (Ivybridge) processor (single core). Left: modeled performance. Right: actual performance. The range of the y-axis does not start at 0 to make the graphs more readable and 28.32 marks theoretical peak performance for this architecture.

- 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: the analysis predicts that, depending on the problem size, a different implementation may have its advantages.

3.3 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).

3.3.1 Single node experiments

Firstly, we reveal the details of single node experiments, using single-core and multi-core STRASSEN implementations.

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) [61].

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 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 [108], using OpenMP directives that parallelize the third loop around the micro-kernel in Figure 3.2.

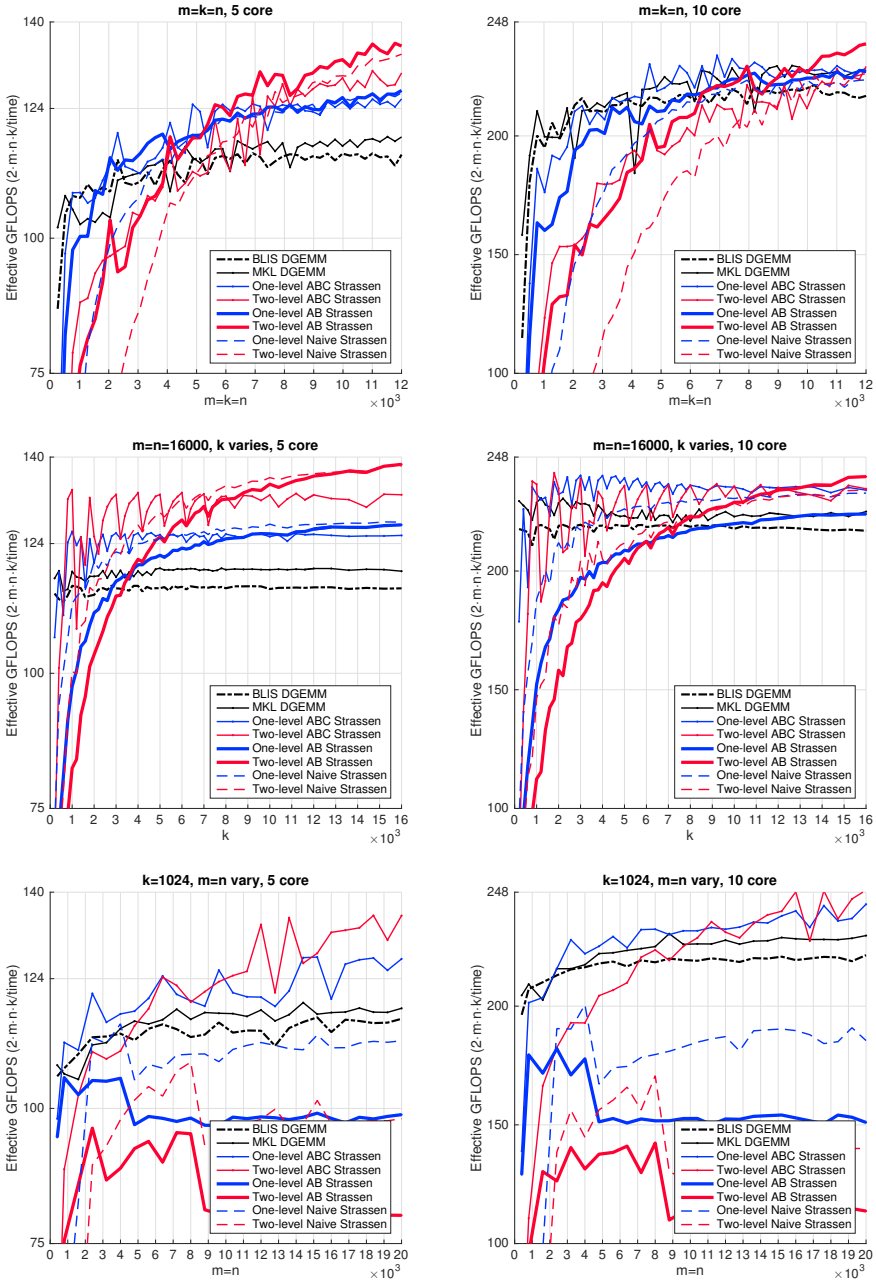


Figure 3.9: Performance of the various implementations on an Intel Xeon E5 2680 v2 (Ivybridge) processor (one and ten cores). Left: 5 core. Right: 10 core. The range of the y-axis does not start at 0 to make the graphs more readable.

Results

Results when using single core are presented in Figure 3.8 (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.

Figure 3.9 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 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.

3.3.2 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 second and third loop around the micro-kernel, as described for BLIS in [108].

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 3.10, relative to the BLIS DGEMM implementation, the one-level **ABC Strassen** shows a nontrivial improvement for a rank-k update with a fixed (large) matrix C. While the BLIS implementation on which our implementation of **ABC Strassen** is based used to be highly

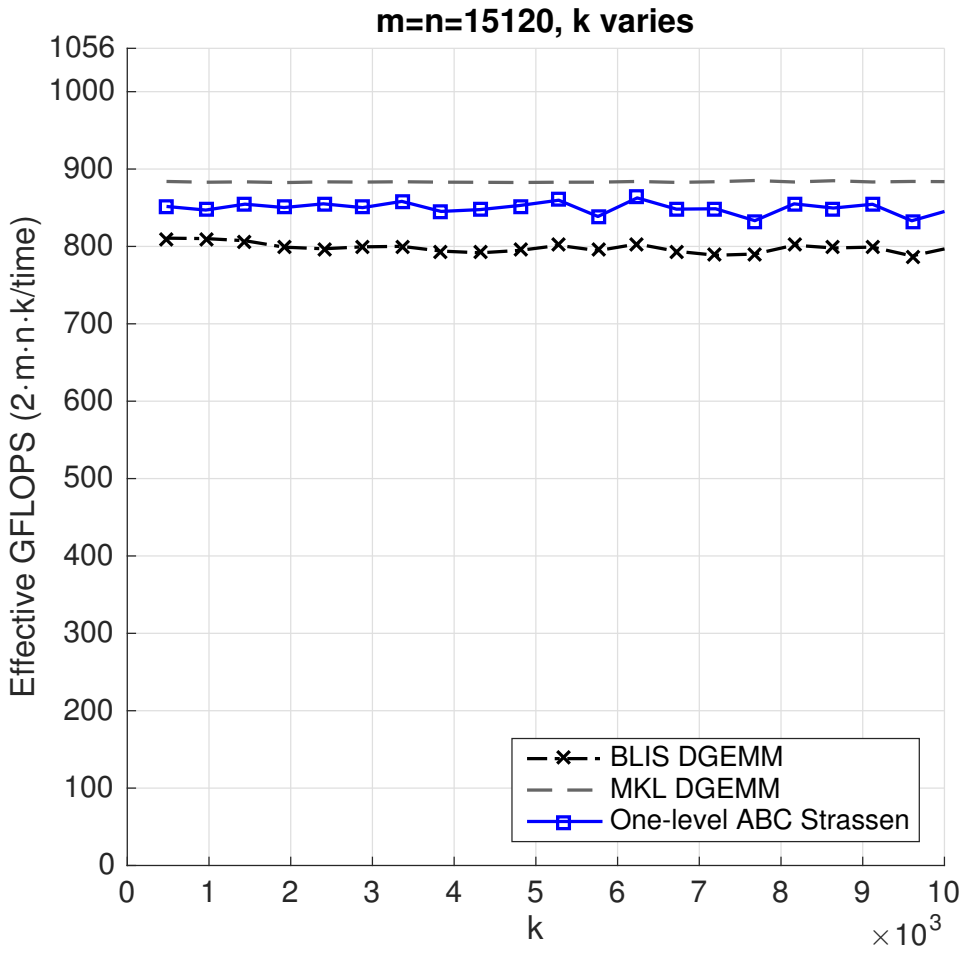


Figure 3.10: Performance of one-level **ABC Strassen**, BLIS, and MKL, on an Intel Xeon Phi (KNC) coprocessor (for 60 cores with 240 threads). The performance results are sampled such that k is a multiple of $480 = 2 \times k_C$.

competitive with MKL's DGEMM (as reported in [108]), 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, since it also implements GotoBLAS algorithm.

3.3.3 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) [120] 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,

$$\begin{aligned} C &:= AB + C = \left(\begin{array}{c|c|c} A_0 & \cdots & A_{K-1} \end{array} \right) \left(\begin{array}{c} \hline B_0 \\ \hline \vdots \\ \hline B_{K-1} \end{array} \right) + C \\ &= A_0 B_0 + \cdots + A_{K-1} B_{K-1} + C \end{aligned}$$

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 3.3.1, using the `mvapich2` version 2.1 [58] implementation of MPI. Each node has two sockets, and each socket has ten cores.

Results

Figure 3.11 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 GEMM 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

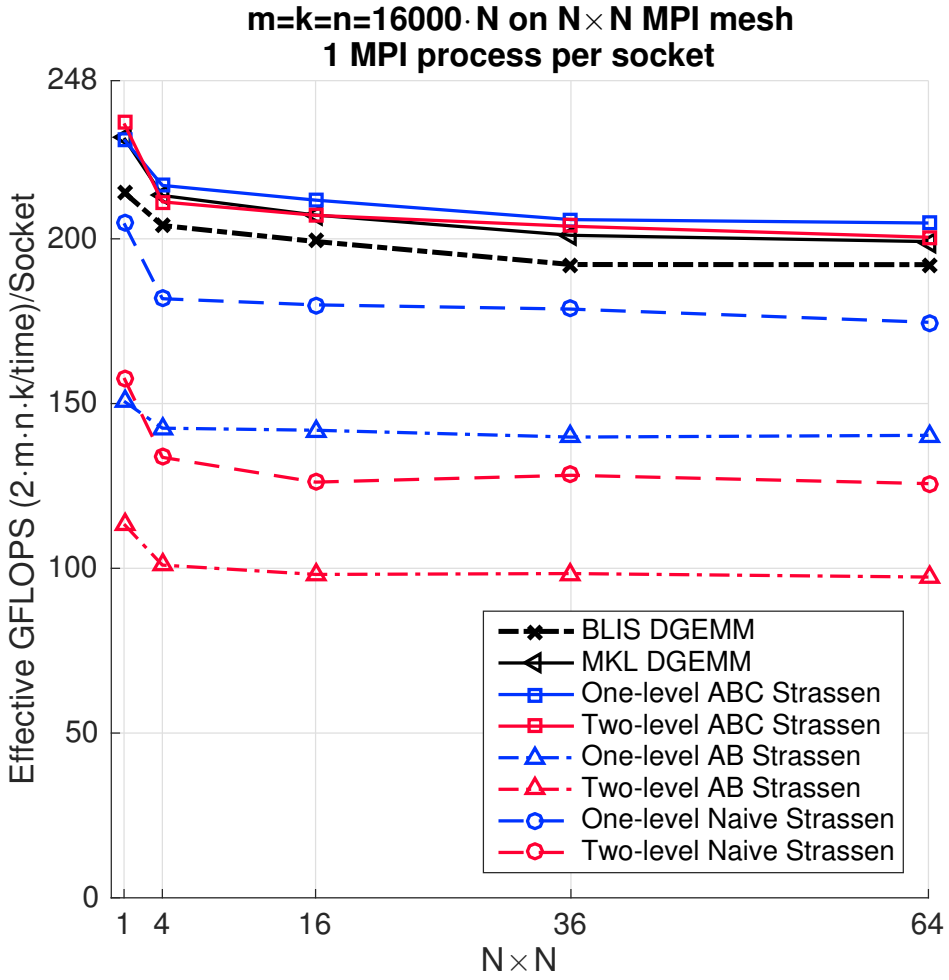


Figure 3.11: Performance of the various implementations on distributed memory (weak scalability).

implementations can be easily transferred to other algorithms that are rich in large rank-k updates.

3.4 Summary

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 GEMM as well as naive 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 to demonstrate performance benefits for single core, multi-core, many-core, and distributed memory parallel implementations. Together, this advances nearly 50 years of research into the theory and practice of STRASSEN.

Our analysis shows that the **ABC Strassen** implementation fulfills our claim that STRASSEN can outperform classical GEMM for small matrices and small k while requiring no temporary buffers beyond those already internal to high-performance GEMM 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.

Chapter 4

Practical Fast Matrix Multiplication Algorithms

Over the last half century, STRASSEN has fueled many theoretical improvements such as other variations of Strassen-like FMM algorithms. In this chapter, we extend insights on how to express multiple levels of STRASSEN in terms of Kronecker products [51] to multi-level FMM algorithms, facilitating a code generator for all FMM methods from [9] (including STRASSEN), in terms of the building blocks created in Chapter 3, 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" [77]. Performance results from single core and multi-core shared memory systems support the theoretical insights.

This chapter is based on the conference paper [55] with minor modifications: "Jianyu Huang, Leslie Rice, Devin A. Matthews, and Robert A. van de Geijn. Generating families of practical fast matrix multiplication algorithms. In *31th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2017)*, pages 656-667, May 2017." I am the main contributor in charge of problem formulation, algorithm development, performance analysis, and experimental validations.

4.1 Fast matrix multiplication basics

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

4.1.1 One-level fast matrix multiplication algorithms

In [9], 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. A $\langle \widetilde{m}, \widetilde{k}, \widetilde{n} \rangle$ FMM algorithm is defined [13, 9, 6] 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,

$$\begin{aligned} &\text{for } \; r = 0, ..., R-1, \\ &\qquad M_r := \left(\sum_{i=0}^{\widetilde{m}\widetilde{k}-1} u_{ir} A_i\right) \times \left(\sum_{j=0}^{\widetilde{k}\widetilde{n}-1} v_{jr} B_j\right); \\ &\qquad C_p \mathrel{+}= w_{pr} M_r \;\; (p = 0, ..., \widetilde{m}\widetilde{n} - 1) \end{aligned} \tag{4.1}$$

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

Figure 4.1: All operations for one-level STRASSEN. Compared to Figure 3.1, the indeces for the submatrices of A, B, and C have been changed.

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,

$$
X = \left(\begin{array}{c|c} X_0 & X_1 \\ \hline X_2 & X_3 \end{array} \right) \quad \text{for } X \in \{A, B, C\} \tag{4.2}
$$

and computations in Figure 4.1,

$$
[\![\begin{pmatrix} 1 & 0 & 1 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & -1 \end{pmatrix}, \begin{pmatrix} 1 & 1 & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & -1 & 0 & 1 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 0 & 1 & -1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & -1 & 1 & 0 & 0 & 1 & 0 \end{pmatrix}]\!] \tag{4.3}
$$

specifies $[\![U, V, W]\!]$ for one-level STRASSEN. Another example of $\langle 3, 2, 3 \rangle$ FMM algorithm can be found in Appendix C.

$\langle \widetilde{m}, \widetilde{k}, \widetilde{n} \rangle$	Ref.	$\widetilde{m}\widetilde{k}\widetilde{n}$	R	Speedup (%)				
				Theory	Practical #1		Practical #2	
					Ours	[9]	Ours	[9]
$\langle 2, 2, 2 \rangle$	[115]	8	7	14.3	11.9	-3.0	13.1	13.1
$\langle 2, 3, 2 \rangle$	[9]	12	11	9.1	5.5	-13.1	7.7	7.7
$\langle 2, 3, 4 \rangle$	[9]	24	20	20.0	11.9	-8.0	16.3	17.0
$\langle 2, 4, 3 \rangle$	[6]	24	20	20.0	4.8	-15.3	14.9	16.6
$\langle 2, 5, 2 \rangle$	[6]	20	18	11.1	1.5	-23.1	8.6	8.3
$\langle 3, 2, 2 \rangle$	[6]	12	11	9.1	7.1	-6.6	7.2	7.5
$\langle 3, 2, 3 \rangle$	[6]	18	15	20.0	14.1	-0.7	17.2	16.8
$\langle 3, 2, 4 \rangle$	[6]	24	20	20.0	11.9	-1.8	16.1	17.0
$\langle 3, 3, 2 \rangle$	[6]	18	15	20.0	11.4	-8.1	17.3	16.5
$\langle 3, 3, 3 \rangle$	[104]	27	23	17.4	8.6	-9.3	14.4	14.7
$\langle 3, 3, 6 \rangle$	[104]	54	40	35.0	-34.0	-41.6	24.2	20.1
$\langle 3, 4, 2 \rangle$	[9]	24	20	20.0	4.9	-15.7	16.0	16.8
$\langle 3, 4, 3 \rangle$	[104]	36	29	24.1	8.4	-12.6	18.1	20.1
$\langle 3, 5, 3 \rangle$	[104]	45	36	25.0	5.2	-20.6	19.1	18.9
$\langle 3, 6, 3 \rangle$	[104]	54	40	35.0	-21.6	-64.5	19.5	17.8
$\langle 4, 2, 2 \rangle$	[6]	16	14	14.3	9.4	-4.7	11.9	12.2
$\langle 4, 2, 3 \rangle$	[9]	24	20	20.0	12.1	-2.3	15.9	17.3
$\langle 4, 2, 4 \rangle$	[6]	32	26	23.1	10.4	-2.7	18.4	19.1
$\langle 4, 3, 2 \rangle$	[6]	24	20	20.0	11.3	-7.8	16.8	15.7
$\langle 4, 3, 3 \rangle$	[6]	36	29	24.1	8.1	-8.4	19.8	20.0
$\langle 4, 4, 2 \rangle$	[6]	32	26	23.1	-4.2	-18.4	17.1	18.5
$\langle 5, 2, 2 \rangle$	[6]	20	18	11.1	7.0	-6.7	8.2	8.5
$\langle 6, 3, 3 \rangle$	[104]	54	40	35.0	-33.4	-42.2	24.0	20.2

Figure 4.2: Theoretical and practical speedup for various FMM algorithms (double precision). $\widetilde{m}\widetilde{k}\widetilde{n}$ 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 algorithms compared with GEMM when $m = n = 14400, k = 480$ (rank-k updates). Practical #2 speedup is the speedup for one-level FMM algorithms compared 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 [9] (linked with Intel MKL) on single core. More details about the experiment setup is described in Section 4.3.1.

Figure 4.2 summarizes a number of such algorithms[1] that can be found in the literature that we eventually test in Section 4.3. We only consider $2 \leq \widetilde{m}, \widetilde{k}, \widetilde{n} \leq 6$ and don't include arbitrary precision approximate (APA) algorithms [12], due to their questionable numerical stability.

4.1.2 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 [41] $X \otimes Y$ 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 $(X \otimes Y)_{p \cdot r+v, q \cdot s+w} = x_{r,s}y_{v,w}$.

4.1.3 Recursive block indexing (Morton-like ordering)

An example of recursive block storage indexing (Morton-like ordering) [37] is given in Figure 4.3. In this example, 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.

[1] In Figure 4.2, the symmetric rotations (e.g., $\langle 2, 3, 4 \rangle$ vs. $\langle 2, 4, 3 \rangle$) may have different performances. 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.

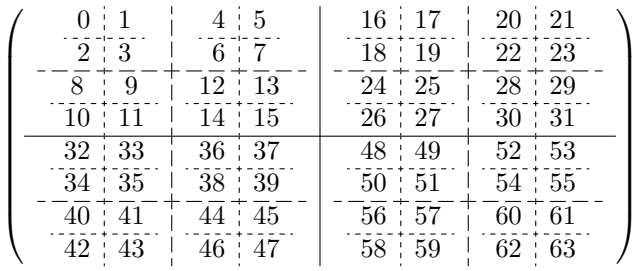


Figure 4.3: Illustration of recursive block storage indexing (Morton-like ordering) [37] on $m \times k$ matrix A where the partition dimensions $\widetilde{m} = \widetilde{k} = 2$ for three-level recursions.

4.1.4 Representing two-level FMM with the Kronecker product

In [51], it is shown that multi-level $\langle 2, 2, 2 \rangle$ STRASSEN can be represented with Kronecker products. In this section, we extend this insight to multi-level FMM algorithms, 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[2] that $C := C + AB$ is computed by,

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

[2]The complete proof is given as Theorem D.1 in Appendix D.

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 Equation (4.3).

4.1.5 Additional levels of FMM

Comparing one-level and two-level FMM algorithms, the same skeleton pattern emerges. The formula for defining L-level FMM algorithms is given by[3],

$$\begin{aligned} &\text{for } \; r = 0, ..., \textstyle\prod_{l=0}^{L-1} R_l - 1, \\ &M_r := \left(\sum_{i=0}^{\prod_{l=0}^{L-1} \widetilde{m}_l \widetilde{k}_l - 1} (\bigotimes_{l=0}^{L-1} U_l)_{i,r} A_i \right) \times \left(\sum_{j=0}^{\prod_{l=0}^{L-1} \widetilde{k}_l \widetilde{n}_l - 1} (\bigotimes_{l=0}^{L-1} V_l)_{j,r} B_j \right); \\ &C_p \mathrel{+}= (\bigotimes_{l=0}^{L-1} W_l)_{p,r} M_r (p = 0, ..., \textstyle\prod_{l=0}^{L-1} \widetilde{m}_l \widetilde{n}_l - 1) \end{aligned} \tag{4.4}$$

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 $[\![\bigotimes_{l=0}^{L-1} U_l, \bigotimes_{l=0}^{L-1} V_l, \bigotimes_{l=0}^{L-1} W_l]\!]$.

[3]The complete proof is given as Theorem D.2 in Appendix D.

4.2 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 algorithm for a matrix of given size and shape. This code generator can thus generate code for arbitrary levels of FMM algorithms that can use different FMM choices at each level. In this way, we have generated and compared more than 20 FMM algorithms (Figure 4.2).

4.2.1 Code generation

Our code generator generates various implementations of FMM algorithms with double precision arithmetic, 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 (4.1).

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 [117] code to handle the remaining "fringes" after invoking FMM, which requires no additional memory.

Generating the typical operations

To generate the code for the typical operations in (4.1), the 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 2.1 and Figure 3.2.
- Assembles a specialized micro-kernel comprised of a hand-coded optimized GEMM kernel and automatically generated updates to multiple submatrices of C.

Further variations

In Chapter 3, a number of variations on the theme illustrated in Figure 3.2 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 100 FMM implementations.[4]

4.2.2 Performance model

In this subsection, we provide a 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

[4] 23 FMM algorithms (Figure 4.2) $\times$ 3 variations $\times$ 2 levels.

can embed our performance model to guide the selection of an 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.

Assumption

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 algorithms are decomposed into three parts:

- memory packing shown in Figures 2.1 and 3.2;
- reading/writing the submatrices of C in Figures 2.1 and 3.2; and
- for **Naive FMM** and **AB FMM**, reading/writing of the temporary buffer.

Notation

Notation is summarized in Figure 4.4. The total execution time, T, is dominated by arithmetic time T_a and memory time T_m (② in Figure 4.5).

τ_a	Time (in seconds) of one arithmetic (floating point) operation.
τ_b	(Bandwidth) Amortized time (in seconds) of 8 Bytes contiguous data movement from DRAM to cache.
T	Total execution time (in seconds).
T_a	Time for arithmetic operations (in seconds).
T_m	Time for memory operations (in seconds).
$T_a^{\times}$	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. 2.1 and 3.2).
$T_m^{\tilde{A}\times}$,$T_m^{\tilde{B}\times}$	T_m for writing submatrices in packing routines (Fig. 2.1 and 3.2).
$T_m^{C\times}$	T_m for reading *and* writing submatrices in micro-kernel (Fig. 2.1 and 3.2).
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**.
N_a^X/N_m^X	Coefficient for the corresponding T_a^X/T_m^X.
$nnz(X)$	Non-zero entry number in matrix or vector X.

Figure 4.4: Notation table for performance model.

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	$T_m = N_m^{A\times} \cdot T_m^{A\times} + N_m^{B\times} \cdot T_m^{B\times} + N_m^{C\times} \cdot T_m^{C\times}$ $+N_m^{A+} \cdot T_m^{A+} + N_m^{B+} \cdot T_m^{B+} + N_m^{C+} \cdot T_m^{C+}$

Figure 4.5: The equations for computing the execution time T and *Effective* GFLOPS in our performance model.

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 4.5). 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 4.7. For instance, N_a^{A+}

	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	$mk\lceil\frac{n}{n_c}\rceil$	$\frac{m}{\widetilde{M_L}}\frac{k}{\widetilde{K_L}}\lceil\frac{n/N_L}{n_c}\rceil$
$T_m^{\widetilde{A}\times}$	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	$2\lambda mn\lceil\frac{k}{k_c}\rceil$	$2\lambda\frac{m}{\widetilde{M_L}}\frac{n}{\widetilde{N_L}}\lceil\frac{k/K_L}{k_c}\rceil$
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	$\frac{m}{\widetilde{M_L}}\frac{n}{\widetilde{N_L}}$

Figure 4.6: The various components of arithmetic and memory operations for BLAS GEMM and various implementations of FMM algorithms. The time shown in the first column for GEMM and L-level FMM algorithms can be computed separately by multiplying the parameter in τ column with the arithmetic/memory operation number in the corresponding entries.

$= nnz(\bigotimes U) - R_L$ for L-level FMM, because computing $\sum((\bigotimes U)_{i,r}A_i)$ in Equation (4.4) 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 2.1 and Figure 3.2, because the mem-

	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 4.7: The coefficient N_a^X/N_m^X mapping table for computing T_a/T_m in the performance model. Here $\widetilde{M_L} = \prod_{l=0}^{L-1} \widetilde{m_l}$, $\widetilde{K_L} = \prod_{l=0}^{L-1} \widetilde{k_l}$, $\widetilde{N_L} = \prod_{l=0}^{L-1} \widetilde{n_l}$, $\bigotimes U = \bigotimes_{l=0}^{L-1} U_l$, $\bigotimes V = \bigotimes_{l=0}^{L-1} V_l$, $\bigotimes W = \bigotimes_{l=0}^{L-1} W_l$, $R_L = \prod_{l=0}^{L-1} R_l$.

ory operation can repeat multiple times according to the loop in which they reside. T_m is broken down into several components, as shown in ④ in Figure 4.5. Each memory operation term is characterized in Figure 4.6 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. This λ is adapted to match GEMM perfor-

mance. 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 2.1 and Figure 3.2. The corresponding coefficients N_m^X are tabulated in Figure 4.7. For example, for **Naive FMM** and **AB FMM**, computing $C_p += (\bigotimes W)_{p,r} M_r (p = 0, ...)$ in Equation (4.4) involves 2 read and 1 write related to temporary buffer in slow memory. Therefore, $N_m^{C\times} = 3nnz(\bigotimes W)$.

4.2.3 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 Figures 4.5, 4.6 and 4.7. We use *effective* GFLOPS (defined in ① in Figure 4.5) as the metric to compare the performance of these various FMM implementations. The architecture-dependent parameters for the model are given in Section 4.3.1. We demonstrate the performance of two representative groups of experiments in Figures 4.8 and 4.9.

- Contrary to what was observed in Chapter 3, **Naive FMM** may perform better than **ABC FMM** and **AB FMM** for relatively large problem sizes. For example, in Figure 4.8, $\langle 3, 6, 3 \rangle$ (with the maximum theoretical speedup among all FMM algorithms we test, given in Figure 4.2) 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** and **ABC FMM**.

