Copyright
by
Tyler Michael Smith
2017

The Dissertation Committee for Tyler Michael Smith
certifies that this is the approved version of the following dissertation:

Theory and Practice of Classical Matrix-Matrix Multiplication for Hierarchical Memory Architectures

Committee:

Robert van de Geijn, Supervisor

Enrique Quintana-Ortí, Supervisor

Keshav Pingali

Donald Fussell

Theory and Practice of Classical Matrix-Matrix Multiplication for Hierarchical Memory Architectures

by

Tyler Michael Smith, B.S.

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
December 2017

Dedicated to Anisha.

Acknowledgments

I wish to thank the multitudes of people who helped me. Time would fail me to tell of . . .

Theory and Practice of Classical Matrix-Matrix Multiplication for Hierarchical Memory Architectures

Publication No. ________________

Tyler Michael Smith, Ph.D.
The University of Texas at Austin, 2017

Supervisors: Robert van de Geijn
Enrique Quintana-Ortí

Matrix-matrix multiplication is perhaps the most important operation used as a basic building block in dense linear algebra. A computer with a hierarchical memory architectures has memory that is organized in layers, with small and fast memories close to the processor, and big and slow memories further away from it. Classical MMM is an operation particularly suited for such architectures, as it exhibits a large degree of data reuse, so expensive data movements can be amortized over a lot of computation. This dissertation advances the theory of how to optimally reuse data during MMM on hierarchical memory architectures, and it uses this understanding to develop new practical algorithms for matrix-matrix multiplication that exhibit improved properties related to data movement.

Table of Contents

Acknowledgments **v**

Abstract **vi**

List of Tables **xi**

List of Figures **xii**

Chapter 1. Introduction **1**

1.1 Motivation 1
1.2 Problem definition 3
1.3 Contributions 4
1.4 Organization 5

Chapter 2. Related Work **7**

2.1 A history of linear algebra packages 7
2.2 Literature on theoretical I/O lower bounds 9
2.3 Literature on practical MMM 10
  2.3.1 Strategy and tactics for practical MMM 11
2.4 State-of-the-art matrix multiplication 14
  2.4.1 A brief description of Goto's algorithm 16
  2.4.2 BLIS 18

Chapter 3. I/O Lower Bounds **20**

3.1 Introduction 20
3.2 Problem Definition 22
  3.2.1 Prior approaches 22
  3.2.2 Our approach 23
3.3 Lower Bound Proof 24

3.3.1 High-level strategy 24
3.3.2 Employing the Loomis-Whitney inequality 25
3.3.3 A lower bound for $C := AB + C$ 26
3.3.4 Improving the lower bound for $C := AB$ 27
3.4 Summary 28

Chapter 4. Optimal Algorithms or: The Lower Bound is Tight 30
4.1 Introduction 30
4.2 Optimal and read-optimal Algorithms 30
4.2.1 Algorithms for one level of cache 31
4.2.2 Algorithms for different shapes of MMM 36
4.2.3 A balancing act 38
4.3 Summary 39

Chapter 5. A Family of Algorithms for Multiple Levels of Cache 41
5.1 Introduction 41
5.2 A Family of Algorithms 42
5.2.1 The outer loop for the L_{h-1} cache 43
5.2.2 The inner loop for the L_{h-1} cache 47
5.2.3 Classifying matrix partitions. 48
5.2.4 Special cases for the family of algorithms 51
5.3 Multilevel Cache Tradeoffs 52
5.3.1 Impact of optimizing for the L_h cache on the L_{h-1} I/O cost 52
5.3.2 Impact of optimizing for L_{h-1} on the L_h I/O cost 56
5.3.3 Skipping caches 59
5.4 Experiments 62
5.4.1 Experimental setup 62
5.4.2 Optimizing for the L3 cache 64
5.4.3 Optimizing for the L4 cache 72
5.4.4 Algorithms for different shapes of matrices 77
5.5 Summary 80

Chapter 6. Cache Aware Multithreaded Parallelization **84**
6.1 Introduction 84
6.2 Parallelization of loops within Goto's Algorithm 85
 6.2.1 The first loop around the micro-kernel 85
 6.2.2 The second loop around the micro-kernel 88
 6.2.3 The third loop around the inner-kernel 90
 6.2.4 The fourth loop around the inner-kernel 91
 6.2.5 The outer-most loop 93
 6.2.6 Parallelism within the micro-kernel 93
6.3 MMM Parallelization for Intel Xeon Phi KNC 94
 6.3.1 Architectural Details 94
 6.3.2 The BLIS implementation on the Intel Xeon Phi 95
 6.3.3 Which loops to parallelize 96
 6.3.4 Parallelism within cores 97
 6.3.5 Parallelism between cores 98
 6.3.6 Performance results 99
6.4 MMM Parallelization for IBM PowerPC A/2 102
 6.4.1 Architectural Details 102
 6.4.2 The BLIS implementation on the IBM PowerPC A2 . . 103
 6.4.3 Which loop to parallelize 104
 6.4.4 Performance results 105
6.5 Paralleling the family of algorithms 107
 6.5.1 Parallelizing outside the L_{h-1} subproblem 108
 6.5.2 Parallelizing within the L_{h-1} subproblem 109
 6.5.3 Summary 110
6.6 Summary 110

Chapter 7. Conclusion **112**
7.1 Results 112
7.2 Future work 114

Appendices **116**

Appendix A. Table of Symbols 117

Appendix B. Constrained Global Maximum of $\sqrt{xyz}$ 118

Appendix C. Bandwidth requirements for different datatypes 120

Bibliography 122

Vita 136

List of Tables

5.1 Machines used in experiments in this paper. 63

C.1 Typical element sizes, computation rate, and bandwidth requirements for MMM relative to single precision. 121

List of Figures

2.1 Diagram of Goto's Algorithm implemented in BLIS. 15

4.1 Three algorithms for matrix multiplication that attain the lower bound for a single level of cache. 32

5.1 Illustration showing the possible scenarios when partitioning for the L_h and L_{h-1} caches in the family of MMM algorithms when Resident A or Resident B is encountered at the L_h cache. . . . 44

5.2 Illustration showing the possible scenarios when partitioning for the L_h and L_{h-1} caches in the family of MMM algorithms when Resident C is encountered at the L_h cache. 45

5.3 This plot determines how much worse in terms of number of inputs to the L_h cache an algorithm will be if it optimizes for the L_{h-1} cache as well. If it is too costly, one may choose not to optimize for the L_{h-1} cache. Vertical lines mark cache size ratios for the Intel i7-7700K. For example, $\frac{S_2}{S_1}$ is too large to optimize for both the L_1 and L_2 caches, but it is reasonable to optimize for both L_2 and L_3. 53

5.4 An algorithm's L_{h-1} versus L_h efficiency in terms of flops per I/O. Moving from left to right, the L_{h-1} blocksizes increase, decreasing the I/O costs associated with the L_{h-1} streamed matrices. This causes the L_h blocksizes to decrease, increasing the I/O costs associated with the L_h streamed matrices and the L_{h-1} resident matrix. In this plot, S_h is 256^2, and S_{h-1} is 128^2. 58

5.5 Two algorithms for MMM. Left: $B_3A_2C_0$. Right: Goto's Algorithm. 65

5.6 Roofline models for an Intel i7-7700K at 4.2GHz with 4 cores under two bandwidth conditions. For algorithms displayed on the plots, the y-axis is measured and the x-axis is theoretical. Bottom: Two channels of DDR4 is set to DDR-3200, for a peak bandwidth of 51200 MB/s. Top: One channel of DDR4 is set to DDR-800, for a peak bandwidth of 6400 MB/s. 67

5.7 Performance of matrix-matrix multiplication on an Intel i7-7700K for square matrices, varying problem size and available bandwidth. Matrices are stored prepacked. 70

5.8 Practical comparision of performance against state-of-the-art open source and vendor libraries, for both high and low bandwidth scenarios. Matrices are passed in as column-major matrices, so packing is performed. 71

5.9 An algorithm for MMM optimizing the number of inputs to both the L4 and the L2 caches. It places a square block of C in the L4 cache, and a square block of A in the L2 cache. The L3 cache is "skipped", but the blocksizes are such that the L4 guest matrix, B, is reused from the L3 cache. 73

5.10 Performance of matrix-matrix multiplication for square matrices, varying problem size and available bandwidth. Matrices are stored prepacked. 74

5.11 Matrix-matrix multiplication performance on an Intel i7-5775C with an 128MB L4 cache for a low bandwidth scenario (1 channel of DDR3-800) and a high bandwidth scenario (2 channels of DDR3-2400). 75

5.12 Two algorithms for MMM for three levels of cache. Left: $A_3B_2C_0$. Right: $C_3A_2C_0$. 78

5.13 Matrix-matrix multiplication performance for different problem shapes on an Intel i7-7700K under a low-bandwidth scenario (1 channel of DDR4-800). 79

6.1 Loops implementing Goto's Algorithm in BLIS. 86

6.2 Illustration of which parts of the memory hierarchy each block of A and B reside in during the execution of the micro-kernel with BLIS. 86

6.3 Diagram of Goto's Algorithm implemented in BLIS. 87

6.4 Left: the micro-kernel. Right: the first loop around the micro-kernel. 88

6.5 The second loop around the micro-kernel. 89

6.6 The third loop around the micro-kernel (first loop around Goto's inner kernel). 90

6.7 The fourth loop around the micro-kernel (second loop around Goto's inner kernel). 92

6.8 Parallelization of the p_c loop requires local copies of the block of C to be made, which are summed upon completion of the loop. 92

6.9 The fifth (outer) loop around the micro-kernel. 93

6.10 MMM performance using different parallelization schemes within BLIS on the Intel Xeon Phi. 'i_c:n way' indicates n-way parallelization of the third loop (indexed by i_c) around the micro-kernel, and 'j_r:n way' indicates n-way parallelization of the second loop (indexed by j_r) around the micro-kernel. 99

6.11 MMM performance comparision of BLIS and MKL on the Intel Xeon Phi KNC. 100

6.12 MMM with different parallelization schemes for the IBM PowerPC A2. 'j_r:n way' indicates n-way parallelization of the second loop (indexed by j_r) around the micro-kernel, and 'i_r:n way' indicates n-way parallelization of the first loop (indexed by i_r) around the micro-kernel. 106

6.13 MMM performance comparision of BLIS and ESSL on the PowerPC A2 Blue Gene Q. 107

Chapter 1

Introduction

Hierarchical memory architectures have been used for decades to reduce the cost of data access. One can conceive of a memory hierarchy being laid out as a pyramid, with several layers of memory. The layer of memory closes to the processor is the smallest and fastest, and each subsequent layer of memory is bigger, slower, and further from the processor compared to the layer above it. The goal for this dissertation is efficient computation for matrix operations on machines with hierarchical memories. The thesis is that there are three algorithms for matrix-matrix multiplication that, considering one layer of memory, are optimal in terms of the number of reads from slower layers. These three algorithms can be composed in order to encounter one of them at each layer of the memory hierarchy. This dissertation contains both theoretican and practical advances related to this thesis.

1.1 Motivation

High-performance dense linear algebra libraries are of great practical importance, as they are often used as building blocks of applications used in scientific computing and data analysis. Efficient implementations are widely available on almost any platform, and dense linear algebra operations are expected by application programmers to achieve efficiencies close to the peak performance of the machine. In addition to its practical importance, the implementation of high-performance matrix operations are of pedagogical significance, since they are often used to illustrate how to attain high performance on a novel architecture.

Research regarding practical dense linear algebra software has been ongoing for at least the past fourty years [33], and research regarding theory of optimal use of hierarchical memory for linear algebra has been ongoing for at least the past three decades [46]. Costs related to data movement are often the most significant overhead when efficient computing is a goal. Caches are small and fast buffers used to facilitate a reduction in data movement costs. When data has good temporal locality, input-output (I/O) costs can be reduced by reusing data while it is in a cache. Modern computers use multiple levels of cache, where higher levels of cache are smaller, faster, and nearer to the processor, and lower levels of cache are bigger, slower, and further.

Matrix-matrix multiplication (MMM) is an operation that many applications depend upon to attain high performance. Scientists and engineers have come to expect high-performance implementations of the Basic Linear Algebra Subprograms (BLAS) [53, 27, 26] interface, providing functionality for vector-vector, matrix-vector, and matrix-matrix operations. In turn, dense linear algebra (DLA) libraries such as LAPACK [5] and libflame [39, 76, 77] use BLAS operations as building blocks to provide high-performance implementations of more sophisticated operations like matrix factorizations for single-node hierarchical memory systems. Then, distributed-memory DLA libraries such as ScaLAPACK [22], PLAPACK [7], and Elemental [63] rely both on the BLAS and on single-node DLA libraries. As such, MMM is one of the most important operations on the bottom of a rich ecosystem of DLA libraries that are depended upon by scientists in high-performance and scientific computing.

When developing a dense linear algebra library, an engineer will typically employ a set of heuristics in order to optimize for performance, energy consumption, or workspace, or to achieve some other goal. Ideally, heuristics should have firm theoretical underpinnings such that they can be shown to be optimal according to a model of computation. Therefore this dissertation has developed realistic but idealized models for computation and then has used these models to develop theoretically optimal results. These results have been be used to understand practical algorithms that attain high-performance and are efficient on modern and future hardware in the domain of dense linear algebra.

1.2 Problem definition

In this dissertation, we will focus on the conventional (classic) MMM operation $C \mathrel{+}= AB$. In all cases, C is $m \times n$, A is $m \times k$, and B is $k \times n$. We focus on the case where all matrices are dense and unstructured, however, the algorithms presented in this dissertation should apply to the cases where matrices are dense and structured as well. Classic MMM requires $2mnk$ floating point operations for dense unstructured matrices.

The goal of this dissertation is efficient algorithms for matrix operations for hierarchical memory architectures. We will describe memory hierarchies in the following manner: We name the n levels of the memory hierarchy $L_0, L_1, ..., L_{n-1}$, for every h, $1 < h < n$, the cost of accessing L_h is more expensive than accessing L_{h-1}. Each level of memory L_h has a size S_h, and $S_h > S_{h-1}$ for every h, $1 < h < n$.

1.3 Contributions

This dissertation makes the following contributions.

- It proves a new and improved theoretical I/O lower bound for MMM. For a machine with two layers of memory –one fast layer with capacity S, and one slow layer with unlimited capacity– MMM requires at least $\frac{2mnk}{\sqrt{S}}$ reads and writes to and from slow memory.
- It analyzes the I/O optimality of a family of three algorithms for a single layer of fast memory. One of these three algorithms has an I/O cost of $\approx \frac{2mnk}{\sqrt{S}} + mn$. Since it attains the lower bound, the algorithm is I/O optimal and this proves that the I/O lower bound is tight. The other two algorithm attain the lower in a weaker sense; that is if one ignores the I/O cost associated with writes to slow memory, which often occurs if the slow memory being written to is a cache.
- It shows that considering different "shapes" of MMM operations, where some of m, n, and k are small and other large, some of those three algorithms will be optimal (when ignoring the cost of writes), and others will be suboptimal. It gives the conditions of optimality for each.
- It describes a new family of algorithms for multiple levels of cache. By composing two loops per level of cache, one of the three algorithms for a single level of cache can be encountered at each layer of the memory hierarchy. It then demonstrates the performance of a practical algorithm for MMM that is 45% more efficient than the state-of-the-art Goto's algorithm on a conventional architecture when the matrices are too large to fit into cache and the ratio between the rate of computation and the

rate of data movement from main memory is made artificially high (and representative of possible future architectures). This performance benefit is accomplished by utilizing the L_3 cache more effectively than Goto's algorithm does.

- It shows how the cache hierarchy can be taken into account when parallelizing matrix operations. When caches are shared between threads, it is beneficial to parallelize differently than when threads have independent caches. It then shows how these ideas can be implemented by demonstrating practical performance for the many-core IBM Blue Gene/Q PowerPC A2 and Intel Xeon Phi Knight's Corner architectures.
- It derives algorithms to utilize aggregate fast memories. That is, on a multiprocessor system where each processor has its own proprietary fast memory, the algorithm is optimal both for the number of elements moved into and out of each individual fast memory, and also optimal for the number of elements read into and out of slow memory. This is advantageous if it is less expensive for a processor to access another processor's fast memory than it is to access slow memory.

1.4 Organization

This dissertation is organized as follows:

- Chapter 2 it outlines the literature related to this present work. The related work is treated with more detail in later chapters.
- Chapter 3 gives a proof of an improved I/O lower bound for MMM.

- Chapter 4 analyzes three algorithms for MMM for machines with a single layer of fast memory. An algorithm is shown to obtain the lower bound from Chapter 3, so the lower bound is said to be tight, and the algorithm is said to be optimal.

- Chapter 5 derives a family of algorithms for MMM for machines with multiple levels of cache. By composing two loops per level of cache, one of the three algorithms from Chapter 4 can be encountered at each level of cache.

- Chapter 6 Discusses how parallelism can be obtained within the family of algorithms presented in Chapter 5. It then describes in detail the properties of parallelizing each of the loops within the BLIS implementation of Goto's algorithm, a member of the family of algorithm from Chapter 5.

In this dissertation, Chapter 3 is very theoretical, and each chapter is more practical than the one before it.

Chapter 2

Related Work

In this chapter, we will look at related work, including a brief history of linear algebra packages, followed by a brief description of the state-of-the-art implementation and algorithms for matrix multiplication.

2.1 A history of linear algebra packages

First we will describe a history of linear algebra packages and the building blocks of the dense linear algebra software stack. One of the earliest examples of such a package is EISPACK by Garbow [33], which provided routines for obtaining the eigenvalues and eigenvectors of a matrix. Lawson et al. [53] introduced the Basic Linear Algebra Subprograms (BLAS). This was a set of 38 simple FORTRAN routines operating on vectors. Examples include the dot product of two vectors, the scaling of a vector, and a Givens rotation. The point of such a library was to standardize an interface for a set of routines such that one could always expect an efficient implementation to exist on any given machine. This standard BLAS interface facilitated portable high performance for the vector computers of the 1970s. Around the same time, Dongarra et al. [25] introduced the LINPACK package that implemented higher level functionality such as matrix factorizations in terms of the BLAS.

In the late 1980s, with the cost of a floating-point arithmetic operation (flop) getting cheaper relative to the cost of a memory operation (memop), and with the advent of hierarchical memory, operations solely on vectors could no longer run efficiently, as such operations provide little opportunity for data reuse. Within the BLAS, this was rectified by the introduction of the level-2 BLAS, extending the BLAS with a set of matrix-vector routines (such as a matrix-vector multiplication or a rank-1 update) [27]. This provided more opportunities for data reuse.

Shortly thereafter, Dongarra et al. [26] introduced the level-3 BLAS. This contained matrix-matrix operations (such as MMM) that, when working with data that can fit into fast memory, can amortize the $\mathcal{O}(n^2)$ memops needed to perform the matrix-matrix operation over $\mathcal{O}(n^3)$ flops. The BLAS level-1, 2, and 3 interfaces have become so ubiquitous that often DLA operations are referred to as their BLAS function names. For example the MMM operation $C := \alpha AB + \beta C$ (where α and β are scalars) is often referred to as GEMM. LINPACK, which cast its computation in terms of the level-1 BLAS, was no longer efficient, leading to the introduction of LAPACK [5].

Gunnels et al. [39, 38] introduced the Formal Linear Algebra Methods Environment or FLAME. It made two important contributions. First, it encodes computation in terms of partitioned matrices and performs operations on these partitions, rather than in terms of indices into arrays. This allowed libflame to be more pedagogical, as the code itself looks like what one might write on a whiteboard when describing an algorithm. Secondly, libflame provided a method of deriving algorithms for linear algebra operations that are proven to be correct [12, 75, 13].

Libraries such as PLASMA [17] and SuperMatrix [19] build on top of libraries like LAPACK and libflame, respectively, to provide runtime parallelization of DLA operations, where matrices are broken into blocks, and operations are implemented as directed acyclic graphs operating on those blocks. The advantage is that this technique avoids tricky load balancing issues that can arise when parallelizing complicated operations.

Finally, there are several libraries targeted towards distributed-memory clusters of computers. Much as LAPACK and FLAME cast computation in terms of the BLAS, distributed dense linear algebra libraries such as ScaLAPACK [22], PLAPACK [7], and Elemental [63] cast much of their computation in terms of BLAS, LAPACK, and libflame, but this dissertation is not concerned with distributed-memory architectures.

2.2 Literature on theoretical I/O lower bounds

The seminal paper by Hong and Kung [46] describes the *red-blue pebble game*, in which computation is described as a directed acyclic graph (DAG). If a blue pebble is placed on a vertex, the value associated with that vertex is in slow memory, and if the vertex instead has a red pebble, the value is in fast memory. There are a finite number of red pebbles, and a small set of rules for playing the game. The goal of the game is to minimize the number of transitions between red and blue to complete computation. Then, this minimum number of transitions is equivalent to the lower bound on the I/O complexity for a computation. Using this model, the minimum number of memory movements required for matrix-matrix multiplication is $\mathcal{O}(\frac{n^3}{\sqrt{S}})$, where S is the number of red pebbles. Savage [64] extends the Hong-Kung model to memories with multiple layers of fast memory. For MMM, it only proves the same lower bound from [46] is true at every level of the memory hierarchy.

Irony, et al. [49] uses the same strategy as [46] to prove lower bounds for MMM for both hierarchical and distributed memory architectures. The main concern of Irony, et al. is attaining the lower bound for MMM on distributed memory architectures using 3D algorithms. An innovation of this paper was using the Loomis-Whitney inequality [55] to find an upper bound on the amount of computation that can be performed using a certain number of elements each of A, B and C.

Dongarra, et al. [24] improved upon the coefficient of the lower bounds for MMM, showing that MMM must incur an I/O cost of $\frac{3\sqrt{3}mnk}{2\sqrt{2}}$. This improvement came from the implicit assumption that computation is performed via fused multiply-add instructions.

Ballard, et al. [8] extended the lower bounds results in [49] to operations beyond MMM, like the LU and Cholesky factorizations. Generalizing the lower bounds to these other operations made the coefficient on the leading term less tight. This is because in order to generalize the proof to handle these operations, the matrices A, B, and C could no longer be assumed to be distinct.

2.3 Literature on practical MMM

Numerous papers have been written about how to implement the operations in the aforementioned linear algebra packages. Here, we focus on the papers concerning the practical implementation of MMM. Much of this work has to do with answering the question: How can data be moved through the memory hierarchy optimally?

2.3.1 Strategy and tactics for practical MMM

Autotuning Autotuning is a technique where the best values of various parameters like blocksizes are determined by empirically measuring performance when those parameters are adjusted. Autotuning was introduced in dense linear algebra by the code generator system PHiPAC [14], which stands for Portable High Performance ANSI C.

The ideas behind PHiPAC were built upon by Whaley and Dongarra [82], in the BLAS library Automatically Tuned Linear Algebra Software (ATLAS). The major contribution of ATLAS was introducing the idea of implementing matrix-matrix multiplication in terms of a basic unit of computation, often known today as a kernel. In ATLAS it was originally called an on-chip multiply.

The algorithm described in [82] specifies two possible loop orderings used by ATLAS, depending on the problem size. Regardless, of the loop ordering chosen, computation is cast in terms of block dot-products where C is $n_b \times n_b$, and the k dimension is much larger. The next step partitions this block-dot product along the k dimension with blocksize n_b. Then the on-chip multiply occurs, updating the block of C with an $n_b \times n_b$ block of A times an $n_b \times n_b$ block of B. During the on-chip multiply, computation is arranged so that either the $n_b \times n_b$ block of A or the $n_b \times n_b$ block of B fills most of cache, streaming the other operands.

The loop around ATLAS's on-chip multiply is in the k dimension, and thus each iteration uses a different block of A and B. Therefore each time an on-chip multiply executes, an $n_b \times n_b$ block of each matrix A, B, and C must be read from a slower level of memory. There are $3n_b^2$ reads and n_b^2 writes per on-chip multiply, so there are $\frac{3}{2n_b}$ reads per flop from slow memory. Since one

$n_b \times n_b$ block must fit into cache, $n_b \leq \sqrt{S}$. Thus the algorithm from [82] has an I/O cost of at least $\frac{3mnk}{\sqrt{S}}$. This is 50% larger than the tight lower bound in [68]. This present paper improves upon [82] by introducing algorithms that attain much closer to the I/O lower bound.

Model-based MMM Gunnels et al. [40] describes a family of algorithms for GEMM. It shows that given the shape of matrix-matrix multiplication that is being executed at some level of cache, there are two locally optimal choices for what shape of matrix-matrix multiplication must happen at the next highest level of cache. This forms a tree of locally optimal decisions, and each path from root to leaf is a member of a family of algorithnms.

Yotov et al. [87] takes an algorithm similar to that of ATLAS [82], but adopts the opposite stance: Identifying optimal or near-optimal block sizes for GEMM can be done analytically, and thus empirical search is not necessary. More recently, Low et al. [56] took the ideas behind Yotov et al., and applied them to the more modern GotoBLAS approach, to be discussed later [35].

Cache oblivious algorithms. Most of the work presented above is related to algorithms that explicitly partition matrices to fit into cache. These are called *cache-aware* algorithms. An alternate technique is to use a divide-and-conquer algorithm, where each divide step reduces the size of working set of data. At some point, the size of this working set becomes small enough that it naturally fits into cache. Algorithms that attempt to use cache optimally in this manner are called *c*ache oblivious algorithms.

An early cache oblivious algorithm is presented in Aggarwal et al, [3] which proposes a model for hierarchical memory and then shows that a divide-

and-conquer algorithm for square matrix-multiplication is optimal under this model. Later, Gustavson [41] showed that such recursive divide-and-conquer algorithms could provide cache blocking for more complicated LAPACK routines such as Cholesky factorization as well. Finally, Frigo et al. [29] popularized the idea of cache-oblivious algorithms beyond the domain of linear algebra and named them. Frigo et al. also presents an algorithm that is shown to be optimal for a matrix multiplication of any shape. They show their cache-oblivious algorithms are optimal only for a single level of cache. However they also show that any cache-oblivious algorithm that is optimal for one level of cache is also a locally optimal cache-oblivious at every level of cache on systems that have multiple levels of cache. This local optimality should be compared to Gunnels et al. [40], which presents a family of cache-aware algorithms that are shown to be locally optimal at every level of cache.

Tactics for practical MMM There are some papers that are not as concerned with the theory of how to optimally move matrices through memory, but address more practical considerations that arise when implementing MMM. Henry [45] discusses how to implement high-performance BLAS on the IBM superscalar RISC S/6000. More importantly, it introduces what it calls a Block Data Structured GEMM. Today we know this concept as packing, where matrix partitions copied into special buffers in order to maximize spatial locality as well as arrange the data in a convenient format for accessing using SIMD instructions.

Kågstöm et al. [50] showed that the level-3 BLAS operations can be implemented mostly in terms of MMM, and in some cases a small amount of computation must be performed in terms of level-2 BLAS operations. This shows that it is only necessary to optimize general matrix-matrix multiplication, and then the rest of the level-3 BLAS comes “for free”.

Agarwal et al. [1] is not specific on how loops are structured for cache blocking, but rather discusses the low level details of how to exploit the various hardware resources available on the IBM Power2 to effectively use instruction level parallelism, yielding high performance.

This is just a small sample of the papers in this area that influenced our own work.

2.4 State-of-the-art matrix multiplication

Goto and van de Geijn [35] described what is currently accepted to be the most effective approach to implementing MMM in terms of rank-k updates. We call this Goto’s algorithm, and it was first implemented in the BLAS implementation GotoBLAS. GotoBLAS is currently maintained as a package called OpenBLAS. The primary innovation was in realizing that there is enough bandwidth from the L2 cache, so that one of the operands can be streamed from the L2 cache, if the other operand resides in the L1 cache, and optimized for the L2 cache by putting a square block in the L2 cache. Previously it was believed that one must optimize for the L1 cache. The effect was that blocksizes could be increased, leading to better amortization of memory movements. [1]

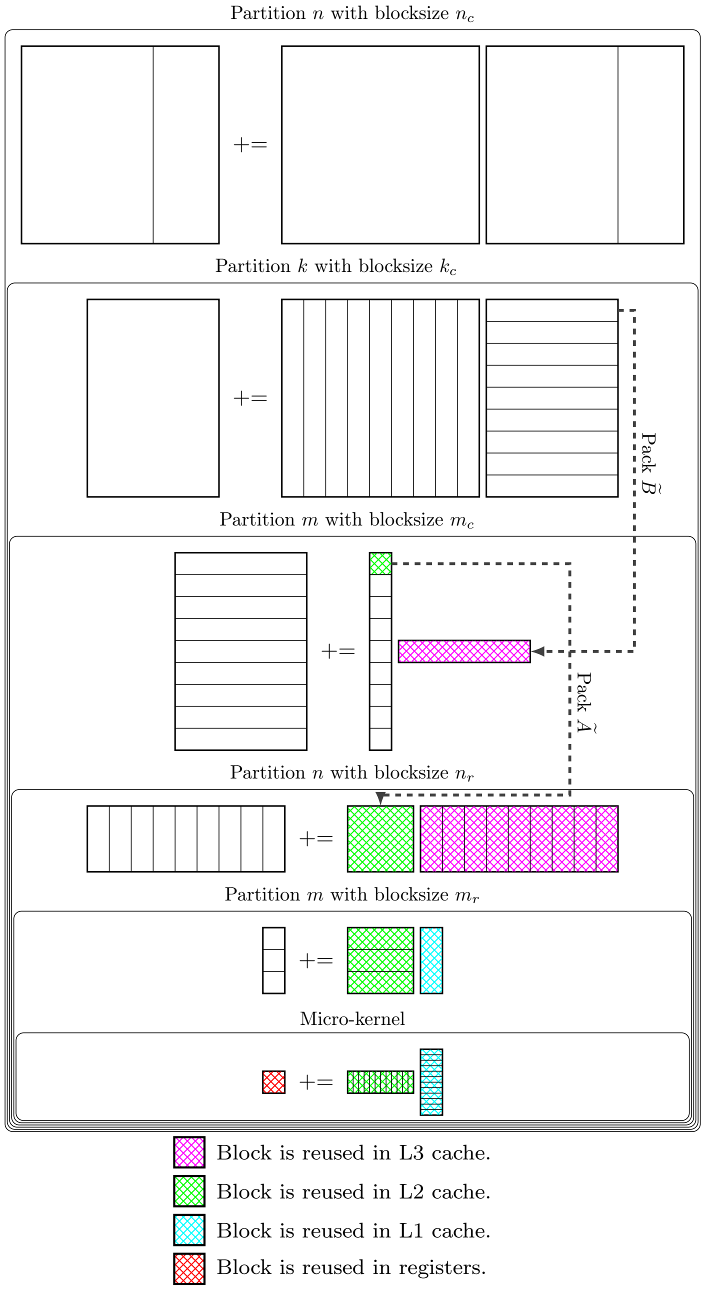

Partition n with blocksize n_c
+=
Partition k with blocksize k_c
+=
Pack $\widetilde{B}$
Partition m with blocksize m_c
+=
Pack $\widetilde{A}$
Partition n with blocksize n_r
+=
Partition m with blocksize m_r
+=
Micro-kernel
+=
Block is reused in L3 cache.
Block is reused in L2 cache.
Block is reused in L1 cache.
Block is reused in registers.

Figure 2.1: Diagram of Goto's Algorithm implemented in BLIS.

2.4.1 A brief description of Goto's algorithm

Let us focus on the simplified case $C := AB + C$, or $C \mathrel{+}= AB$, where A, B, and C are $m \times k$, $k \times n$, and $m \times n$ matrices, respectively. Goto's algorithm, illustrated in Figure 2.1, successively partitions the matrices so that different tiles of A, B, and C are moved through levels of cache in different ways. A chunk of memory (or matrix partition) may either *reside* in some layer of memory or it may be *streamed* through that layer of memory. If the computation is arranged such that some matrix partition is moved into a layer of memory and then reused while it is still in that layer of memory, we say that it resides in that layer. If instead the computation is arranged such that some matrix partition is moved into a layer of memory, used once, and then evicted from that layer of memory before it is used again, we say that it is streamed through that layer of memory.

The partitioning of the matrices and where they reside in memory is shown on the left, and the loops that implement the matrix partitions are shown on the right. Starting from the top, the outer-most loop, indexed by j_c, partitions C and B into (wide) column panels. Next, the loop indexed by p_c partitions A and the current column panel of B into column panels and row panels, respectively. Thus the current column panel of C (of size $m \times n_c$) is updated as a sequence of rank-k updates (with $k = k_c$). At this point, Goto's algorithm packs the current row panel of B into a contiguous buffer, $\widetilde{B}$. Assuming there is an L3 cache, $\widetilde{B}$ will reside in the L3 cache. The primary reason for the outer-most loop, indexed by j_c, is to limit the amount

---

[1] At the time, the amount of data addressable by the pages in the level-1 translation lookaside buffer (TLB) limited the size of the block in the L2 cache.

of workspace required for $\widetilde{B}$ A secondary reason is to allow $\widetilde{B}$ to remain in the L3 cache.

Next, the current panel of A is partitioned into blocks, indexed by i_c. The current block is then packed into a contiguous buffer, $\widetilde{A}$. The block is sized to occupy a substantial part of the L2 cache, leaving enough space to ensure that other data does not evict the block. Goto's algorithm then implements the "block-panel" multiplication of $\widetilde{A}\widetilde{B}$ as its inner kernel, making this the basic unit of computation.

Now let us describe the workings of this inner kernel. In GotoBLAS, this inner-kernel is a black-box implementation, often coded in assembly language. At this point, $\widetilde{A}$ resides in the L2 cache and $\widetilde{B}$ in the L3 cache. The next loop, indexed by j_r, partitions $\widetilde{B}$ into column micro-panels of width n_r. During one iteration of this loop, the current micro-panel of $\widetilde{B}$ resides in the L1 cache. Finally, the inner-most loop, indexed by i_r, partitions $\widetilde{A}$ into row micro-panels of height m_r.

At each iteration of the inner-most loop, a block dot product computation occurs: the current micro-panel of $\widetilde{A}$ is multiplied by the current micro-panel of $\widetilde{B}$ to update the corresponding $m_r \times n_r$ block of C. This is performed as a sequence of rank-1 updates (outer products) with columns from the micro-panel of $\widetilde{A}$ and rows from the micro-panel of $\widetilde{B}$. During the execution of this block dot product, an $m_r \times n_r$ block of C resides in registers, a $k_c \times n_r$ micro-panel of $\widetilde{B}$ resides in the L1 cache, and the $m_r \times k_c$ micro-panel of $\widetilde{A}$ is streamed from the L2 cache.

The key takeaway here is that different tiles of A, B, and C are placed in different layers of memory, and reused from different layers of memory in an attempt to amortize or hide the memory movements required to implement MMM. These memory movements can be summarized as follows:

- $\widetilde{B}$ is moved into the L3 cache from main memory. This memory movement is amortized over many block-panel matrix multiplications.
- $\widetilde{A}$ is moved into the L2 cache from main memory, amortized over many block-micropanel matrix multiplications.
- A micro-panel of $\widetilde{B}$ is moved into the L1 cache from the L3 cache, amortized over many block dot products.
- A micro-tile of C is moved into registers from main memory, amortized over many rank-1 updates.

It is through data reuse that we can amortize the cost of memory movements. With effective amortization, the cost of the MMM is then dependent only on the cost of the computation. The goal is to reuse data optimally to amortize the memory movements the best that we can.

2.4.2 BLIS

Van Zee and van de Geijn [80] introduced the BLAS-like Library Instantiation Software (BLIS). This is a systematic reimplementation of GotoBLAS, focusing on reducing the amount of effort required to port BLIS to a new architecture, as demonstrated in [79]. It shows that the GotoBLAS inner kernel (known in BLIS terminology as the *macro-kernel*) can be implemented as two loops around a much smaller *micro-kernel*. The effect is that only a single micro-kernel must be implemented for each data type on a given architecture, whereas GotoBLAS requires 12 inner kernels to be implemented for each data type on a given architecture in order to support all level-3 BLAS operations.

Chapter 3

I/O Lower Bounds

3.1 Introduction

The goal of this chapter is to find theoretical lower bounds on the I/O cost for MMM. An I/O lower bound gives us the minimum number of reads and writes that must occur during the execution of an MMM operation, and the greater the I/O lower bound, the better. When developing algorithms, we can evaluate them by analyzing their I/O cost and comparing this to the I/O lower bounds. When the costs are equal, we can say that the I/O lower bounds are tight and the algorithm is optimal.

The text for this chapter has been taken in part from [68]. We wish to acknowledge that the results in this chapter were independently discovered by Julien Langou and Bradley Lowery, however they have not been published at the time of writing this dissertation. A joint paper containing these results is in submission at the time of writing.

Deriving lower bounds starts with the assumption that a processor has two layers of memory hierarchy, a small *fast* memory and large *slow* memory. The fast and slow memory could represent the cache(s) and main memory of a processor, respectively, or main memory and disk. Practical implementations attempt to minimize the movement of data between these (and more) layers.

Hong and Kung [46] introduced what they called the red-blue pebble game model for a machine with two layers of memory. A limited number of blue pebbles represented fast memory while an unlimited number of red pebbles represented slow memory. By reasoning how at different times blue and red pebbles could be associated with subsets of data, a lower bound of $\Omega(mnk/\sqrt{S})$ for MMM was obtained, where S is the size of the fast memory (in matrix elements). In 1995, Savage [64] extended the red-blue pebble game to an arbitrary number of layers, and showed that the lower bound from [46] applies at every layer of the memory hierarchy.

In 2004, Irony et al. [49] use a very similar technique to extend the lower bounds to communication between nodes of a distributed memory parallel computer. Importantly, they provide a constant for the leading term of the I/O lower bound, $mnk/(2\sqrt{2}\sqrt{S})$. More recently, in 2014, Ballard et al. [8] generalized the techniques in [46] and [49] so that they can be applied to many operations in numerical linear algebra, not only MMM.

This chapter provides a better constant for the leading term of the lower bound for this problem. In Chapter 4, we will show that there are algorithms that attain this lower bound with the same constant, hence the lower bound is tight. While prior papers obtained lower bounds by reasoning about the multiplications that must be performed as part of an MMM operation, this chapter observes that in practice *fused multiply add* (FMA) operations are employed, and that in practice $C := AB + C$ (matrix-matrix multiplication and accumulation or MMMA) is more representative of how matrix-matrix operations are implemented in high-performance libraries. It shows that by targeting MMMA instead of MMM, a superior lower bound of $2mnk/\sqrt{S} - 2S$ can be obtained. The constant on the leading term is $4\sqrt{2}$ times greater than that of the previous lower bound. It shows how the new result for $C := AB + C$

can be translated into a lower bound of $2mnk/\sqrt{S} - 2S - \mathcal{O}(mn)$ for MMM ($C := AB$).