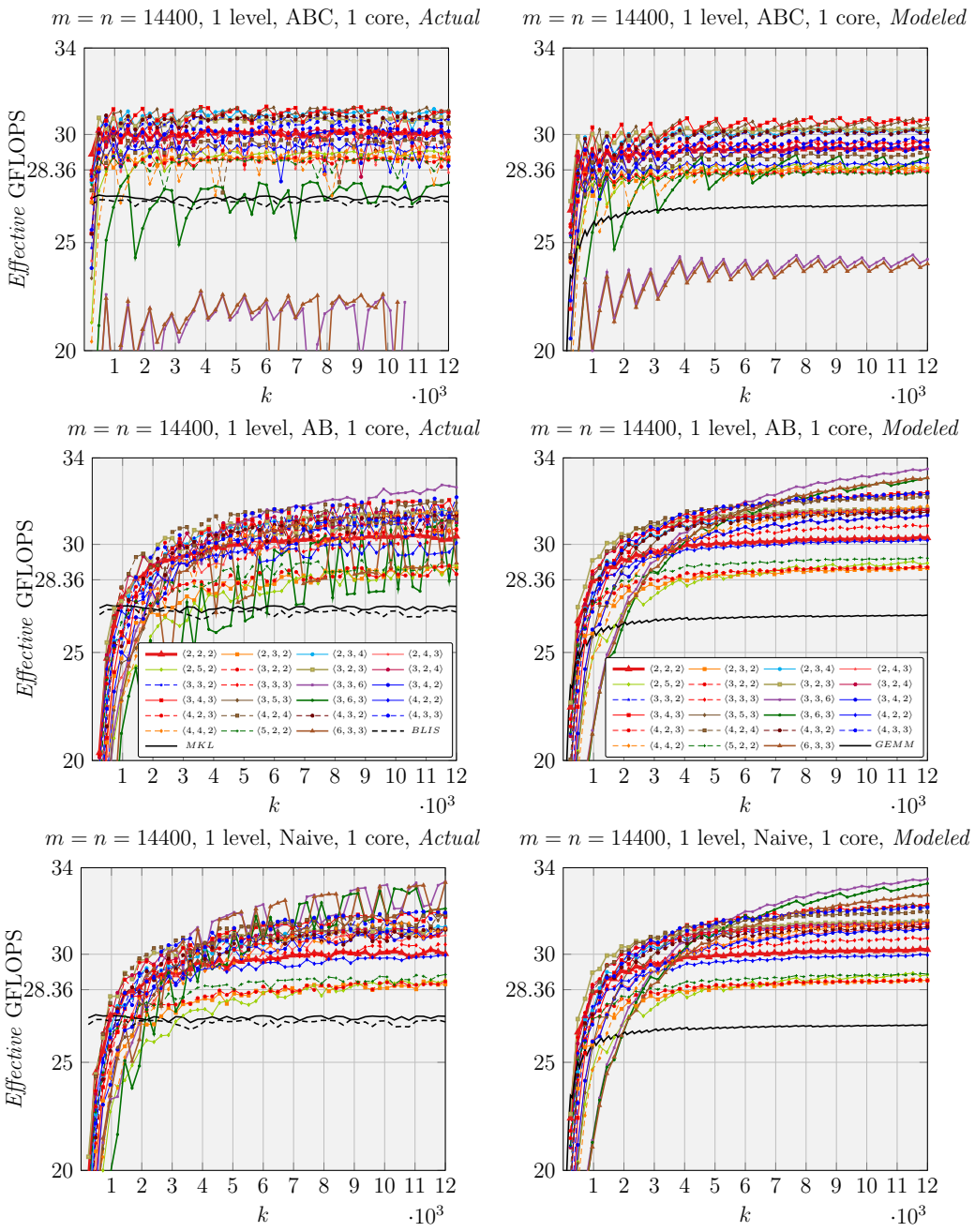


Figure 4.8: Performance of generated one-level ABC, AB, Naive FMM implementations on single core when m=n=14400, k varies. Left column: actual performance; Right column: modeled performance. Top row: one-level, ABC; Middle row: one-level, AB; Bottom row: one-level, Naive.

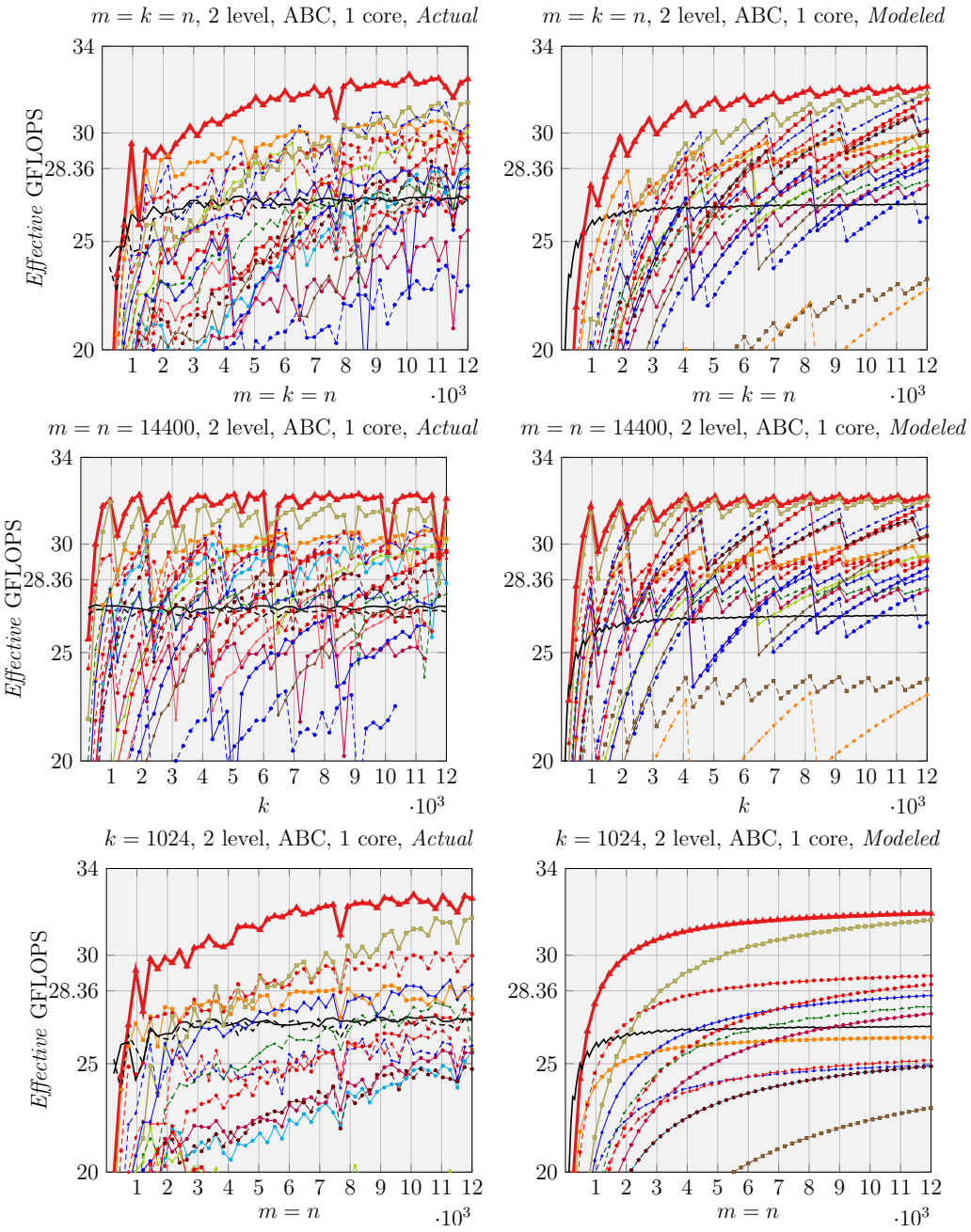


Figure 4.9: Performance of generated two-level ABC FMM implementations on single core when m=k=n; m=n=14400, k varies; k=1024, m=n vary. Left column: actual performance; Right column: modeled performance. Top row: m=k=n; Middle row: m=n=14400, k varies; Bottom row: k=1024, m=n vary.

- Contrary to what was observed in [9], for rank-k updates (middle column, right column, Figure 4.9), $\langle 2, 2, 2\rangle$ still performs the best with the **ABC FMM** implementations ([9] 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 4.8 shows that for small problem size, when k is small, **ABC FMM** performs best; when k is large, **AB FMM** and **Naive FMM** perform better. That can be quantitatively explained by comparing the coefficients of N_m^X in Figure 3.7.

- The graph for $m = n = 14400$, k varies, ABC, 1 core (left column, Figure 4.8; middle row, Figure 4.9) shows that for k equal to the appropriate multiple of k_C ($k = \prod_{l=0}^{L-1} \widetilde{k}_l \times k_C$), **ABC FMM** achieves the best performance.

4.2.4 Incorporating the performance model into the 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.

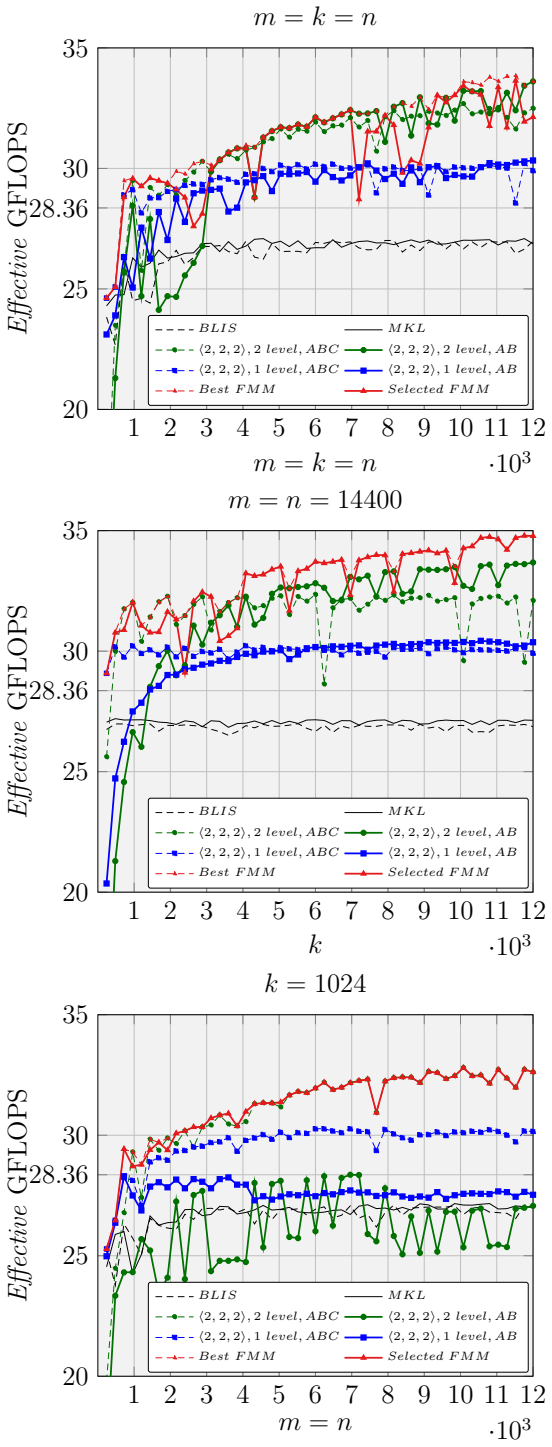


Figure 4.10: Selecting FMM implementations with the performance model.

In Figure 4.10 we show the performance results on a single core by selecting the generated FMM implementation with the guide of the 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 an exhaustive empirical search.

4.3 Performance experiments

We present performance evaluations for various generated FMM implementations.

4.3.1 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 [61] (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 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 [108], using OpenMP directives that parallelize the third loop around the micro-kernel in Figure 3.2.

4.3.2 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 4.11, when k is fixed to 1200 and $m = n$ vary. As suggested and illustrated in Section 4.2.3, **ABC FMM**

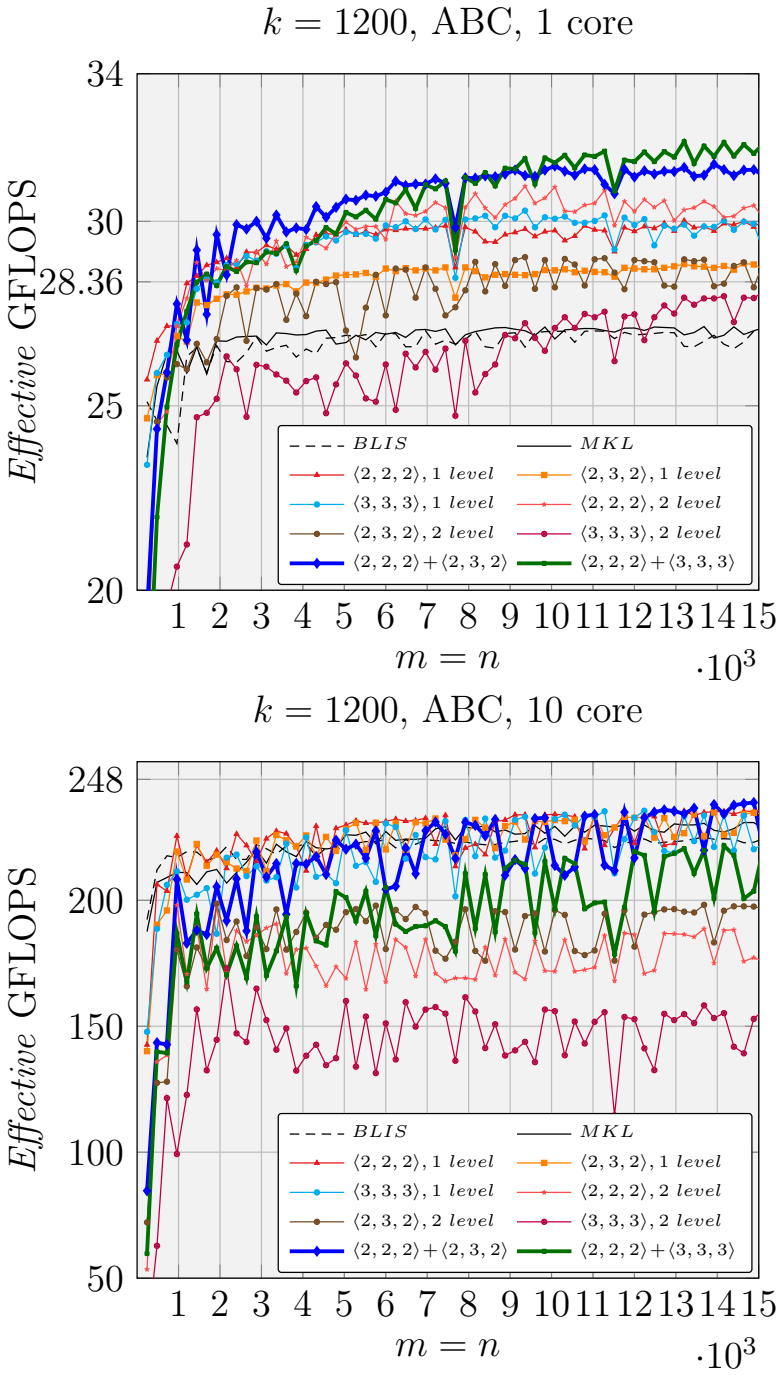


Figure 4.11: Benefit of hybrid partitions over other partitions.

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 demonstrates the benefit of hybrid partitions, facilitated by the Kronecker product representation.

4.3.3 Sequential and parallel performance

Results when using a single core are presented in Figure 4.2 and Figures 4.8 and 4.9. Our generated **ABC FMM** implementation outperforms **AB FMM** and **Naive FMM** and reference implementations from [9] for rank-k updates (when k is small). For very large square matrices, our generated **AB FMM** and **Naive FMM** can achieve competitive performance with reference implementations [9] that are linked with Intel MKL. These experiments validate our performance model.

Figure 4.12 reports performance results for ten cores within the same socket. Memory bandwidth contention impacts the performance of various FMM algorithms when using many cores. Nonetheless we still observe the speedup of FMM algorithms over GEMM. For smaller matrices and special

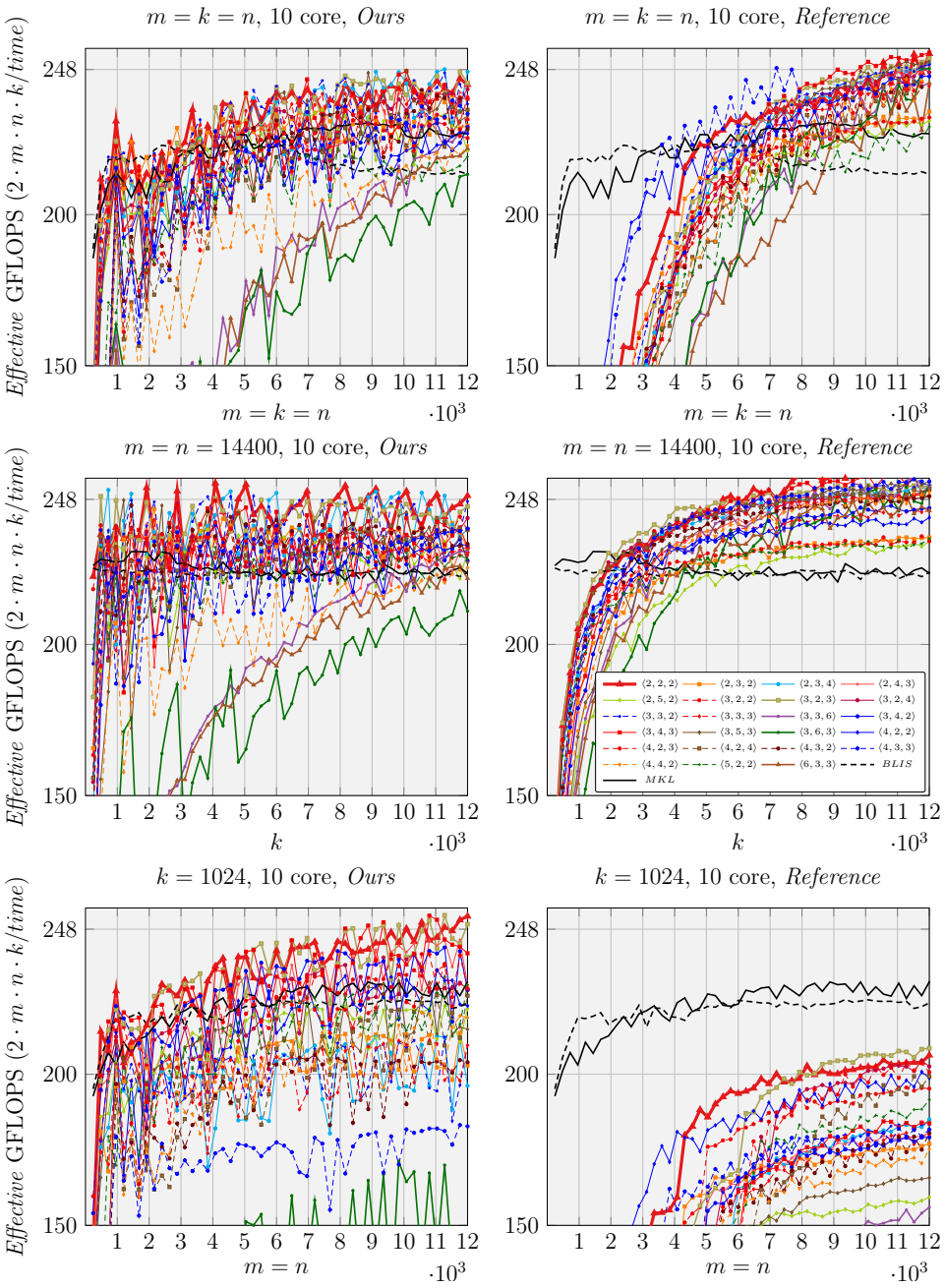


Figure 4.12: Performance of the best implementation of our generated FMM code and reference implementations [9] on one socket (10 core). Top row: our implementations; Bottom row: reference implementations from [9] (linked with Intel MKL). Left column: $m{=}k{=}n$; Middle column: $m{=}n{=}14400$, k varies; Right column: $k{=}1024$, $m{=}n$ vary.

shapes such as rank-k updates, our generated implementations achieve better performance than reference implementations [9].

4.4 Summary

We have discussed a code generator framework that can automatically implement families of Strassen-like fast matrix multiplication algorithms in a vast design space. To explore this space, automatic generation coupled with analytic performance analysis is a necessity. On the one hand, the prototype code generator generates various FMM implementations by expressesing the composition of multi-level FMM algorithms as Kronecker products. It incorporates the matrix summations that must be performed for FMM algorithms into the inherent packing and micro-kernel operations inside GEMM, avoiding extra workspace requirement and reducing the overhead of memory movement. On the other hand, 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. Compared 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.

Chapter 5

A Practical Strassen's Algorithm for Tensor Contraction

Tensor contraction (TC) is an important computational kernel widely used in numerous applications. It is a multi-dimensional generalization of matrix multiplication (GEMM). While Strassen's algorithm for GEMM is well studied in theory and practice, extending it to accelerate TC has not been previously pursued. Thus, we believe this to be the first work to demonstrate how one can in practice speed up tensor contraction with Strassen's algorithm. By adopting a block-scatter-matrix format, a novel matrix-centric tensor layout, we can conceptually view TC as GEMM for a general stride storage, with an implicit tensor-to-matrix transformation. This insight enables us to tailor the novel insights for the practical implementation of STRASSEN in Chapter 3 to TC, avoiding explicit transpositions (permutations) and extra workspace, and reducing the overhead of memory movement that is incurred. Performance benefits are demonstrated with a performance model as well as in practice on

This chapter is based on the journal paper [54] with minor modifications: "Jianyu Huang, Devin A. Matthews, and Robert A. van de Geijn. Strassen's algorithm for tensor contraction. *SIAM Journal on Scientific Computing*, 40(3):C305-C326, 2018." I am the main contributor in charge of problem formulation, algorithm development, performance analysis, and experimental validations.

modern single core, multi-core, and distributed memory parallel architectures, achieving up to 1.3× speedup. The resulting implementations can serve as a drop-in replacement for various applications with significant speedup.

In this chapter, we use the the special calligraphic font for tensors (Section 5.1.1), the upright Roman boldface letters for matrix views of tensors or normal matrices (Section 5.1.4), the normal math italic font for the notation (time, coefficients, etc.) in the performance model (Section 5.4). For example, $\mathcal{T}$ represents a tensor, $\mathbf{T}$ represents the matrix view of $\mathcal{T}$ or a normal matrix, and T represents the total execution time (in seconds) in our performance model in Section 5.4.

5.1 Background on high-performance tensor contraction

The definition and notation of tensors and tensor contraction are briefly reviewed before describing the tensor layouts that enable high-performance tensor contraction.

5.1.1 Tensor

The concept of matrices is extended to multiple dimensions through the use of tensors. For example, consider a 3-dimensional (3-D) tensor $\mathcal{T}$ of size $4 \times 6 \times 3$. $\mathcal{T}$ can be thought of as a 3-dimensional array of elements, where each element is given by indexing: $\mathcal{T}_{i,j,k} \in \mathbb{R}$. The possible values for i, j, and k are determined by the lengths of the dimensions as given in the tensor size,

i.e., $0 \leq i < N_i = 4$, $0 \leq j < N_j = 6$, and $0 \leq k < N_k = 3$.

In general, a d-dimensional tensor $\mathcal{T} \in \mathbb{R}^{N_{i_0}} \times \cdots \times \mathbb{R}^{N_{i_{d-1}}}$ has elements indexed as $\mathcal{T}_{i_0,\ldots,i_{d-1}} \in \mathbb{R}$ for all $(i_0, \ldots, i_{d-1}) \in N_{i_0} \times \ldots \times N_{i_{d-1}}$, where $M \times \ldots \times N$ is a shorthand notation for the set of all tuples $(i, \ldots, j),\ 0 \leq i < M \wedge \ldots \wedge 0 \leq j < N$. The length of the dimension indexed by some symbol x is given by $N_x \in \mathbb{N}$. The indices may be collected in an ordered *index bundle* $I_d = (i_0, \ldots, i_{d-1})$, such that $\mathcal{T}_{I_d} \in \mathbb{R}$ for all $I_d \in N_{i_0} \times \ldots \times N_{i_{d-1}}$. In general we will denote the dimension of a tensor $\mathcal{T}$ as $d_{\mathcal{T}}$, and the bundle length $N_{I_d} \in \mathbb{N}$ as the total length of an index bundle I_d, i.e., $N_{I_d} = \prod_{i \in I_d} N_i = N_{i_0} \cdot \ldots \cdot N_{i_{d-1}}$.

5.1.2 Tensor contraction

Tensor contraction is the generalization of matrix multiplication to many dimensions. As an example, consider the tensor contraction illustrated in Figure 5.1a, $\mathcal{C}_{a,b,c} += \sum_{d=0}^{N_d-1} \mathcal{A}_{d,c,a} \cdot \mathcal{B}_{d,b}$. The summation is usually suppressed and instead implied by the Einstein summation convention, where indices that appear twice (once for each of $\mathcal{A}$ and $\mathcal{B}$) on the right-hand side are summed over. In contrast to the definition of matrix multiplication in Chapter 2, tensor contraction may have more than one index summed over and more than one non-summed index in each of $\mathcal{A}$ and $\mathcal{B}$. The groups of indices that correspond to i, j, and p in the matrix case are grouped into index bundles I_m, J_n, and P_k. For this example, the bundles are (a, c), (b), and (d), respectively. Other than involving more indices, tensor contraction is precisely the same mathematical operation as matrix multiplication.

For general tensor contractions, let $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ be tensors of any dimensionality satisfying $d_{\mathcal{A}} + d_{\mathcal{B}} - d_{\mathcal{C}} = 2k,\ k \in \mathbb{N}$. Then, let I_m, J_n, and P_k be index bundles with $m = d_{\mathcal{A}} - k$ and $n = d_{\mathcal{B}} - k$. Last, let the index reordering $\Pi_{\mathcal{A}}((i_0, \ldots, i_{d_{\mathcal{A}}-1})) = (i_{\pi_{\mathcal{A}}(0)}, \ldots, i_{\pi_{\mathcal{A}}(d_{\mathcal{A}}-1)})$ be defined by the bijective map $\pi_{\mathcal{A}} \colon \{0, \ldots, d_{\mathcal{A}} - 1\} \to \{0, \ldots, d_{\mathcal{A}} - 1\}$, and similarly for $\Pi_{\mathcal{B}}$ and $\Pi_{\mathcal{C}}$. The general definition of tensor contraction is then given by

$$\mathcal{C}_{\Pi_{\mathcal{C}}(I_m J_n)} \mathrel{+}= \sum_{P_k \in N_{p_0} \times \ldots \times N_{p_{k-1}}} \mathcal{A}_{\Pi_{\mathcal{A}}(I_m P_k)} \cdot \mathcal{B}_{\Pi_{\mathcal{B}}(P_k J_n)},$$

where juxtaposition of two index bundles (e.g., $I_m J_n$) denotes concatenation. The indices in the bundles I_m and J_n are generally called *free*, *external*, or *uncontracted* indices, while the indices in the P_k bundle are called *bound*, *internal*, or *contracted* indices.[1] In the following we will suppress the explicit summation over P_k. The number of leading-order floating point operations required for tensor contraction is $2N_{I_m} \cdot N_{J_n} \cdot N_{P_k} = 2(\prod_{i \in I_m} N_i) \cdot (\prod_{j \in J_n} N_j) \cdot (\prod_{p \in P_k} N_p)$. Thus, if the length of each dimension is $O(N)$, the tensor contraction operation requires $O(N^{m+n+k})$ flops.

The example illustrated in Figure 5.1a has index bundles as noted above and index reordering given by $\Pi_{\mathcal{A}}((i_0, i_1, i_2)) = (i_2, i_1, i_0)$, $\Pi_{\mathcal{B}}((i_0, i_1)) = (i_0, i_1)$, and $\Pi_{\mathcal{C}}((i_0, i_1, i_2)) = (i_0, i_2, i_1)$. Note that, for example, defining I_m as (c, a) would give different index reorderings—the choice of ordering withing the index

[1] The preferred terms depend on context and specific field of research. In some cases, these terms have specific meaning beyond the indication of how summation is performed; for example in quantum chemistry the terms internal and external refer to the diagrammatic representation of tensor contractions [24].

bundles and the index reorderings is not unique. The number of floating point operations and memory accesses for this contraction is identical to that for a matrix multiplication of $(N_a \cdot N_c) \times N_d$, $N_d \times N_b$, and $(N_a \cdot N_c) \times N_b$ matrices, if performed entirely in-place (i.e., without transposition).