3.2 Problem Definition

In this section, we give a formal description of MMM for which we will derive I/O lower bounds. The computation that must be performed can be described as follows: Consider $C := AB$ and let $\gamma_{i,j}$, $\alpha_{i,j}$, and $\beta_{i,j}$ equal the (i,j) elements of the respective matrices.

Then $\gamma_{i,j} := \sum_{p=0}^{k-1} \alpha_{i,p}\beta_{p,j}$. This requires mnk scalar multiplications and $mn(k-1)$ scalar additions.

3.2.1 Prior approaches

Previous work, including both Hong and Kung [46], Irony et al. [49], obtained lower bounds for MMM and described computation in terms of a *directed acyclic graph* (DAG). In those papers, the DAGs have input vertices corresponding to the elements of the matrices A and B, output vertices corresponding to the elements of the result matrix C, and computation vertices corresponding to the the mnk elementary multiplications $\alpha_{i,p}\beta_{p,j}$. Each computation vertex has as inputs an element of A and an element of B, and as as an output a scalar that must be summed with others to form an element of C. These DAGs allow reasoning about dependencies but do not expose the costs associated with reading elements of C from slow memory.

3.2.2 Our approach

In order to achieve tight lower bounds, our problem definition must expose the costs of reading elements of C. To achieve this, we model computation in terms of FMAs. Unfortunately, the MMM operation can not be directly modeled in terms of FMA operations. This is because of the mismatch between the mnk elementary multiplications and the $mn(k-1)$ elementary additions. Because of this, we will instead find lower bounds for a different problem, that of the operation $C \mathrel{+}= AB$, or MMMA. A conventional MMMA has mnk scalar multiplications and mnk scalar additions. We assume that each scalar multiplication $\alpha_{i,p}\beta_{p,j}$ is paired with a corresponding scalar addition, and they are executed via an FMA instruction that has three inputs (a variable in which contributions to $\gamma_{i,j}$ are accumulated, and the elements $\alpha_{i,p}$ and $\beta_{p,j}$) and one output (the variable in which contributions to $\gamma_{i,j}$ are accumulated).

When describing a DAG for an MMMA that casts computation in terms of FMAs, each computation vertex would depend on another vertex that contributes to the same element of C. Thus the definition of such a DAG would impose a partial ordering on the computation. We wish to avoid such an ordering, which leads us to *not* describe computation in terms of a DAG.

Our problem definition is as follows. A conventional MMMA executes mnk FMAs. Each FMA has three inputs: an element of A, B, and C, and all inputs must be in fast memory in order for the FMA to be executed. Our proofs reason only about the input costs, and so our problem definition does not include anything about outputs from instructions. Each FMA must be of the form $\tau_{i,j} = \delta_{i,j} + \alpha_{i,p}\beta_{p,j}$ where $\tau_{i,j}$ and $\delta_{i,j}$ each are a partial accumulations of $\gamma_{i,j}$. For simplicity, the rest of this chapter uses either $\gamma_{i,j}$ or the phrase "an element of C" as a shorthand to refer to any partial result to be accumulated into that element of C, except where a distinction between $\tau_{i,j}$, $\delta_{i,j}$, and $\gamma_{i,j}$ needs to be made.

3.3 Lower Bound Proof

In this section we prove that an MMMA must have an I/O cost of at least $2mnk/\sqrt{S} - 2S$.

3.3.1 High-level strategy

In order to obtain I/O lower bounds, we will think of computation as being divided into phases, where there are exactly M loads and stores during each phase (except for the last phase). That is to say, each phase has an I/O cost of at most M. If one can prove that there must be at least N phases for any algorithm, then it follows that the algorithm must have a total I/O cost of at least $(N-1)M$. The $N-1$ comes from the fact that the last phase may have less than M loads and stores.

How many phases must there be? Since it is a conventional MMMA, we know that mnk FMAs must be executed. Let F be an upper bound on the number of FMAs that can occur during a single phase. Then there must be at least $(mnk)/F$ phases. This gives an overall I/O lower bound of $((mnk)/F - 1)M$.

How do we find F? We know that the size of fast memory is S, and there are at most M loads during a single phase. This means that there are $S+M$ elements of A, B, and C that can be used as inputs to FMAs during a phase. Thus placing an upper bound on the number of FMAs that can be computed using $S+M$ elements gives an upper bound on F.

This is nearly the same strategy that was used by [46] and [49]. The important difference is that in the previous papers, the number of loads and stores to and from fast memory per phase is always equal to the size of fast memory, S. We will show that one can achieve greater lower bounds by allowing M to be a value other than S.

3.3.2 Employing the Loomis-Whitney inequality

The Loomis-Whitney inequality [55] was used in [49] to determine how many elementary operations involved in an MMM can be executed with some number of elements.

Theorem 3.3.1 (Loomis-Whitney)**.** *Let m be the measure of an open subset $\mathcal{O}$ of Euclidean n-space, and let $m_1, ..., m_n$ be the $(n-1)$-dimensional measures of the projections of $\mathcal{O}$ on the coordinate hyper planes. Then $m^{n-1} \leq m_1 m_2 \cdots m_n$.*

To apply this theorem to our situation, let $\mathcal{O}$ represent a three dimensional set of some FMAs that occur during an MMMA. Each FMA has a coordinate (i, j, p) in the m, n, and k dimensions: $\gamma_{i,j} + := \alpha_{i,p}\beta_{p,j}$. The projection of $\mathcal{O}$ in each of the m, n, and k dimensions respectively corresponds to the elements of B, A, and C that are inputs to the FMAs in $\mathcal{O}$. If F is the number of FMAs that occur during an MMMA using x elements of A, y elements of B, and z elements of C then the Loomis-Whitney inequality tells us that $F^2 \leq xyz$ and hence $F \leq \sqrt{xyz}$.

There are at most $S + M$ elements that can be used as inputs to computation during a phase, because there are at most S elements of A, B, and C in fast memory at the start of the phase and at most M elements read from slow memory during the phase. Similarly there can be at most $S + M$ elements that are outputs of computation during a phase, because there are at most S elements in fast memory at the end of the phase and at most M elements written to slow memory during the phase.

The authors of [49] reason separately about x, y, and z, showing that since there can be at most $S + M$ inputs to computation during phase, $x \leq S + M$ and $y \leq S + M$. Similarly, since there can be at most $S + M$ elements written during a phase, $z \leq S + M$. Working with FMAs, we do better because we know that the x elements of A, the y elements of B, and the z elements of C must all be inputs to the same phase. Therefore, we can reason about x, y, and z all together: $x + y + z \leq S + M$.

The above can be formulated as a constrained maximization problem,

$$\text{maximize } F \text{ under the constraints } \left\{ \begin{array}{c} F \leq \sqrt{xyz} \\ 0 \leq x, y, z \\ x + y + z \leq S + M. \end{array} \right.$$

Application of standard Langrange multiplier methods, detailed in Appendix B, tells us that the global maximum occurs when

$$x = y = z = \frac{S + M}{3} \quad \text{so that} \quad F = \frac{(S + M)\sqrt{S + M}}{3\sqrt{3}}.$$

3.3.3 A lower bound for $C := AB + C$

The upper bound on F gives us the following lower bound for the I/O cost of MMMA:

$$\left(\frac{mnk}{F} - 1 \right) M = \left(\frac{3\sqrt{3}}{(S + M)\sqrt{S + M}} mnk - 1 \right) M.$$

In this lower bound, M is a free variable meaning that different choices for M yield different lower bounds. In [46], [49], and [8], M is always equal to S. If we also make this choice, we obtain the lower bound:

$$\frac{3\sqrt{3}}{2\sqrt{2}}\frac{mnk}{\sqrt{S}} - S.$$

This lower bound can be found in [24].

It is possible to do better still. In order to find the greatest lower bound for any M, our goal is to find the M that maximizes:

$$\max_{M>0}\left(\frac{3\sqrt{3}Mmnk}{(S+M)\sqrt{S+M}} - 1\right) \approx \max_{M>0}\left(\frac{3\sqrt{3}Mmnk}{(S+M)\sqrt{S+M}}\right)$$

when m, n, and k are large. Standard maximization techniques from calculus yield that the global maximum is attained when $M = 2S$ so that the I/O lower bound (when m, n, and k are large so that the $-M$ term can be ignored) becomes:

$$\left(\frac{3\sqrt{3}mnk}{(S+M)\sqrt{S+M}} - 1\right)M = \left(\frac{3\sqrt{3}mnk}{3S\sqrt{3S}} - 1\right)(2S) =$$

$$\frac{2mnk}{\sqrt{S}} - 2S.$$

3.3.4 Improving the lower bound for $C := AB$

Above, we found a lower bound for a different problem than was considered in previous work [46, 49], which studied $C := AB$. We will now use the lower bound for $C := AB + C$ to obtain a new lower bound for the operation $C := AB$.

Consider the MMA $C := AB + C$. It can be implemented by the sequence of operations:

$$D := AB; \quad C := D + C.$$

Let Q_{AB} be the I/O cost for the first operation and Q_{D+C} be the I/O cost for the second. Then since these operations implement an MMMA find that $Q_{AB} + Q_{D+C} \geq 2mnk/\sqrt{S} - 2S$. It is easy to show that there exists an algorithm for $D + C$ such that its I/O cost is $\mathcal{O}(mn)$. Then:

$$Q_{AB} + \mathcal{O}(mn) \geq \frac{2mnk}{\sqrt{S}} - 2S \quad \text{or, equivalently,} \quad Q_{AB} \geq \frac{2mnk}{\sqrt{S}} - 2S - \mathcal{O}(mn).$$

Thus, our new lower bound for $C := AB + C$ yields a new lower bound on the I/O cost of $C := AB$.

3.4 Summary

In this chapter, proved that the operation $C := AB + C$ where computation must be performed in terms of FMAs, and A, B, and C are disticnt must have an I/O cost of at least $2mnk/\sqrt{S} - 2S$. Then, $C := AB$ must have an I/O cost of at least $2mnk/\sqrt{S} - 2S - \mathcal{O}(mn)$. These lower bounds are of interest by themselves as a theoretical result. In the next chapter, we use them to help gain fundamental insight into how MMM must be implemented.

We believe that the proof techniques presented in this paper can apply to algorithms outside of matrix multiplication. Generalizing the size of the phase is one technique that can apply to lower bound proofs that follow the same general strategy for any operation. In the domain of linear algebra, we believe this techniqe can be used in order to find the best possible constant coefficient for lower bounds as long as the matrix operands are distinct. For matrix operations where the operands are not necessarily distinct, we believe that these techniques can be combined with those from [8] to improve the lower bounds.

While the proof in this chapter only applies to algorithms that do not use FMA operations, we hypothesize that the lower bound applies in this case. In order to prove this to be true, we believe that a different high level proof strategy must be used, because the strategy does not provide a mechanism to reason about long range dependencies, like if a scalar multiplication occurs during a different phase from its corresponding scalar addition.

As a testament to its relevance, we note that techniques from this work have alrady been used by others [9].

Chapter 4

Optimal Algorithms
or: The Lower Bound is Tight

4.1 Introduction

In Chapter 3, we obtained I/O lower bounds for MMM and MMMA. In this chapter, we use those lower bounds to derive properties of algorithms that could obtain those lower bounds. From those properties, we derive an algorithm that has an I/O cost where the highest ordered term is the same, including its coefficient. We derive two other algorithms that attain the lower bound when ignoring the cost of writing to main memory. The three algorithms are in some sense symmetric to each other and so we say that they form a family of algorithms. We then show that for different "shapes" of MMM, that is when different combinations of m, n, and k are large or small, which of these three algorithms will be favorable.

Portions of the text in this chapter are taken from [68].

4.2 Optimal and read-optimal Algorithms

In Chapter 3, we obtained a tight constant on the I/O cost for matrix-matrix multiplication for machines with one level of cache. Assuming that computation is performed in terms of fused multiply-adds (FMAs), matrix-matrix multiplication must have an I/O cost of at least $\frac{2mnk}{\sqrt{S}}$, where S is the capacity of the cache. The paper then presents three algorithms for matrix-matrix multiplication. One of these attains the I/O lower bound, and the other two are optimal only in terms of the number of inputs into cache, however they can be considered to be optimal if a read plus a write costs the same as a read.

In this section, we describe and analyze these three algorithms. We discuss how to choose blocksizes for the algorithms. We then show that each of these algorithms is favorable for different shapes of matrix-matrix multiplication.

4.2.1 Algorithms for one level of cache

Matrix-matrix multiplication that casts computations in terms of FMAs requires an I/O cost of least $\frac{2mnk}{\sqrt{S}}$. The goal for the algorithms in this section is to incur an I/O cost of only $\frac{2mnk}{\sqrt{S}}$ plus a quadratic term. To attain this goal, it is sufficient that: (1) each element of one of the operands is read from slow memory once and (2) each element of the other two operands is involved in $\approx \sqrt{S}$ FMAs each time it is read from slow memory. We now present three algorithms for MMM that attain this goal.

Resident C. We now describe an algorithm that keeps a block of C resident in fast memory during computation. See the top algorithm illustrated in Figure 4.1. Consider $C := AB + C$. Partition:

$$C \rightarrow \left(\begin{array}{c|c|c} C_{0,0} & \cdots & C_{0,n-1} \\ \hline \vdots & & \vdots \\ \hline C_{m-1,0} & \cdots & C_{m-1,n-1} \end{array} \right), \quad A \rightarrow \left(\begin{array}{c} A_0 \\ \hline \vdots \\ \hline A_{m-1} \end{array} \right), \quad B \rightarrow \left(\begin{array}{c|c|c} B_0 & \cdots & B_{n-1} \end{array} \right),$$

where $C_{i,j}$ is $m_c \times n_c$, A_i is $m_c \times k$, and B_j is $k \times n_c$. Then compute $C_{i,j} \mathrel{+}= A_i B_j$ such that $C_{i,j}$ is read from slow memory once, and then updated with a series

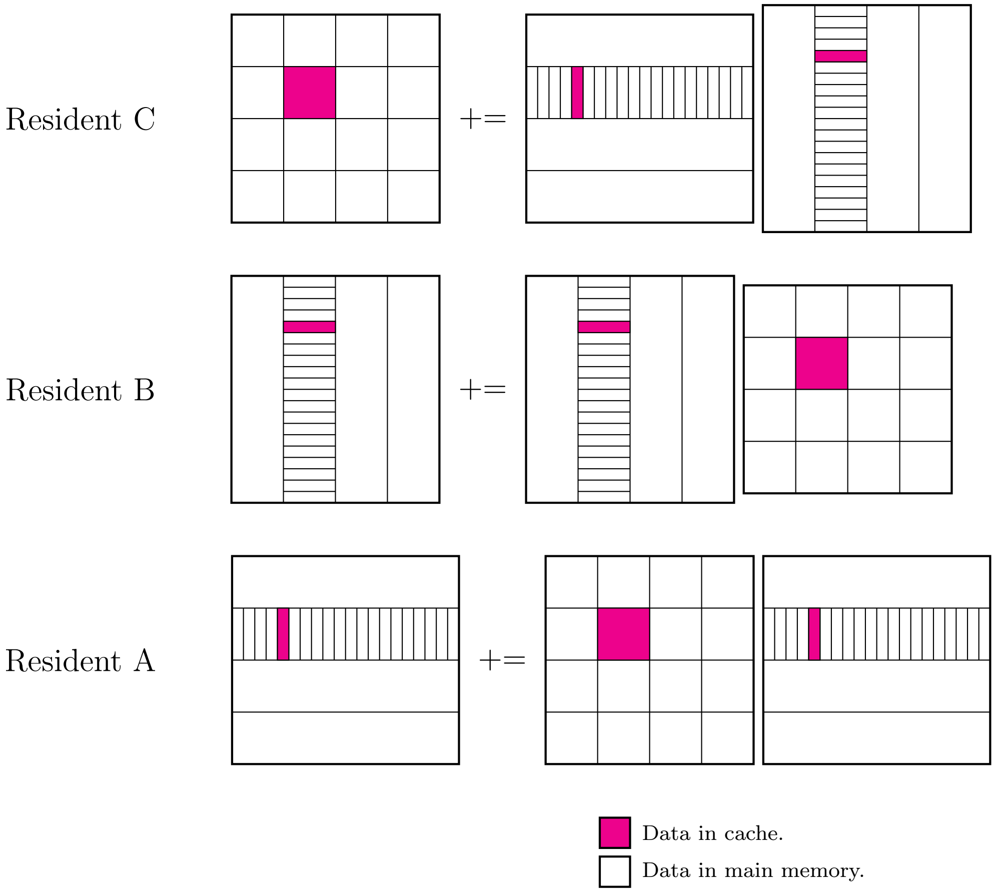


Figure 4.1: Three algorithms for matrix multiplication that attain the lower bound for a single level of cache.

of rank-1 updates. During $C_{i,j}$ += A_iB_j, each element of $C_{i,j}$, A_i, and B_j is read from slow memory one time, and each element of $C_{i,j}$ is written to slow memory once. This gives the following I/O costs for each operand:

- $C_{i,j}$: m_cn_c reads and m_cn_c writes.
- A_i: m_ck reads.
- B_j: kn_c reads.

With $\lceil \frac{mn}{m_cn_c} \rceil$ such $C_{i,j}$ += A_iB_j suboperations performed during the overall MMM operation C += AB, the total input and output costs associated with each matrix are:

- C: mn reads and mn writes.
- A: $\lceil \frac{mnk}{n_c} \rceil$ reads.
- B: $\lceil \frac{mnk}{m_c} \rceil$ reads.

When $m_c \approx n_c \approx \sqrt{S}$ [1], the total input cost is $\frac{2mnk}{\sqrt{S}} + mn$, and the total output cost is mn. Choosing $m_c = n_c$ equalizes the input costs associated with A and B, and minimizes $\lceil \frac{mnk}{n_c} \rceil + \lceil \frac{mnk}{m_c} \rceil$. The highest ordered term in the cost of the Resident C algorithm is the same as the I/O lower bound for MMM. Thus the algorithm is optimal and the lower bound is tight.

---

[1] Note that m_c and n_c must be slightly less than $\sqrt{S}$ so that there is room for a row of A_i and a column of B_j in cache.

Resident B. Another possibility is an algorithm that keeps a block of B resident in fast memory during computation. See the middle algorithm illustrated in Figure 4.1.

Partition:

$$C \rightarrow \left(\begin{array}{c|c|c} C_0 & \cdots & C_{n-1} \end{array} \right), \quad A \rightarrow \left(\begin{array}{c|c|c} A_0 & \cdots & A_{n-1} \end{array} \right),$$

$$B \rightarrow \left(\begin{array}{c|c|c} B_{0,0} & \cdots & B_{0,n-1} \\ \hline \vdots & & \vdots \\ \hline B_{m-1,0} & \cdots & B_{m-1,n-1} \end{array} \right),$$

where C_j is $m \times n_c$, A_p is $m \times k_c$, and $B_{p,j}$ is $k_c \times n_c$. $C_j \mathrel{+}= A_p B_{p,j}$ is implemented as a loop over vector-matrix multiplications.

During $C_j \mathrel{+}= A_p B_{p,j}$, each element of C_j, A_p, and $B_{p,j}$ is read from slow memory one time, and each element of C_j is written to slow memory once. This gives the following I/O costs for each operand:

- C_j: mn_c reads and mn_c writes.
- A_i: mk_c reads.
- $B_{p,j}$: $k_c n_c$ reads.

With $\lceil \frac{kn}{k_c n_c} \rceil$ such $C_j \mathrel{+}= A_p B_{p,j}$ suboperations performed during the overall MMM operation $C \mathrel{+}= AB$, the total input and output costs associated with each matrix are:

- C: $\lceil \frac{mnk}{k_c} \rceil$ reads and $\lceil \frac{mnk}{k_c} \rceil$ writes.
- A: $\lceil \frac{mnk}{n_c} \rceil$ reads.
- B: nk reads.

If $k_c \approx n_c \approx \sqrt{S}$, the input cost is approximately $\frac{2mnk}{\sqrt{S}} + nk$, and the output cost is approximately $\frac{mnk}{\sqrt{S}}$. The input cost now attains near the I/O lower bound.

Resident A. A third possibility is an algorithm that keeps a block of A resident in fast memory during computation, as illustrated in Figure 4.1.

Partition:

$$C \rightarrow \left(\begin{array}{c} C_0 \\ \hline \vdots \\ \hline C_{m-1} \end{array}\right), \quad A \rightarrow \left(\begin{array}{c|c|c} A_{0,0} & \cdots & A_{0,k-1} \\ \hline \vdots & & \vdots \\ \hline A_{m-1,0} & \cdots & A_{m-1,k-1} \end{array}\right), \quad B \rightarrow \left(\begin{array}{c} B_0 \\ \hline \vdots \\ \hline B_{k-1} \end{array}\right),$$

where C_i is $m_c \times n$, $A_{i,p}$ is $m_c \times k_c$, and B_p is $k_c \times n$. $C_i \mathrel{+}= A_{i,p}B_p$ is implemented as a loop over matrix-vector multiplications.

During $C_i \mathrel{+}= A_{i,p}B_p$, each element of C_i, $A_{i,p}$, and B_p is read from slow memory one time, and each element of C_i is written to slow memory once. This gives the following I/O costs for each operand:

- C_i: $m_c n$ reads and $m_c n$ writes.
- $A_{i,p}$: $m_c k_c$ reads.
- B_p: $k_c n$ reads.

With $\lceil \frac{mk}{m_c k_c} \rceil$ such $C_i \mathrel{+}= A_{i,p}B_p$ suboperations performed during the overall MMM operation $C \mathrel{+}= AB$, the total input and output costs associated with each matrix are:

- C: $\lceil \frac{mnk}{k_c} \rceil$ reads and $\lceil \frac{mnk}{k_c} \rceil$ writes.
- A: mk reads.

- B: $\lceil \frac{mnk}{m_c} \rceil$ reads.

If $m_c \approx k_c \approx \sqrt{S}$, the input cost is approximately $\frac{2mnk}{\sqrt{S}} + mk$, and the output cost is approximately $\frac{mnk}{\sqrt{S}}$. The input cost attains near the I/O lower bound.

Discussion. These three algorithms and the shapes of the subproblems exposed by them have been described before, however their optimality has not been able to be analyzed in the context of the $\frac{2mnk}{\sqrt{S}}$ lower bound before now. The Resident A algorithm was described as early as 1991 [52], and each of Resident A, Resident B, and Resident C appears in [40].

4.2.2 Algorithms for different shapes of MMM

The number of reads and writes from slow memory during the algorithms Resident A, Resident B, and Resident C depend on the shape of the input matrices. There are cases where one of the algorithms is more *efficient* than the other two, where we define efficiency by flops per I/O cost. The higher this value is, the more efficient the algorithm is. There are $2mnk$ flops performed during MMM, and the I/O lower bound is $\frac{2mnk}{\sqrt{S}}$. Thus our goal for efficiency is $\sqrt{S}$ flops per I/O. We will now examine the cases for which algorithms are efficient, assuming that m, n, and k are at least $\sqrt{S}$.

Resident C is efficient if and only if k is large. The Resident C algorithm reads $\lceil \frac{mnk}{n_c} \rceil + \lceil \frac{mnk}{m_c} \rceil + mn$ elements from slow memory during MMM. If $m_c = n_c = \sqrt{S}$, this is approximately $\frac{2mnk}{\sqrt{S}} + mn$. This gives an efficiency of $\left(\frac{1}{\sqrt{S}} + \frac{mn}{2k} \right)^{-1}$. When k is large, this is approximately $\sqrt{S}$.

We can analyze Resident A and Resident B similarly. Here we ignore the I/O cost for writes. If blocksizes are chosen to be equal to $\sqrt{S}$, Resident B has an efficiency of $\left(\frac{1}{\sqrt{S}} + \frac{nk}{2m}\right)^{-1}$, which is approximately $\sqrt{S}$ when m is large. Resident A has an efficiency of $\left(\frac{1}{\sqrt{S}} + \frac{mk}{2n}\right)^{-1}$, which is approximately $\sqrt{S}$ when n is large.

This shows that one must choose the right algorithm depending on the shape of the problem. For each of Resident A, Resident B, and Resident C, there is a minimal shape that can be implemented efficiently. For Resident C it is when m and n are $\approx \sqrt{S}$, and k is large. For Resident B it is when k and n are $\approx \sqrt{S}$, and m is large. For Resident A it is when m and k are $\approx \sqrt{S}$, and n is large. In each of these cases, the resident matrix must fit into fast memory, and the "other dimension" must be large so that the cost of moving the resident matrix into fast memory can be amortized. In fact, these minimal shapes are the exact shapes that are exposed by each of the algorithms after they have been partitioned by their outer two loops. Because these problems are the smallest, most basic problems that can be implemented efficiently, and because they are the same problem shapes that are exposed by the algorithms, we call these problem shapes *optimal subproblems*.

The fact that one must choose a different algorithm for MMM depending on problem shape and size has been noted before for distributed memory MMM [65, 54], and for hierarchical memory MMM [40, 82], but our analysis of the algorithms in terms of the tight I/O lower bounds is novel. In practice, shapes of MMM where one or two dimensions are small often arise as subproblems during matrix factorizations [39, 5]. Outside of DLA, other applications may use highly skewed matrix-matrix multiplication. In quantum chemistry, coupled cluster applications use tensors of one to eight dimensions, and from hundreds of bytes to hundreds of gigabytes in size[10]. Tensor contractions on these tensors, which are equivalent to MMM under permutations, range widely in size and shape as well, with contractions varying from perfectly "square" (m=n=k) to highly skewed, with ratios between m, n, and k as high as 10000 in higher-order coupled cluster methods[61].

4.2.3 A balancing act

In the previous parts of this section, we have assumed that I/O costs associated with all three matrices are equal. We analyzed algorithms assuming that a read and a write costs the same as a read, and we only looked at the read costs of all three algorithms.

In doing so, we reached the conclusion that we should take the following strategy for the algorithms Resident A, Resident B, and Resident C: Place a square block of the resident matrix in fast memory, and stream the other two operands through fast memory. This amortizes the I/O costs associated with the resident matrix, and equalizes the I/O costs associated with the streamed matrices.

What if the operands are not symmetric in terms of I/O costs? Suppose accessing elements of one of the matrices is more expensive than accessing elements of another. For instance, accessing elements of C may be inherently more expensive than accessing elements of A or B, since elements of C must be read and written. In this case, it may make sense to use the algorithm that has the expensive matrix reside in fast memory.

However this is not always possible depending on problem shape. Suppose we are using either the Resident A, B, or C algorithm. Let's call the streamed matrices W and V, and suppose the algorithm has blocksizes q and r such that during each optimal subproblem of the algorithm, each element of W is reused q times and each element of V is reused r times. If the cost of accessing elements W and V are equal, then we should choose $q = r \approx \sqrt{S}$. If instead the cost of accessing an element of W costs α, and accessing an element of V costs β, then when m, n, and k are large, the efficiency of our chosen algorithm is $\frac{\alpha}{2q} + \frac{\beta}{2r}$. This is minimized when $q = \sqrt{\frac{\alpha S}{\beta}}$ and $r = \sqrt{\frac{\beta S}{\alpha}}$.

For example, when choosing blocksizes for Resident A, (and n is large), if a write and a read costs the same as two reads, then the efficiency of Resident A is $\frac{2}{2k_c} + \frac{1}{2m_c}$. In this case, instead of equalizing the blocksizes so that $m_c = k_c$ and a square block resides in fast memory, we must balance the cost of reading and writing C with the cost of reading A, so k_c should be chosen to be $\sqrt{2S}$, and m_c should be chosen to be $\sqrt{\frac{S}{2}}$.

In an ideal situation, input costs associated with all matrices are equal, and we can pick square blocksizes. This is not always the case in practice. To summarize this section, the ingredients to an efficient algorithm are: (1) Fill fast memory with a submatrix of one of the operands (the resident matrix), (2) Amortize the I/O cost associated with (1) over a lot of computation, (3) Choose blocksizes that balance the I/O costs associated with the other two matrices with each other (the streamed matrices).

4.3 Summary

In this chapter, we described three algorithms for MMM. The analysis of these algorithms shows that the I/O lower bounds from Chapter 3 have the best possible coefficient, and it shows that one of these algorithms is optimal, and the other two are read-optimal. In all three algorithms, a square block of one of the three operands is moved into fast memory, occupying most of it, and panels of the other two matrices are streamed through fast memory. We show that when k is large, the Resident C algorithm is efficient, when m is large, the Resident B algorithm is efficient, and when n is large, the Resident A algorithm is efficient.

Chapter 5

A Family of Algorithms for Multiple Levels of Cache

5.1 Introduction

Typically, modern computer architectures have several levels of cache. In this chapter, we show how to adapt the results from Chapter 4 to develop practical algorithms that are efficient on such architectures. The three algorithms (Resident A, Resident B, and Resident C) are used as building blocks to develop a new family of algorithms for architectures for multiple levels of cache. By composing two loops for each level of the memory hierarchy, a modified version one of those three algorithms can be encountered at each level of cache. This lets us optimize the I/O cost for each level of cache, letting us effectively use a multilevel cache hierarchy.

Unfortunately, tradeoffs occur when trying to simultaneously optimize the I/O cost at different levels of the cache hierarchy within this family of algorithms. We analyze these tradeoffs and provide insight into how to resolve them. Finally we show that members of this family of algorithms outperform the state-of-the-art Goto's Algorithm [35] in low bandwidth situations when data movement to and from main memory becomes a more significant cost.

5.2 A Family of Algorithms

Given the ideas about how to attain a near-optimal I/O cost for MMM for a single layer of fast memory presented in Section 4.2, we will use those ideas to develop a methodology for implementing MMM for machines with multiple layers of fast memory. The basic idea is as follows: We will assume there is an optimal subproblem targeting some layer of cache, the L_h cache. That is, a subproblem where two dimensions are small and one is large, and the matrix that has two small dimensions resides in the L_h cache. Then, we will implement this L_h optimal subproblem such that for the next smaller and faster level of cache, the L_{h-1} cache, there is an optimal algorithm targeting it. In doing this, we attempt to simultaneously optimize the I/O cost for both the L_h and L_{h-1} caches.

We use tiled loops in order to implement the L_h subproblem, where each loop partitions the matrices along one of the dimensions m, n, or k with some blocksize. We will show that two loops can be used to implement the L_h subproblem such that one encounters an optimal subproblem targeting the L_{h-1} cache. We will show that the directions of the loops depend on the shape of the subproblem encountered at the L_h cache. With this methodology, if one shape of optimal subproblem is encountered at the L_h cache, then one of the other two shapes will be encountered at the L_{h-1} cache. The loops partitioning the L_h subproblem into the L_{h-1} subproblem are illustrated in Figures 5.1 and 5.2.

We note that [40] claimed that it was locally optimal to encounter a subproblem that corresponds to one of the three optimal subproblems at every level of the memory hierarchy. However that paper did not give details on how this could be accomplished, nor did it analyze the claim in terms of any I/O lower bounds.

5.2.1 The outer loop for the L_{h-1} cache

In this section, we will analyze the outermost of the two loops for the L_{h-1} cache. The scenarios for this L_{h-1} outer loop are illustrated in Figures 5.1 and 5.2, under the leftmost two columns, labeled L_h optimal subproblem and L_{h-1} outer loop. We will show that the outermost of the two tiled loops must be along the "long" dimension of the L_h subproblem. The tiled loops within the L_h optimal subproblem should be designed in such a way that ensures that the L_h I/O cost is as close to the I/O lower bound as possible. This means that:

1. The L_h resident matrix must remain in the L_h cache during the entire subproblem.
2. Each element of each of the non-resident matrices must be used in $\approx \sqrt{S_h}$ FMAs each time it is loaded into cache.
3. To attain close to the I/O lower bound, the matrix resident in the L_h cache should occupy as much of the cache as possible, so only a small portion of the non-resident matrices should be in cache at a time.

We will now present two arguments that show that the outer loop must partition the matrices along the long dimension of the optimal subproblem.

The first argument is as follows, and holds when the L_h cache has a least recently used (LRU) replacement policy. Consider a loop partitioning the matrices along one of the small dimensions of the L_h subproblem. Such a loop would partition the L_h resident matrix, so not every element of the L_h

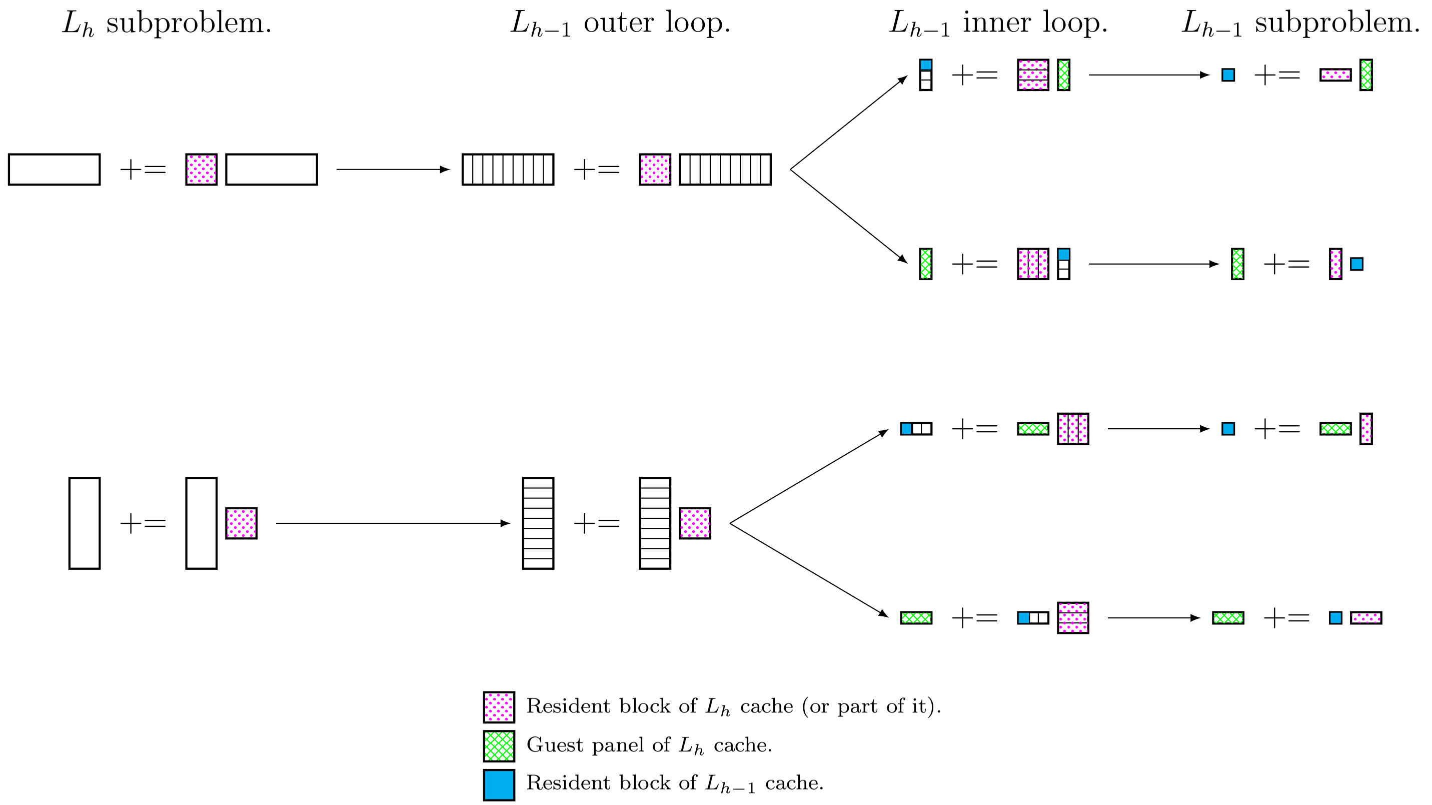

L_h subproblem.
L_{h-1} outer loop.
L_{h-1} inner loop.
L_{h-1} subproblem.
+=
Resident block of L_h cache (or part of it).
Guest panel of L_h cache.
Resident block of L_{h-1} cache.

Figure 5.1: Illustration showing the possible scenarios when partitioning for the L_h and L_{h-1} caches in the family of MMM algorithms when Resident A or Resident B is encountered at the L_h cache.

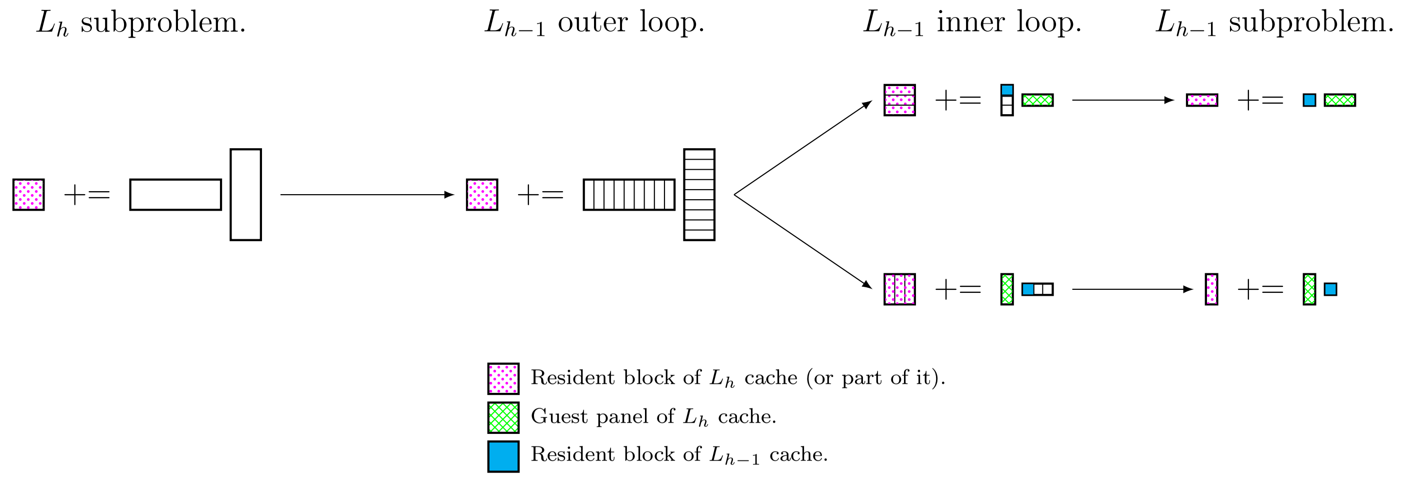

L_h subproblem.
L_{h-1} outer loop.
L_{h-1} inner loop.
L_{h-1} subproblem.
+=
+=
+=
+=
+=
+=
Resident block of L_h cache (or part of it).
Guest panel of L_h cache.
Resident block of L_{h-1} cache.