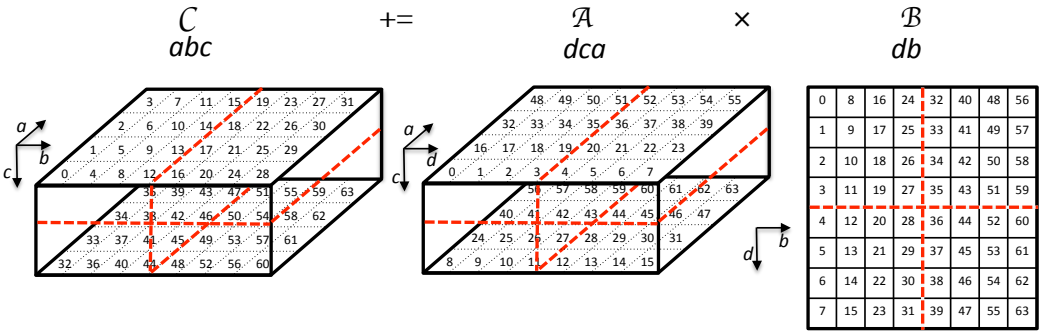
5.1.3 General stride layouts

The well-known column-major and row-major matrix layouts may be extended to tensors as the generalized column- and row-major tensor layouts, where elements are stored contiguously along the first dimension or last dimension, respectively. However, in general we may assume only a *general tensor layout*, which extends the general matrix layout [124] by replacing matrix row and column strides (e.g., $rs_{\mathbf{M}}$ and $cs_{\mathbf{M}}$) with a stride associated to each tensor dimension. For a d-dimensional tensor $\mathcal{T}$ indexed by I_d, the strides $s_{\mathcal{T};i_k} \in \mathbb{N}$ for all $0 \leq k < d$ form the set $S_{\mathcal{T}} = (s_{\mathcal{T};i_0}, \ldots, s_{\mathcal{T};i_{d-1}})$, which gives General Stride element LOCations relative to $\mathcal{T}_{0,\ldots,0}$,

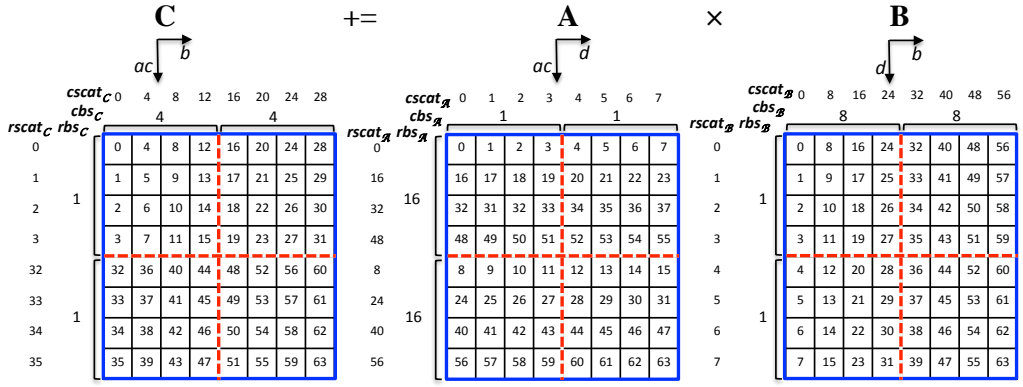
$$\mathrm{LOC}_{GS}(\mathcal{T}_{I_d}, S_{\mathcal{T}}) = \sum_{k=0}^{d-1} i_k \cdot s_{\mathcal{T};i_k}.$$

In general, the stride of the dimension indexed in $\mathcal{T}$ by a particular symbol x is denoted by $s_{\mathcal{T};x}$. The generalized column-major and row-major layouts can also be represented using a general stride layout, in which case $s_{\mathcal{T};i_k} = \prod_{l=0}^{k-1} N_{i_l}$ and $s_{\mathcal{T};i_k} = \prod_{l=k+1}^{d-1} N_{i_l}$, respectively.

In Figure 5.1a, $\mathcal{C}$ is stored in the generalized column-major layout. The numbers represent the location of the element $\mathcal{C}_{a,b,c}$ relative to the element



(a) Tensor contraction $\mathcal{C}_{a,b,c} += \mathcal{A}_{d,c,a} \cdot \mathcal{B}_{d,b}$ with $N_a = 4$, $N_b = N_d = 8$, and $N_c = 2$. The relative location of each data element in memory is given assuming a generalized column-major layout. For example, $\mathcal{C}$ is stored in the generalized column-major layout. The numbers represent the location of the element $\mathcal{C}_{a,b,c}$ relative to the element $\mathcal{C}_{0,0,0}$ in the tensor storage layout. The strides are $s_{\mathcal{C};a} = 1$, $s_{\mathcal{C};b} = N_a = 4$, and $s_{\mathcal{C};c} = N_a \cdot N_b = 32$. The element location of $\mathcal{C}_{a,b,c}$ is $a \cdot s_{\mathcal{C};a} + b \cdot s_{\mathcal{C};b} + c \cdot s_{\mathcal{C};c} = a + 4b + 32c$.



(b) Block scatter matrix view of **(a)**, where $\mathcal{A}_{d,c,a}$, $\mathcal{B}_{d,b}$, and $\mathcal{C}_{a,b,c}$ are mapped to matrices $\mathbf{A}_{i,p}$, $\mathbf{B}_{p,j}$, and $\mathbf{C}_{i,j}$: $rscat_{\mathcal{T}}$ and $cscat_{\mathcal{T}}$ denote the scatter vectors; $rbs_{\mathcal{T}}$ and $cbs_{\mathcal{T}}$ denote the block scatter vectors. Element locations are given by the sum of the row and column scatter vector entries.

Figure 5.1: An example to illustrate Strassen's algorithm for tensor contraction. The dashed lines denotes STRASSEN 2×2 partitions mapping from block scatter matrix view (bottom) to the original tensor (top). In this example the partitions are regular subtensors, but this is not required in general.

$\mathcal{C}_{0,0,0}$ in the tensor storage layout. The strides are $s_{\mathcal{C};a} = 1$, $s_{\mathcal{C};b} = N_a = 4$, and $s_{\mathcal{C};c} = N_a \cdot N_b = 32$. The element location of $\mathcal{C}_{a,b,c}$ is $a \cdot s_{\mathcal{C};a} + b \cdot s_{\mathcal{C};b} + c \cdot s_{\mathcal{C};c} = a + 4b + 32c$.

5.1.4 Block scatter matrix view

In [85] it is shown that tensors can be represented in a matrix-centric layout that allows for a simple but efficient implementation of tensor contraction using the BLIS framework. The main idea of that work is that the locations of tensor elements of $\mathcal{T}$ can be described in a matrix format, the Scatter Matrix layout, for an $N_i \times N_j$ matrix view of $\mathcal{T}$, $\mathbf{T}$, very similarly to the general stride matrix layout,

$$\mathrm{LOC}_{SM}(\mathbf{T}_{i,j}, rscat_{\mathcal{T}}, cscat_{\mathcal{T}}) = rscat_{\mathcal{T};i} + cscat_{\mathcal{T};j}, \tag{5.1}$$

where $rscat_{\mathcal{T}} \in \mathbb{N}^{N_i}$ and $cscat_{\mathcal{T}} \in \mathbb{N}^{N_j}$. If we define the index bundle I_p of size p as the set of indices of $\mathcal{T}$ that map to columns of $\mathbf{T}$ and the index bundle J_q of size $q = d_{\mathcal{T}} - p$ as the set of indices that map to rows of $\mathbf{T}$, then (by inspection of the general stride layout) we can see that the scatter vector $rscat_{\mathcal{T}}$ with respect to I_p is given by

$$rscat_{\mathcal{T};i} = \sum_{k=0}^{p-1} i_k \cdot s_{\mathcal{T};i_k},\ i = \sum_{k=0}^{p-1} i_k \cdot \prod_{l=0}^{k-1} N_{i_l},$$
$$\forall\, (i_0, \ldots, i_{p-1}) \in N_{i_0} \times \ldots \times N_{i_{p-1}};$$

and similarly for $cscat_{\mathcal{T}}$ with respect to J_q.

The relative location of $\mathcal{C}_{a,b,c}$ in Figure 5.1a, or $\mathbf{C}_{i,j}$ in the matrix view of $\mathcal{C}$ in Figure 5.1b is $rscat_{\mathcal{C};i} + cscat_{\mathcal{C};j}$ (e.g., $\mathrm{LOC}_{SM}(\mathcal{C}_{2,3,1}) = \mathrm{LOC}_{SM}(\mathbf{C}_{6,3}) =$

$rscat_{\mathcal{C};6} + cscat_{\mathcal{C};3} = 34 + 12$). Here, (1) $rscat_{\mathcal{C};i} = a \cdot s_{\mathcal{C};a} + c \cdot s_{\mathcal{C};c} = a + 32c, i = a + c \cdot N_a = a + 4c, \forall\,(a,c) \in N_a \times N_c$; (2) $cscat_{\mathcal{C};j} = b \cdot s_{\mathcal{C};b} = 4b, j = b, \forall\,(b) \in N_b$. These scatter vectors are shown on the top and the left of the matrix view of $\mathcal{C}$ in Figure 5.1b.

The general definition of tensor contractions gives a natural mapping from tensors to matrices through the index bundles I_m, J_n, and P_k. Thus, the bundle I_m defines $rscat_{\mathcal{A}}$ and $rscat_{\mathcal{C}}$, J_n defines $cscat_{\mathcal{B}}$ and $cscat_{\mathcal{C}}$, and P_k defines $cscat_{\mathcal{A}}$ and $rscat_{\mathcal{B}}$. If we define matrices $\mathbf{A}_{i,k}$, $\mathbf{B}_{k,j}$, and $\mathbf{C}_{i,j}$ and imbue them with scatter matrix layouts using the scatter vectors from the corresponding tensors, we can perform tensor contraction using the high-performance matrix multiplication algorithm introduced in Section 2.1.5, without explicitly forming those matrices in extra working buffers and incurring the associated cost of data movement.

Since we are using the BLIS implementation of the GotoBLAS algorithm, we can leverage the fact that these matrices will be partitioned and packed to introduce further optimizations. In the *micro-kernel* (Figures 2.1 and 3.2), the matrix $\mathbf{C}$ will be partitioned into $m_R \times n_R$ blocks and the matrices $\mathbf{A}$ and $\mathbf{B}$ will be partitioned into $m_R \times k_C$ and $k_C \times n_R$ slivers, respectively. If we further partition k_C into smaller increments of a new parameter k_R, on the order of m_R and n_R, then we will end up with only matrix blocks of very small size. As in [85], we can partition the scatter vectors into very small blocks of size m_R, n_R, and k_R as well and use optimized algorithms in the *packing* kernels (i.e., packing process in Section 2.1.5) and *micro-kernel* when

the scatter values for the current block are regularly spaced (i.e., strided). The regular strides for each $\{m,n,k\}_R$-sized block of $\{r,c\}scat_{\mathcal{T}}$ (m_R for $rscat_{\mathcal{A}}$ and $rscat_{\mathcal{C}}$, n_R for $cscat_{\mathcal{B}}$ and $cscat_{\mathcal{C}}$, k_R for $cscat_{\mathcal{A}}$ and $rscat_{\mathcal{B}}$), or zero if no regular stride exists, are collected in a row/column block scatter vector $\{r,c\}bs_{\mathcal{T}}$ of length $\lceil \frac{N_i}{\{m,n,k\}_R} \rceil$ and similarly for the other row/column scatter vectors. With these block scatter vectors, we can then utilize efficient `SIMD` vector load/store instructions for the stride-one index, or vector gather/scatter fetch instructions for the stride-n index, in a favorable memory access pattern.

In Figure 5.1b, assuming $m_R = n_R = k_R = 4$, $rbs_{\mathcal{C}} = (1,1)$, and $cbs_{\mathcal{C}} = (4,4)$, since the regular strides for each four elements of $rscat_{\mathcal{C}}$ and $cscat_{\mathcal{C}}$ are 1 and 4, respectively.

5.2 Strassen's algorithm for tensor contraction

The operations summarized in Figure 4.1 are all special cases of

$$\mathbf{M} = (\mathbf{X} + \delta\mathbf{Y})(\mathbf{V} + \epsilon\mathbf{W}); \quad \mathbf{D} \mathrel{+}= \gamma_0\mathbf{M}; \quad \mathbf{E} \mathrel{+}= \gamma_1\mathbf{M}; \tag{5.2}$$

for appropriately chosen $\gamma_0, \gamma_1, \delta, \epsilon \in \{-1, 0, 1\}$. Here, $\mathbf{X}$ and $\mathbf{Y}$ are submatrices of $\mathbf{A}$, $\mathbf{V}$ and $\mathbf{W}$ are submatrices of $\mathbf{B}$, and $\mathbf{D}$ and $\mathbf{E}$ are submatrices of $\mathbf{C}$. As in Chapter 3, this scheme can be extended to multiple levels of Strassen.

Instead of partitioning the tensor $\mathcal{A}$ into subtensors $\mathcal{X}$ and $\mathcal{Y}$ and so on for $\mathcal{B}$ and $\mathcal{C}$, we partition the matrix representations $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ (block scatter matrix view of $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$) as in the matrix implementation of Strassen. Figure 5.1 provides an example to illustrate the partition mechanism. Block

scatter matrix layouts for these submatrices may be trivially obtained by partitioning the scatter and block scatter vectors of the entire matrices along the relevant dimensions. Once imbued with the appropriate layouts, these submatrices may then be used in the BLIS-based STRASSEN of Chapter 3 along with modifications the packing kernels and micro-kernel as in [85].

In fusing these two methodologies, we need to further address the consideration of multiple block scatter vectors as required when packing and executing the micro-kernel. Methods for dealing with this issue are described in Section 5.3.1. The advantage of using matrix partitions (which is enabled by the block scatter layout) instead of tensor partitions is primarily that only the product of the lengths of each index bundle, $\{N_{I_m},\ N_{J_n},\ N_{P_k}\}$, must be considered when partitioning, and not the lengths of individual tensor dimensions. For example, STRASSEN may be applied to any tensor contraction where *at least* one dimension in each bundle is even in our approach, whereas the *last* dimension (or rather, the dimension with the longest stride) should be even when using subtensors.[2] Additionally, when applying techniques such as zero-padding or dynamical peeling [59, 117] in order to address edge cases, the overhead is magnified for subtensor-based algorithms because the padding or peeling applies to only a single tensor dimension; in our algorithm padding or peeling may be applied based on the length of the entire index bundle, which is necessarily longer and therefore incurs less overhead.

[2]A dimension other than the last could also be chosen for partitioning, but the spatial locality of the partitioning would be destroyed.

5.3 Implementations

We now detail the modifications to the block scatter matrix-based packing kernel and micro-kernel as described in [85] for STRASSEN. We focus on the double precision arithmetic and data.

5.3.1 Packing

When packing submatrices for STRASSEN using Equation (5.2), multiple scatter and block scatter vectors must be considered. In our implementation, the block scatter vector entries for the corresponding blocks in both input submatrices (or all submatrices for L-level STRASSEN) are examined. If *all* entries are non-zero, then the constant stride is used in packing the current block.[3] Otherwise, the scatter vectors are used when packing the current block, even though one or more of the input submatrix blocks may in fact have a regular stride. In future work, we plan to exploit these cases for further performance improvements.

5.3.2 Micro-kernel

As in Chapter 3, we use assembly-coded micro-kernels that include the update to several submatrices of **C** from registers. In order to use this efficient update, *all* block scatter vector entries for the relevant submatrix blocks of **C** must be non-zero. Unlike in the packing kernel implementation, the case

[3]Note that when non-zero, the block scatter vector entries for different submatrices will always be equal.

where only one or more of the submatrix blocks is regular stride would be more difficult to take advantage of, as the micro-kernel would have to be modified to flexibly omit or redirect individual submatrix updates.

5.3.3 Variations

We implement three variations of STRASSEN for tensor contraction on the theme illustrated in Figure 3.2, extending Chapter 3.

- **Naive Strassen**: A classical implementation with temporary buffers. Submatrices of matrix representations of $\mathcal{A}$ and $\mathcal{B}$ (**A** and **B**) are explicitly copied and stored as regular submatrices. Intermediate submatrices **M** are explicitly stored and then accumulated into submatrices of matrix representation of $\mathcal{C}$ (**C**). We store the **M** submatrices as regular, densely-stored matrices, and handle their accumulation onto block scatter matrix layout submatrices of **C**. Thus, the **Naive Strassen** algorithm for tensor contraction is extremely similar to a TTDT-based STRASSEN algorithm (see Section 5.6), except that the tensors are not required to be partitioned into regular subtensors.

- **AB Strassen**: The packing routines incorporate the summation of submatrices of matrix representations of $\mathcal{A}$ and $\mathcal{B}$ with implicit tensor-to-matrix transformation into the packing buffers (see Section 5.3.1), but explicit temporary buffers for matrices **M** are used.

- **ABC Strassen**: **AB Strassen**, but with a specialized micro-kernel (see

Section 5.3.2) that incorporates additions of $\mathbf{M}$ to multiple submatrices of matrix representation of $\mathcal{C}$ with implicit matrix-to-tensor transformation. Thus, the **ABC Strassen** algorithm for tensor contraction requires no additional temporary buffers beyond the workspace already incorporated in conventional GEMM implementations.

5.4 Performance model

In Chapter 3, a performance model was proposed to predict the execution time T for variations of STRASSEN for matrices. In this section, we extend that performance model to estimate the execution time T of **ABC**, **AB**, and **Naive** variations of L-level STRASSEN for TC and the high-performance non-STRASSEN TC routine we build on (see Section 5.1; using TBLIS implementation [85, 84] introduced in Section 5.5; denoted as TBLIS henceforth). Due to the high dimensionality of tensors and enormous types and combinations of permutations (transpositions) in TC, it is impractical to exhaustively search for every tensor shape and tensor problem size to find the best variation using empirical performance timings. Performance modeling helps us to better understand the memory footprint and computation of different STRASSEN implementations for TC and at least reduce the search space to pick the right implementation. In our new model, besides input problem size, block sizes, and the hardware parameters such as the peak GFLOPS and bandwidth, T also depends on the shape of the tensors and the extra permutations in the packing routines and in the micro-kernel. In [55] we showed that a similar

τ_a	Time (in seconds) of one arithmetic (floating point) operation.
τ_b	(Bandwidth) Amortized time (in seconds) of 8 Bytes contiguous data movement from slow main memory to fast cache.
ρ_a	Penalty factor for arithmetic operation effciency.
ρ_b	Penalty factor for bandwidth.
λ	Prefetching efficiency.
T	Total execution time (in seconds).
T_a	Time for arithmetic operations (in seconds).
T_m	Time for memory operations (in seconds).
$T_a^{\times}$	T_a for (sub)tensor contractions.
$T_a^{\mathcal{A}+}$, $T_a^{\mathcal{B}+}$, $T_a^{\mathcal{C}+}$	T_a for extra (sub)tensor addtions/permutations.
$T_m^{\mathcal{A}\times}$, $T_m^{\mathcal{B}\times}$	T_m for reading (sub)tensors in packing routines (Figure 3.3).
$T_m^{\mathcal{C}\times}$	T_m for reading *and* writing (sub)tensors in micro-kernel (Figure 3.3).
$T_m^{\mathcal{A}+}$, $T_m^{\mathcal{B}+}$, $T_m^{\mathcal{C}+}$	T_m for reading *or* writing (sub)tensors, related to the temporary buffer as part of **Naive Strassen** and **AB Strassen**.
W_a^X/W_m^X	Coefficient for the corresponding T_a^X/T_m^X.

Figure 5.2: Notation table for performance model.

①	*Effective* GFLOPS $= 2 \cdot N_{I_m} \cdot N_{J_n} \cdot N_{P_k}/T \cdot 10^{-9}$
②	$T = T_a + T_m$
③	$T_a = W_a^{\times} \cdot T_a^{\times} + W_a^{\mathcal{A}+} \cdot T_a^{\mathcal{A}+} + W_a^{\mathcal{B}+} \cdot T_a^{\mathcal{B}+} + W_a^{\mathcal{C}+} \cdot T_a^{\mathcal{C}+}$
④	$T_m = W_m^{\mathcal{A}\times} \cdot T_m^{\mathcal{A}\times} + W_m^{\mathcal{B}\times} \cdot T_m^{\mathcal{B}\times} + W_m^{\mathcal{C}\times} \cdot T_m^{\mathcal{C}\times}$ $+W_m^{\mathcal{A}+} \cdot T_m^{\mathcal{A}+} + W_m^{\mathcal{B}+} \cdot T_m^{\mathcal{B}+} + W_m^{\mathcal{C}+} \cdot T_m^{\mathcal{C}+}$
⑤	$\tau_a = 1/(\rho_a \cdot \text{Peak GFLOPS})$
⑥	$\tau_b = 8/(\rho_b \cdot \text{Bandwidth})$

Figure 5.3: Equations for computing the execution time T and *Effective* GFLOPS in our performance model.

model is capable of predicting the best-performing fast matrix multiplication algorithms from a large set of candidates in most circumstances. The same predictive power should be applicable to tensor contractions as well, over a wide range of tensor shapes and sizes.

	type	τ	TBLIS	L-level
$T_a^{\times}$	-	τ_a	$2N_{I_m}N_{J_n}N_{P_k}$	$2\frac{N_{I_m}}{\widetilde{M_L}}\frac{N_{J_n}}{\widetilde{N_L}}\frac{N_{P_k}}{\widetilde{K_L}}$
$T_a^{\mathcal{A}+}$	-	τ_a	-	$2\frac{N_{I_m}}{\widetilde{M_L}}\frac{N_{P_k}}{\widetilde{K_L}}$
$T_a^{\mathcal{B}+}$	-	τ_a	-	$2\frac{N_{P_k}}{\widetilde{K_L}}\frac{N_{J_n}}{\widetilde{N_L}}$
$T_a^{\mathcal{C}+}$	-	τ_a	-	$2\frac{N_{I_m}}{\widetilde{M_L}}\frac{N_{J_n}}{\widetilde{N_L}}$
$T_m^{\mathcal{A}\times}$	`r`	τ_b	$N_{I_m}N_{P_k}\lceil\frac{N_{J_n}}{n_c}\rceil$	$\frac{N_{I_m}}{\widetilde{M_L}}\frac{N_{P_k}}{\widetilde{K_L}}\lceil\frac{N_{J_n}/\widetilde{N_L}}{n_c}\rceil$
$T_m^{\mathcal{B}\times}$	`r`	τ_b	$N_{J_n}N_{P_k}$	$\frac{N_{J_n}}{\widetilde{N_L}}\frac{N_{P_k}}{\widetilde{K_L}}$
$T_m^{\mathcal{C}\times}$	`r/w`	τ_b	$2\lambda N_{I_m}N_{J_n}\lceil\frac{N_{P_k}}{k_c}\rceil$	$2\lambda\frac{N_{I_m}}{\widetilde{M_L}}\frac{N_{J_n}}{\widetilde{N_L}}\lceil\frac{N_{P_k}/\widetilde{K_L}}{k_c}\rceil$
$T_m^{\mathcal{A}+}$	`r/w`	τ_b	$N_{I_m}N_{P_k}$	$\frac{N_{I_m}}{\widetilde{M_L}}\frac{N_{P_k}}{\widetilde{K_L}}$
$T_m^{\mathcal{B}+}$	`r/w`	τ_b	$N_{J_n}N_{P_k}$	$\frac{N_{J_n}}{\widetilde{N_L}}\frac{N_{P_k}}{\widetilde{K_L}}$
$T_m^{\mathcal{C}+}$	`r/w`	τ_b	$N_{I_m}N_{J_n}$	$\frac{N_{I_m}}{\widetilde{M_L}}\frac{N_{J_n}}{\widetilde{N_L}}$

Figure 5.4: Various components of arithmetic and memory operations for TBLIS TC and various implementations of STRASSEN TC. The time shown in the first column for TBLIS TC and L-level STRASSEN can be computed separately by multiplying the parameter in τ column with the arithmetic/memory operation number in the corresponding entries. Here $N_{I_m} = \prod_{i\in I_m} N_i = N_{i_0}\cdot\ldots\cdot N_{i_{m-1}}$, $N_{J_n} = \prod_{j\in J_n} N_j = N_{j_0}\cdot\ldots\cdot N_{j_{n-1}}$, $N_{P_k} = \prod_{p\in P_k} N_p = N_{p_0}\cdot\ldots\cdot N_{p_{k-1}}$.

Assumption

Similar to Chapter 3, we assume two layers of memory hierarchy: slow main memory and fast caches.[4] For write operations, the lazy write-back policy is enforced such that the time for writing into fast caches can be hidden. For

[4]The latency from multiple levels of cache for modern processors is hidden by hardware prefetching. Two layers of memory are good enough for modeling performance of regular applications such as GEMM.

	TBLIS	1-level			2-level		
		ABC	AB	Naive	ABC	AB	Naive
$W_a^{\times}$	1	7	7	7	49	49	49
$W_a^{\mathcal{A}+}$	-	5	5	5	95	95	95
$W_a^{\mathcal{B}+}$	-	5	5	5	95	95	95
$W_a^{\mathcal{C}+}$	-	12	12	12	144	144	144
$W_m^{\mathcal{A}\times}$	1	12	12	7	194	194	49
$W_m^{\mathcal{B}\times}$	1	12	12	7	194	194	49
$W_m^{\mathcal{C}\times}$	1	12	7	7	144	49	49
$W_m^{\mathcal{A}+}$	-	-	-	19	-	-	293
$W_m^{\mathcal{B}+}$	-	-	-	19	-	-	293
$W_m^{\mathcal{C}+}$	-	-	36	36	-	432	432

Figure 5.5: Coefficient W_a^X/W_m^X mapping table for computing T_a^X/T_m^X in the performance model.

read operations, the latency for accessing the slow main memory is counted, while the latency for accessing caches can be ignored.[5]

Notations

We summarize our notation in Figure 5.2. The total execution time, T, can be decomposed into a sum of arithmetic time T_a and memory time T_m (② in Figure 5.3).

[5]Either because it can be overlapped with computation or because it can be amortized over sufficient computations.

Arithmetic operations

As shown in ③, T_a includes (sub)tensor contraction ($T_a^{\times}$) and (sub)tensor additions/permutations ($T_a^{\mathcal{A}+}$, $T_a^{\mathcal{B}+}$, $T_a^{\mathcal{C}+}$). The corresponding coefficients W_a^X for TBLIS TC and L-level various STRASSEN TC are enumerated in Figure 5.5. For example, one-level STRASSEN TC has coefficients $W_a^{\times} = 7$, $W_a^{\mathcal{A}+} = 5$, $W_a^{\mathcal{B}+} = 5$, and $W_a^{C+} = 12$, because it involves 7 submatrix multiplications, 5 additions with subtensors of $\mathcal{A}$, 5 additions with subtensors of $\mathcal{B}$, and 12 additions with subtensors of $\mathcal{C}$. Note that T_a^X is calculated by multiplying the unit time τ_a with the arithmetic operation number in Figure 5.4. We compute τ_a through ⑤. The penalty factor $\rho_a \in (0, 1]$ is introduced, due to the extra computations involved in $\{r, c\}scat_{\mathcal{T}}$ and $\{r, c\}bs_{\mathcal{T}}$, and the slow micro-kernel invocation when the corresponding entries in $rbs_{\mathcal{C}}$ or $cbs_{\mathcal{C}}$ are 0 (see Section 5.3.2; non-regular stride access).

Memory operations

Based on the above assumptions, T_m can be broken down into three parts (④ in Figure 5.3):

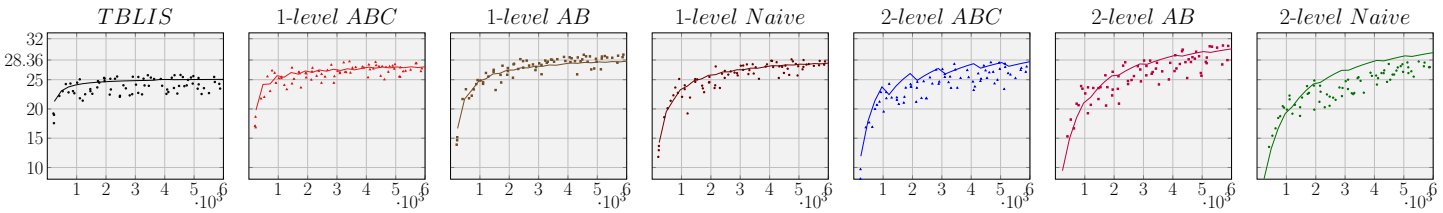
- updating the temporary buffer that are parts of **Naive Strassen** and **AB Strassen** ($W_m^{\mathcal{T}+} \cdot T_m^{\mathcal{T}+}$);
- memory packing shown in Figures 2.1 and 3.2 ($W_m^{\mathcal{A}\times} \cdot T_m^{\mathcal{A}\times}$, $W_m^{\mathcal{B}\times} \cdot T_m^{\mathcal{B}\times}$);
- updating the submatrices of C shown in Figures 2.1 and 3.2 ($W_m^{\mathcal{C}\times} \cdot T_m^{\mathcal{C}\times}$).

The coefficients W_m^X are tabulated in Figure 5.5. T_m^X is a function of block sizes $\{m_C, k_C, n_C\}$ in Figures 2.1 and 3.2, and the bundle lengths $\{ N_{I_m}/2^L, N_{J_n}/2^L, N_{P_k}/2^L \}$ because the memory operation can repeat multiple times according to which loop they reside in. Figure 5.4 characterizes each memory operation term by its read/write type and the amount of memory in units of 64-bit double precision elements. In order to get T_m^X, the memory operation number needs to be multiplied by the bandwidth τ_b. We compute τ_b through ⑥. We penalize the effect of permutations without stride-one index accesss[6] by setting $\rho_b = 0.7$. A similar parameter is introduced in [112] for regular non-STRASSEN TC. Because of the software prefetching effects, $T_m^{\mathcal{C}_\times} = 2\lambda \frac{N_{I_m}}{\widetilde{M_L}} \frac{N_{J_n}}{\widetilde{N_L}} \lceil \frac{N_{P_k}/\widetilde{K_L}}{k_c} \rceil \tau_b$ has an extra parameter $\lambda \in (0.5, 1]$, which denotes the prefetching efficiency. $T_m^{\mathcal{C}_\times}$ is a ceiling function proportional to N_{P_k}, since rank-k updates for accumulating submatrices of C recur $\lceil \frac{N_{P_k}/\widetilde{K_L}}{k_c} \rceil$ times in fourth loop (Figures 2.1 and 3.2).

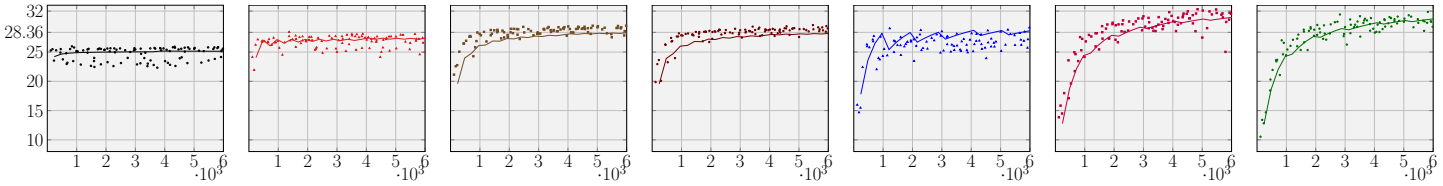
Discussion

We can estimate the run time performance of various implementations, based on the performance model presented in Figures 5.3, 5.4 and 5.5. Here we define *Effective* GFLOPS (① in Figure 5.3) for TC as the metric to compare the performance of various STRASSEN TC and TBLIS TC. The theoretical peak GFLOPS and bandwidth information are given in Section 5.5. In Figure 5.6, we demonstrate the modeled and actual performance for a wide range

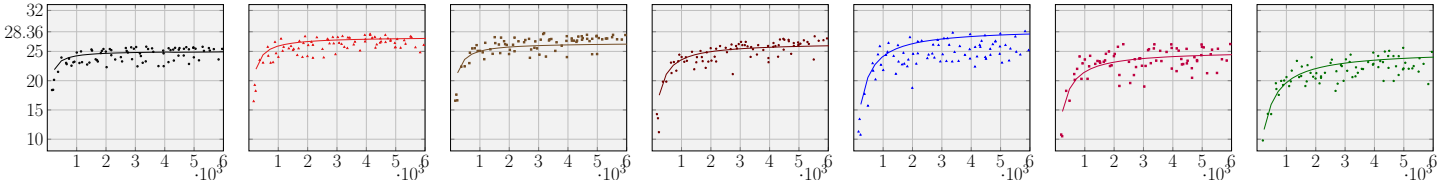
[6]See Section 5.3.1; the corresponding entries in neither $rbs_{\mathcal{T}}$ or $cbs_{\mathcal{T}}$ are 1, i.e., using scatter/gather operation, or indirect memory addressing with (5.1).



(a) m=k=n; x-axis for all figures in this row is $\widetilde{N_{I_m}} = \widetilde{N_{P_k}} = \widetilde{N_{J_n}}$.

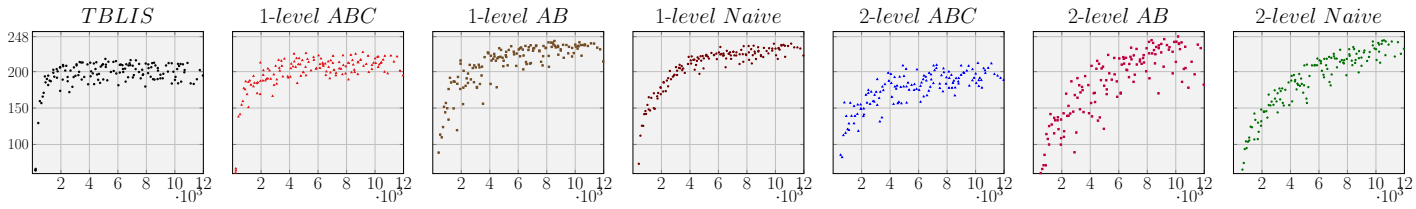


(b) m=n=14400, k varies; x-axis for all figures in this row is $\widetilde{N_{P_k}}$.

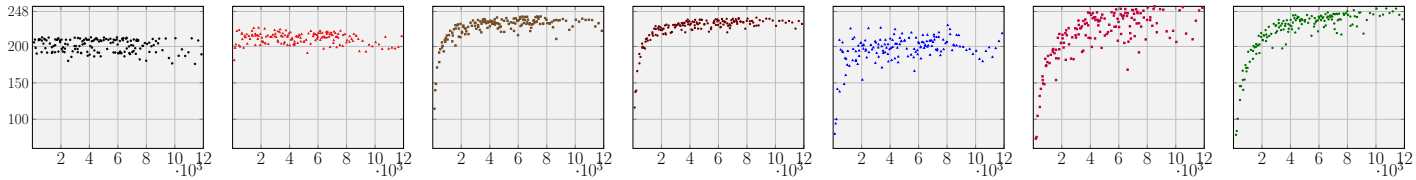


(c) k=1024, m=n vary; x-axis for all figures in this row is $\widetilde{N_{I_m}} = \widetilde{N_{J_n}}$.

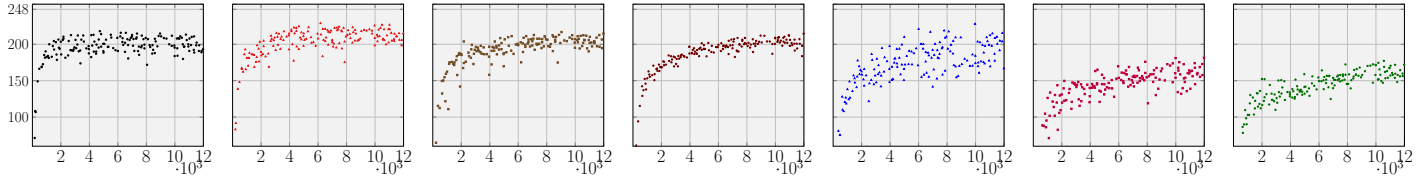
Figure 5.6: Modeled performance (solid line) and actual performance (dots) of various implementations for synthetic data on single core. y-axis for all figures is *Effective* GFLOPS ($2 \cdot N_{I_m} \cdot N_{J_n} \cdot N_{P_k}/time$).



(a) m=k=n; x-axis for all figures in this row is $\widetilde{N_{I_m}} = \widetilde{N_{P_k}} = \widetilde{N_{J_n}}$.



(b) m=n=14400, k varies; x-axis for all figures in this row is $\widetilde{N_{P_k}}$.



(c) k=1024, m=n vary; x-axis for all figures in this row is $\widetilde{N_{I_m}} = \widetilde{N_{J_n}}$.

Figure 5.7: Actual performance (dots) of various implementations for synthetic data on one socket. y-axis for all figures is *Effective* GFLOPS ($2 \cdot N_{I_m} \cdot N_{J_n} \cdot N_{P_k}/time$).

TC shapes	NRMSE (%)						
	TBLIS	1-level			2-level		
		ABC	AB	Naive	ABC	AB	Naive
m=k=n	5.26	4.27	3.23	3.49	7.82	5.64	8.65
m=n=14400, k varies	4.88	3.95	5.31	4.81	7.17	5.68	4.57
k=1024, m=n vary	4.55	4.64	5.39	5.23	9.08	7.65	7.26

Figure 5.8: Normalized root-mean-square error (NRMSE) between the actual and modeled performance for synthetic data on single core. NRMSE is defined as the root of mean square error normalized by the mean value of the measurements, which shows the model prediction accuracy.

of synthetic tensor sizes and shapes: m=k=n; m=n=14400, k varies; k=1024, m=n vary. How we generate synthetic data is detailed in Section 5.5.1. In Figure 5.8, we quantitatively show the model prediction accuracy.

- The model can predict the relative performance for various implementations within 10% error bound.

- For m=k=n (Figure 5.6a), the **ABC Strassen** implementations outperform TBLIS, when N_{I_m}, N_{J_n}, N_{P_k} are as small as $2k_C$, nearly 500, while **Naive Strassen** cannot beat TBLIS until the problem size is larger than 2000.

- The "m=n=14400, k varies" graphs (Figure 5.6b) shows that when N_{P_k} is small, **ABC Strassen** performs best; when N_{P_k} is large, **AB Strassen** performs better. The coefficients W_m^X in Figure 5.5 help to illustrate the reasons quantitatively. Two-level **AB Strassen** can achieve over 30% speedup compared with TBLIS.

- According to the model, when N_{P_k} is equal to appropriate multiple of k_C ($N_{P_k} = 2^L \cdot k_C$ for L-level), **ABC Strassen** achieves the best performance. We will leverage this observation in our distributed memory experiment.

5.5 Performance experiments

We perform our experimental evaluations for synthetic data and real-world benchmarks on a single node and on a distributed memory architecture. The implementations are written in `C++`, utilizing `AVX` assembly, based on the open source TBLIS framework [84]. We compare against TBLIS's tensor contraction routine (marked as TBLIS) as well as the TTT routine from the MATLAB Tensor Toolbox [4] (linked with Intel MKL [61], marked as TTT) for single node and the tensor contraction routine from the Cyclops Tensor Framework [110] (also linked with Intel MKL, marked with CTF) for distributed memory.

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) processor with 256 GB memory (peak bandwidth: 59.7 GB/s with four channels) and a three-level cache (32 KB L1 data; 256 KB L2; 25.6 MB L3). 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 all 10 cores are in use (24.8 GFLOPS/core peak). We disable hyper-threading explicitly and set thread affinity with `KMP_AFFINITY=compact` which also ensures the computation and the memory allocation all reside on the same socket.

The cache blocking parameters, $m_C = 96$, $n_C = 4096$, $k_C = 256$, and the register block sizes, $m_R = 8$, $n_R = 4$, are consistent with parameters used for the standard BLIS DGEMM implementation for this architecture. We use the default value of $k_R = 4$ as defined in TBLIS. This makes the size of the packing buffer $\widetilde{\mathbf{A}}_i$ 192 KB and $\widetilde{\mathbf{B}}_p$ 8192 KB, which then fits the L2 cache and the L3 cache, respectively. Parallelization is implemented mirroring that described in [108], but with the number of threads assigned to each of the loops in Figures 2.1 and 3.2 automatically determined by the TBLIS framework.

5.5.1 Single node experiments

We report the experimental results on single node for the synthetic tensor contraction, real-world benchmark, and shape dependence.

Synthetic tensor contractions

To evaluate the overall performance of various STRASSEN TC comparing against TBLIS TC for different tensor problem sizes, shapes, and permutations, we randomly generate TC test cases with 2-D to 6-D randomly permuted tensors as operands and test all these implementations for each synthetic test case, as shown in Figures 5.6 and 5.7. We choose step size 256 to sample uniformly $\{N_{I_m}, N_{J_n}, N_{P_k}\}$ for various tensor bundle lengths: *square*: m=k=n; *rank-*N_{P_k}: m=n=14400, k varies; *fixed-*N_{P_k}: k=1024, m=n vary. For each bundle length $\{N_{I_m}, N_{J_n}, N_{P_k}\}$, we randomly generate three $\{I_m, J_n, P_k\}$ 1-D, 2-D, or 3-D bundles, such that the product of each index length is close to

$\{N_{I_m}, N_{J_n}, N_{P_k}\}$. The order of $\{I_m, J_n, P_k\}$ is then randomly permuted.

The generated bundle lengths may not exactly match the original sampled bundle lengths. When we plot the actual performance of these synthetic test cases, we set effective bundle lengths $\widetilde{N_{I_m}} = \widetilde{N_{J_n}} = \widetilde{N_{P_k}} = (N_{I_m} \cdot N_{J_n} \cdot N_{P_k})^{1/3}$ for the *square* bundle lengths; $\widetilde{N_{P_k}} = N_{I_m} \cdot N_{J_n} \cdot N_{P_k}/(16000 \cdot 16000)$ for *rank-*N_{P_k} bundle lengths; and $\widetilde{N_{I_m}} = \widetilde{N_{J_n}} = (N_{I_m} \cdot N_{J_n} \cdot N_{P_k}/1024)^{1/2}$ for *fixed-*N_{P_k} bundle lengths.

For the *square* and *rank-*N_{P_k} tensor shapes on one core, TBLIS is rapidly outpaced by **ABC Strassen**, with a crossover point of about $500 \approx 2 \cdot k_C$. **ABC Strassen** is then shortly overtaken by **AB Strassen** and then by two-level **AB Strassen**. As predicted by the performance model, the **AB Strassen** implementation is best for very large problem sizes due to repeated updates to **C** in the **ABC Strassen** algorithm. The **Naive Strassen** implementations are never the best in these experiments, although they may become more efficient than **AB Strassen** for extremely large, square problems. These trends are repeated in the 10-core experiments, although the crossover points are moved to larger tensor sizes.

For the *fixed-*N_{P_k} shapes, total performance is lower for **AB Strassen** and **Naive Strassen** with scalability for the algorithms being especially impacted by the relatively smaller N_{I_m} and N_{J_n} sizes. For these shapes **ABC Strassen** is always the fastest method above the crossover point with standard TBLIS.

The actual performance data matches the predicted performance very well, with some variation due to the randomization of the tensor lengths and permutations. Using these performance models, it may be possible to analytically decide on which algorithm to apply for a given tensor contraction to achieve the highest performance, allowing an automated and seamless inclusion of STRASSEN into a TBLIS-like tensor framework.

Real-world benchmark

In Figure 5.9, we measure the performance of various implementations for a subset of tensor contractions from the Tensor Contraction Benchmark [111] on a single core and one socket. We present representative use cases where N_{P_k} is nearly equal to or larger than $2k_C$ (512), for which STRASSEN can show performance benefits, as illustrated in Section 5.4. The right three test cases represent various regularly blocked tensor contractions from coupled cluster with single and double excitations (CCSD) [103, 47, 102], a workhorse quantum chemistry computational method. The fourth case from the right illustrates the performance of TBLIS and STRASSEN TC for a pure matrix case. Comparing this case and the CCSD contractions highlights some of the performance issues that exist in the current implementation of the packing and matrix-to-block scatter matrix copy kernels (see Section 5.3.1 for details). On one core, all STRASSEN implementations improve on TBLIS for these right four cases, and in parallel one-level STRASSEN implementations give a speedup as well, exceeding TTT performance especially in the case of **AB Strassen**.

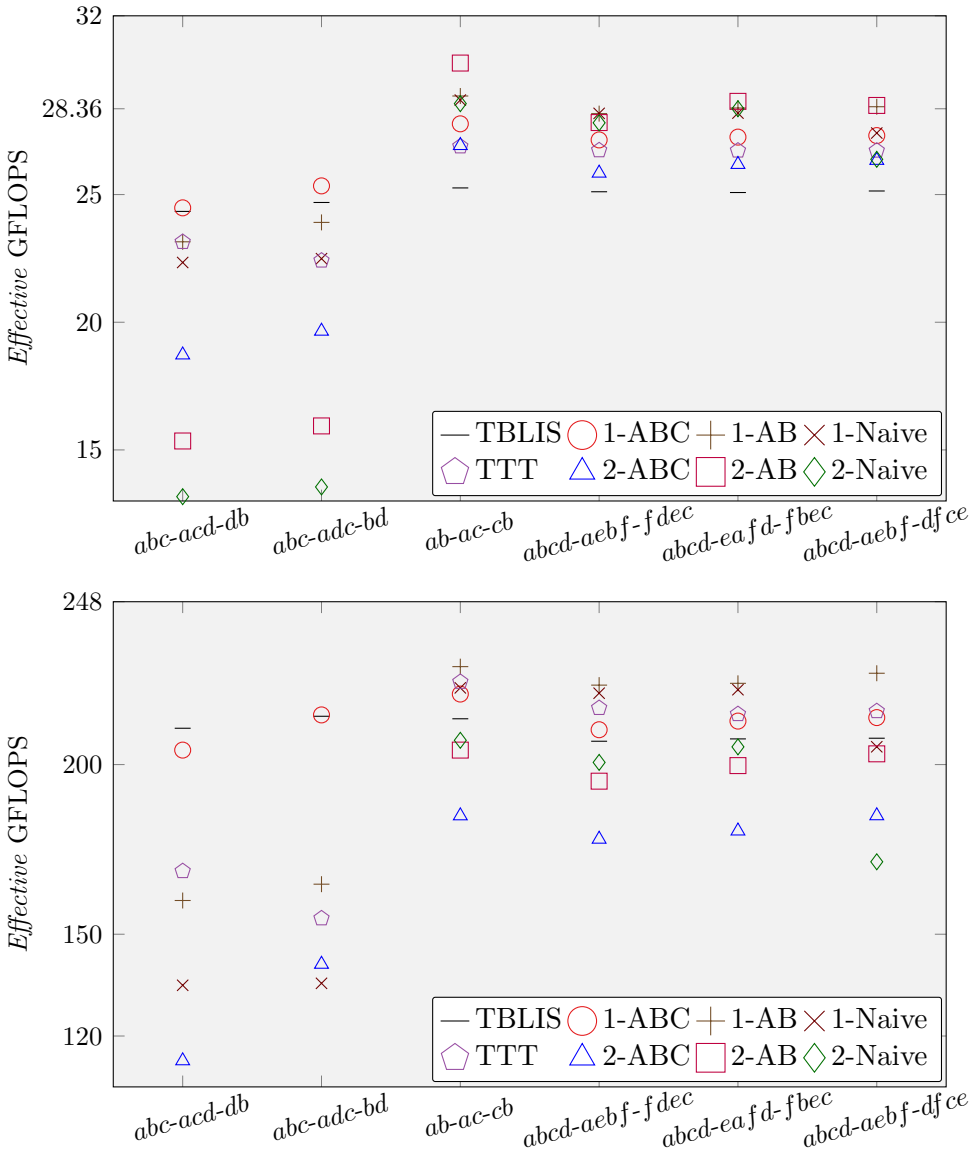


Figure 5.9: Performance for representative user cases of benchmark from [112]. TC is identified by the index string, with the tensor index bundle of each tensor in the order $\mathcal{C}$-$\mathcal{A}$-$\mathcal{B}$, e.g., $\mathcal{C}_{abcd} += \mathcal{A}_{aebf}\mathcal{B}_{dfce}$ is denoted as $abcd$-$aebf$-$dfce$. Top: performance on single core. Bottom: performance on one socket.

The gap between TBLIS and TTT for these contractions is due to TTT's use of Intel's MKL library, which is more highly optimized than the BLIS/TBLIS framework.

The left two benchmarks are again quantum chemistry applications using 3-D tensors that arise in density-fitting (DF) calculations [127, 34]. These contractions are also structurally equivalent to certain contractions from the coupled cluster with perturbative triples (CCSD(T)) method [96], where the *occupied* (see Section 5.5.2) indices have been sliced. These cases show the improvement of TBLIS over TTT as noted in [85] but do not show a speedup from STRASSEN except for one-level **ABC Strassen** on one core. Our STRASSEN implementation performs the submatrix multiplications sequentially, with only parallelization of each submatrix multiplication step. A more comprehensive parallelization scheme, for example, using task-based parallelism [9], may show better performance. Additionally, since the DF/CCSD(T) contractions are highly "non-square," an alternate fast matrix multiplication algorithm [9, 55] may perform better.

Shape-dependence experiments

The performance of the "particle-particle ladder" tensor contraction from CCSD, $\mathcal{Z}_{abij} += \mathcal{W}_{abef} \cdot \mathcal{T}_{efij}$, is reported for a range of tensor shapes in Figure 5.10. In these experiments, the length of the *virtual* dimensions $\{a, b, e, f\}$ is varied with respect to the length of the *occupied* dimensions $\{i, j\}$ such that the total number of FLOPs is roughly similarly to a 16000×16000

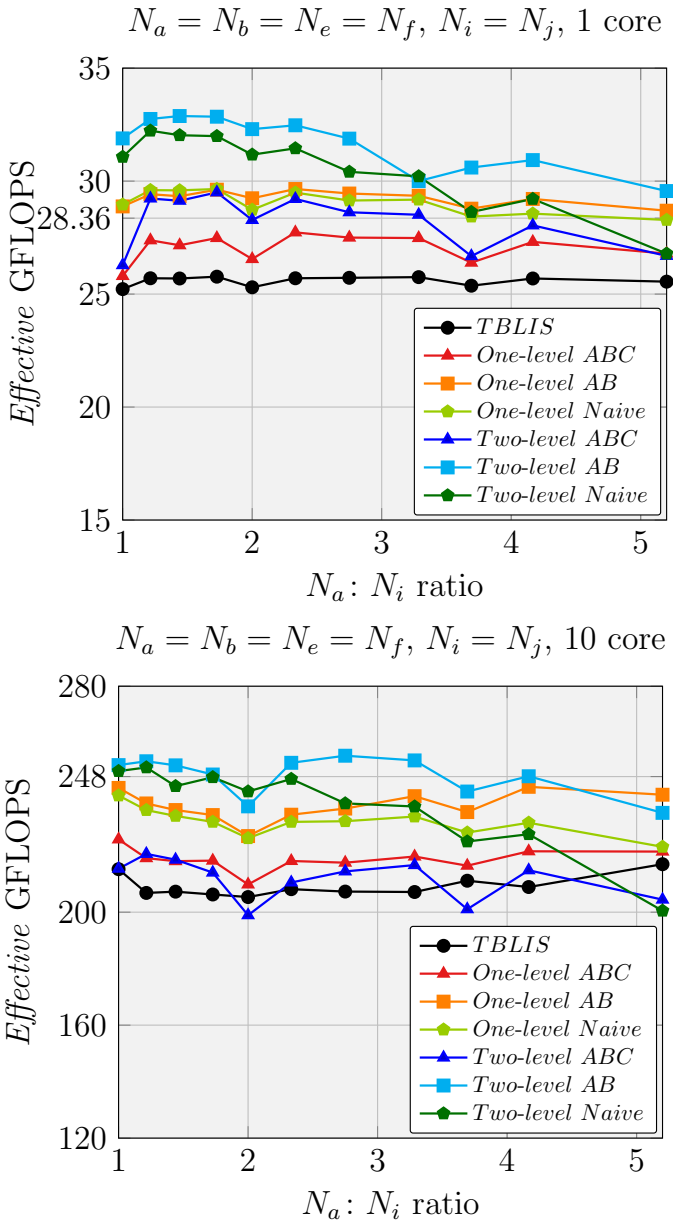


Figure 5.10: Performance for the contraction $\mathcal{Z}_{abij} \mathrel{+}= \mathcal{W}_{abef} \cdot \mathcal{T}_{efij}$ with varying N_a : N_i ratio. Left: performance on single core. Right: performance on one socket.

matrix multiplication, and the ratio $N_a : N_i$ is used as a proxy for tensor shape. A ratio of 1:1 would reflect an extremely poor quality of basis set for the overall calculation but is common when the calculation employs *regular blocking*. The other end of the scale, with a ratio of $\sim 5 : 1$, would then correspond to *uneven blocking*. This type of blocking allows for better load balancing and lower overhead when N_a and N_i are very unequal in the overall calculation.

The performance of TBLIS and all of the one-level STRASSEN algorithms shows essentially no performance degradation across the entire range tested. The two-level STRASSEN algorithms show some performance degradation at larger ratios but still show improvement over TBLIS. Eventually, all STRASSEN algorithms will cross over and perform worse than TBLIS, as evidenced by the left two contractions in Figure 5.9 (these correspond to a ratio of about 22). However, the good performance of STRASSEN out to reasonably large ratios shows that it could be beneficial in both regular blocking and uneven blocking scenarios.

5.5.2 Distributed memory experiments

We demonstrate how to use the STRASSEN TC implementations to accelerate a distributed memory implementation of 4-D tensor contraction that exemplifies the two-particle "ring" terms from CCSD. In our tests we set the length of *virtual* indices $\{a, b, e\}$ to $10\times$ that of *occupied* indices $\{i, j, m\}$, which is a ratio commonly encountered in quantum chemistry calculations using popular basis sets such as 6-311++G** [69] and cc-pVTZ [35]. The

problem sizes tested here correspond to calculations on systems with 80, 112, 160, 192, and 224 electrons (i.e., $N_i = N_j = N_m \in \{40, 56, 80, 96, 112\}$ and $N_a = N_b = N_e = 10 \cdot N_i$). We use the contraction $\mathcal{Z}_{abij} := \mathcal{W}_{bmej}\mathcal{T}_{aeim}$ as a demonstration example to show the performance benefit.

We implement a SUMMA-like [120] algorithm for 4-D tensor contraction with MPI, similar to the distributed memory implementation in Section 3.3.3. Initially the tensors $\mathcal{W}$, $\mathcal{T}$, and $\mathcal{Z}$ are distributed to a $P \times P$ mesh of MPI processes using a 2-D block distribution over the a, b, and e dimensions, with the i, j, and m dimensions stored locally (i.e., not distributed). After slicing $\mathcal{W}$ and $\mathcal{T}$ along the e dimension, the contraction is broken down into a sequence of K contractions of tensor slice pairs,

$$\mathcal{Z} \;:=\; \left(\; \mathcal{W}_{e;0} \;\middle|\; \cdots \;\middle|\; \mathcal{W}_{e;K-1} \;\right) \left(\begin{array}{c} \mathcal{T}_{e;0} \\ \hline \vdots \\ \hline \mathcal{T}_{e;K-1} \end{array}\right)$$

such that the e index length for each tensor slice pair $\{\mathcal{W}_{e;p}, \mathcal{T}_{e;p}\}$ is $N'_e = N_e/K$. For each tensor slice pair, $0 \leq p < K$, $\mathcal{W}_{e;p}$ is broadcast within rows of the mesh, and $\mathcal{T}_{e;p}$ is broadcast within columns of the mesh. Then a local tensor contration for the received tensor slice pair is performed to update the local block. Here TBLIS TC and various STRASSEN TC are used as a drop-in replacement for this local tensor contraction.