Figure 5.2: Illustration showing the possible scenarios when partitioning for the L_h and L_{h-1} caches in the family of MMM algorithms when Resident C is encountered at the L_h cache.

resident matrix would be accessed during every iteration. Each iteration of this loop accesses every element of one of the non-resident matrices. Because the non-resident matrices are too big to fit into cache, accessing that many elements would cause the partitions of the resident matrix not accessed by the current iteration to be evicted from cache. Therefore, in an LRU cache, first partitioning along one of the short dimensions of the resident subproblem causes portions of the resident matrix to be evicted from cache, and so Condition (1) for implementing the optimal subproblem is not met.

The second argument even holds when the L_h cache is ideal and one can explicitly control what data is in it. Suppose that the loop partitions along one of the small dimensions of the L_h subproblem with blocksize s_{h-1}. Then, one of the non-resident matrices will not be partitioned by this loop. During each iteration of the loop, every element of the non-resident matrix that is not partitioned is used in s_{h-1} FMAs. In order to satisfy condition (2) above, each element of the non-resident matrices must be used $\approx \sqrt{S_h}$ times each time they are read into the L_h cache. Since the L_{h-1} outer loop blocks for the L_{h-1} cache, s_{h-1} will be significantly smaller than $\sqrt{S_h}$. In order for this to occur, the entire non-resident matrix would need to be brought into the L_h cache, and stay in cache during the entire loop. This is not possible since the non-resident matrix is too big.

We have now established that the outer of the two loops targeting the L_{h-1} cache must be along the long dimension. If we name the blocksize for the L_{h-1} outer loop s_{h-1}, then we can summarize the cases as follows.

- If the L_h subproblem is Resident A, then partition the n dimension with blocksize s_{h-1}.

- If the L_h subproblem is Resident B, then partition the m dimension with blocksize s_{h-1}.
- If the L_h subproblem is Resident C, then partition the k dimension with blocksize s_{h-1}.

5.2.2 The inner loop for the L_{h-1} cache

The next step is to further partition the matrices to target the L_{h-1} cache. The possibilities for this L_{h-1} inner loop are illustrated in Figures 5.1 and 5.2, in the columns labeled L_{h-1} inner loop and L_{h-1} subproblem. Each iteration of the L_{h-1} outer loop is a subproblem where two dimensions are approximately $\sqrt{S_h}$ and the other is s_{h-1}. The L_{h-1} inner loop will partition this subproblem one of the two dimensions that the L_{h-1} outer loop did not.

If the L_h subproblem is:

- Resident A, then the subproblem exposed by each iteration of the L_{h-1} outer loop is a block of A times a skinny panel of B updating a skinny panel of C. Then the L_{h-1} inner loop should partition either the m or the k dimension with blocksize t_{h-1}.
- Resident B, then the subproblem exposed by each iteration of the L_{h-1} outer loop is a short panel of A times a block of B updating a short panel of C. Then the L_{h-1} inner loop should partition either the n or the k dimension with blocksize t_{h-1}.
- Resident C, then the subproblem exposed by each iteration of the L_{h-1} outer loop is a skinny panel of A times a short panel of B updating a block of C (also called a rank-k update). Then the L_{h-1} inner loop should partition either the m or the n dimension with blocksize t_{h-1}.

This L_{h-1} inner loop exposes a new subproblem that we will call the L_{h-1} subproblem. The pair of loops for the L_{h-1} cache have partitioned the matrices such that one of the matrices is $s_{h-1} \times t_{h-1}$ or $t_{h-1} \times s_{h-1}$, and the other dimension is longer, roughly $\sqrt{S_h}$. Thus at the L_{h-1} cache, we encounter a problem that is similar to one of the three optimal subproblems.

5.2.3 Classifying matrix partitions.

The two loops for the L_{h-1} have exposed partitions of each matrices that differ in terms of access frequency and size. From these properties, we can classify these different matrix partitions.

L_h resident matrix. We have already classified the L_h resident matrix. The L_{h-1} outer loop partitions along the large dimension of the L_h subproblem. Each element of the L_h resident matrix is accessed in every iteration of the loop, and a comparatively smaller amount of the other two matrices are accessed. This is why we classify it as the resident matrix. The elements of the other two operands are not all accessed every iteration of the L_{h-1} outer loop, so we can classify them as the L_h *streamed matrices*.

There are two choices for the dimension that the L_{h-1} inner loop iterates in. Depending on the direction of this loop, the elements of different matrix partitions are reused at different rates. The L_{h-1} inner loop always partitions the L_h resident matrix and one of the L_h streamed matrices. This allows us to classify the two L_h streamed matrices differently, taking into account the properties from Section 5.2.1 that must hold so that the implementation of the L_h subproblem is still close to optimal.

L_h guest matrix. The operand that is not partitioned by the L_{h-1} inner loop is used during every iteration of the L_{h-1} inner loop. In order to meet the conditions for implementing the L_h optimal subproblem, the operand not partitioned by the L_{h-1} operand can only be inputted to the L_h cache one time. Therefore it must remain in cache during the entire inner L_{h-1} loop. We name this operand the *guest matrix* of the L_h cache to contrast it with the resident matrix of the L_h cache. The elements of the L_h guest matrix, like the elements of the L_h resident matrix, are reused from the L_h cache across iterations of a loop. The difference is that the L_h resident matrix is reused across every iteration of the outer L_{h-1} loop, and the L_h guest matrix is reused across the iterations of the inner L_{h-1} loop. The resident matrix therefore remains in the L_h cache for a long period of time, whereas the guest matrix remains in the L_h cache for a comparatively short period.

L_{h-1} resident matrix. We have discussed the L_h resident matrix and the L_h guest matrix, and will now discuss the remaining operand. The purpose of the outer and inner L_{h-1} loops is to effectively utilize the L_{h-1} cache. This remaining operand has been partitioned twice, and if the blocksizes for these two loops are chosen properly, the partition can fit inside the L_{h-1} cache, and it is possible to implement the subproblem exposed by the L_{h-1} inner loop such that it remains in cache. Therefore we name this matrix partition the L_{h-1} resident matrix.

Suppose that t_h is the blocksize for the L_h cache that is not partitioned by the L_{h-1} inner loop. Then these are the cases that we may encounter for the L_{h-1} subproblem.

- If A is the L_{h-1} resident matrix, then the L_{h-1} subproblem is a $t_{h-1} \times s_{h-1}$

block of A times a $s_{h-1} \times t_h$ wide panel of B to update a $t_{h-1} \times t_h$ wide panel of C.

- If B is the L_{h-1} resident matrix, then the L_{h-1} subproblem is a $t_h \times t_{h-1}$ tall panel of A times a $t_{h-1} \times s_{h-1}$ block of B to update a $t_h \times s_{h-1}$ tall panel of C.
- If C is the L_{h-1} resident matrix, then the L_{h-1} subproblem is a $t_{h-1} \times t_h$ wide panel of A times a $t_h \times s_{h-1}$ tall panel of B to update a $t_{h-1} \times s_{h-1}$ block of C.

Suppose $s_{h-1} \approx t_{h-1}$. In this case the subproblem is a roughly square block with dimensions s_{h-1} and t_{h-1} residing in the L_{h-1} cache. The other operands have dimensions s_{h-1} and t_h and t_{h-1} and t_h, with t_h much larger than s_{h-1}, and they both remain in the L_h cache during the operation. If the resident matrix occupies most of the L_{h-1} cache, and the other two operands are streamed into it, then this is an optimal subproblem for the L_{h-1} cache.

Naming algorithms. We have now described a procedure to optimize for the I/O cost for the L_{h-1} cache, given that one of three shapes of subproblems is encountered at the L_h cache. One of the other two shapes of subproblems then is encountered at the L_{h-1} cache. One can then apply the same procedure to optimize for the next, smaller and faster level of cache, the L_{h-2} cache. The algorithms that arise from this methodology can be identified by which operand is the resident matrix in each level of cache. Therefore, we introduce a naming convention for the algorithms that states the level of cache followed by the operand resides in it. For instance if an algorithm has B as the resident matrix of the L_2 cache, A as the resident matrix of the L_1 cache, and C as the resident matrix in registers, it is called $B_2A_1C_0$.

5.2.4 Special cases for the family of algorithms

When applying this family of algorithms to different levels of the memory hierarchy, some special cases may arise.

Avoiding costly writes. When reading and writing from and to certain levels of the memory hierarchy, reads and write can be overlapped. For other levels of the memory hierarchy, they cannot. For example, it is often the case that reads and writes from and to caches can happen simultaneously. In contrast, reads and writes to and from DRAM utilize the same resources, and so a read plus a write to and from main memory can cost the same as two reads. Because of this, if there is a level of the memory hierarchy where a read plus a write costs more than a read from that level, it may be preferred to use an algorithm where the Resident C shape is encountered at the next smaller and faster level of the memory hierarchy.

Optimizing for registers. In our family of algorithms, we think of the register file as simply the smallest and fastest level of cache. However for practical reasons, it should be treated as a special case. In many implementations of MMM, the innermost kernel implements the Resident C algorithm [35, 80, 81, 43]. There are good reasons for this. The latency of the computation instructions dictates that there is a minimum number of register that must be used to store elements of C to avoid the instruction latency becoming a bottleneck. The number of elements of C that are stored in registers must be at least the product of the instruction latency and the number of elementary additions that can be performed per cycle [87, 57]. Often this means that a significant portion of the registers must be dedicated to storing elements of C, such it would be unnatural to use the Resident A or Resident B algorithms for the registers.

Therefore, it is often the case that there is no choice but to use Resident C for the registers. A side effect is that only Resident A or Resident B can be used for the next slower level of cache for which ones optimizes.

5.3 Multilevel Cache Tradeoffs

In this family of algorithms, when simultaneously optimizing for multiple levels of cache, there are tensions between I/O costs at the different levels of cache. That is, when there is an optimal subproblem at the L_h level of cache, and two loops partition the matrices so there is an optimal subproblem at the L_{h-1} level of cache, there will be more L_h cache misses than when only optimizing for the L_h cache, and there will be more L_{h-1} cache misses than when only optimizing for the L_{h-1} cache.

5.3.1 Impact of optimizing for the L_h cache on the L_{h-1} I/O cost

When blocking only for one level of cache, the streamed matrices are each associated with an aggregate I/O cost of $\mathcal{O}\left(\frac{mnk}{\sqrt{S}}\right)$. The I/O cost associated with the resident matrix is a lower ordered term and we often ignore it. When optimizing for both the L_h and L_{h-1} caches, however, the I/O cost associated with the L_{h-1} resident matrix becomes cubic because each element of the L_{h-1} resident matrix is moved into the L_{h-1} cache once per L_h subproblem. For illustration, suppose that the Resident A shape is encountered

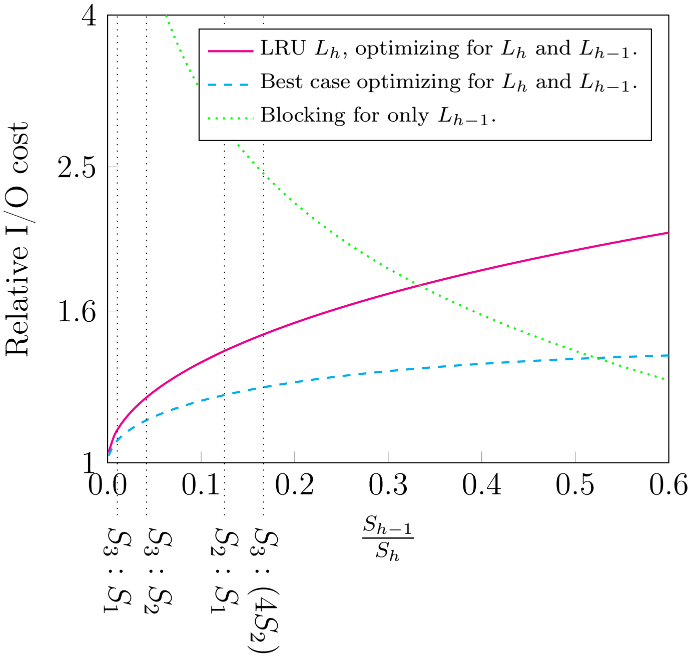

I/O cost relative to lower bound for different scenarios

Figure 5.3: This plot determines how much worse in terms of number of inputs to the L_h cache an algorithm will be if it optimizes for the L_{h-1} cache as well. If it is too costly, one may choose not to optimize for the L_{h-1} cache. Vertical lines mark cache size ratios for the Intel i7-7700K. For example, $\frac{S_2}{S_1}$ is too large to optimize for both the L_1 and L_2 caches, but it is reasonable to optimize for both L_2 and L_3.

at the L_h cache and the Resident C shape is encountered at the L_{h-1} cache. Suppose furthermore that m_c is the blocksize in the m dimension for the L_h cache and k_c is the blocksize in the k dimension for the L_h cache, and that m_r and n_r are the blocksizes in the m and n dimensions for the L_{h-1} cache. During the overall MMM operation, there are $\frac{mk}{m_c k_c}$ L_h subproblems, and each L_h subproblem is made up of $\frac{m_c n}{m_r n_r}$ L_{h-1} subproblems. An $m_r n_r$ block of C gets loaded into the L_{h-1} cache during each L_{h-1} subproblem. Multiplying by the number of subproblems gives a total I/O cost of $\frac{mnk}{k_c}$. Now consider the I/O cost associated with the L_{h-1} streamed matrix B. Per L_h subproblem, there is an I/O cost of $\frac{m_c k_c n}{m_r}$ associated with it. The overall I/O cost is $\frac{mnk}{m_r}$. This is the same as if we were only blocking for the L_h cache. In the general case, when optimizing for both L_h and L_{h-1}, the I/O cost associated with the L_{h-1} resident matrix will be $\approx \frac{mnk}{\sqrt{S_h}}$, whereas when only optimizing for the L_{h-1} cache, the I/O cost associated with the L_{h-1} resident matrix is either $\mathcal{O}(mn)$, $\mathcal{O}(mk)$ or $\mathcal{O}(nk)$. On the other hand, the I/O costs associated with the streamed matrices are not affected.

Tradeoffs are local. Optimizing for further levels of cache L_{h+1}, L_{h+2}, etc, does not affect the L_{h-1} I/O cost. We will now illustrate this claim with an example. Suppose we encounter the Resident B algorithm at the L_{h+1} cache, Resident A at the L_h cache, and Resident C at the L_{h-1} cache. Then suppose the blocksizes for the L_{h+1} cache are labeled k_{h+1} and n_{h+1}, the blocksizes for the L_h cache are labeled m_h and k_h, and the blocksizes for the L_{h-1} cache are labeled n_{h-1} and m_{h-1}. Consider an L_h subproblem. The dimensions associated with that subproblem are m_h, n_{h+1}, and k_h. Thus the L_{h-1} I/O costs associated with that subproblem are:

- A: $\frac{m_h k_h n_{h+1}}{n_{h-1}}$
- B: $\frac{m_h k_h n_{h+1}}{m_{h-1}}$
- C: $m_h n_{h+1}$

There are $\frac{m k_{h+1}}{m_h k_h}$ L_h subproblems per L_{h+1} subproblem. Multiplying that number with the L_{h-1} I/O costs above gives us the following L_{h-1} I/O costs per L_{h+1} subproblem.

- A: $\frac{m k_{h+1} n_{h+1}}{n_{h-1}}$
- B: $\frac{m k_{h+1} n_{h+1}}{m_{h-1}}$
- C: $\frac{m k_{h+1} n_{h+1}}{k_h}$

Finally, there are $\frac{nk}{n_{h+1} k_{h+1}}$ L_{h+1} subproblems overall. Multipying this number with the L_{h-1} I/O cost per L_{h+1} subproblem gives us the following overall L_{h-1} I/O costs.

- A: $\frac{mnk}{n_{h-1}}$
- B: $\frac{mnk}{m_{h-1}}$
- C: $\frac{mnk}{k_h}$

This illustrates that when looking at tradeoffs for the L_{h-1} cache, we only need to look at the blocking for the L_h cache, because blocking for further (bigger and slower) levels of cache does not harm the L_{h-1} I/O cost.

5.3.2 Impact of optimizing for L_{h-1} on the L_h I/O cost

Blocksizes must be chosen such that every matrix partition that must be able to fit into the L_h cache can do so. Different matrix partitions must fit into the L_h cache depends on the situation, and blocking for the L_{h-1} cache means that there is less room in the L_h cache for the L_h resident matrix, and blocksizes must be reduced.

Suppose that the L_h resident matrix is $s_h \times t_h$, and the L_{h-1} resident matrix is $s_{h-1} \times t_{h-1}$. We will now discuss what matrices must fit into the L_h cache, and the conditions on the blocksizes that this implies.

- At minimum, the L_h resident matrix and L_h guest matrix must fit into the L_h cache. The L_h resident matrix is of size $s_h \times t_h$ and the L_h guest matrix is of size $s_h \times s_{h-1}$ or $t_h \times s_{h-1}$. Thus, the blocksizes must satisfy the condition $s_h t_h + s_h s_{h-1} \leq S_h$ in the first case, and $s_h t_h + t_h s_{h-1} \leq S_h$ in the second.

- If the L_h cache is inclusive, meaning that anything in the L_{h-1} cache must also be in the L_h cache, then the L_{h-1} resident matrix must fit into the L_h cache as well. In this case an additional $t_{h-1} s_{h-1}$ elements must fit into cache.

- If the L_h cache is inclusive and LRU, then in order for an element to remain in the L_h cache, fewer than S_h elements may be accessed in between accesses of the element that we desire to remain in cache. This means that every matrix partition that is exposed by the L_{h-1} outer loop must fit into cache, so the blocksizes must satisfy the condition $s_h t_h + s_h s_{h-1} + t_h s_{h-1} \leq S_h$.

These conditions represent a tradeoff between optimizing for the L_h and L_{h-1} caches. The larger the L_{h-1} cache is, the more data must fit into the L_h cache, and the smaller the blocksizes s_h and t_h must be. If we make the simplifying assumptions that $s_h = t_h$, and that $t_{h-1} = s_{h-1} = \sqrt{S_{h-1}}$, and then adjust s_h and t_h to account for the extra data that must be in cache, the I/O cost for the L_h cache when optimizing for both the L_h and L_{h-1} caches can be determined by the ratio S_h/S_{h-1}.

For example, if the L_h is LRU and inclusive, then the L_h resident block and the panels of the L_h streamed matrices that are exposed by each iteration of the L_{h-1} outer loop must fit into the L_h cache. Therefore $s_h^2 + 2s_h t_{h-1} \leq S_h$. If we assume $t_{h-1} = \sqrt{S_{h-1}}$, then this condition becomes $s_h^2 + 2s_h\sqrt{S_{h-1}} \leq S_h$. Solving for s_h in the case that attains the equality, we obtain $s_h = -\sqrt{S_{h-1}} + \sqrt{S_{h-1} + S_h}$. In this case the I/O cost is $\frac{2mnk}{\sqrt{S_h + S_{h-1}} - \sqrt{S_h - 1}}$. If we divide this by the I/O lower bound, we obtain $\sqrt{1 + \frac{S_{h-1}}{S_h}} + \sqrt{\frac{S_{h-1}}{S_h}}$. This tells how many times worse than the L_h I/O lower bound we have when applying our family of algorithms to both the L_h and L_{h-1} caches. Figure 5.3 plots this curve along with the curve corresponding to the ideal case and the curve corresponding to blocking only for the L_{h-1} cache.