We perform the distributed memory experiment on the same machine as the single node experiment. The dual-socket processor has 10 cores on each socket. We run one MPI process for each socket and leverage all 10 cores in a

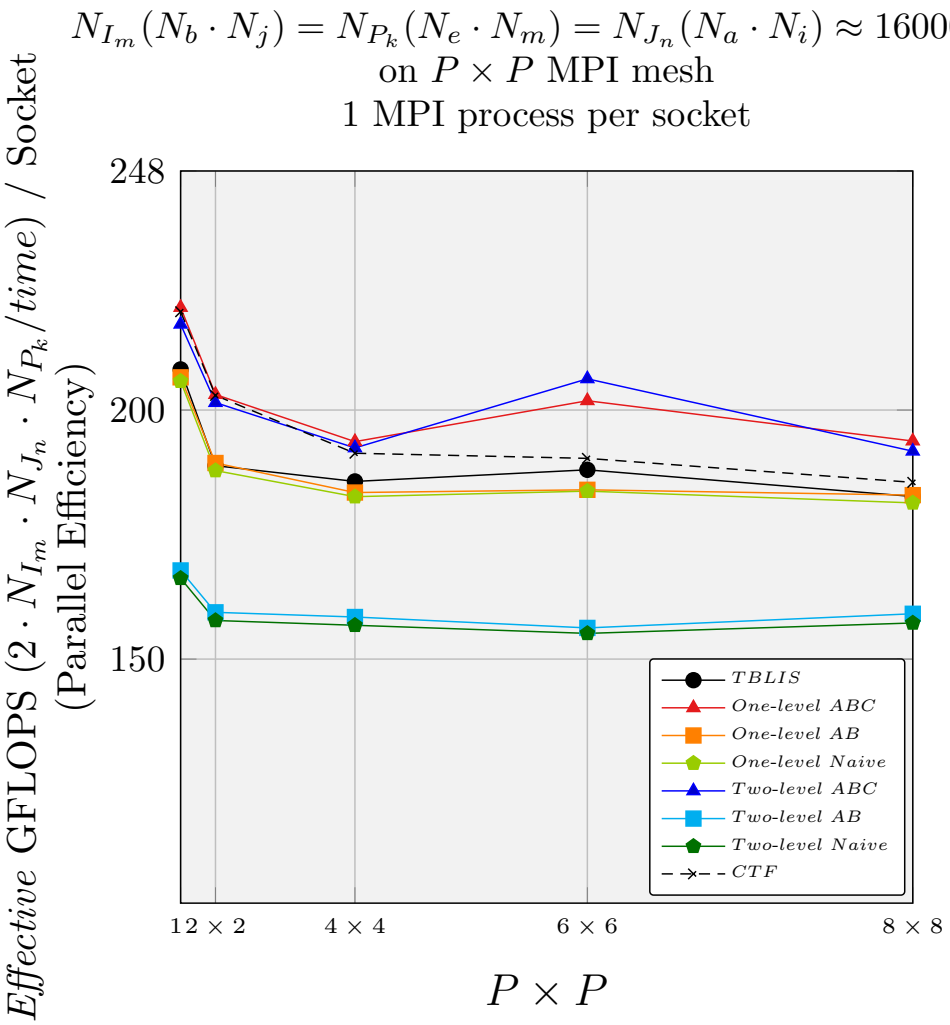


Figure 5.11: Weak scalability performance result of the various implementations for a 4-D tensor contraction CCSD application on distributed memory: $\mathcal{Z}_{abij} := \mathcal{W}_{bmej}\mathcal{T}_{aeim}$. CTF: the performance of the Cyclops Tensor Framework [110] (linked with Intel MKL).

socket with thread parallelism for all implementations. Figure 5.11 reports the weak scalability performance result on up to 640 cores (32 nodes, 64 sockets).

In our experiments on $P \times P$ mesh of sockets (MPI processes), the lengths of virtual indices are set to equal $N_a = N_b = N_e \approx 400\sqrt{P}$ and the lengths of occupied indices are set to equal $N_i = N_j = N_m \approx 40\sqrt{P}$, which make $N_{I_m} = N_{J_n} = N_{P_k} \approx 16000 \cdot P$. This guarantees the local memory buffer allocated to $\mathcal{Z}, \mathcal{W}, \mathcal{T}$ is constant. Our experiments verify that the above SUMMA-like algorithm is weakly scalable on this constant local memory setup, regardless of which local TC implementation we use. The local e index length N'_e is chosen close to $N'_e = 1024/N_m$ (i.e., $N'_e \in \{25, 18, 12, 10, 9\}$) such that the local TC computations are performed with $N_{P_k} = N'_e \cdot N_m \approx 4 \cdot k_C$. The tensor slice pairs in the local TC computations matches the shape when **ABC Strassen** achieves the best performance. Therefore, the one-level and two-level **ABC Strassen** implementations outperform all other implementations.

We also tested the Cyclops Tensor Framework (CTF) [110], which also uses a SUMMA or nested SUMMA algorithm but with possibly different block sizes and tensor distributions, as well as using the TTDT algorithm for local tensor contractions. We show it here as a reference for state-of-the-art performance.

5.6 Related work on tensor contraction

To the best of our knowledge, this work represents the first implementation of Strassen's algorithm for tensor contraction.

For tensor contraction, recent work on high-performance tensor contraction [85, 112] serves as the motivation and basis for our present work, while other research has focused on algorithms using tensor slicing [29, 94, 75, 83] or on improving the efficiency of the so-called TTDT algorithm for tensor contraction [46, 45, 82, 113], where input tensors $\mathcal{A}$ and $\mathcal{B}$ are Transposed (permuted) and then used in a standard GEMM algorithm, with the output then being Transposed and accumulated onto the tensor $\mathcal{C}$. TTDT could be used to construct a STRASSEN algorithm for TC by transposing subtensors into submatrices and vice versa and using a matrix implementation of STRASSEN instead of GEMM. However, we showed that this algorithm is essentially the same as our **Naive Strassen** algorithm (see Section 5.3.3), which is often less efficient than the other algorithms that we have implemented.

The GETT algorithm [112] is a high-performance tensor contraction implementation similar in many ways to the BLIS-based implementation by Matthews [85]. As in our current implementation, formation of linear combinations of input subtensors of $\mathcal{A}$ and $\mathcal{B}$ and output to multiple subtensors of $\mathcal{C}$ could be fused with the internal tensor transposition and micro-kernel steps of GETT. However, the implementation would be restricted to regular subtensors rather than more general submatrices (see Section 5.2), which could have possible negative performance implications (e.g., false sharing).

5.7 Summary

We have presented what we believe to be the first work to demonstrate how to leverage Strassen's algorithm for tensor contraction, and have shown practical performance speedup on single core, multi-core, and distributed memory implementations. Using a block scatter matrix layout enables us to partition the matrix view of the tensor, instead of the tensor itself, with automatic (implicit) tensor-to-matrix transformation, and the flexibility to facilitate Strassen's 2-D matrix partition to multi-dimensional tensor spaces. Fusing the matrix summation that must be performed for STRASSEN and the transposition that must be conducted for tensor contraction with the packing and micro-kernel operations inside high-performance implementation of GEMM avoids extra workspace requirements and reduces the cost of additional memory movement. We provided a performance model which can predict the speedup of the resulting family of algorithms for different tensor shapes, sizes, and permutations, with enough accuracy to reduce the search space to pick the right implementation. We evaluated our families of implementations for various tensor sizes and shapes on synthetic and real-world datasets, both observing significant speedups comparing to the baseline (TBLIS) and naive implementations (**Naive Strassen**), particularly for smaller problem sizes ($N_{I_m}, N_{J_n}, N_{P_k} \approx 2k_C, 4k_C$), and irregular shape ($N_{P_k}$ is much smaller comparing to N_{I_m}, N_{J_n}). Together, this work demonstrates Strassen's algorithm can be applied for tensor contraction with practical performance benefit.

Chapter 6

A Practical Strassen's Algorithm on GPUs

In the previous chapters, we explored how to develop practical implementations of STRASSEN, extensions to other Strassen-like FMM algorithms, and extensions to higher-dimensional tensor contraction, all targeting CPUs. In this chapter, we show how similar techniques extend the practical STRASSEN approach to GPUs. Unlike CPUs, GPUs are designed as parallel, throughput-oriented computing engines. We will show that the high-performance implementation of matrix multiplication on GPUs requires a somewhat different philosophy from that on CPUs.

Several challenges must be overcome for a practical implementation of STRASSEN on GPUs. First, the GPU architecture and programming model are different from their counterparts for a CPU. In order to achieve high performance, a practical implementation of STRASSEN needs to leverage the memory hierarchy and thread hierarchy (Section 6.1.1) of a GPU. Second, a GPU has a limited physical memory capacity. As previously discussed, conventional STRASSEN implementations require some extra temporary memory for storing intermediate submatrices, which limit the maximum problem size that can be computed compared to GEMM because of the GPU memory capacity. Third,

a GPU is a highly parallel, multi-threaded, many-core processor. STRASSEN needs to be parallelized in multiple ways to fully utilize the computational horsepower of GPUs. However, there is a tension between reducing the memory and exploiting more parallelism with the conventional implementation of STRASSEN. Finally, the ratio between the theoretical peak performance and peak memory bandwidth of a GPU is even higher (less favorable) than that of a CPU. STRASSEN has a lower ratio of arithmetic operations to memory operations compared to GEMM, which means STRASSEN only becomes advantageous when the problem sizes are sufficiently large. Thus, the practical implementation of STRASSEN needs to reduce the extra data movement to save the bandwidth and outperform GEMM for small or moderate problem sizes.

6.1 Background on Nvidia GPUs

In this section, the GPU programming model and the newest Nvidia GPUs are briefly reviewed before describing the high-performance implementation of GEMM on such GPUs.

6.1.1 GPU programming model

The CUDA programming environment [88] assumes that the CUDA program (*kernel*) is executed on physically independent *devices* (GPUs) as coprocessors to the *host* (CPU). Figure 6.1 shows the memory and thread hierarchy on the GPU device.

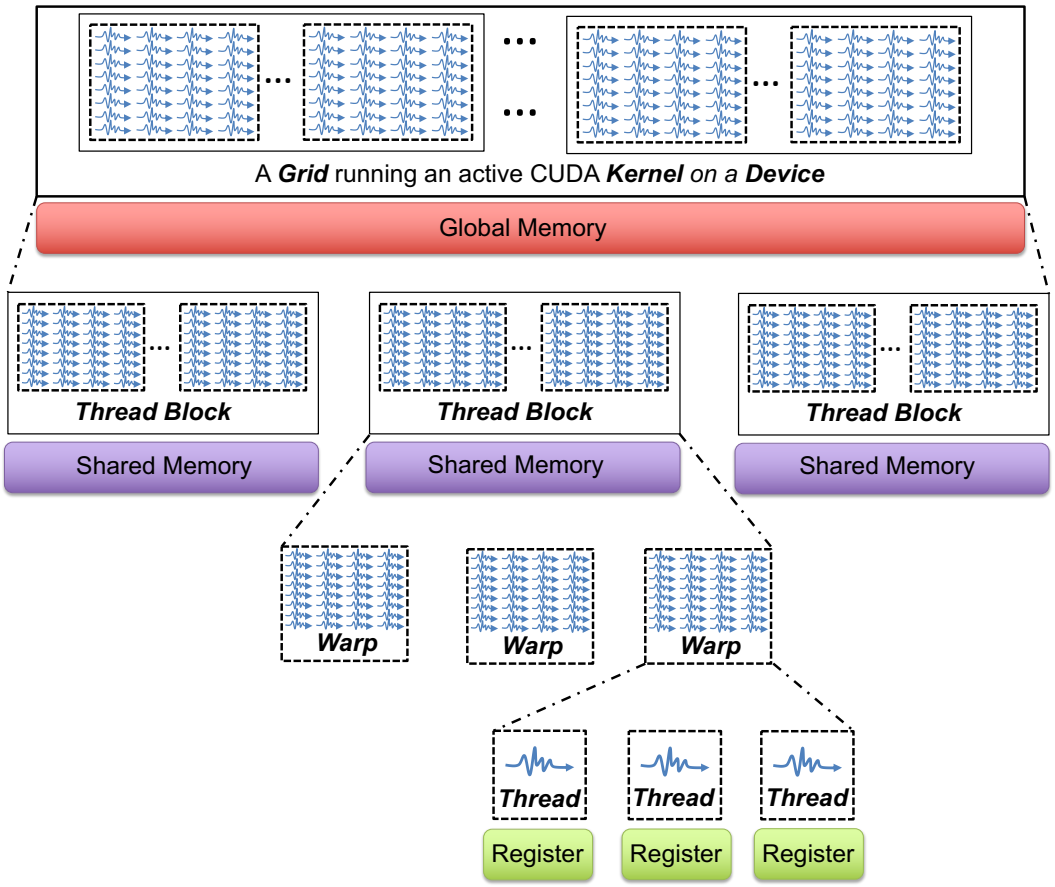


Figure 6.1: Memory and thread hierarchy in the CUDA programming environment.

Memory hierarchy

The memory hierarchy on the GPU device includes three levels: global memory, shared memory, and register files. The latency decreases while the bandwidth increases through the memory hierarchy from global memory to registers.

Thread hierarchy

A *thread* is the smallest execution unit in a CUDA program. A *thread block* is a group of threads that run on the same core and shares a partition of resources such as shared memory. Thread blocks communicate through barrier synchronization. Multiple blocks are combined to form a *grid*, which corresponds to an active CUDA kernel on the device. At runtime, a thread block is divided into a number of *warps* for execution on the cores. A warp is a set of 32 threads to execute the same instructions while operating on different data in lockstep.

6.1.2 Nvidia Volta GPUs

Recently, Nvidia released the Tesla V100 accelerator (V100) with Volta GV100 GPUs to boost high-performance computing (HPC), artificial intelligence (AI), and graphics applications [36]. It is reported that the incoming pre-exascale systems, such as the Summit and Sierra supercomputers, will be equipped with Nvidia Tesla V100 GPUs [89]. Therefore, we foresee the importance of improving the performance of V100 GPUs for the HPC community.

The Tesla V100 GPU accelerator comprises 80 streaming multiprocessors (SMs). Each SM is partitioned into 4 processing blocks. Each processing block consists of 2 Tensor Cores, 8 FP64 (double precision) cores, 16 FP32 (single precision) cores, and 16 INT32 cores, of which the last three are also called *CUDA cores*. Each Tensor Core performs matrix multiplication and accumulation operations on small matrices with a size of 4×4, achieving up to 64 floating point Fused-Multiply-Add (FMA) operations in one clock cycle. It multiplies two FP16 (half precision) matrices of a size of 4×4 and adds to the accumulation FP16 (half precision) or FP32 (single precision) matrices. The tested Tesla V100 PCIe GPU accelerator has the base clock frequency 1.254 GHz and boost clock frequency 1.38 GHz. The theoretical peak performance can reach 14.13 TFLOPS[1] with single precision and 7.065 TFLOPS[2] with double precision, while Tensor Cores can deliver 113 TFLOPS[3] for FP16/FP32 mixed precision. The tested Tesla V100 GPU is built using 16 GB HBM2 memory with 900 GB/s of bandwidth.

6.1.3 Matrix multiplication on GPUs

We review the high-performance implementation of matrix multiplication on Nvidia GPUs, based on Nvidia's CUDA Templates for Linear Algebra

[1] 1 FMA/cycle × 2 FLOP/FMA × 1.38G (boost clock frequency) × 16 (# FP32 core) × 4 (# processing block/SM) × 80 (# SM).

[2] 1 FMA/cycle × 2 FLOP/FMA × 1.38G (boost clock frequency) × 8 (# FP64 core) × 4 (# processing block/SM) × 80 (# SM).

[3] 64 FMA/cycle × 2 FLOP/FMA × 1.38G (boost clock frequency) × 2 (# Tensor Core) × 4 (# processing block/SM) × 80 (# SM).

Subroutines (CUTLASS) [67, 86], a collection of CUDA C++ templates and abstractions to perform high-performance GEMM operations. CUTLASS incorporates strategies for hierarchical partition and data movement similar to cuBLAS [87], the state-of-the-art implementation of the BLAS implementation on Nvidia GPU, and can reach more than 90% of cuBLAS performance on V100. Without loss of generality, we will focus on single precision arithmetic henceforth.

Blocking strategies

CUTLASS organizes the computation by partitioning the operands into blocks in the different levels of the device, thread block, warp, and thread. Figure 6.2 illustrates the GEMM implementation in CUTLASS.

- Device level: blocking for the thread blocks and shared memory.

  The three operand matrices, A, B, and C, are partitioned into $m_S \times k_S$, $k_S \times n_S$, and $m_S \times n_S$ blocks. Each thread block computes an $m_S \times n_S$ block of C by accumulating the results of matrix products of an $m_S \times k_S$ block of A and a $k_S \times n_S$ block of B. Therefore, the $m_S \times n_S$ block of C, the output of the thread block, is referred as the *C Accumulator*. Since the *C Accumulator* is updated many times, it needs to be lifted into the fastest memory in the SM: the register file. The global memory in which the parts of C corresponding to the *C Accumulator* only needs to be updated once after the *C Accumulator* has accumulated the results of all matrix products along the k dimension. Furthermore, to improve data locality, blocks of A

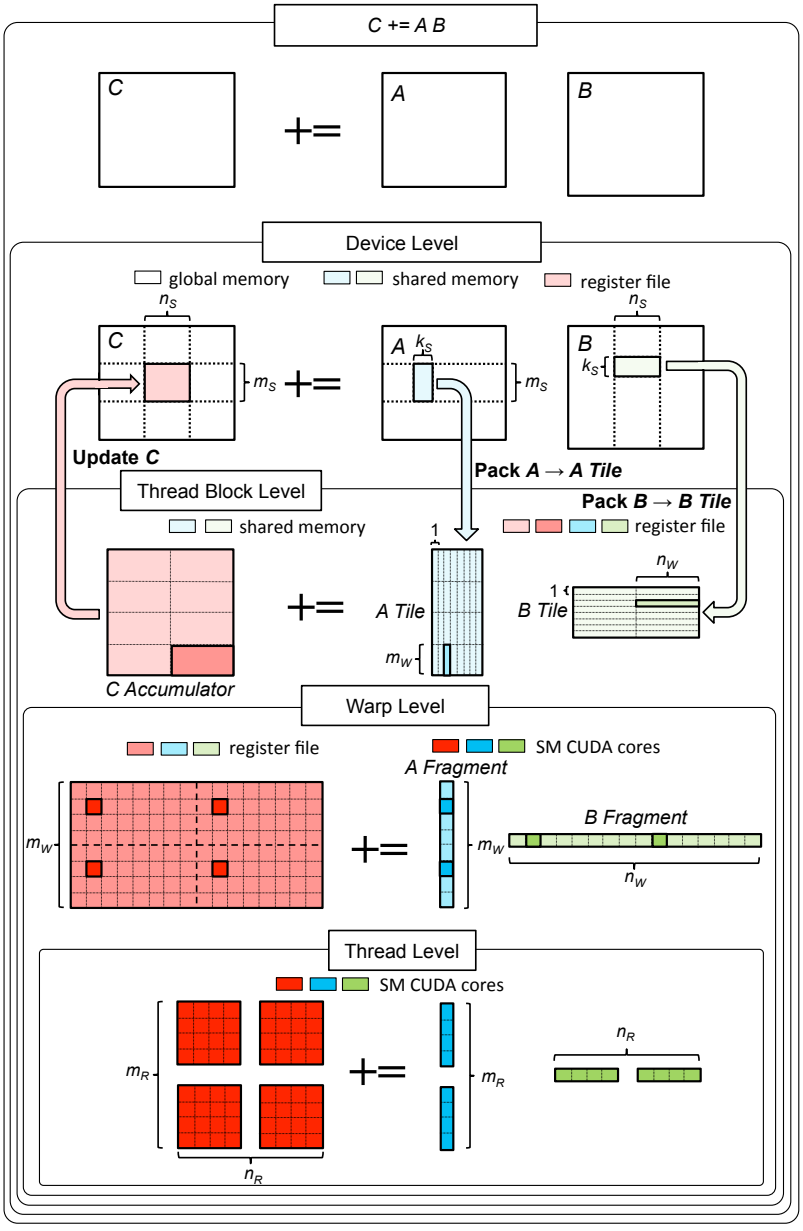


Figure 6.2: Illustration of the GEMM implementation in CUTLASS [67]. CUTLASS partitions the operand matrices into blocks in the different levels of the device, thread block, warp, and thread. Here we show block sizes typical for the large SGEMM: $m_S = 128$, $n_S = 128$, $k_S = 8$; $m_W = 4 \times m_R = 32$, $n_W = 8 \times n_R = 64$; $n_R = 8$, $n_R = 8$.

and B are "*packed*" (copied) from global memory into shared memory as the *A Tile* and *B Tile* for data reuse, accessible by all threads in the same thread block.

- Thread block level: blocking for the warps.

  After the *A Tile* and *B Tile* are stored in shared memory, each individual warp computes a sequence of accumulated outer product by iteratively loading an *A Fragment* (a column of the *A Tile* with height m_W) and a *B Fragment* (a row of the *B Tile* with width n_W) from the corresponding shared memory into register files along the k dimension and performing a rank-1 update. Note that the *C Accumulator* is spatially partitioned across all the warps within the same thread block, with each warp storing a non-overlapping 2-D block in the register files.

- Warp level: blocking for the threads.

  Each thread in a warp computes an $m_R \times n_R$ outer product with subvectors of the *A Fragment* and subvectors of the *B Fragment* in a "strip-mining" (cyclic) pattern. Each piece has a size of 4, because the largest granularity of vector load is 128 bits (4 single precision floating point numbers), and this helps to maximize the effective bandwidth. The total length of all pieces for an individual thread in m dimension is m_R, while the total length in n dimension is n_R. Since each warp has 32 threads (Section 6.1.1), CUTLASS organizes the threads within the same warp in a 4×8 or 8×4 fashion such that $m_W/m_R = 4$, $n_W/n_R = 8$, or $m_W/m_R = 8$, $n_W/n_R = 4$.

Strategy	m_S	n_S	k_S	m_R	n_R	m_W/m_R	n_W/n_R
Small	16	16	16	2	2	4	8
Medium	32	32	8	4	4	4	8
Large	64	64	8	8	8	4	8
Tall	128	32	8	8	4	8	4
Wide	32	128	8	4	8	4	8
Huge	128	128	8	8	8	4	8

Figure 6.3: CUTLASS specifies six strategies of block sizes at each level in Figure 6.2 for different matrix shapes and problem sizes.

- Thread level: executing on the CUDA cores.

  Each thread issues a sequence of independent FMA instructions to the CUDA cores and accumulates an $m_R \times n_R$ outer product.

Choices of block sizes

The block sizes at each level $\{m_S, n_S, k_S, m_R, n_R, m_W, n_W\}$ in Figure 6.2 are constrained by hardware parameters such as the register size on a SM, the maximum thread number for a thread block, the shared memory size for a SM, and the maximum register number per thread. To optimize the performance of GEMM on such a GPU, these block sizes also need to be chosen to match the maximum bandwidth of the global memory and shared memory, and the theoretical peak performance of the device. Towards this goal, CUTLASS customizes six different strategies of block sizes for different matrix shapes and problem sizes, as shown in Figure 6.3. We report the performance of these different strategies for square matrices on V100 in Figure 6.4.

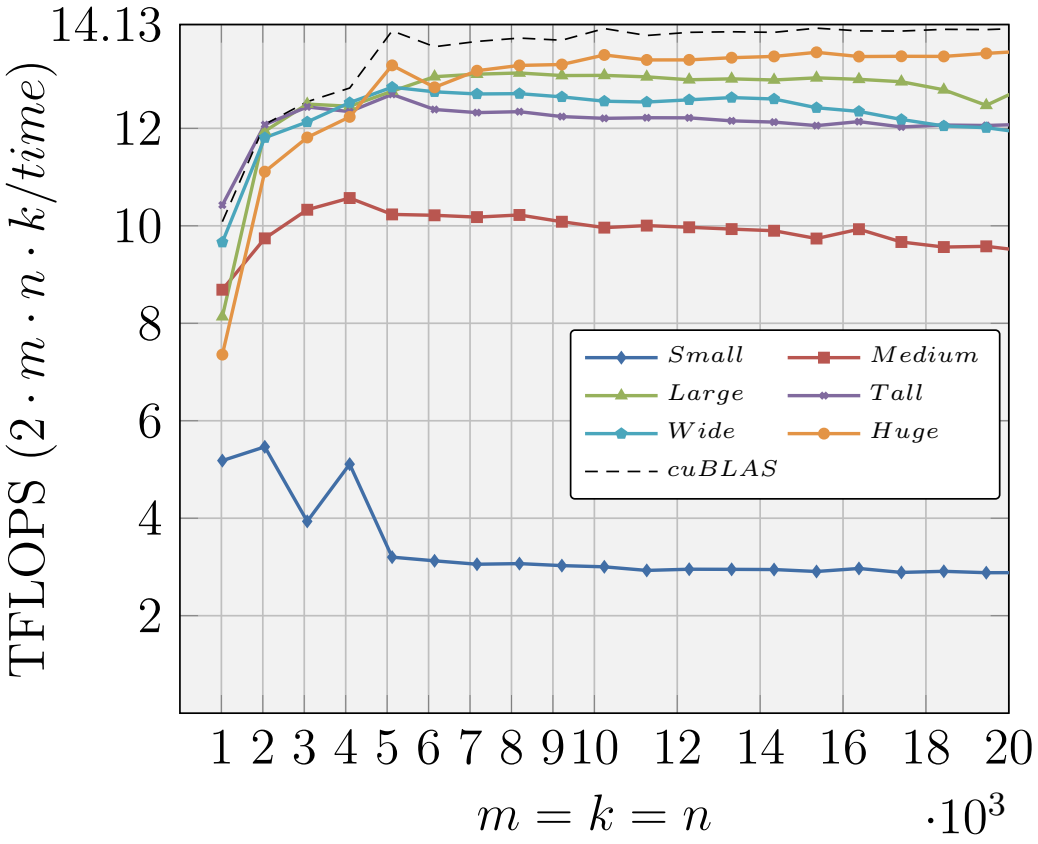


Figure 6.4: CUTLASS performance with different strategies of block sizes.

Algorithm 1 GEMM on GPUs with software pipelining.

```
//Register:  frag_A[2][m_R], frag_B[2][n_R], next_A[m_R], next_B[n_R]
//Register:  accum_C[m_R × n_R]
//Shared memory:  tile_A[k_S × m_S], tile_B[k_S × n_S]
load one m_S × k_S of A into tile_A[k_S][m_S]
load one k_S × n_S of B into tile_B[k_S][n_S]
__syncthreads()
load subvectors of first column in tile_A into frag_A[0][m_R]
load subvectors of first row in tile_B into frag_B[0][n_R]
for block_k = 0 : k_S : k do
    prefetch one subcolumn of next m_S × k_S block of A into next_A[m_R]
    prefetch one subrow of next k_S × n_S block of B into next_B[n_R]
    for warp_k = 0 : 1 : k_S do
        prefetch subvectors of next column in tile_A into frag_A[(warp_k + 1)%2][m_R]
        prefetch subvectors of next row in tile_B into frag_B[(warp_k + 1)%2][n_R]
        accum_C[m_R][n_R] += frag_A[warp_k%2][m_R]frag_B[warp_k%2][n_R]
    end for
    store next_A[m_R] into tile_A[k_S][m_S]
    store next_B[n_R] into tile_B[k_S][n_S]
    __syncthreads()
end for
update m_S × n_S block of C with accum_C[m_R][n_R]
```

Software prefetching

As shown in Algorithm 1, to keep the SM busy, CUTLASS uses global and local software prefetching to hide the data movement latency. The computations on the CUDA cores are overlapped with the data preloading from the global memory and from the shared memory. A synchronization after the data is stored to the shared memory is required to avoid the race conditions of read between warp for the next iteration.[4]

[4]CUTLASS also provides the option of double buffering on the thread block level to enable concurrent reading for the current iteration and writing for the next iteration. It eliminates the synchronization but also doubles the cost of the shared memory and the number of registers to hold the global memory fetches. On the Tesla V100 GPUs, the option of double buffering on the thread block level is disabled.

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

Figure 6.5: All operations for one-level STRASSEN. Duplicate of Figure 4.1 for easy reference.

6.2 Strassen's algorithm on Nvidia GPUs

Recall that the operations encountered in Figure 6.5 are all special cases of

$$M = (X + \delta Y)(V + \epsilon W); \quad D \mathrel{+}= \gamma_0 M; \quad E \mathrel{+}= \gamma_1 M; \tag{6.1}$$

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 D and E are submatrices of C. As in Chapter 3, this scheme can be extended to multiple levels of STRASSEN.

We will modify the GEMM implementation for GPUs illustrated in Figure 6.2 to accommodate the representative computation

$$M = (X + Y)(V + W); D \mathrel{+}= M; E \mathrel{+}= M. \tag{6.2}$$

As shown in Figure 6.6, the key insights are that we develop a specialized kernel for the representative operation (6.2) utilizing the GPU memory hierarchy and thread hierarchy:

- The summation of matrices $X + Y$ can be incorporated into the packed

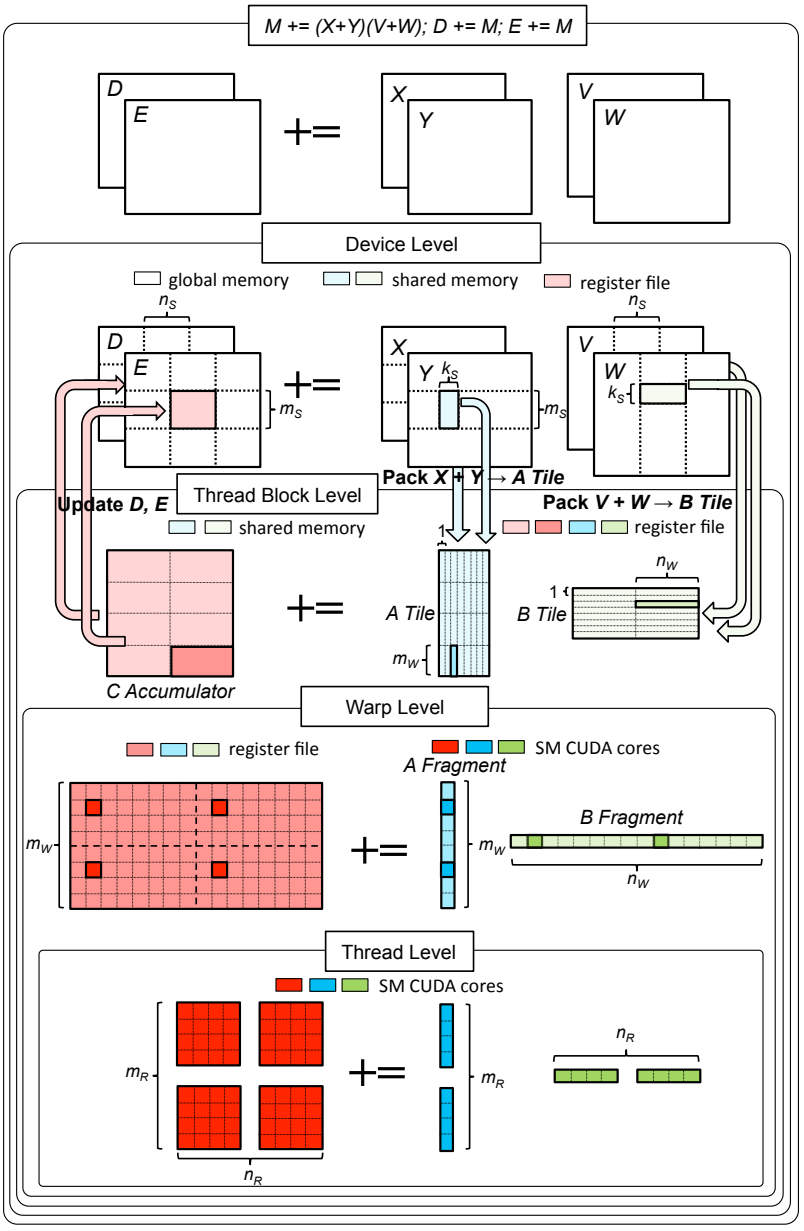


Figure 6.6: Specialized kernel that implements the representative computation $M = (X + Y)(V + W); D+= M; E+= M$ of each row of computations in Figure 6.5 based on Figure 6.2. X, Y are submatrices of A; V, W are submatrices of B; D, E are submatrices of C; M is the intermediate matrix product.

A Tile during the packing process (Section 6.1.3), avoiding the extra workspace requirement, and reducing the additional memory movement since the *A Tile* is reused for the temporary matrix sum, which is held in the shared memory.

- Similarly, the summation of matrices $V + W$ can be also incorporated into the packed *B Tile* during the packing process.

- After the *C Accumulator* has accumulated its result of $(X + Y)(V + W)$ along the k dimension, it can update the appropriate parts of D and E in the global memory once. This optimization avoids the required workspace for temporary intermediate matrices M_i and reduces the additional memory movement since the *C Accumulator* is kept in the register files: it is fetched from the global memory into the register once in the beginning, and it is written to D and E only after its computation completes.

6.3 Implementations

6.3.1 Exploiting more parallelism

A straightforward implementation of Strassen's algorithm based on our specialized kernel (Section 6.2) would invoke a sequence of kernels sequentially (7 kernels for one level, 49 kernels for two levels). This approach has already achieved the intra-kernel parallelism across the thread blocks, warps, and threads, which is utilized in the GEMM implementation on a GPU. How-

ever, it is further possible to improve concurrency by exploiting more inter-kernel parallelism. A careful observation of Figure 6.5 reveals that

- the ordering of these operations can be arbitrary;
- the dependencies between the kernels for these operations only occur for the concurrent writes to different submatrices of C.

By invoking multiple independent kernels without write dependencies to different parts of C, we can achieve inter-kernel parallelism, which is especially important for small problem sizes when there is limited intra-kernel parallelism such that each kernel cannot saturate the workload for the GPU device and for multi-level Strassen when the partitioned block size is small. We exploit the inter-kernel parallelism in the following ways:

Multi-kernel streaming

CUDA programs can manage the concurrency across kernels through *streams* [88], each of which is a sequence of commands that execute in order. While the instructions within the same stream must be scheduled in sequential order, the commands from different streams may run concurrently out of order. This helps to overlap computation with communication, as well as run multiple kernels at the same time. To ensure every command in a particular stream has finished execution, `cudaDeviceSynchronize` can be used to enforce synchronization points.

Stages	Operation		Stream
0	$M_1{=}(A_2 + A_3)B_0;$	$C_2{+}{=}\ M_1; C_3{-}{=}\ M_1;$	0
	$M_4{=}(A_0 + A_1)B_3;$	$C_1{+}{=}\ M_4; C_0{-}{=}\ M_4;$	1
	$M_5{=}(A_2 - A_0)(B_0 + B_1);$	$C_3{+}{=}\ M_5;$	0
	$M_6{=}(A_1 - A_3)(B_2 + B_3);$	$C_0{+}{=}\ M_6;$	1
1	$M_2{=}A_0(B_1 - B_3);$	$C_1{+}{=}\ M_2; C_3{+}{=}\ M_2;$	0
	$M_3{=}A_3(B_2 - B_0);$	$C_0{+}{=}\ M_3; C_2{+}{=}\ M_3;$	1
2	$M_0{=}(A_0 + A_3)(B_0 + B_3);$	$C_0{+}{=}\ M_0; C_3{+}{=}\ M_0;$	0

Figure 6.7: Reordered operations based on Figure 4.1 with multi-kernel streaming.

Figure 6.7 illustrates the multi-kernel streaming implementation with the reordered operations based on Figure 6.5. We organize the kernels for these operations into three stages with two streams. Synchronization of the streams is imposed between the different stages to guarantee the correctness of the operations because of the concurrent write dependencies to different parts of C. The kernels for these operations in different streams within the same stage can be interleaved or executed concurrently. We note that [72] also leverage multi-kernel streaming to make use of concurrency across kernels for the bottom level STRASSEN. However, it requires an additional workspace of 15 extra submatrices, while our approach doesn't require any additional workspace due to our specialized kernels.

Element-wise atomic write to C

Since the dependencies across the kernels only exist during the concurrent write to different parts of C, we can "partially" remove the dependencies and execute all operations fully concurrently in one stage with one operation

per stream by replacing the post-processing step[5] of submatrices of C with element-wise atomic write operation through `atomicAdd`. We find that this way will benefit small problem sizes. However, when the problem sizes become larger, the performance drops because element-wise atomic operations incur a greater overhead.

Atomic write to C with larger granularity

Each thread finally updates an $m_R \times n_R$ block of C, so it is possible to switch the granularity of atomic write from 1 (element-wise) to $m_R \times n_R$ (block-wise). We need to construct a device array for the purpose of a mutex/lock. Each entry of the device array corresponds to an $m_R \times n_R$ block of C. Once a thread completes the accumulation of the C *Accumulator* in the register, it will first lock the corresponding mutex/lock entries of $m_R \times n_R$ block of D and E (different submatrices of C) before updating D and E, and unlock the corresponding mutex/lock entries after the updates finish. The locks can be implemented with `atomicCAS` and `atomicExch` functions.

Reducing the multi-kernel launch overhead

With the multi-kernel streaming, we still need to launch seven kernels for one-level Strassen. For small problem sizes, the overhead for launching multiple kernels cannot be ignored. To further reduce the overhead for ker-

[5]The post-processing step involves loading each element of D and E in the global memory, adding or subtracting corresponding element in M_i residing in the C *Accumulator* in the register, and storing back to the original location of D and E in the global memory.

nel launching and avoid the possible inefficiency caused by the CUDA stream implementation, it is possible to just launch a kernel for all operations shown in Figure 6.5 with a 3-D grid of thread blocks, while the GEMM and specialized kernels in Figures 6.2 and 6.6 only require a 2-D grid. The additional z dimension for the grid (`blockIdx.z`) corresponds to different operations in Figure 6.5. The inter-kernel parallelism for multi-kernel streaming has thus transformed into one extra dimensional concurrency inside the kernel. We can utilize the atomic write with the granularity of either 1 or $m_R \times n_R$ to guarantee the correctness of the result.

6.3.2 Reducing memory requirement

The conventional implementations of STRASSEN on GPUs based on the functions provided by cuBLAS for matrix multiplication and matrix addition (i.e., `cublasSgemm, cublasSgeam`) led to the "street wisdom" that there is a trade-off between reducing the effective available memory and exploiting more parallelism. More temporary workspace for intermediate result is traditionally required to eliminate the dependency and increase the concurrency. With our approach, however, we can achieve both reduced memory and more homogeneous parallelism, similar to data parallelism except the dependencies for concurrent writes to different parts of C. We reuse the shared memory to store the temporary sum and the register file to store the temporary matrix product M_i (Section 6.2). The different operations in Figure 6.5 can be easily parallelized with the help of multi-kernel streaming or a kernel with a 3-D grid

(Section 6.3.1).

6.3.3 Handling the fringes

Traditionally, for matrices with odd dimensions, we need to handle the remaining fringes before applying STRASSEN. There are some well-known approaches such as padding (i.e., adding rows or columns with zeros to get matrices of even dimensions) and peeling (i.e., deleting rows or columns to obtain even dimensioned matrices) [59, 117] followed by post-processing. In our approach, fringes can be internally handled by padding the A *Tile* and B *Tile* with zeros, and aligning the $m_C \times n_C$ C *Accumulator* along the fringes. This trick avoids the handling of the fringes with extra memory or computations because the packing and accumulation processes always occur for high-performance implementation of GEMM on GPUs, and we reuse the same buffers.

6.3.4 Adapting software prefetching

As illustrated in Algorithm 2, instead of prefetching one $m_S \times k_S$ block of A and one $k_S \times n_S$ block of B, STRASSEN requires the preloading of two $m_S \times k_S$ blocks of X and Y and two $k_S \times n_S$ blocks of V and W. The required register number per thread has since doubled, but the required sizes of shared memory and global memory remain the same.

Algorithm 2 $M = (X + Y)(V + W); D+= M; E+= M$ on GPUs with software prefetching

```
//Register:  frag_A[2][m_R], frag_B[2][n_R]
//Register:  next0_A[m_R], next1_A[m_R], next0_B[n_R], next1_B[n_R]
//Register:  accum_C[m_R × n_R]
//Shared memory:  tile_A[k_S × m_S], tile_B[k_S × n_S]
load the sum of one m_S × k_S of X and corresponding m_S × k_S of Y into tile_A[k_S][m_S]
load the sum of one k_S × n_S of V and corresponding k_S × n_S of W into tile_B[k_S][n_S]
__syncthreads()
load subvectors of first column in tile_A into frag_A[0][m_R]
load subvectors of first row in tile_B into frag_B[0][n_R]
for block_k = 0 : k_S : k do
    prefetch one subcolumn of next m_S × k_S block of X into next0_A[m_R]
    prefetch one subcolumn of next m_S × k_S block of Y into next1_A[m_R]
    prefetch one subrow of next k_S × n_S block of V into next0_B[n_R]
    prefetch one subrow of next k_S × n_S block of W into next1_B[n_R]
    for warp_k = 0 : 1 : k_S do
        prefetch subvectors of next column in tile_A into frag_A[(warp_k + 1)%2][m_R]
        prefetch subvectors of next row in tile_B into frag_B[(warp_k + 1)%2][n_R]
        accum_C[m_R][n_R] += frag_A[warp_k%2][m_R]frag_B[warp_k%2][n_R]
    end for
    store next0_A[m_R] + next1_A[m_R] into tile_A[k_S][m_S]
    store next0_B[n_R] + next1_B[n_R] into tile_B[k_S][n_S]
    __syncthreads()
end for
update m_S × n_S block of D with accum_C[m_R][n_R]
update m_S × n_S block of E with accum_C[m_R][n_R]
```

6.4 Performance experiments

Experimental setup

We perform our experiments on a Tesla V100 PCIe accelerator which is connected to an Intel Xeon Gold 6132 Skylake server. The Operating System is CentOS Linux version 7.4.1708. We use CUDA Driver/Runtime Version 9.1/9.0 with CUDA Capability 7.0 and cuBLAS with version 9.0. The GNU compiler version for compiling the host code is 6.4.0. The nvcc compiler flags `-O3 -Xptxas -v -std=c++11 -gencode arch=compute_70,code=sm_70` are

used. As presented in Section 6.1.2, the tested Tesla V100 PCIe accelerator has a theoretical peak performance of 14.13 TFLOPS with single precision.

Measurement

We report the single precision floating point results using square matrices with the size of $m = k = n$ for each dimension. To time the CUDA execution of kernels running on the GPU, we use the CUDA events that have a resolution of approximately half a microsecond. As before, we take *Effective* TFLOPS as the main metric to compare the performance of various implementations.

$$\textit{Effective}\ \text{TFLOPS} = \frac{2 \cdot m \cdot n \cdot k}{\text{time (in seconds)}} \cdot 10^{-12}. \tag{6.3}$$

CUTLASS and our implementations of Strassen based on CUTLASS are tested with different strategies of block sizes to select the highest performing setup.

Result

Figure 6.8 reports the single precision floating point performance of cuBLAS, CUTLASS, and various Strassen implementations on V100. The 1-level and 2-level reference implementations [72] are linked with cuBLAS 9.0, with the operations in Strassen restructured to have only two temporary matrices.

For the 2-level hybrid implementation, we use reference implementation

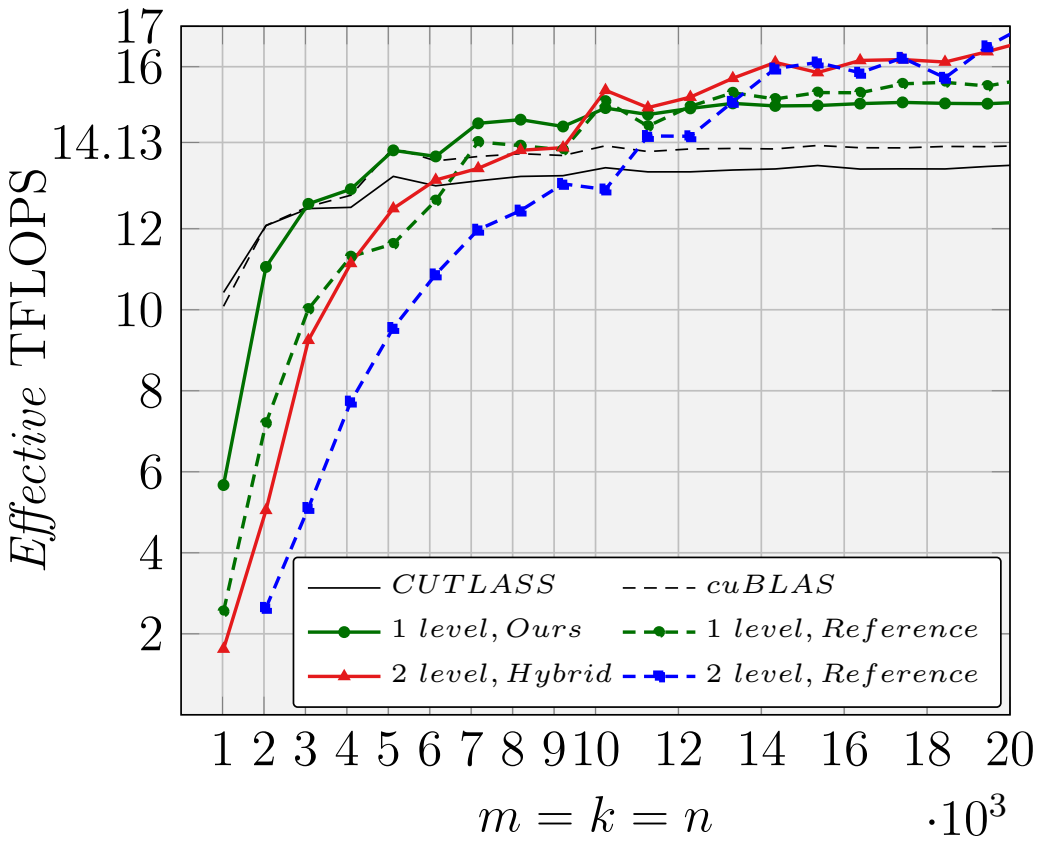


Figure 6.8: Performance of various STRASSEN implementations on V100 with single precision. The theoretical peak for the machine is 14.13 TFLOPS. The 1-level and 2-level reference implementations are from [72] (linked with cuBLAS 9.0). The 2-level hybrid implementation replaces the cublasSgemm function in the 1-level reference implementation [72] with our 1-level STRASSEN implementation.

in the top level, and our 1-level implementation in the bottom level.[6]

By comparing the performance of various implementations, we make the following observations:

- For 1-level, our implementation of STRASSEN outperforms CUTLASS and cuBLAS when the problem sizes $m = k = n$ are as small as 3000. The reference implementation cannot get the comparable performance with our implementation until the problem sizes are larger than 10000.

- For 2-level, the reference implementation cannot beat the hybrid implementation until the problem size is larger than 15000.

- Our implementation has the same memory consumption as CUTLASS, while the 1-level reference implementation consumes much more memory.

- Our 1-level STRASSEN implementation and 2-level hybrid implementation achieve the best performance over the entire spectrum of problem sizes compared to the reference implementation, with no or less additional memory consumption.

[6]We also tried to extend the insights from Figure 6.6 to 2-level STRASSEN implementation. However, it requires up to four times more registers per thread compared to GEMM. Due to the capacity of the register number per thread on GPUs, we cannot get a reasonable performance.

6.5 Summary

We have presented a practical implementation of Strassen's algorithm on GPUs, which outperforms the state-of-the-art implementation on small problem sizes and consumes no additional memory compared to GEMM. By developing a specialized kernel, we utilized the GPU memory hierarchy and thread hierarchy. By reusing the shared memory to store the temporary matrix sum during the packing process and the register files to hold the temporary matrix product during the accumulation process, we avoided the extra workspace requirement and reduced the additional memory movement. Besides the intra-kernel parallelism across the thread blocks, warps, and threads similar to GEMM implementation on GPUs, we also exploited the inter-kernel parallelism with multi-kernel streaming, atomic write with different granularity, and launching a kernel with a 3-D grid to reduce the multi-kernel launch overhead. The fringes can be handled internally during the packing and accumulation process. We also leveraged the software prefetching to hide the latency of data movement across the memory hierarchy. Together, we achieved both less memory and more parallelism with our customized kernels.

Chapter 7

Conclusion

In this dissertation, we explored practical implementations of Strassen's algorithm and other Strassen-like fast matrix multiplication algorithms on CPU and GPU architectures. The conventional implementations of these Strassen and similar Strassen-like FMM algorithms led to the "street wisdom" that they are only practical for large, relatively square matrices, that they require considerable workspace, and that they are difficult to achieve thread-level parallelism. We dispelled these notions, demonstrating significant benefits for small and non-square matrices, requiring no workspace beyond what is already incorporated in high-performance implementations of matrix multiplication, and achieving performance benefits on the Intel Xeon Phi processor with 60 cores executing 240 threads and on the latest Volta GPU device with over 14 TFLOPS theoretical peak performance for single precision. The resulting families of algorithms can serve as a drop-in replacement for matrix multiplication, which has numerous scientific applications. This study showed that STRASSEN and Strassen-like fast matrix multiplication algorithms can be incorporated into libraries for practical use.

7.1 Results

In this dissertation, a number of novel contributions have been reported.

- **Practical implementations of STRASSEN on various CPU architectures.** Key to the work is that the linear combinations of submatrices that underlie STRASSEN can be incorporated in and composed with the primitives of high-performance GEMM implementations exposed by the BLAS-like Library Instantiation Software (BLIS), which discloses the fundamental building blocks below the BLAS interface that can be used to build the new algorithms and fuse a sequence of BLAS-like operations to avoid repeated memory movements that constitute overhead. Incorporating the matrix additions that must be performed for STRASSEN into the inherent packing and micro-kernel operations inside high-performance implementations of GEMM avoids extra workspace and reduces the cost of extra memory movement. Adopting the same loop structures as high-performance GEMM implementations allows parallelizations of STRASSEN with simple but efficient data parallelization without the expenses of task parallelism. Our implementations demonstrated performance benefits over the conventional GEMM on a variety of architectures, such as single core, multi-core, many-core, and distributed memory parallel CPU architectures.

- **Code and model generation for the practical implementations of Strassen-like FMM algorithms.** We developed a code generator framework which can automatically implement families of FMM 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 algorithms into the inherent packing and micro-kernel operations inside GEMM, avoiding extra workspace requirement and reducing the overhead of memory movement. Importantly, it generates a performance model that is accurate enough 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. Without the requirement for exhaustive empirical search, our generated implementations of various FMM algorithms outperformed the state-of-the-art implementations, especially for smaller matrices and special matrix multiplication shapes such as rank-k updates.

- **Generalization of practical implementations of STRASSEN for higher-dimensional tensor contraction.** This work presents the first efficient implementation of Strassen's algorithm for tensor contraction, a significant problem with numerous applications. It describes how to extend Strassen's algorithm to tensor contraction without the explicit transposition of data that inherently incurs significant memory movement and workspace overhead; it provides a performance model for the cost of the resulting family of algorithms; it details the practical implementation of these algorithms, including how to exploit variants of the primitives that underlie BLIS and a data layout to memory for the tensors; it demonstrates practical speedup on modern single core and multi-core CPUs; it illustrates how the local use of

the Strassen's tensor contraction algorithm on each node improves performance of a simple distributed memory tensor contraction. Together, these results unlock a new frontier for the research and application of Strassen's algorithm.

- **Practical implementations of** STRASSEN **on Nvidia GPUs.** The GPU architecture is different from a traditional CPU in a number of aspects: a GPU has its own programming model; it has limited physical memory size; it is designed as a parallel, throughput-oriented computing engine; it has a higher ratio of the computation power to the memory bandwidth. We overcame these challenges for a practical implementation of STRASSEN on GPUs, which outperforms the state-of-the-art implementations on small problem sizes and consumes the same memory compared to the conventional GEMM. We leveraged the GPU thread and memory hierarchy by designing a dedicated kernel, reduced the additional memory consumption through reusing the shared memory for the temporary matrix addition result and the register files for the temporary matrix product in STRASSEN, and exploited the inter-kernel parallelism as well as the intra-kernel parallelism. Finally, our specialized kernel for STRASSEN on GPUs can attain both more parallelism and less memory consumption, compared to the state-of-the-art result.

7.2 Future work

There are several avenues of research for future work suggested by this dissertation.

- **Special structures of matrix multiplication and tensor contraction.** In this dissertation we target dense matrix multiplication and tensor contraction, which have numerous applications. However, the structure of the matrix and tensor operands may be symmetric [63, 39, 21, 98], which yields a number of new challenges, like more efficient storage or layout format. How to explore those structure patterns and combine with STRASSEN and similar Strassen-like FMM algorithms can be investigated.

- **Other level-3 BLAS.** Kågström *et al.* [63] and Higham [49] demonstrated that GEMM and FMM can be the basis for level-3 BLAS. In [39, 124, 122], it is discussed how the GotoBLAS algorithm and BLIS framework for GEMM can be extended to implement other level-3 BLAS (matrix-matrix operations). This sets the stage for a more thorough investigation of how the various modifications of STRASSEN and Strassen-like FMM algorithms can similarly be applied ot the other level-3 BLAS.

- **Higher-level linear algebra functionality.** Unique to our implementation of STRASSEN and FMM algorithms is the fact that it already attains high performance for a rank-k update for a relatively small k. Many operations in libraries like LAPACK [2], ScaLAPACK [19], `libflame` [121, 43],

PLAPACK [118], and Elemental [95], cast higher level linear algebra functionality like LU (with pivoting), QR, and Cholesky factorization in terms of rank-k updates. A natural question becomes how to accelerate these important operations with the new STRASSEN or other FMM algorithms. This takes the subject full circle in the sense that the original Strassen paper actually discussed how LU factorization could be accelerated [115].

- **Future hierarchical memory architectures.** In Section 2.1.6, we briefly reviewed the related work on implementing a family of algorithms on general hierarchical memory architectures. Future computer architectures are expected to have a lower ratio between the rate of data movement from the main memory to the caches and the rate of computation, requiring alternative GEMM algorithms, e.g., presented in [44, 106]. We intend to explore how to extend these algorithms to STRASSEN and other Strassen-like FMM algorithms with the goal of achieving good performance in a low bandwidth scenario.

- **Machine learning primitives.** Building upon BLIS, fusing GEMM with other operations can benefit the performance of operations encountered in machine learning such as the "K-Nearest Neighbor" computation [131]. The idea is that many memory movements can be avoided by incorporating the processing of the output of a GEMM operation into the implementation, not unlike how we incorproate the addition of submatrices into the packing of those submatrices in our STRASSEN implementation. We plan to investigate

whether such machine learning operations can be further accelerated by incorporating STRASSEN or other FMM algorithms rather than classical GEMM.

- **Mixed precision.** More levels of STRASSEN and Strassen-like FMM algorithms may lose precision due to numerical instability issues. It may be possible to combine the proposed techniques with Extended and Mixed Precision BLAS [78] to get higher speedup and maintain precision.

- **Task parallelism.** Task parallelism and various parallel schemes are proposed in the recent literature [26, 9]. In [9], the authors of that paper discuss alternative ways for achieving thread-level parallelism: parallelism with the GEMM calls for the individual multiplications with submatrices; task-level parallelism where each multiplication with submatrices is viewed as a task; and a combination of these techniques. In Chapter 6, we exploited such inter-kernel and the intra-kernel parallelism. We need to pursue how our techniques compare to these and how to combine these with our advances in STRASSEN and Strassen-like FMM algorithms. It may also be possible to utilize our performance model to help better task scheduling.

- **Searching for new FMM algorithms with the performance model.** Finding new FMM algorithms by searching the coefficient matrix $[\![U, V, W]\!]$ is an NP-hard problem [68]. It may be possible to prune branches with the performance model as the cost function during the search process.

- **Communication lower bound.** The asymptotic communication lower bound for STRASSEN and the conventional matrix multiplication has been characterized, in [62, 7, 101, 10]. We would like to investigate how to apply our performance model to constrain the coefficients of the cubic and quadratic terms and get more precise lower bound for specific architectures.

- **Pedagogical outreach.** We have created a pedagogical "sandbox" we call *BLISlab* [57, 52], which teaches relative novices to the science of high-performance computing how to optimize GEMM within a simplified BLIS-like framework in courses [119, 93]. We intend to extend this exercise to also guide the participants through the optimizations that underlie STRASSEN and the Strassen-like FMM implementations.

- **Distributed memory.** Practical distributed memory parallel algorithms for GEMM are invariably variations on the Scalable University Matrix Multiplication Algorithm (SUMMA) [120, 99]. In Chapter 3, we discuss how a SUMMA algorithm can benefit from a call to STRASSEN for the node-level matrix-matrix multiplication. In contrast, in [42], SUMMA was extended to incorporate STRASSEN, with STRASSEN applied at the top level, parallelism within each multiplication with submatrices, and a call to a standard DGEMM for the local call. On the one hand, the performance of our approach that incorporates STRASSEN in the local GEMM needs to be compared to these implementations. On the other hand, it may be possible to add a local STRASSEN GEMM into these parallel implementations. Alternatively,

the required packing may be incorporated into the communication of the data.

In short: there are many variations of the STRASSEN and similar Strassen-like FMM algorithm themes yet to be explored.

Appendices

Appendix A

Table of Acronyms

AI	Artificial Intelligence.
APA	Arbitrary Precision Approximate.
ATLAS	Automatically Tuned Linear Algebra Software.
BLAS	Basic Linear Algebra Subprogram.
BLIS	BLAS-like Library Instantiation Software.
CCSD	Coupled Cluster with Single and Double excitations.
CUTLASS	CUDA Templates for Linear Algebra Subroutines.
CTF	Cyclops Tensor Framework.
DLA	Dense Linear Algebra.
DF	Density-fitting.
DRAM	Dynamic Random Access Memory.
FMA	Fused Multiply Add.
FMM	Fast Matrix Multiplication.
GEMM	General Matrix Multiplication.
GFLOPS	Giga (10^9) FLoating-point Operations Per Second.
HPC	High-Performance Computing.
MOMMS	Multilevel Optimized Matrix-matrix Multiplication Sandbox.
NRMSE	Normalized Root-Mean-Square Error.
PHiPAC	Portable High Performance ANSI C.
SM	Streaming Multiprocessor.
SUMMA	Scalable Universal Matrix Multiplication Algorithm.
TACC	Texas Advanced Computing Center.
TC	Tensor Contraction.
TFLOPS	Tera (10^{12}) FLoating-point Operations Per Second.