Sometimes, it is counterproductive or of limited value to optimize for the L_{h-1} cache if one is optimizing for the L_h cache. There are a few options for this case.

1. The simplest option is to simply treat the L_{h-1} cache as if it were smaller than it is.
2. Another option is to tweak the blocksizes for the L_{h-1} cache in a slightly more sophisticated way, as the number of elements in cache that are

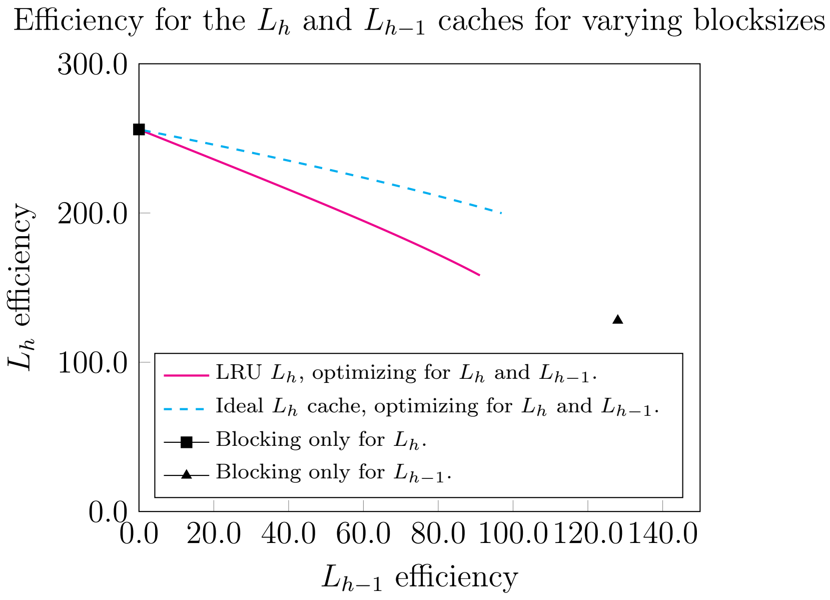


Figure 5.4: An algorithm's L_{h-1} versus L_h efficiency in terms of flops per I/O. Moving from left to right, the L_{h-1} blocksizes increase, decreasing the I/O costs associated with the L_{h-1} streamed matrices. This causes the L_h blocksizes to decrease, increasing the I/O costs associated with the L_h streamed matrices and the L_{h-1} resident matrix. In this plot, S_h is 256^2, and S_{h-1} is 128^2.

not part of the resident matrix depends on s_{h-1} if the L_h cache is LRU. Therefore, one can tweak the blocksizes so that s_{h-1} is smaller than t_{h-1}.

3. A third option is that instead of optimizing for the L_{h-1} cache, one could "skip" it and instead simultaneously optimize for the L_h and L_{h-2} caches.

Tradeoffs are local. When examining the impact of blocking for L_h upon the L_{h-1} I/O cost we found that this adversely effected the L_{h-1} I/O cost, but blocking for further (bigger and slower) levels of cache did not. Similarly, blocking for the L_{h-1} cache adversely affects the L_h I/O cost but blocking for further (smaller and faster) levels of cache does not. This is because the entire L_{h-2} subproblem fits within the data that must be in the L_h cache. These two properties mean that tradeoffs are local. When trying to optimize for one level of cache, one only has to consider the level of cache above it and the level of cache below it.

Pareto optimal solutions. In Figure 5.4, we consider a pair of caches, varying the L_{h-1} cache sizes of an algorithm, and comparing its L_h efficiency to its L_{h-1} efficiency (where efficiency is as defined in Section 4.2.2). If we assume that the resident matrices are square, and we only consider algorithms within this family of algorithms, then this plot gives us Pareto optimal solutions to the L_h and L_{h-1} tradeoff problem. When trying to resolve multilevel cache tradeoffs, this can be done for every pair of levels in the memory hierarchy.

5.3.3 Skipping caches

We have seen that tradeoffs occur when simultaneously optimizing for the I/O cost of multiple levels of cache. Sometimes these tradeoffs are too great, so instead of optimizing for both the L_h and L_{h-1} cache, one may forego the L_{h-1} cache, and instead simultaneously optimize for L_h and L_{h-2} I/O cost, where the L_{h-1} cache is intermediate between L_h and L_{h-2}. We call this *skipping* the L_{h-1} cache.

When the L_{h-1} cache is skipped, an optimal subproblem is encountered at the L_h level and at the L_{h-2} level, but not at the L_{h-1} level, and there are two loops targeting the L_{h-2} cache, but none for the L_{h-1} cache. Thus I/O cost for the L_{h-1} cache is not optimized in this case. However, this does not mean that the L_{h-1} is not useful. Recall that the L_h guest matrix is reused during each iteration of the L_{h-2} inner loop. With the right circumstances, this L_h guest matrix may be instead reused in the L_{h-1} cache, if that level of cache is skipped.

The L_h guest matrix will have on the order of $\sqrt{S_h S_{h-2}}$ elements, and so reusing the L_h guest matrix in the L_{h-1} cache may be appropriate if S_{h-1} is on the order of $\sqrt{S_h S_{h-2}}$. The exact blocksizes enabling the use of the L_h guest matrix in the L_{h-1} cache depend on the nature of the L_{h-1} cache. We will now analyze the blocksizes, assuming that $s_h = t_h = \sqrt{S_h}$, and $t_{h-1} = s_{h-1} = \sqrt{S_{h-2}}$

- In idealized circumstances, only the L_h guest matrix should need to be in the L_{h-1} cache. In this case, the condition $\sqrt{S_h S_{h-2}} \leq S_{h-1}$ must be satisfied.

- If the L_{h-1} cache is LRU, then a panel of the L_h resident matrix must also fit into the L_{h-1} cache. In this case, the condition $2\sqrt{S_h S_{h-2}} \leq S_{h-1}$ must be satisfied.

- If the L_{h-1} cache is inclusive, then the L_{h-2} resident block must also fit into the L_{h-1} cache. If it is inclusive and LRU, the condition $2\sqrt{S_h S_{h-2}} + S_{h-2} \leq S_{h-1}$ must be satisfied.

While this scheme uses the L_{h-1} cache, it does not optimize for it. The L_h guest matrix is reused from the L_{h-1} cache, but its dimensions are no where close to square, so its blocksizes do not balance the I/O costs of loading the other two operands into the L_{h-1} cache. Furthermore, if the L_{h-1} cache is LRU, then it occupies at most half of the L_{h-1} cache, so blocksizes must be much smaller than optimal. For these reasons, this skipping of the L_{h-1} cache, but still using it, does not satisfy the properties of an optimal subproblem, and furthermore we do not call it the L_{h-1} resident matrix.

We have seen that sometimes it is inadvisable to optimize for every level of cache, but that sometimes it is possible to still use a level of cache without optimizing the I/O cost for that cache. Doing so has some notable side effects.

- It is particularly favorable to place the L_h guest matrix in the L_{h-1} cache when the L_h cache is exclusive. Exclusive caches do not contain any data that is contained in smaller faster levels of the memory hierarchy. In this case the L_h guest matrix does not take up space in the L_h cache, so the L_h resident matrix can be increased in size, reducing the L_h I/O cost.

- The L_h guest panel, and one panel of the L_h resident matrix will be streamed through the L_{h-2} cache. Elements of the L_h guest panel will be less expensive to access since they will be in the L_{h-1} cache. One must take this asymmetry into account when choosing blocksizes for the L_{h-2} cache, as in Section 4.2.3.

- It is possible that one may wish to skip the biggest and slowest level of cache, but still use it. In this case, one may introduce a single outermost loop in order to constrain the size of a panel that can fit into the last level of cache. We characterize Goto's Algorithm as fitting into this family of algorithms, where there are two loops optimizing for the number of L_2 cache misses, and two loops optimizing for the number of loads into registers. The L_1 cache is skipped, and an outermost loop is added to use but not optimize for the L_3 cache. Since A is the resident matrix of the L_2 cache, and C is the resident matrix of the registers, the name of Goto's Algorithm according to the naming convention proposed in Section 5.2.3 is A_2C_0.

5.4 Experiments

5.4.1 Experimental setup

In this section, we will evaluate the algorithms that can be created by our methodology. We will do so by performing experiments on architectures with a varying number of levels of cache. In many practical circumstances, Goto's Algorithm attains excellent performance [79] that is difficult to exceed, despite the fact that it does not attain close to the I/O lower bound on machines with an L3 cache. Therefore, in order to evaluate the I/O cost of different algorithms, we will vary the cost of accessing main memory. This allows us to demonstrate that as reading and writing to main memory become more expensive, algorithms that optimize for the L3 cache become more efficient relative to those that do not.

In all of our experiments, we perform MMM in double precision. We are interested in scenarios where I/O is important, and double precision MMM requires the most bandwidth of the four common data types (single precision, complex, double precision, and double complex). We are not aware of this observation being in the literature, and so we present an analysis in Appendix C.

We created the Multilevel Optimized Matrix-matrix Multiplication Sandbox (MOMMS) in order to implement the algorithms for this paper. MOMMS implements algorithms for MMM by composing building blocks like matrix partitioning, packing, and parallelization at compile time. MOMMS is written in Rust [59]. Rust is a modern systems programming language focusing on memory safety; in safe Rust, there are no null or dangling pointers and there

Component	**Machine 1**	**Machine 2**
CPU	Intel i7-7700K	Intel i7-5775C
Memory	2x8GB DDR4-3200	2x8GB DDR3-2400
Motherboard Chipset	Intel Z270	Intel Z97

Table 5.1: Machines used in experiments in this paper.

are no data races. Most of this safety is enforced at compile-time through Rust's borrow checker. In Rust, memory is freed when it goes out of scope, so there is no garbage collector. From Rust, one can call C functions with very low overhead. For low-level kernels, MOMMS calls the BLIS micro-kernel. Memory safety, lack of garbage collection, zero cost abstractions, and easy low-overhead interface to C make Rust an appealing choice for high performance computing applications.

In our experiments, we compare against the BLAS [26] libraries ATLAS, BLIS, and Intel's Math Kernel Library (MKL). We used the 2017 release 2 of MKL, version 0.2.1 of BLIS, and 3.10.3 of ATLAS in our experiments.

Experimental platforms. We built two machines for our experiments. Relevant components are shown in Table 5.1. We will refer to these machines by their processor names. We chose the Z270 and Z97 chipset motherboards because these are enthusiast motherboards for consumers interested in overclocking, and so they provide the ability to change the memory multiplier. The i7-7700K machine is a 4-core Intel Kaby Lake machine with a 64KB L1 cache, a 256KB L2 cache, and a 6MB L3 cache. We chose this because it is a recent readily available Intel processor with an L3 cache. The i7-5775C is a 4-core Intel Broadwell machine. It also has a 64KB L1 cache, a 256KB L2 cache, and a 6MB L3 cache. More importantly it has 128MB of eDRAM, functioning as an L4 cache.

On both machines, all experiments were performed with hyperthreading disabled. A userspace CPU governor was used to set the CPUs to the nominal CPU frequency: 4.2 GHz for the i7-7700K, and 3.3 GHz for the i7-5775C.

Varying Bandwidth. The bandwidth to main memory can be determined by the product of the number of memory channels, the base clock rate, the number of bytes per transfer, and the memory multiplier. With DDR RAM, this is doubled since it transfers on both the leading and trailing edges of the clock signal. For our experiments we wish to increase the ratio of the rate of I/O to the rate of computation. Decreasing the base clock rate decreases both the rate of computation and the rate of I/O, but reducing the memory multiplier and the number of memory channels decreases the rate of I/O without changing the rate of computation, and they can be changed by tweaking a computer's BIOS settings.

5.4.2 Optimizing for the L3 cache

In this section, we will describe an algorithm that optimizes for both the L_3 and L_2 cache. We have implemented this algorithm using MOMMS, and will compare this algorithm to our implementation Goto's Algorithm (using MOMMS), and to vendor and state-of-the-art open source BLAS [26] implementations. Figure 5.5 compares Goto's Algorithm with our algorithm optimizing for the I/O cost of the L_3 and L_2 caches. We call our algorithm $B_3A_2C_0$, because B is the resident matrix of the L_3 cache, and A is the resident matrix of the L_2 cache.

We now describe the $B_3A_2C_0$ algorithm as implemented for the i7-

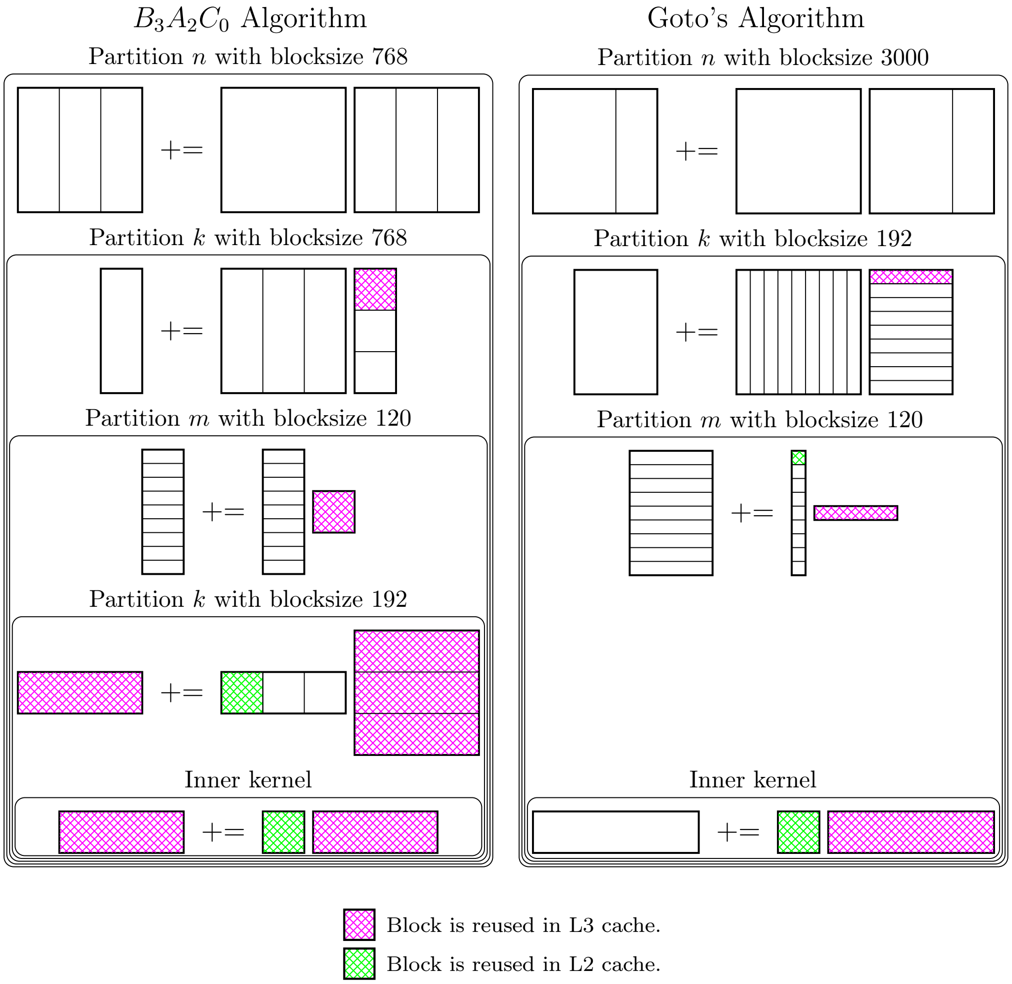

$B_3A_2C_0$ Algorithm
Goto's Algorithm
Partition n with blocksize 768
Partition n with blocksize 3000
+=
Partition k with blocksize 768
Partition k with blocksize 192
Partition m with blocksize 120
Partition m with blocksize 120
Partition k with blocksize 192
Inner kernel
Inner kernel
Block is reused in L3 cache.
Block is reused in L2 cache.

Figure 5.5: Two algorithms for MMM. Left: $B_3A_2C_0$. Right: Goto's Algorithm.

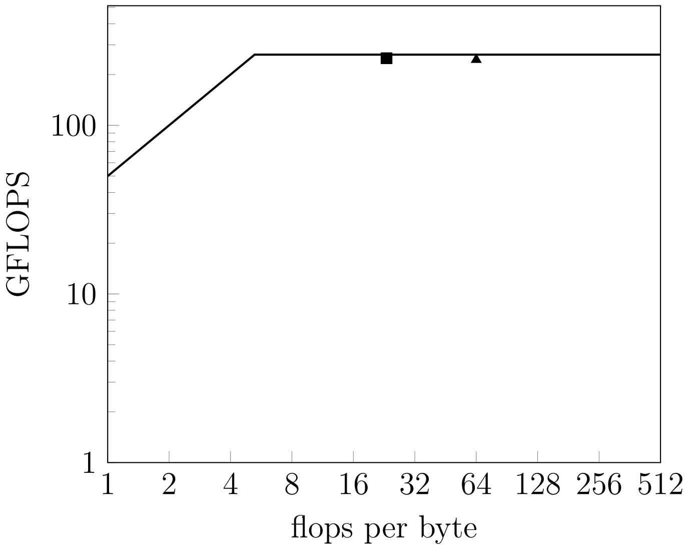
7700K and illustrated in Figure 5.5. First, we partition for the L_3 cache. The L_3 outer loop partitions the matrices in the n dimension with blocksize 768. Then the L_3 inner loop partitions in the k dimension, also with blocksize 768. This reveals a 768×768 block of B that will be the resident matrix of the L_3 cache. Next, we partition for the L_2 cache. Since B is the L_3 resident matrix, the L_2 outer loop must be in the m dimension, and it is with blocksize 120. The L_2 inner loop then partitions the k dimension with blocksize 192, making a block of A resident in the L_2 cache, and a 120×768 panel of C the guest matrix of the L_3 cache. We skip the L_1 cache, since it is a quarter the size of the L_2 cache, and therefore it is not beneficial to optimize for both the L_2 and L_1. The next two loops make a 4×12 block of C the resident matrix of registers, and a 192×12 panel of B the guest matrix of the L_2 cache. Since we skipped the L_1 cache, the 192×12 panel of B can reside in the L_1 cache as it is appropriately sized. Finally, we call a 4×12 micro-kernel provided by BLIS [80].

We compare against Goto's Algorithm with similar blocksizes as follows: n_c is 3000, k_c is 192, m_c is 120, m_r is 4, and n_r is 12. Our implementation of Goto's Algorithm uses the same micro-kernel from BLIS as does $B_3A_2C_0$. For both algorithms, we parallelize the second loop around the micro-kernel with 4 threads. The effect of this is to quadruple the bandwidth requirements of our algorithms.

Rooflines. In Figure 5.6, we show the roofline model [83] for the i7-7700K machine for the case of one channel of DDR4-800 RAM, and for the case of two channels of DDR4-3200 RAM.