Appendix B

Table of Symbols

$\alpha, \beta, ...$	Scalar variables.
$a, b, c, ...$	Vector variables.
$A, B, C, ...$	Matrix variables.
$\mathcal{A}, \mathcal{B}, \mathcal{C}, ...$	Tensor variables.
$\mathbf{A}, \mathbf{B}, \mathbf{C}, ...$	Matrix views of tensors.
m, n, k	Matrix dimensions.
m_C, n_C, k_C	Cache blocking parameters for GotoBLAS.
m_R, n_R	Register blocking parameters for GotoBLAS or CUTLASS.
m_S, n_S, k_S	Shared memory blocking parameters for CUTLASS.
m_W, n_W	Warp blocking parameters for CUTLASS.
I_m, J_n, P_k	Index bundles. See Section 5.1.1.
$\langle \widetilde{m}, \widetilde{k}, \widetilde{n} \rangle$	Partitions for FMM algorithms. See Section 4.1.1.
$[\![U, V, W]\!]$	The set of coefficients that determine the $\langle \widetilde{m}, \widetilde{k}, \widetilde{n} \rangle$ algorithm.
$rscat, cscat$	Scatter vectors for the matrix view of tensors.
rbs, cbs	Block scatter vectors for the matrix view of tensors.
T	Time.

Appendix C

$\langle 3, 2, 3 \rangle$ FMM algorithm

In this appendix, we present an example of a FMM algorithm with $\langle 3, 2, 3 \rangle$ partition, originally from [6, 8, 9].

Consider $C := C + AB$, where C, A, and B are $m \times n$, $m \times k$, and $k \times n$ matrices, respectively. Assuming that m and n can be evenly divided by 3, and k is even, this $\langle 3, 2, 3 \rangle$ FMM algorithm has the partition

$$C = \left(\begin{array}{c|c|c} C_0 & C_1 & C_2 \\ \hline C_3 & C_4 & C_5 \\ \hline C_6 & C_7 & C_8 \end{array}\right), A = \left(\begin{array}{c|c} A_0 & A_1 \\ \hline A_2 & A_3 \\ \hline A_4 & A_5 \end{array}\right), B = \left(\begin{array}{c|c|c} B_0 & B_1 & B_2 \\ \hline B_3 & B_4 & B_5 \end{array}\right), \qquad \text{(C.1)}$$

where A_i, B_j, and C_p are the $\frac{m}{3} \times \frac{k}{2}$, $\frac{k}{2} \times \frac{n}{3}$, and $\frac{m}{3} \times \frac{n}{3}$ submatrices of A, B and C, respectively, with a single index in the row major order. It can be verified that the operations in Figure C.1 also compute $C := C + AB$, requiring only 15 multiplications with submatrices instead of $3 \times 2 \times 3 = 18$ submatrix multiplication. Theoretically this algorithm can reach $(18 - 15)/15 = 20\%$ speedup, if we ignore the lower-order term of submatrix additions.

The following $[\![U, V, W]\!]$ specifies the set of coefficients that determine this one-level $\langle 3, 2, 3 \rangle$ FMM algorithm, where U, V, and W are 6×15, 6×15, and 9×15 matrices. Note that the entries u_{ir}, v_{jr}, and w_{pr} in the rth column of $[\![U, V, W]\!]$, specify the cofficients for the operation in rth row in Figure C.1,

and i, j, p are mapped to the submatrix index in (C.1). For example, In the first column of $[\![U, V, W]\!]$, $-1, 1, 1$ of U are corresponding to A_1, A_3, and A_5; $1, 1$ of V are corresponding to B_4 and B_5; and -1 of W are corresponding to C_2. These coefficients are mapped to the operation in the first row:

$$M_0 = (-A_1 + A_3 + A_5)(B_4 + B_5); C_2 -= M_0;$$

$$U = \begin{pmatrix} 0 & -1 & -1 & 0 & -1 & 0 & 0 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & -1 \\ -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & -1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & -1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & -1 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 \end{pmatrix},$$

$$V = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & -1 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 & -1 & 0 \end{pmatrix},$$

$$W = \begin{pmatrix} 0 & 0 & -1 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & -1 & 0 & 1 & 0 & 0 & -1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 1 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & -1 & 0 \\ 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \end{pmatrix}.$$

Other FMM algorithms with different partitions found in the literature and tested in our experiment in Chapter 4 are listed in Figure 4.2 and more details can be found in [53].

$M_0 = (-A_1 + A_3 + A_5)(B_4 + B_5);$	$C_2 \mathrel{-}= M_0;$
$M_1 = (-A_0 + A_2 + A_5)(B_0 - B_5);$	$C_2 \mathrel{-}= M_1;\ C_7 \mathrel{-}= M_1;$
$M_2 = (-A_0 + A_2)(-B_1 - B_4);$	$C_0 \mathrel{-}= M_2;\ C_1 \mathrel{+}= M_2;\ C_7 \mathrel{+}= M_2;$
$M_3 = (A_2 - A_4 + A_5)(B_2 - B_3 + B_5);$	$C_2 \mathrel{+}= M_3;\ C_5 \mathrel{+}= M_3;\ C_6 \mathrel{-}= M_3;$
$M_4 = (-A_0 + A_1 + A_2)(B_0 + B_1 + B_4);$	$C_0 \mathrel{-}= M_4;\ C_1 \mathrel{+}= M_4;\ C_4 \mathrel{+}= M_4;$
$M_5 = (A_2)(B_0);$	$C_0 \mathrel{+}= M_5;\ C_1 \mathrel{-}= M_5;\ C_3 \mathrel{+}= M_5;\ C_4 \mathrel{-}= M_5;\ C_6 \mathrel{+}= M_5;$
$M_6 = (-A_2 + A_3 + A_4 - A_5)(B_3 - B_5);$	$C_2 \mathrel{-}= M_6;\ C_5 \mathrel{-}= M_6;$
$M_7 = (A_3)(B_3);$	$C_2 \mathrel{+}= M_7;\ C_3 \mathrel{+}= M_7;\ C_5 \mathrel{+}= M_7;$
$M_8 = (-A_0 + A_2 + A_4)(B_1 + B_2);$	$C_7 \mathrel{+}= M_8;$
$M_9 = (-A_0 + A_1)(-B_0 - B_1);$	$C_1 \mathrel{+}= M_9;\ C_4 \mathrel{+}= M_9;$
$M_{10} = (A_5)(B_2 + B_5);$	$C_2 \mathrel{+}= M_{10}; C_6 \mathrel{+}= M_{10}; C_8 \mathrel{+}= M_{10};$
$M_{11} = (A_4 - A_5)(B_2);$	$C_2 \mathrel{+}= M_{11}; C_5 \mathrel{+}= M_{11}; C_7 \mathrel{-}= M_{11}; C_8 \mathrel{+}= M_{11};$
$M_{12} = (A_1)(B_0 + B_1 + B_3 + B_4);$	$C_0 \mathrel{+}= M_{12};$
$M_{13} = (A_2 - A_4)(B_0 - B_2 + B_3 - B_5);$	$C_6 \mathrel{-}= M_{13};$
$M_{14} = (-A_0 + A_1 + A_2 - A_3)(B_5);$	$C_2 \mathrel{-}= M_{14}; C_4 \mathrel{-}= M_{14};$

Figure C.1: All operations for an example of one-level $\langle 3, 2, 3 \rangle$ FMM algorithm.

Appendix D

Derivation of Kronecker Product

In this appendix, we derive that the coefficient matrices of a multi-level FMM algorithm can be represented as the Kronecker product of the coefficient matrices of each level.

Theorem D.1. (**2-level**) For a 2-level FMM algorithm, assuming the set of coefficients that determine the first level $\langle \widetilde{m}, \widetilde{k}, \widetilde{n} \rangle$ FMM algorithm is $[\![U, V, W]\!]$, the set of coefficients that determine the second level $\langle \widetilde{m'}, \widetilde{k'}, \widetilde{n'} \rangle$ FMM algorithm is $[\![U', V', W']\!]$, and the submatrices of A, B and C are indexed with a 2-level recursive block storage indexing, then the set of coefficients of this two-level FMM algorithm is $[\![U \otimes U', V \otimes V', W \otimes W']\!]$.

Proof. Following our representations in Section 4.1, the first level $\langle \widetilde{m}, \widetilde{k}, \widetilde{n} \rangle$ FMM algorithm with the set of coefficients $[\![U, V, W]\!]$ can be rewritten as,

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

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. Note that here A_i, B_j, and C_p are the submatrices of A, B and C, with a single index in the row major order.

Similarly, the second level $\langle \widetilde{m'}, \widetilde{k'}, \widetilde{n'} \rangle$ FMM algorithm with the set of coefficients $[\![U', V', W']\!]$ can be rewritten as,

$$\begin{aligned} &\text{for } \; r' = 0, ..., R' - 1, \\ &\qquad M'_r := \left(\sum_{i'=0}^{\widetilde{m'}\widetilde{k'}-1} u'_{i'r'} A'_i \right) \times \left(\sum_{j'=0}^{\widetilde{k'}\widetilde{n'}-1} v_{j'r'} B'_j \right); \\ &\qquad C'_p \mathrel{+}= w'_{p'r'} M'_r \; (p' = 0, ..., \widetilde{m'}\widetilde{n'} - 1) \end{aligned} \tag{D.2}$$

where $u'_{i'r'}$, $v'_{j'r'}$, and $w'_{p'r'}$ 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. Note that here A'_i, B'_j, and C'_p are the submatrices of A_i, B_j and C_p of the first level, with a single index in the row major order.

By passing (D.2) into (D.1), the two-level FMM algorithm composed of the above first level and second level FMM algorithms can be written as,

$$\begin{aligned} &\text{for } \; r = 0, ..., R - 1, \\ &\quad \text{for } r' = 0, ..., R' - 1, \\ &\qquad M_{r \cdot R' + r'} := \left(\sum_{i=0}^{\widetilde{m}\widetilde{k}-1} u_{ir} \left(\sum_{i'=0}^{\widetilde{m'}\widetilde{k'}-1} u'_{i'r'} A_{(i,i')} \right) \right) \\ &\qquad \times \left(\sum_{j=0}^{\widetilde{k}\widetilde{n}-1} v_{jr} \left(\sum_{j'=0}^{\widetilde{k'}\widetilde{n'}-1} v'_{j'r'} B_{(j,j')} \right) \right); \\ &\qquad C_{(p,p')} \mathrel{+}= w_{pr} \left(w'_{p'r'} M_{r \cdot R' + r'} \right) \\ &\qquad (p = 0, ..., \widetilde{m}\widetilde{n} - 1; p' = 0, ..., \widetilde{m'}\widetilde{n'} - 1) \end{aligned} \tag{D.3}$$

where $A_{(i,i')}$, $B_{(j,j')}$, and $C_{(p,p')}$ are the i'th subblock in the second level partition of ith submatrix in the first level partition of A, the j'th subblock in

the second level partition of jth submatrix in the first level partition of B, the p'th subblock in the second level partition of pth submatrix in the first level partition of C, respectively.

Through interchanging the order of summation, the two-level FMM algorithm can be rewritten as, (D.3) can be rewritten as,

$$
\begin{aligned}
&\text{for } \; r = 0, ..., R-1, \\
&\quad \text{for } r' = 0, ..., R'-1, \\
&\qquad M_{r \cdot R' + r'} := \left(\sum_{i=0}^{\widetilde{m}\widetilde{k}-1} \sum_{i'=0}^{\widetilde{m'}\widetilde{k'}-1} \left(u_{ir} u'_{i'r'}\right) A_{(i,i')} \right) \\
&\qquad \times \left(\sum_{j=0}^{\widetilde{k}\widetilde{n}-1} \sum_{j'=0}^{\widetilde{k'}\widetilde{n'}-1} \left(v_{jr} v'_{j'r'}\right) B_{(j,j')} \right); \\
&\qquad C_{(p,p')} \mathrel{+}= \left(w_{pr} w'_{p'r'}\right) M_{r \cdot R' + r'} \\
&\qquad (p = 0, ..., \widetilde{m}\widetilde{n} - 1; p' = 0, ..., \widetilde{m'}\widetilde{n'} - 1).
\end{aligned}
\tag{D.4}
$$

Since the submatrices of A, B and C are indexed with a 2-level recursive block storage indexing, $A_{(i,i')}$, $B_{(j,j')}$, and $C_{(p,p')}$ can be indexed with a single index in the two-level recursive blocking block storage indexing, that is, $A_{i \cdot \widetilde{m'}\widetilde{k'} + i'}$, $B_{j \cdot \widetilde{k'}\widetilde{n'} + j'}$, $C_{p \cdot \widetilde{m'}\widetilde{n'} + p'}$, respectively. Therefore, (D.4) can be rewritten as,

$$
\begin{aligned}
&\text{for } \; r = 0, ..., R-1, \\
&\quad \text{for } r' = 0, ..., R'-1, \\
&\qquad M_{r \cdot R' + r'} := \left(\sum_{i=0}^{\widetilde{m}\widetilde{k}-1} \sum_{i'=0}^{\widetilde{m'}\widetilde{k'}-1} \left(u_{ir} u'_{i'r'}\right) A_{i \cdot \widetilde{m'}\widetilde{k'} + i'} \right) \\
&\qquad \times \left(\sum_{j=0}^{\widetilde{k}\widetilde{n}-1} \sum_{j'=0}^{\widetilde{k'}\widetilde{n'}-1} \left(v_{jr} v'_{j'r'}\right) B_{j \cdot \widetilde{k'}\widetilde{n'} + j'} \right); \\
&\qquad C_{p \cdot \widetilde{m'}\widetilde{n'} + p'} \mathrel{+}= \left(w_{pr} w'_{p'r'}\right) M_{r \cdot R' + r'} \\
&\qquad (p = 0, ..., \widetilde{m}\widetilde{n} - 1; p' = 0, ..., \widetilde{m'}\widetilde{n'} - 1).
\end{aligned}
\tag{D.5}
$$

Let's use $a\%b$ to denote the remainder of a/b, and assume that $r^* = r \cdot R' + r'$, $i^* = i \cdot \widetilde{m}\widetilde{k} + i'$, $j^* = j \cdot \widetilde{k}\widetilde{n} + j'$, and $p^* = p \cdot \widetilde{m}\widetilde{n} + p'$. Then, $r = \lfloor r^*/R' \rfloor$, $r' = r^*\%R'$; $i = \left\lfloor i^*/(\widetilde{m'}\widetilde{k'}) \right\rfloor$, $i' = i^*\%(\widetilde{m'}\widetilde{k'})$; $j = \left\lfloor j^*/(\widetilde{k'}\widetilde{n'}) \right\rfloor$, $j' = j^*\%(\widetilde{k'}\widetilde{n'})$; and $p = \left\lfloor p^*/(\widetilde{m'}\widetilde{n'}) \right\rfloor$, $p' = p^*\%(\widetilde{m'}\widetilde{n'})$. By replacing r, r', i, i', j, j', p, and p' with r^*, i^*, j^*, and p^*, and merging the summation, (D.5) can be rewritten as,

$$
\begin{aligned}
&\text{for} \quad r^* = 0, ..., R \cdot R' - 1, \\
&M_{r^*} := \left(\sum_{i^*=0}^{\widetilde{m}\widetilde{k} \cdot \widetilde{m'}\widetilde{k'} - 1} \left(u_{\lfloor i^*/(\widetilde{m'}\widetilde{k'}) \rfloor, \lfloor r^*/R' \rfloor} u'_{i^*\%(\widetilde{m'}\widetilde{k'}), r^*\%R'} \right) A_{i^*} \right) \\
&\times \left(\sum_{j^*=0}^{\widetilde{k}\widetilde{n} \cdot \widetilde{k'}\widetilde{n'} - 1} \left(v_{\lfloor j^*/(\widetilde{k'}\widetilde{n'}) \rfloor, \lfloor r^*/R' \rfloor} v'_{j^*\%(\widetilde{k'}\widetilde{n'}), r^*\%R'} \right) B_{j^*} \right); \qquad \text{(D.6)} \\
&C_{p^*} \mathrel{+}= \left(w_{\lfloor p^*/(\widetilde{m'}\widetilde{n'}) \rfloor, \lfloor r^*/R' \rfloor} w'_{p^*\%(\widetilde{m'}\widetilde{n'}), r^*\%R'} \right) M_{r^*} \\
&(p^* = 0, ..., \widetilde{m}\widetilde{n} \cdot \widetilde{m'}\widetilde{n'} - 1).
\end{aligned}
$$

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 [41] $X \otimes Y$ 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 $(X \otimes Y)_{p \cdot r+v, q \cdot s+w} = x_{r,s}y_{v,w}$, or, $(X \otimes Y)_{i,j} = x_{\lfloor i/p \rfloor, \lfloor j/q \rfloor} y_{i\%p, j\%q}$. With this definition of Kronecker product, (D.6) can be rewritten as,

$$
\begin{aligned}
&\text{for } \; r^* = 0, ..., R \cdot R' - 1, \\
&\qquad M_{r^*} := \left(\sum_{i^*=0}^{\widetilde{m}\widetilde{k} \cdot \widetilde{m'}\widetilde{k'}-1} (U \otimes U')_{i^*, r^*} \, A_{i^*} \right) \\
&\qquad \times \left(\sum_{j^*=0}^{\widetilde{k}\widetilde{n} \cdot \widetilde{k'}\widetilde{n'}-1} (V \otimes V')_{j^*, r^*} \, B_{j^*} \right); \\
&\qquad C_{p^*} \mathrel{+}= (W \otimes W')_{\underbrace{p^*}, \underbrace{r^*}} \, M_{r^*} \\
&\qquad (p^* = 0, ..., \widetilde{m}\widetilde{n} \cdot \widetilde{m'}\widetilde{n'} - 1).
\end{aligned}
\tag{D.7}
$$

Thus, 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 we have proved (D.7), that is, $C := C + AB$ can be computed by,

$$
\begin{aligned}
&\text{for } \; r = 0, ..., R \cdot R' - 1, \\
&\qquad M_r := \left(\sum_{i=0}^{\widetilde{m}\widetilde{k} \cdot \widetilde{m}'\widetilde{k}'-1} (U \otimes U')_{i,r} A_i \right) \times \left(\sum_{j=0}^{\widetilde{k}\widetilde{n} \cdot \widetilde{k}'\widetilde{n}'-1} (V \otimes V')_{j,r} B_j \right); \\
&\qquad C_p \mathrel{+}= (W \otimes W')_{p,r} M_r (p = 0, ..., \widetilde{m}\widetilde{n} \cdot \widetilde{m}'\widetilde{n}' - 1).
\end{aligned}
\tag{D.8}
$$

□

Theorem D.2. (***L*-level generalization**) Assuming that the set of coefficients that determine the l-level $\langle \widetilde{m_l}, \widetilde{k_l}, \widetilde{n_l} \rangle$ algorithm is $[\![U_l, V_l, W_l]\!]$ ($l = 0, 1, ..., L-1; L \geq 1$), and the submatrices of A, B and C are indexed with a L-level recursive block storage indexing, then the set of coefficients of this L-level FMM algorithm is $[\![\bigotimes_{l=0}^{L-1} U_l, \bigotimes_{l=0}^{L-1} V_l, \bigotimes_{l=0}^{L-1} W_l]\!]$.

Proof. This can be derived by mathematical induction.

Base Step. For $L = 1$, following our representations in Section 4.1, since 1-level recrusive block storage indexing is the same as row major order indexing, the set of coefficients of 1-level $\langle \widetilde{m_0}, \widetilde{k_0}, \widetilde{n_0} \rangle$ algorithm is $[\![U_0, V_0, W_0]\!]$.

Inductive Hypothesis. Assume that for some $L = K$ ($K \geq 1$), the set of coefficients of this L-level FMM algorithm is $[\![\bigotimes_{l=0}^{K-1} U_l, \bigotimes_{l=0}^{K-1} V_l, \bigotimes_{l=0}^{K-1} W_l]\!]$. we want to show that the proposition holds for $L = K + 1$.

Inductive Step. For $L = K + 1$, the $(K + 1)$-level recursive block storage indexing can be viewed as the K-level recursive block storage indexing and a following 1-level block storage indexing on top of the K-level partition. the $(K + 1)$-level FMM algorithm can also be viewed as the K-level FMM algorithm and a following 1-level $\langle \widetilde{m_K}, \widetilde{k_K}, \widetilde{n_K} \rangle$ algorithm with the set of coefficients $[\![U_K, V_K, W_K]\!]$. By Theorem D.1 and the inductive hypothesis, we have that the set of coefficients of this L-level FMM algorithm is $[\![\left(\bigotimes_{l=0}^{K-1} U_l\right) \otimes U_K, \left(\bigotimes_{l=0}^{K-1} V_l\right) \otimes V_K, \left(\bigotimes_{l=0}^{K-1} W_l\right) \otimes W_K]\!]$, or, $[\![\bigotimes_{l=0}^{K} U_l, \bigotimes_{l=0}^{K} V_l, \bigotimes_{l=0}^{K} W_l]\!]$.

We showed that $L = 1$ satisfies the property and that if some K ($K \geq 1$) satisfies it, then $K + 1$ satisfies it as well. This shows, by the induction principle, this theorem holds for any $L \geq 1$, so this completes the proof.

The formula for defining the L-level FMM algorithm is given by,

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

$$
\begin{aligned}
M_r := & \left(\sum_{i=0}^{\prod_{l=0}^{L-1} \widetilde{m}_l \widetilde{k}_l - 1} (\bigotimes_{l=0}^{L-1} U_l)_{i,r} A_i \right) \times \left(\sum_{j=0}^{\prod_{l=0}^{L-1} \widetilde{k}_l \widetilde{n}_l - 1} (\bigotimes_{l=0}^{L-1} V_l)_{j,r} B_j \right); \\
C_p += & (\bigotimes_{l=0}^{L-1} W_l)_{p,r} M_r (p = 0, ..., \prod_{l=0}^{L-1} \widetilde{m}_l \widetilde{n}_l - 1)
\end{aligned}
\tag{D.9}
$$

where A_i, B_j, C_p are the submatrices of A, B and C, with a single index in L-level recursive block storage indexing.

□

Appendix E

Numerical Stability

In this appendix, we analyze the theoretical and empirical numerical stability result of STRASSEN and similar Strassen-like FMM algorithms, and discuss some possible algorithmic techniques to improving the numerical accuracy.

E.1 Numerical stability for Strassen's algorithm

E.1.1 Theoretical bound

There once was a widespread viewpoint that STRASSEN is severely numerically unstable. However, this viewpoint is unfounded [16]. The fact is that STRASSEN does meet a somewhat weaker error bound than the classical matrix multiplication algorithm (Section 2.1) [50, 28, 6]:

- STRASSEN only meets a norm-wise bound, while the classical matrix multiplication algorithm further meets an element-wise bound.
- The constant term in the norm-wise bound for STRASSEN is larger than that for the classical algorithm.

As illustrated in [50], The forward norm-wise error bound for L-level

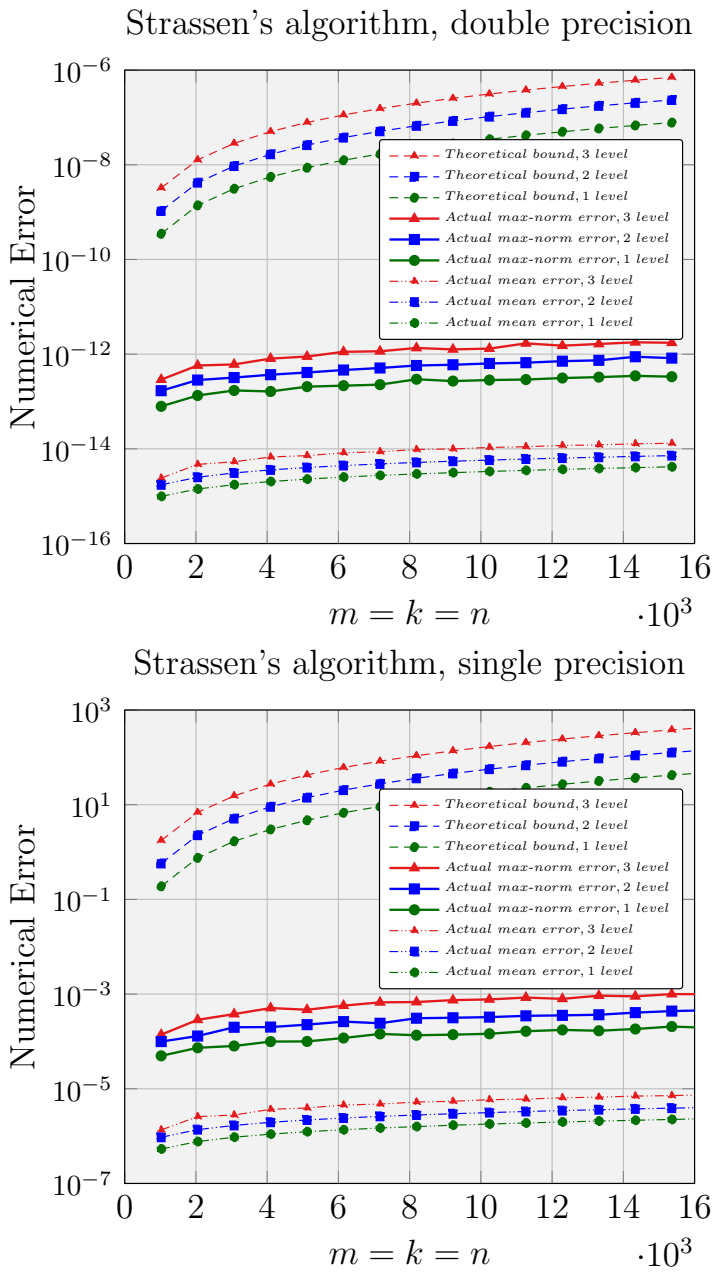


Figure E.1: Empirical stability experiment for STRASSEN. The actual max-norm error and mean error vs. theoretical error bound for square matrices in double precision and single precision.

STRASSEN is

$$\|C - \hat{C}\| \leq (12^L((\frac{n}{2^L})^2 + \frac{5n}{2^L}) - 5n)\|A\|\|B\|\mathbf{u} + O(\mathbf{u}^2) \tag{E.1}$$

where M is an $n \times n$ matrix, $\|M\| := max_{i,j}|M_{i,j}|$ for $M \in \{A, B, C\}$, and $\mathbf{u}$ is the unit roundoff. For single precision, $\mathbf{u} = 2^{-24} \approx 5.96 \times 10^{-8}$; For double precision, $\mathbf{u} = 2^{-53} \approx 1.11 \times 10^{-16}$.

As a comparison, the forward norm-wise error bound for the classical algorithm given in [50] is

$$\|C - \hat{C}\| \leq n\|A\|\|B\|\mathbf{u} + O(\mathbf{u}^2).$$

E.1.2 Empirical experiment

The theoretical bound given in [50] is too loose. In Figure E.1, we show the measured max-norm absolute error ($max_{i,j}|C_{i,j} - \hat{C}_{i,j}|$) and mean absolute error ($\frac{1}{n^2}\Sigma_{i,j}|C_{i,j} - \hat{C}_{i,j}|$) compared to the theoretical bound for square matrices with STRASSEN in Chapter 3 in double precision and single precision, where each entry is uniformly randomly generated from $[-1, 1]$.

E.2 Numerical stability for FMM algorithms

E.2.1 Theoretical bound

Strassen-like FMM algorithms are also norm-wise numerical stable. A number of papers [13, 50, 28, 6] reveal that the norm-wise stability bounds for L-level FMM algorithms can be present as

$$\|C - \hat{C}\| \leq f_{alg}(n, L)\|A\|\|B\|\mathbf{u} + O(\mathbf{u}^2) \tag{E.2}$$

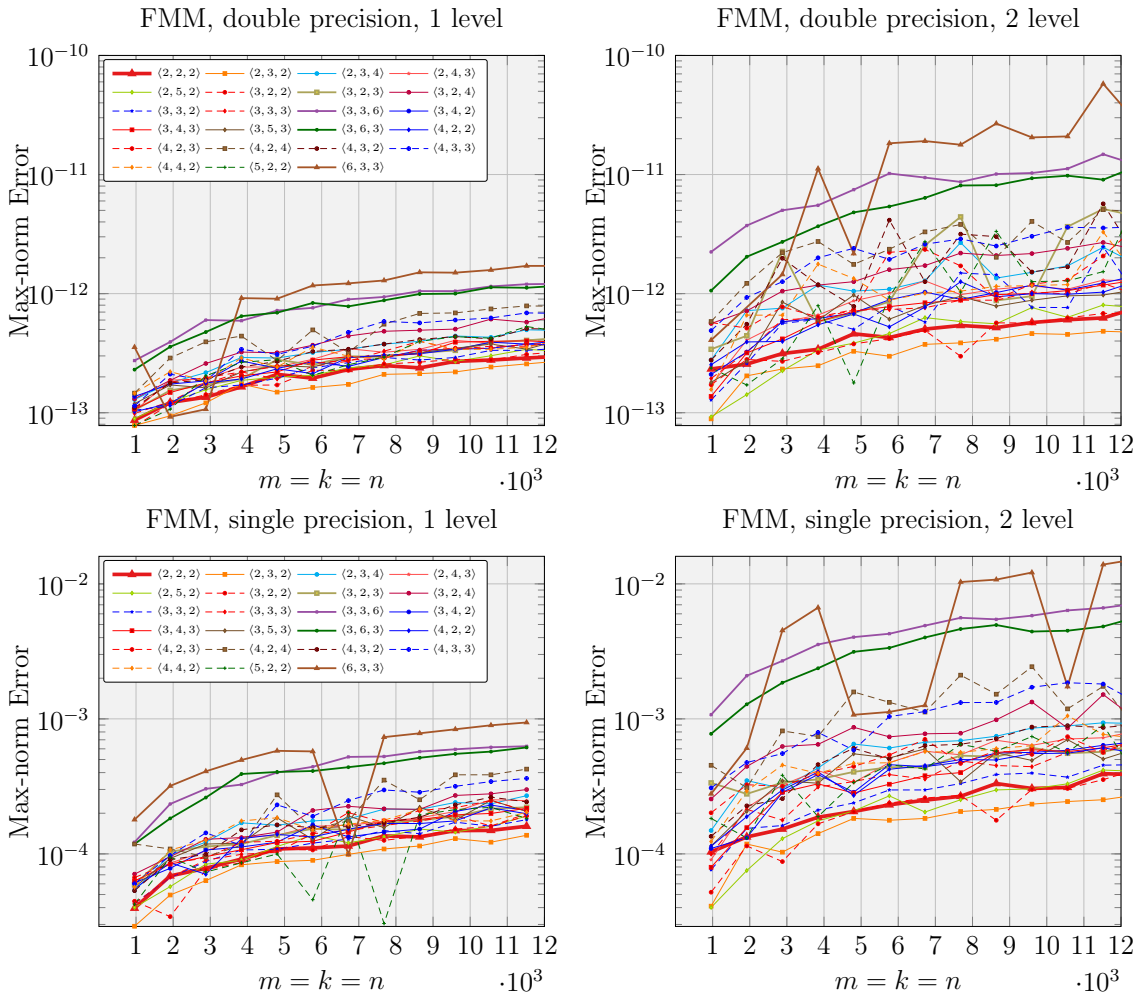


Figure E.2: Empirical stability experiment for 1-level and 2-level FMM algorithms. The actual max-norm error for square matrices in double precision and single precision.

where f_{alg} is a polynomial function of n depending on the algorithm.

E.2.2 Empirical experiment

In Figure E.2, we present the measured max-norm absolute error ($\|C - \hat{C}\|$) for square matrices with 1-level and 2-level FMM algorithms in Chapter 4 in double precision and single precision, where each entry is uniformly randomly generated from $[-1, 1]$.

E.3 Improving the numerical accuracy

The numerical accuracy of FMM algorithms depends on not only the properties of the algorithms, but also the intrinsic structures of the input matrices. There are several strategies for improving the error guarantees.

- Reducing the number of levels of FMM algorithms. The numerical error decreases as fewer levels of FMM algorithms are employed, as observed in Figure E.1 and Figure E.2. However, fewer levels may mean less performance improvement with FMM algorithms. There is a trade-off between the levels of FMM algorithms and the performance.

- Selecting the FMM algorithm with better numerical stability. This may involve an exhaustive search and more details can be found in [6].

- Diagonal scaling by preprocessing A and B and postprocessing C [28, 6] in order to improve the intrisic structure of the input matrices, that is, to decrease $\|A\|$ and $\|B\|$ in Equation (E.1) and Equation (E.2). The

basic idea follows the equation,

$$C = D_A D_A^{-1} A D D^{-1} B D_B^{-1} D_B = D_A((D_A^{-1} A D)(D^{-1} B D_B^{-1}))D_B$$

for any nonsingular diagonal scaling matrices, D_A, D_B, and D. With the associativity property of matrix multiplication, these diagonal scaling matrices can be first applied to the input matrices A and B as a preprocessing step, and later applied to the matric product as a postprocessing step, which involve an extra $O(N^2)$ cost. The scaling technique can be incorporated in the the packing and micro-kernel routines for CPUs, and in the packing and accumulating process for GPUs.

Finally, we note that the numerical accuracy loss caused by using STRASSEN or similar Strassen-like FMM algorithms instead of the classical matrix multiplication is likely no worse than the other computations encountered in a larger application.

Bibliography

[1] AMD rocBLAS. GitHub Repository, 2018. https://github.com/ROCmSoftwarePlatform/rocBLAS.

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

[3] ATLAS. Available Online, 2016. http://math-atlas.sourceforge.net.

[4] Brett W. Bader, Tamara G. Kolda, et al. Matlab tensor toolbox version 2.6. Available Online, February 2015. http://www.sandia.gov/~tgkolda/TensorToolbox/.

[5] D. H. Bailey. Extra high speed matrix multiplication on the Cray-2. *SIAM Journal of Scientific and Statistical Computing*, 8(3):603–607, 1988.

[6] Grey Ballard, Austin R Benson, Alex Druinsky, Benjamin Lipshitz, and Oded Schwartz. Improving the numerical stability of fast matrix multiplication. *SIAM Journal on Matrix Analysis and Applications*, 37(4):1382–1418, 2016.

[7] Grey Ballard, James Demmel, Olga Holtz, Benjamin Lipshitz, and Oded 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)*, pages 193–204. ACM, 2012.

[8] Austin R. Benson. Fast matrix multiplication. GitHub Repository. `https://github.com/arbenson/fast-matmul`.

[9] Austin R. Benson and Grey 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)*, pages 42–53. ACM, 2015.

[10] Gianfranco Bilardi and Lorenzo De Stefani. The I/O complexity of Strassen's matrix multiplication with recomputation. *arXiv preprint arXiv:1605.02224*, 2016.

[11] Jeff Bilmes, Krste Asanović, Chee whye Chin, and Jim Demmel. Optimizing matrix multiply using PHiPAC: a Portable, High-Performance, ANSI C coding methodology. In *Proceedings of International Conference on Supercomputing*, Vienna, Austria, July 1997.

[12] Dario Bini, Milvio Capovani, Francesco Romani, and Grazia Lotti. O $(n^{2.7799})$ complexity for $n \times n$ approximate matrix multiplication. *Information Processing Letters*, 8(5):234–235, 1979.

[13] Dario Bini and Grazia Lotti. Stability of fast algorithms for matrix multiplication. *Numerische Mathematik*, 36(1):63–72, 1980.

[14] BLAS-like Library Instantiation Software Framework (BLIS). GitHub Repository, 2018. `https://github.com/flame/blis`.

[15] Brice Boyer, Jean-Guillaume Dumas, Clément Pernet, and Wei Zhou. Memory efficient scheduling of Strassen-Winograd's matrix multiplication algorithm. In *Proceedings of the 2009 International Symposium on Symbolic and Algebraic Computation*, ISSAC 09, pages 55–62, New York, NY, USA, 2009. ACM.

[16] R. P. Brent. Error analysis of algorithms for matrix multiplication and triangular decomposition using winograd's identity. *Numerische Mathematik*, 16(2):145–156, Nov 1970.

[17] L.E. Cannon. *A Cellular Computer to Implement the Kalman Filter Algorithm*. PhD thesis, Montana State University, 1969.

[18] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[19] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker. ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. In *Proceedings of the Fourth Symposium on the Frontiers of*

Massively Parallel Computation, pages 120–127. IEEE Comput. Soc. Press, 1992.

[20] clBLAS. GitHub Repository, 2017. `https://github.com/clMathLibraries/clBLAS`.

[21] Pierre Comon, Gene Golub, Lek-Heng Lim, and Bernard Mourrain. Symmetric tensors and symmetric tensor rank. *SIAM Journal on Matrix Analysis and Applications*, 30(3):1254–1279, 2008.

[22] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC 87, pages 1–6, New York, NY, USA, 1987. ACM.

[23] Don Coppersmith and Shmuel Winograd. On the asymptotic complexity of matrix multiplication. *SIAM Journal on Computing*, 11(3):472–492, 1982.

[24] T.D. Crawford and H.F. Schaefer III. An introduction to coupled cluster theory for computational chemists. *Rev. Comp. Chem.*, 14:33–136, 2000.

[25] Paolo D'Alberto. The better accuracy of Strassen-Winograd algorithms (FastMMW). *Advances in Linear Algebra & Matrix Theory*, 4(01):9, 2014.

[26] Paolo D'Alberto, Marco Bodrato, and Alexandru 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.*, 38(1):2:1–2:30, December 2011.

[27] Paolo D'Alberto and Alexandru Nicolau. Adaptive Winograd's matrix multiplications. *ACM Trans. Math. Softw.*, 36(1):3:1–3:23, March 2009.

[28] James Demmel, Ioana Dumitriu, Olga Holtz, and Robert Kleinberg. Fast matrix multiplication is stable. *Numerische Mathematik*, 106(2):199–224, 2007.

[29] E. Di Napoli, D. Fabregat-Traver, G. Quintana-Orti, and P. Bientinesi. Towards an efficient use of the BLAS library for multilinear tensor contractions. *Appl. Math. Comput.*, 235:454–468, 2014.

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

[31] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 14(1):1–17, March 1988.

[32] C.C. Douglas, M. Heroux, G. Slishman, and R.M. Smith. GEMMW - a portable level 3 BLAS Winograd variant of Strassen's matrix-matrix

multiplication algorithm. *J. Computational Physics*, pages 1–10, 1994.

[33] Jean-Guillaume Dumas and Victor Pan. Fast matrix multiplication and symbolic computation. *arXiv preprint arXiv:1612.05766*, 2016.

[34] Brett I. Dunlap. Robust and variational fitting. *Phys. Chem. Chem. Phys.*, 2:2113–2116, 2000.

[35] Thom H. Dunning Jr. Gaussian basis sets for use in correlated molecular calculations. i. the atoms boron through neon and hydrogen. *J. Chem. Phys.*, 90:1007–1023, 1989.

[36] Luke Durant, Olivier Giroux, Mark Harris, and Nick Stam. Inside Volta: The world's most advanced data center GPU. Nvidia Developer Blog, 2017. `https://devblogs.nvidia.com/inside-volta`.

[37] Erik Elmroth, Fred Gustavson, Isak Jonsson, and Bo Kågström. Recursive blocked algorithms and hybrid data structures for dense matrix library software. *SIAM review*, 46(1):3–45, 2004.

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

[39] Kazushige Goto and Robert A. van de Geijn. High-performance implementation of the level-3 BLAS. *ACM Trans. Math. Softw.*, 35(1):4:1–4:14, July 2008.

[40] GOTOBLAS. Available Online, 2010. https://www.tacc.utexas.edu/research-development/tacc-software/gotoblas2.

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

[42] Brian Grayson and Robert van de Geijn. A high performance parallel Strassen implementation. *Parallel Processing Letters*, 6(1):3–12, 1996.

[43] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. FLAME: Formal linear algebra methods environment. *ACM Trans. Math. Softw.*, 27(4):422–455, December 2001.

[44] John A. Gunnels, Greg M. Henry, and Robert A. van de Geijn. A family of high-performance matrix multiplication algorithms. In Vassil N. Alexandrov, Jack J. Dongarra, Benjoe A. Juliano, René S. Renner, and C. J. Kenneth Tan, editors, *Computational Science — ICCS 2001*, pages 51–60, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[45] M. Hanrath and A. Engels-Putzka. An efficient matrix-matrix multiplication based antisymmetric tensor contraction engine for general order coupled cluster. *J. Chem. Phys.*, 133(6):064108, 2010.

[46] A. Hartono, Q. Lu, T. Henretty, S. Krishnamoorthy, H. Zhang, G. Baumgartner, D. E. Bernholdt, M. Nooijen, R. Pitzer, J. Ramanujam, and

P. Sadayappan. Performance optimization of tensor contraction expressions for many-body methods in quantum chemistry. *J. Phys. Chem. A*, 113(45):12715–12723, 2009.

[47] Trygve Helgaker, Poul Jorgensen, and Jeppe Olsen. *Molecular Electronic-Structure Theory*. Wiley, Chichester, NY, first edition, February 2013.

[48] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

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

[50] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, Philadelphia, PA, USA, second edition, 2002.

[51] C-H Huang, Jeremy R Johnson, and Robert W Johnson. A tensor product formulation of Strassen's matrix multiplication algorithm. *Applied Mathematics Letters*, 3(3):67–71, 1990.

[52] Jianyu Huang. BLISlab. GitHub Repository, 2016. `https://github.com/flame/blislab`.

[53] Jianyu Huang. Generating families of practical fast matrix multiplication algorithms. GitHub Repository, 2017. `https://github.com/flame/fmm-gen`.

[54] Jianyu Huang, Devin A. Matthews, and Robert A. van de Geijn. Strassen's algorithm for tensor contraction. *SIAM Journal on Scientific Computing*, 40(3):C305–C326, 2018.

[55] Jianyu Huang, Leslie Rice, Devin A. Matthews, and Robert A. van de Geijn. Generating families of practical fast matrix multiplication algorithms. In *31th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2017)*, pages 656–667, May 2017.

[56] Jianyu Huang, Tyler M. Smith, Greg M. Henry, and Robert A. van de Geijn. Strassen's algorithm reloaded. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 16)*, pages 59:1–59:12. IEEE Press, 2016.

[57] Jianyu Huang and Robert A. van de Geijn. BLISlab: A sandbox for optimizing GEMM. FLAME Working Note #80, TR-16-13, The University of Texas at Austin, Department of Computer Science, 2016.

[58] 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*, volume 1, pages 43–48, May 2006.

[59] Steven Huss-Lederman, Elaine M. Jacobson, Anna Tsao, Thomas Turnbull, and Jeremy R. Johnson. Implementation of Strassen's algorithm for matrix multiplication. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, SC 96, Washington, DC, USA, 1996. IEEE.

[60] IBM ESSL. Available Online, 2018. https://www.ibm.com/support/knowledgecenter/en/SSFHY8/essl_welcome.html.

[61] Intel MKL. Available Online, 2018. https://software.intel.com/en-us/intel-mkl.

[62] Dror Irony, Sivan Toledo, and Alexander Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *Journal of Parallel and Distributed Computing*, 64(9):1017–1026, 2004.

[63] Bo Kågström, Per Ling, and Charles van Loan. GEMM-based level 3 BLAS: High-performance model implementations and performance evaluation benchmark. *ACM Trans. Math. Softw.*, 24(3):268–302, September 1998.

[64] Igor Kaporin. The aggregation and cancellation techniques as a practical tool for faster matrix multiplication. *Theoretical Computer Science*, 315(2-3):469–510, 2004.

[65] Elaye Karstadt and Oded Schwartz. Matrix multiplication, a little faster. In *Proceedings of the 29th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA 17, New York, NY, USA, 2017. ACM.

[66] Jeremy Kepner and John Gilbert. *Graph algorithms in the language of linear algebra*. SIAM, 2011.

[67] Andrew Kerr, Duane Merrill, Julien Demouth, and John Tran. CUTLASS: Fast linear algebra in CUDA C++. Nvidia Developer Blog, Dec 2017. https://devblogs.nvidia.com/cutlass-linear-algebra-cuda.

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

[69] R. Krishnan, J. S. Binkley, R. Seeger, and J. A. Pople. Selfconsistent molecular orbital methods. xx. a basis set for correlated wave functions. *J. Chem. Phys.*, 72:650–654, 1980.

[70] Bharat Kumar, C-H Huang, P Sadayappan, and Rodney W Johnson. A tensor product formulation of Strassen's matrix multiplication algorithm with memory reduction. *Scientific Programming*, 4(4):275–289, 1995.

[71] Julian Laderman, Victor Pan, and Xuan-He Sha. On practical algorithms for accelerated matrix multiplication. *Linear Algebra and Its Applications*, 162:557–588, 1992.

[72] P. W. Lai, H. Arafat, V. Elango, and P. Sadayappan. Accelerating Strassen-Winograd's matrix multiplication algorithm on GPUs. In *20th Annual International Conference on High Performance Computing*, pages 139–148, Dec 2013.

[73] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Soft.*, 5(3):308–323, Sept. 1979.

[74] François Le Gall. Powers of tensors and fast matrix multiplication. In *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation*, ISSAC 14, pages 296–303, New York, NY, USA, 2014. ACM.

[75] J. Li, C. Battaglino, I. Perros, J. Sun, and R. Vuduc. An input-adaptive and in-place approach to dense tensor-times-matrix multiply. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC 15, pages 76:1–76:12, New York, NY, USA, 2015. ACM.

[76] J. Li, S. Ranka, and S. Sahni. Strassen's matrix multiplication on GPUs. In *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, pages 157–164, Dec 2011.

[77] 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*, 9(5):345–389, May 1997.

[78] Xiaoye S. Li, James W. Demmel, David H. Bailey, Greg Henry, Yozo Hida, Jimmy Iskandar, William Kahan, Suh Y. Kang, Anil Kapur, Michael C. Martin, Brandon J. Thompson, Teresa Tung, and Daniel J.

Yoo. Design, implementation and testing of extended and mixed precision BLAS. *ACM Trans. Math. Softw.*, 28(2):152–205, June 2002.

[79] Benjamin Lipshitz, Grey Ballard, James Demmel, and Oded Schwartz. Communication-avoiding parallel Strassen: Implementation and performance. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC 12)*, pages 101:1–101:11. IEEE, 2012.

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

[81] Qingshan Luo and John B. Drake. A scalable parallel Strassen's matrix multiplication algorithm for distributed-memory computers. In *Proceedings of the 1995 ACM Symposium on Applied Computing*, SAC 95, pages 221–226, New York, NY, USA, 1995. ACM.

[82] Dmitry I. Lyakh. An efficient tensor transpose algorithm for multicore CPU, Intel Xeon Phi, and NVidia Tesla GPU. *Comput. Phys. Commun.*, 189:84–91, 2015.

[83] W. Ma, S. Krishnamoorthy, O. Villa, K. Kowalski, and G. Agrawal. Optimizing tensor contraction expressions for hybrid CPU-GPU execution. *Cluster Comput.*, 16(1):131–155, 2011.

[84] Devin A. Matthews. TBLIS: Tensor-based library instantiation software. Github Repository, 2016. https://github.com/devinamatthews/tblis.

[85] Devin A. Matthews. High-performance tensor contraction without transposition. *SIAM Journal on Scientific Computing*, 40(1):C1–C24, 2018.

[86] Nvidia. CUTLASS: CUDA templates for linear algebra subroutines (v0.1.0). GitHub Repository, 2017. https://github.com/NVIDIA/cutlass/releases/tag/v0.1.0.

[87] Nvidia. cuBLAS. Available Online, 2018. https://developer.nvidia.com/cublas.

[88] Nvidia. CUDA C programming guide. Available Online, 2018. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[89] Nvidia. Summit GPU supercomputer enables smarter science. Nvidia Developer Blog, 2018. https://devblogs.nvidia.com/summit-gpu-supercomputer-enables-smarter-science/.

[90] OpenBLAS, an optimized BLAS library. Available Online. http://www.openblas.net.

[91] V. Ya. Pan. Strassen's algorithm is not optimal trilinear technique of aggregating, uniting and canceling for constructing fast algorithms

for matrix operations. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, SFCS 78, pages 166–176, Washington, DC, USA, 1978. IEEE Computer Society.

[92] V. Ya. Pan. Trilinear aggregating with implicit canceling for a new acceleration of matrix multiplication. *Computers & Mathematics with Applications*, 8(1):23–34, 1982.

[93] Devangi N. Parikh, Jianyu Huang, Margaret E. Myers, and Robert A. van de Geijn. Learning from optimizing matrix-matrix multiplication. In *8th NSF/TCPP Workshop on Parallel and Distributed Computing Education (EduPar-18)*. IEEE, 2018.

[94] E. Peise, D. Fabregat-Traver, and P. Bientinesi. On the performance prediction of BLAS-based tensor contractions. In S. A. Jarvis, S. A. Wright, and S. D. Hammond, editors, *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, number 8966 in Lecture Notes in Computer Science, pages 193–212. Springer International Publishing, 2014.

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

[96] Krishnan Raghavachari, Gary W. Trucks, John A. Pople, and Martin

Head-Gordon. A fifth-order perturbation comparison of electron correlation theories. *Chemical Physics Letters*, 157(6):479 – 483, 1989.

[97] Leslie Rice. Performance optimization for the K-Nearest Neighbors kernel using Strassen's algorithm. 2017. Undergraduate Honors Thesis, The University of Texas at Austin.

[98] Martin D. Schatz, Tze Meng Low, Robert A. van de Geijn, and Tamara G. Kolda. Exploiting symmetry in tensors for high performance: Multiplication with symmetric tensors. *SIAM Journal on Scientific Computing*, 36(5):C453–C479, 2014.

[99] Martin D. Schatz and Jack Poulson Robert A. van de Geijn. Parallel matrix multiplication: A systematic journey. *SIAM Journal on Scientific Computing*, 38(6):C748–C781, 2016.

[100] Arnold Schönhage. Partial and total matrix multiplication. *SIAM Journal on Computing*, 10(3):434–455, 1981.

[101] Jacob Scott, Olga Holtz, and Oded Schwartz. Matrix multiplication I/O-complexity by path routing. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 15)*, pages 35–45. ACM, 2015.

[102] G. E. Scuseria, A. C. Scheiner, T. J. Lee, J. E. Rice, and H. F. Schaefer. The closed-shell coupled cluster single and double excitation (CCSD) model for the description of electron correlation. A comparison with

configuration interaction (CISD) results. *J. Chem. Phys.*, 85(5):2881, March 1987.

[103] Isaiah Shavitt and Rodney J. Bartlett. *Many-Body Methods in Chemistry and Physics: MBPT and Coupled-Cluster Theory.* Cambridge University Press, Cambridge ; New York, first edition, August 2009.

[104] A. V. Smirnov. The bilinear complexity and practical algorithms for matrix multiplication. *Computational Mathematics and Mathematical Physics*, 53(12):1781–1795, 2013.

[105] Tyler M. Smith. Multilevel Optimized Matrix-matrix Multiplication Sandbox (MOMMS). GitHub Repository. `https://github.com/tlrmchlsmth/momms`.

[106] Tyler M. Smith. Theory and practice of classical matrix-matrix multiplication for hierarchical memory architectures. 2017. PhD thesis, The University of Texas at Austin.

[107] Tyler M. Smith and Robert A van de Geijn. Pushing the bounds for matrix-matrix multiplication. *arXiv preprint arXiv:1702.02017*, 2017.

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

[109] Tyler M. Smith, Robert A. van de Geijn, Mikhail Smelyanskiy, and Enrique S. Quintana-Ortí. Toward ABFT for BLIS GEMM. FLAME Working Note #76. Technical Report TR-05-2015, The University of Texas at Austin, Department of Computer Sciences, June 2015.

[110] Edgar Solomonik, Devin Matthews, Jeff R. Hammond, John F. Stanton, and James Demmel. A massively parallel tensor contraction framework for coupled-cluster computations. *Journal of Parallel and Distributed Computing*, 74(12):3176–3190, 2014.

[111] Paul Springer and Paolo Bientinesi. Tensor contraction benchmark v0.1. GitHub Repository, December 2016. `https://github.com/hpac/tccg/tree/master/benchmark`.

[112] Paul Springer and Paolo Bientinesi. Design of a high-performance GEMM-like tensor-tensor multiplication. *ACM Trans. Math. Softw.*, 44(3):28:1–28:29, January 2018.

[113] Paul Springer, Jeff R. Hammond, and Paolo Bientinesi. TTC: A high-performance compiler for tensor transpositions. *ACM Trans. Math. Softw.*, 44(2):15:1–15:21, August 2017.

[114] Andrew James Stothers. On the complexity of matrix multiplication. 2010. PhD thesis, The University of Edinburgh.

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

[116] Volker Strassen. The asymptotic spectrum of tensors and the exponent of matrix multiplication. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, SFCS 86, pages 49–54, Washington, DC, USA, 1986. IEEE Computer Society.

[117] Mithuna Thottethodi, Siddhartha Chatterjee, and Alvin R. Lebeck. Tuning Strassen's matrix multiplication for memory efficiency. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing (SC 98)*, pages 1–14. IEEE, 1998.

[118] Robert A. van de Geijn. *Using PLAPACK: Parallel Linear Algebra Package*. The MIT Press, 1997.

[119] Robert A. van de Geijn, Jianyu Huang, Margaret E. Myers, Devangi N. Parikh, and Tyler M. Smith. Lowering barriers into HPC through open education. In *2017 Workshop on Education for High-Performance Computing (EduHPC)*. IEEE, 2017.

[120] Robert A. van de Geijn and Jerrell Watts. SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, April 1997.

[121] Field G. Van Zee. `libflame`*: The Complete Reference*. `www.lulu.com`, 2009.

[122] Field G. Van Zee, Tyler Smith, Francisco D. Igual, Mikhail Smelyanskiy, Xianyi Zhang, Michael Kistler, Vernon Austel, John Gunnels, Tze Meng

Low, Bryan Marker, Lee Killough, and Robert A. van de Geijn. The BLIS framework: Experiments in portability. *ACM Transactions on Mathematical Software*, 42(2):12:1–12:19, June 2016.

[123] Field G. Van Zee and Tyler M. Smith. Implementing high-performance complex matrix multiplication via the 3m and 4m methods. *ACM Trans. Math. Softw.*, 44(1):7:1–7:36, July 2017.

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

[125] Pete Warden. Why GEMM is at the heart of deep learning. `https://petewarden.com/2015/04/20/why-gemm-is-at-the-heart-of-deep-learning/`, 2015.

[126] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *SC 98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, pages 38–38, Nov 1998.

[127] J. L. Whitten. Coulombic potential energy integrals and approximations. *The Journal of Chemical Physics*, 58(10):4496–4501, 1973.

[128] V. V. Williams. Multiplying matrices faster than Coppersmith-Winograd. In *Proceedings of the Forty-fourth Annual ACM Symposium on Theory of Computing*, STOC 12, pages 887–898, New York, NY, USA, 2012. ACM.

[129] Shmuel Winograd. On multiplication of 2×2 matrices. *Linear algebra and its applications*, 4(4):381–388, 1971.

[130] Kamen Yotov, Xiaoming Li, Gang Ren, MJS Garzaran, David Padua, Keshav Pingali, and Paul Stodghill. Is search really necessary to generate high-performance BLAS? *Proceedings of the IEEE*, 93(2):358–386, 2005.

[131] Chenhan D. Yu, Jianyu Huang, Woody Austin, Bo Xiao, and George Biros. Performance optimization for the K-Nearest Neighbors kernel on x86 architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC 15, pages 7:1–7:12, New York, NY, USA, 2015. ACM.

Vita

Jianyu Huang was born in Jiangsu, China. He received his Bachelor's degree in Computer Science and Technology from Beihang University (BUAA), Beijing, China in 2013. Since August 2013, he has been a PhD student in the Department of Computer Science at the University of Texas at Austin, working with Robert van de Geijn. His research has been supported by the Microelectronics and Computer Development Fellowship, Graduate School Summer Fellowship, and Graduate Research Assistantship at UT Austin. He was a research and software engineering intern at Microsoft Research Asia, VMware Inc., and Intel Labs.

Email address: jianyu.huang@utexas.edu

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.