The roofline model gives an upper bound on performance based on the

Dual channel DDR 3200 RAM



Single channel DDR 800 RAM

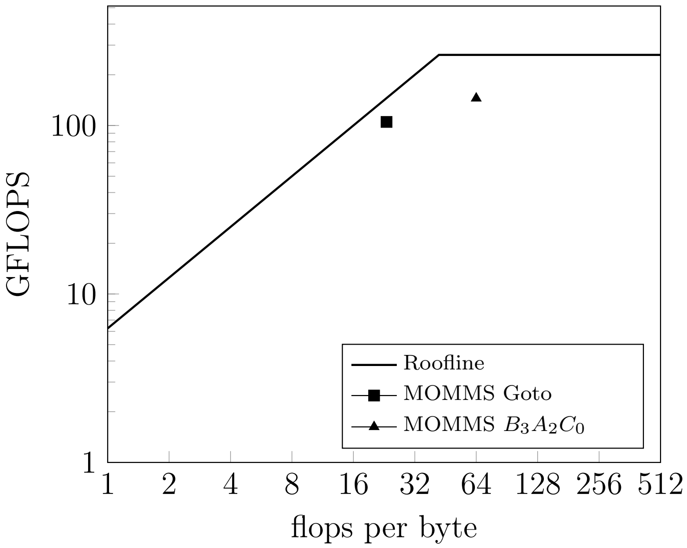


Figure 5.6: Roofline models for an Intel i7-7700K at 4.2GHz with 4 cores under two bandwidth conditions. For algorithms displayed on the plots, the y-axis is measured and the x-axis is theoretical. Bottom: Two channels of DDR4 is set to DDR-3200, for a peak bandwidth of 51200 MB/s. Top: One channel of DDR4 is set to DDR-800, for a peak bandwidth of 6400 MB/s.

arithmetic intensity of an algorithm for a specific machine. The machine is characterized by its rate of computation, and the rate at which it can transfer data between main memory and cache. The arithmetic intensity of an algorithm is the number of flops per byte transferred between memory and cache during the execution of that algorithm. When the arithmetic intensity is low it is bandwidth bound, and when the arithmetic intensity is high it is compute bound. The roofline model is thus a plot where the x-axis is the arithmetic intensity and the y-axis is maximum rate of computation for that arithmetic intensity. The roofline that serves as an upper bound on performance is formed by two linear curves that intersect when the minimum time spent for computation for an algorithm is equal to the minimum time spent for I/O. Algorithms are plotted on the roofline model according to their arithmetic intensity and measured performance as a way to explain their performance and to explain whether or not they could perform better. One can either measure the arithmetic intensity of an algorithm or analyze it. We choose to analyze the arithmetic intensity of the algorithms plotted.

The arithmetic intensities are determined by summing the theoretical I/O cost of each matrix, and dividing by the number of flops. The cost of reading and writing C to and from main memory is $\frac{2mnk}{k_c}$, the cost of loading A from main memory is $\frac{mnk}{n_c}$, and the cost of loading B from main memory is $\frac{mnk}{m}$. When the matrices are large, this gives an efficiency of $\left(\frac{1}{k_c} + \frac{1}{2n_c}\right)^{-1}$ flops per element. For our implementation, this is 23.26 flops per byte. For $B_3A_2C_0$, with a 768×768 block of B in the L_3 cache, the cost of reading and writing of C to and from main memory is $\frac{2mnk}{768}$, the cost of reading A from main memory is $\frac{mnk}{768}$, and the cost of reading B from main memory is $\frac{mnk}{m}$. This gives an I/O cost of $\frac{3mnk}{768} + nk$, and there are $2mnk$ flops. When the matrices are large, this gives an efficiency of $\frac{2*768}{3}$ flops per element or 64 flops per byte.

These cases represent the minimum and the maximum memory bandwidth that we can practically configure the machine for. We plot the modeled efficiency of Goto's Algorithm and the algorithm $B_3A_2C_0$ against each algorithm's measured performance. In the DDR4-3200 case, either algorithm could achieve the computation peak of the machine, but for the DDR4-800 case, only $B_3A_2C_0$ would be able to.

Varying Bandwidth. In Figure 5.7, we compare the achieved performance of Goto's Algorithm and $B_3A_2C_0$ for square matrices, varying the amount of bandwidth to main memory. Packing is often used to achieve spatial locality during an algorithm. Otherwise blocks that are designed to reside in cache may not be able to due to cache conflict issues [45]. Packing incurs extra memory movements that are not fundamentally required to happen during MMM. Therefore, we sidestep the spatial locality issue by storing matrices in a friendly format for our algorithms such that every time a matrix is partitioned, the blocks are stored contiguously.

At low bandwidth, $B_3A_2C_0$ outperforms Goto's Algorithm by thirty to forty percent. As the amount of bandwidth increases, I/O cost to and from main memory becomes less important, and the gap decreases and eventually disappears. With two channels of DDR4-3200, Goto's Algorithm slightly outperforms $B_3A_2C_0$. This is expected because $B_3A_2C_0$ has an additional loop as compared to Goto's Algorithm. It is simply more complicated and should therefore be slower when I/O is not an issue.

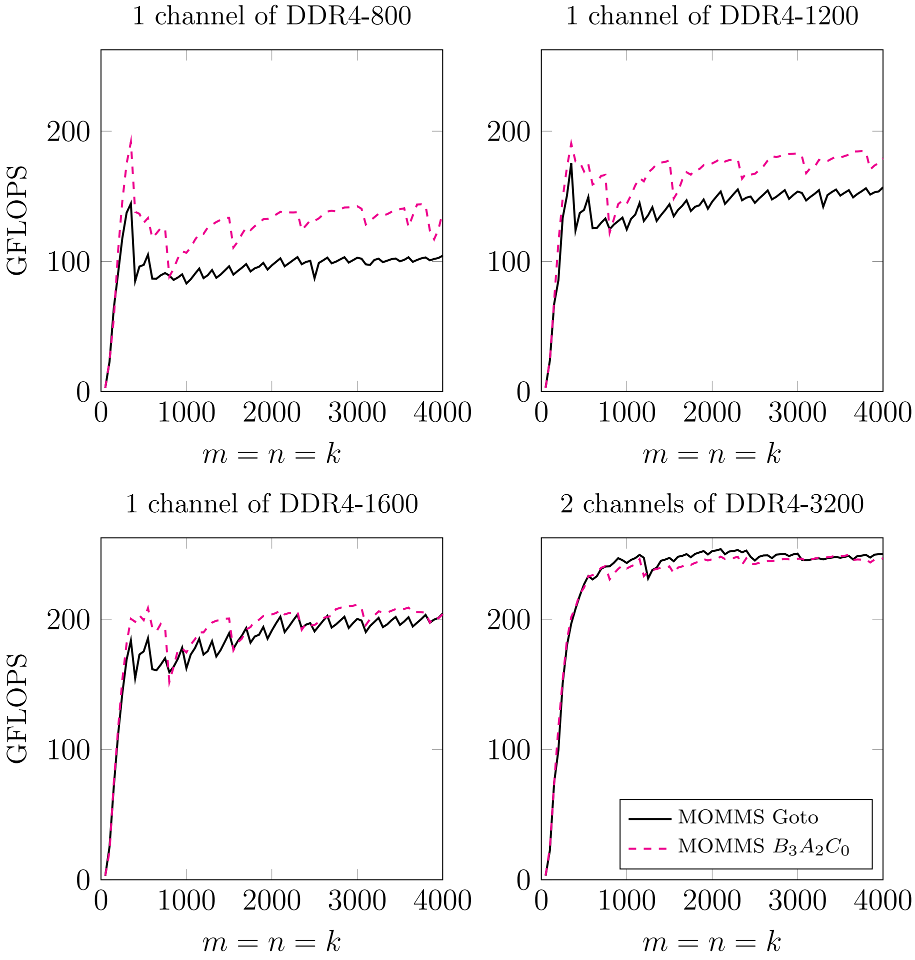


Figure 5.7: Performance of matrix-matrix multiplication on an Intel i7-7700K for square matrices, varying problem size and available bandwidth. Matrices are stored prepacked.

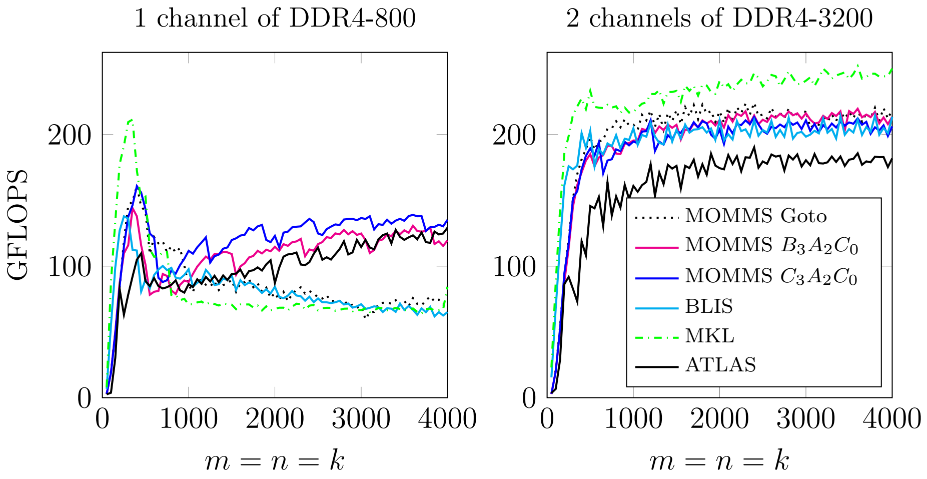


Figure 5.8: Practical comparision of performance against state-of-the-art open source and vendor libraries, for both high and low bandwidth scenarios. Matrices are passed in as column-major matrices, so packing is performed.

Comparing with existing implementations. In Figure 5.8, we compare our implementations of Goto's Algorithm and $B_3A_2C_0$ against the DGEMM routines in MKL and BLIS. It would not be fair to compare against implementations of MMM if we did not need to pack, so for this experiment, input matrices are stored in column major order, and our implementations of Goto's Algorithm and $B_3A_2C_0$ pack matrices the first time they become the resident or guest matrix at some level of cache. This packing (and the fact that C is not stored hierarchically for Goto's Algorithm) account for the performance difference seen for the Goto and $B_3A_2C_0$ curves between Figures 5.7 and 5.8. In this graph, we see that for high bandwidth scenarios, BLIS, the MOMMS implementation of Goto's Algorithm, and $B_3A_2C_0$ all attain roughly 75% of peak, and that MKL greatly outperforms the other implementations. MKL is the gold standard vendor implementation of MMM, and typically boasts the highest performance on any Intel CPU. For low bandwidth, $C_3A_2C_0$ performs the best, with $B_3A_2C_0$ close behind. ATLAS performs almost as well as the algorithms implemented in MOMMS that optimize for the L_3 I/O cost (although its performance ramps up slowly), but it does not perform nearly as well for the high bandwidth case.

This experiment demonstrates that state-of-the-art implementations of MMM do not optimize for the L_3 cache. When bandwidth to main memory is slow, these implementations are suboptimal. Although it greatly outperforms the others, performance even for $B_3A_2C_0$ and $C_3A_2C_0$ is quite low when there is little bandwidth to main memory. The Intel i7-7700K's L_3 cache is only 6MB, whereas L_3 caches on many server CPUs are much larger, suggesting that greater performance is possible for low bandwidth scenarios on such architectures.

5.4.3 Optimizing for the L4 cache

In this section, we demonstrate that our methodology can be efficiently applied to the Intel i7-5775C, which has four levels of cache. The L4 cache is 128MB of eDRAM.

We implemented an algorithm called $C_4A_2C_0$ for this architecture. Figure 5.9 shows the loop ordering and the blocksizes used for $C_4A_2C_0$. In $C_4A_2C_0$, a 3600×3600 block of C resides in the L_4 cache, and a 120×192 block of A resides in the L_2 cache. We decided to skip blocking for the L_3 cache, as there is sufficient bandwidth from the L_4 cache without optimizing for the number of L_3 cache misses. Nevertheless, the guest matrix of the L_4 cache, a 192×3600 panel of B, is appropriately sized to remain in the L_3. $C_4A_2C_0$ uses the same inner kernel as $B_3A_2C_0$.

In Figure 5.10, we compare the performance of Goto's Algorithm and

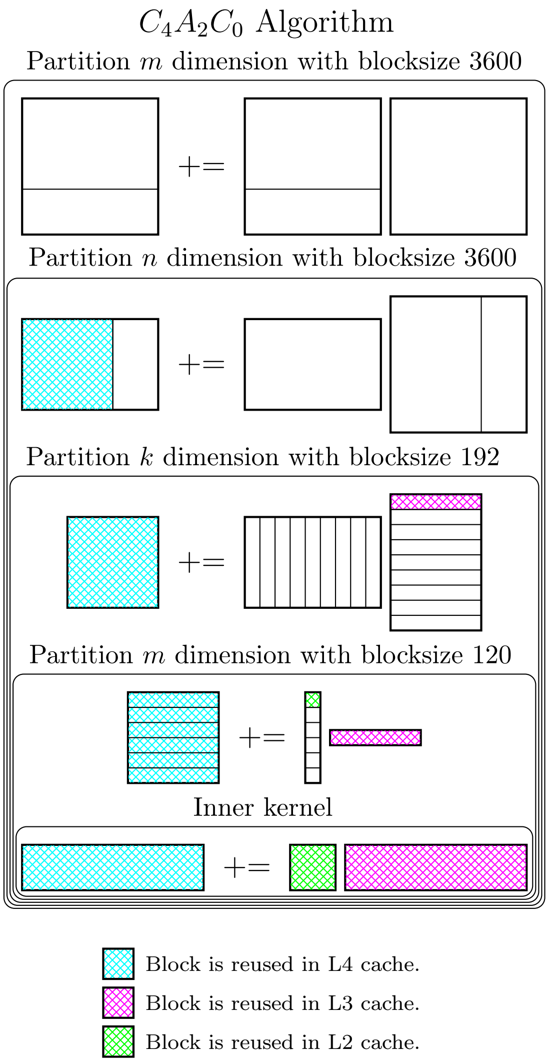

$C_4A_2C_0$ Algorithm
Partition m dimension with blocksize 3600
+=
Partition n dimension with blocksize 3600
+=
Partition k dimension with blocksize 192
+=
Partition m dimension with blocksize 120
+=
Inner kernel
+=
Block is reused in L4 cache.
Block is reused in L3 cache.
Block is reused in L2 cache.

Figure 5.9: An algorithm for MMM optimizing the number of inputs to both the L4 and the L2 caches. It places a square block of C in the L4 cache, and a square block of A in the L2 cache. The L3 cache is “skipped”, but the blocksizes are such that the L4 guest matrix, B, is reused from the L3 cache.

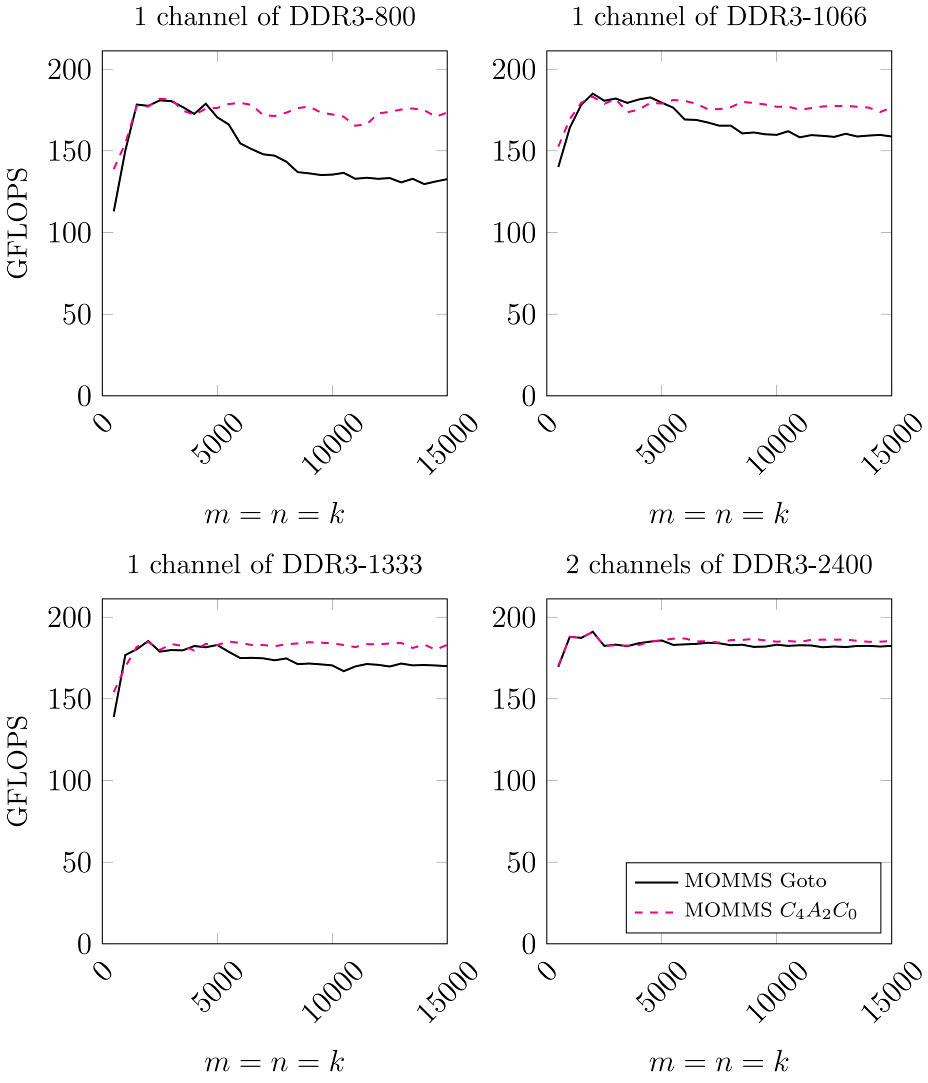


Figure 5.10: Performance of matrix-matrix multiplication for square matrices, varying problem size and available bandwidth. Matrices are stored prepacked.

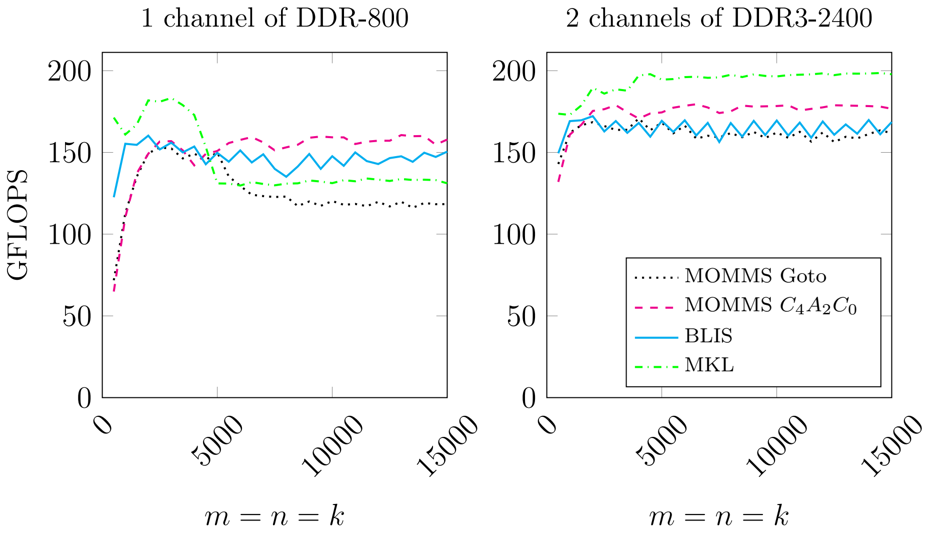


Figure 5.11: Matrix-matrix multiplication performance on an Intel i7-5775C with an 128MB L4 cache for a low bandwidth scenario (1 channel of DDR3-800) and a high bandwidth scenario (2 channels of DDR3-2400).

$C_4A_2C_0$ for square matrices across several bandwidths. In this experiment, matrices are stored hierarchically, and so packing is not performed. For high bandwidths, Goto's Algorithm and $C_4A_2C_0$ exhibit similar performance, but when bandwidth is low, $C_4A_2C_0$ outperforms Goto's Algorithm.

In Figure 5.11, we compare the performance on square matrices of our implementations of Goto's Algorithm and $C_4A_2C_0$, with Intel MKL and BLIS. In this experiment, matrices are stored in column-major order, and so packing is performed when partitions of A and B become resident or guest matrices of some level of cache. In $C_4A_2C_0$, C is unpacked when is no longer resident in L4. In both Figures 5.10 and 5.11, the top of the graphs is the peak computational rate of the machine.

Because the L4 cache is so large, we ran quite large problems, otherwise the matrices would completely fit into cache. Performance for Goto's Algorithm and MKL do not fall off until the problem size becomes $m = n = k \approx 5000$. From these graphs, we can see that Goto's Algorithm does not optimally use the L_4 cache, and neither does MKL.

BLIS's performance does not fall off as severely as for the other impementations when the problem size grows, however its performance overall is not as high. The algorithmic differences between BLIS and the MOMMS implemenation of Goto's Algorithm are parallelism and blocksizes. BLIS uses a larger k_c and smaller m_c than MOMMS and parallelizes the 2nd and 3rd loops around the micro-kernel, whereas MOMMS parallelizes the 2nd loop around the micro-kernel. Modifying either the parallelism or the blocksizes so that they match that of the MOMMS implementation of Goto's Algorithm adversely affects performance for the low bandwidth case, causing a noticeable dropoff for larger matrices. We can only conclude that somehow the way that data is shared by the threads within BLIS, coupled with the larger value of k_c within BLIS, (or the smaller m_c) fosters better reuse of data within the L4 cache. This is a reminder that it is possible for an algorithm to have better data reuse than was designed.

With DDR-800, all implementations of MMM on the i7-5775C outperform those on the i7-7700K, despite the fact that the processor is two generations older. The large L_4 cache means that blocksizes for the $C_4A_2C_0$ algorithm can be very large, so the algorithm does not need much bandwidth from main memory, but even for algorithms that do not take advantage of the L_4 cache by using such large blocksizes benefit from having the 128MB cache. This is because the large amount of cache space facilitates the hiding of latency to main memory, through techniques such as hardware prefetching.

5.4.4 Algorithms for different shapes of matrices

We will now compare different algorithms for different shapes of MMM. Algorithm $A_3B_2C_0$ partitions the matrices such that a square block of A is resident in the L_3 and a block of B is resident in the L_2 cache. It then calls an inner kernel updating a panel of C whose elements are in the L_3 cache by multiplying a block of A whose elements are in the L_2 cache times a panel of B whose elements are in the L_3 cache. Algorithm $C_3A_2C_0$ partitions the matrices such that a square block of C is resident in the L_3 and a block of A is resident in the L_2 cache. It then calls the same inner kernel as the algorithm $B_3A_2C_0$ does. Blocksizes and loop orderings for algorithms $A_3B_2C_0$ and $C_3A_2C_0$ are shown in Figure 5.12.

Algorithms $A_3B_2C_0$, $B_3A_2C_0$, and $C_3A_2C_0$ represent three choices for blocking for the L_3 cache. In Section 4.2.2, we argued that each of these choices may be optimal for a specific problem shape where two dimensions are equal to $\sqrt{S_3}$ and the other dimension is large, and that using the wrong algorithm with the wrong problem shape can result in an I/O cost that is 50% greater.

On a machine with three levels of cache and low bandwidth, we claim the following: $A_3B_2C_0$ casts its computation in terms of a block-panel multiply, with a block of A in the L_3 cache, and so it should be the best choice of the three algorithms when $m = k \approx \sqrt{S_3}$, and n is large. Similarly, $B_3A_2C_0$ casts its computation in terms of a panel-block multiply, with a block of B in the L_3 cache, and so it should be the best choice of the three algorithms when $n = k \approx \sqrt{S_3}$, and m is large. Finally, $C_3A_2C_0$ casts its computation in terms of a block dot product multiply, with a block of C in the L_3 cache, and so it should be the best choice of the three algorithms when $m = n \approx \sqrt{S_3}$, and k is large.

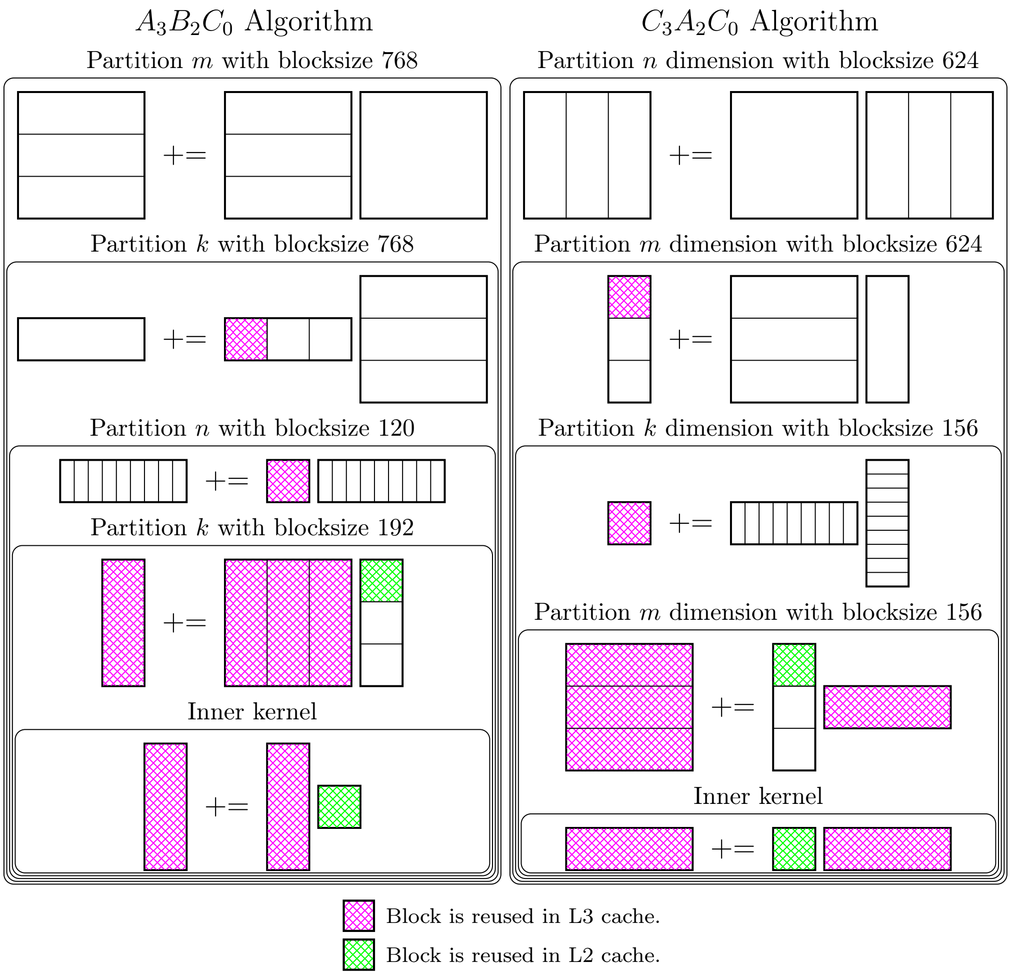

$A_3B_2C_0$ Algorithm
Partition m with blocksize 768
+=
Partition k with blocksize 768
+=
Partition n with blocksize 120
+=
Partition k with blocksize 192
+=
Inner kernel
+=
$C_3A_2C_0$ Algorithm
Partition n dimension with blocksize 624
+=
Partition m dimension with blocksize 624
+=
Partition k dimension with blocksize 156
+=
Partition m dimension with blocksize 156
+=
Inner kernel
+=
Block is reused in L3 cache.
Block is reused in L2 cache.

Figure 5.12: Two algorithms for MMM for three levels of cache. Left: $A_3B_2C_0$. Right: $C_3A_2C_0$.

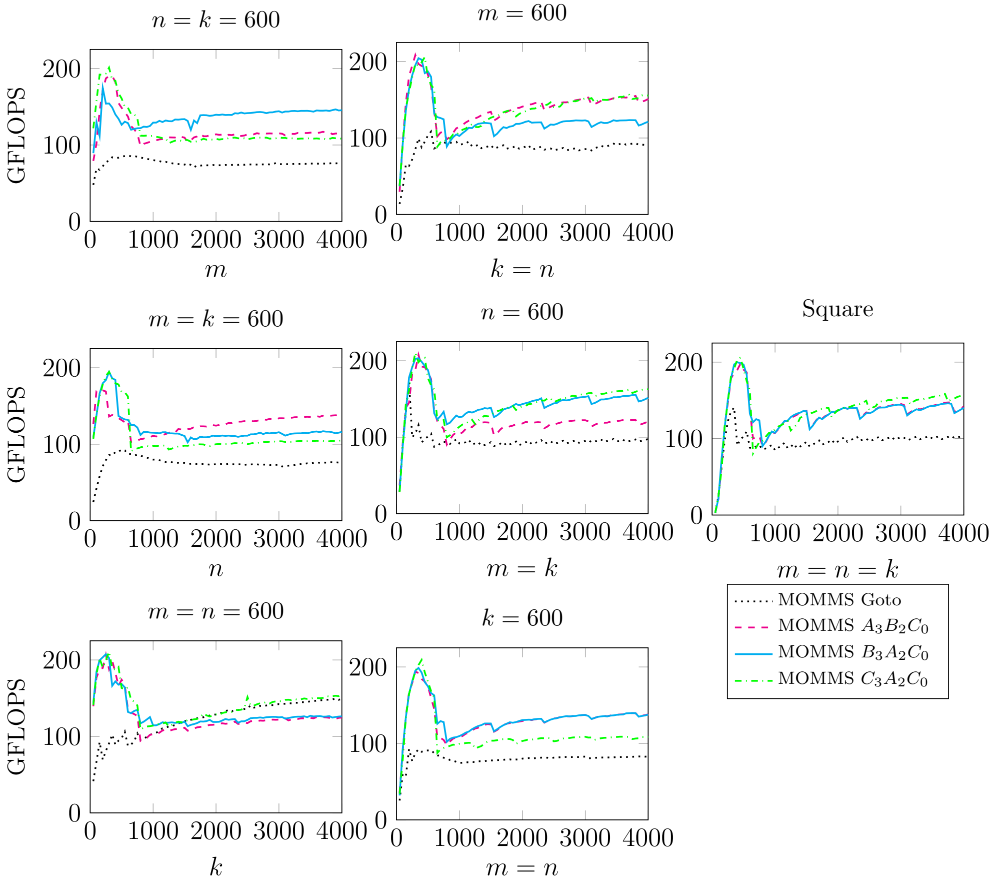


Figure 5.13: Matrix-matrix multiplication performance for different problem shapes on an Intel i7-7700K under a low-bandwidth scenario (1 channel of DDR4-800).

In Figure 5.13, we show performance results for MMM using $A_3B_2C_0$, $B_3A_2C_0$, $C_3A_2C_0$, and Goto's Algorithm, with matrices stored hierarchically and no packing. We vary the shape of the matrices. In each case, two of the dimensions are set to 768, and one of the dimensions is varied along the x-axis. The experiment performed on the Intel i7-7700K machine, with the DDR speed set to DDR4-800. When the dimension that is allowed to vary is large, the predicted algorithm outperforms the others. We also show performance when the matrices are square, and the size varies along the x-axis. For our algorithms that optimize for the L_3 cache, there is very little performance difference between the square case and the case where an algorithm is the "correct" choice. This shows that to get the best I/O properties when executing MMM for large matrices, it should be sufficient for an application to have two dimensions approximately equal to the the size of the square root of the last level of cache.

The algorithms $A_3B_2C_0$, $B_3A_2C_0$, and $C_3A_2C_0$ outperform Goto's Algorithm for larger problem sizes in this low bandwidth scenario. This is because even when the algorithm is wrong for the problem shape, the I/O cost is only 50% higher. In comparison, on this machine, Goto's Algorithm has an I/O cost that is approximately two times higher than the "correct" algorithm.

5.5 Summary

In this chapter, we have developed a new family of algorithms for simultaneously optimizing the I/O cost of MMM at multiple levels of the memory hierarchy. In Chapter 4, we analyzed the problem shapes that naturally arise from algorithms that attain the theoretical I/O lower bounds for MMM for a single level of cache that [68], and we named these shapes optimal subproblems. We then showed how one can use two loops at each level of the memory hierarchy in order to encounter an optimal subproblem at each level of cache. We analyzed the tradeoffs that occur when doing so, showing that one may not want to optimize for every level of cache. Finally we quantified the effectiveness of our family of algorithms by demonstrating performance improvements over state-of-the-art implementations of MMM when I/O cost to main memory is a limiting factor. While we only described algorithms for general MMM, we believe that the same techniques can be applied to cases where matrices have structure, for example when some matrices are triangular or symmetric.

In this chapter, we often focused on potential performance benefits of these algorithms, and in our experiments, we demonstrated differences in I/O costs by examining performance differences. However, even when two algorithms exhibit the same performance, we believe that the one with lower I/O cost is inherently better. This is because memory movements cost far more energy than flops do [66], so all things being equal, an algorithm that moves less data around will cost less money to execute and be better for the environment.

Many algorithms in libraries such as LAPACK [5] or libflame [39] take advantage of the fact that MMM implementations in BLAS libraries are efficient when the k dimension is relatively small, on the order of a couple of hundred, as Goto's Algorithm reaches its maximal efficiency when k is equal to the blocksize k_c. We expect future machines to be bandwidth bound when executing MMM in such situations, and algorithms for MMM that have larger blocksizes will be used. To take advantage of algorithms that use larger blocksizes, LAPACK and FLAME can use larger blocksizes, however this is currently disadvantageous because the larger their blocksizes, the more time is spent during inefficient unblocked subproblems. According to this line of thought, LAPACK and FLAME will need to use different algorithms that eliminate this weakness. One possible solution is to use recursive algorithms, as advocated in Peise and Bientinesi [62].

This chapter uses I/O lower bounds to make arguments about algorithms targeting hierarchical memory. A great deal of previous work uses I/O lower bounds to make arguments for algorithms targeting distributed memory instead [49, 70, 65, 8]. In this dissertation, and in the so-called 2.5D or 3D algorithms for MMM, the goal is to fill fast memory with one operand, and stream the other operands from slow memory. The difference is that for distributed memory systems, there is not enough data to fill local memory with one operand because data is distributed. To solve this, data can be duplicated between processors, allowing local memories to be filled with copies of data. Because of this duplication of data and because of limitations of the proof technique used in [68], the $\frac{2mnk}{\sqrt{S}}$ lower bound is not known to apply to these 2.5D or 3D algorithms. Despite this, it would be interesting to compare the 2.5D and 3D algorithms to the $\frac{2mnk}{\sqrt{S}}$ lower bound, and future work is to compare and contrast those algorithms for distributed MMM to the algorithms for hierarchical MMM in this chapter.

Hard drives and other similarly slow storage devices can be thought of as another layer of the memory hierarchy. Because of this, we believe that out-of-core algorithms for matrix-matrix multiplication can be considered part of this family of algorithms. A major difference between such out-of-core algorithms and the ones in this chapter targeting LRU caches is that out-of-core algorithms may require explicit reads and writes to disk and explicit overlapping of I/O and computation.

We believe that the algorithms in this chapter can be easily generalized to other dense linear algebra operations, much in the way that Goto's Algorithm [35] was generalized to the rest of the Level-3 BLAS [36]. The key point is that most suboperations during the other level 3 BLAS operations (that operate on structured matrices) are just regular, unstructured MMM operations.

Chapter 6

Cache Aware Multithreaded Parallelization

6.1 Introduction

When parallelizing MMM, one should keep in mind how hardware resources are shared between threads and use that to guide how data should be shared between threads. We will now provide a brief example that illustrates that if the goal is to minimize the I/O cost for MMM, one should to parallelize in a cache-aware fashion. Suppose that there is a machine N threads, and the N threads share a fast memory of size S. If one parallelizes such that each thread works on completely independent data, and the cache is split equally between the threads, then each thread has its own fast memory of size S/N. In this scheme, an algorithm for MMM has an I/O cost of at least $\frac{2mnk}{\sqrt{S/N}} = \frac{2mnk\sqrt{N}}{\sqrt{S}}$. If we were instead to parallelize such that all the threads work with the same resident block at the same time, then we could perhaps we could parallelize such that the parallel MMM algorithm still attains the I/O lower bound of $\frac{2mnk}{\sqrt{S}}$.

The goal of this chapter is how to implement shared-memory parallel MMM when accounting for the fact that on different computer architectures, different levels of cache may be shared by threads. We do so by first investigating the properties of parallelizing the different loops within the BLIS implementation of Goto's Algorithm (in Section 6.2). Sections 6.3 and 6.4 form a case study detailing which of these loops to parallelize for the the IBM

Blue Gene/Q PowerPC A2 (BGQ), and the Intel Xeon Phi Knight's Corner (KNC) architectures. In Section 6.5 we take the insights from parallelizing Goto's Algorithm and apply them to the family of algorithms introduced in Chapter 5.

The text and figures from this chapter are taken in part from [67].

6.2 Parallelization of loops within Goto's Algorithm

We will now break down the properties of the loops within the BLIS implementation of Goto's Algorithm, applying the ideas of how to parallelize the family of algorithms from Chapter 5. In the original GotoBLAS implementation, the inner kernel is the basic unit of computation. Parallelization is not incorporated within it, and it would not be feasible to do so since it is typically implemented in assembly. The BLIS framework exposes two loops within that inner kernel that are implemented in C, and casts computation in terms of a smaller basic unit of computation, the micro-kernel. This smaller basic unit of computation exposes two new loops for which it is feasible to parallelize. The loops that we consider parallelizing within BLIS are shown in Figure 6.1, an illustration of Goto's Algorithm within BLIS with the loops labeled is shown in Figure 6.3, and a diagram showing which layer of the memory hierarchy that different matrix partitions reside in is shown in Figure 6.2.

6.2.1 The first loop around the micro-kernel

First consider Loop 1 in Figure 6.1, illustrated in Figure 6.4. If one parallelizes the first loop around the micro-kernel (indexed by i_r), different instances of the micro-kernel are assigned to different threads. Our objective is to optimally use fast memory resources. In this case, the different threads

```
Loop 5  for $j_c = 0, \ldots, n-1$ in steps of $n_c$, $\mathcal{J}_c = j_c : j_c + n_c - 1$
Loop 4    for $p_c = 0, \ldots, k-1$ in steps of $k_c$, $\mathcal{P}_c = p_c : p_c + k_c - 1$
            $B(\mathcal{P}_c, \mathcal{J}_c) \rightarrow B_c$                // Pack into $B_c$
Loop 3      for $i_c = 0, \ldots, m-1$ in steps of $m_c$, $\mathcal{I}_c = i_c : i_c + m_c - 1$
              $A(\mathcal{I}_c, \mathcal{P}_c) \rightarrow A_c$              // Pack into $A_c$
Loop 2        for $j_r = 0, \ldots, n_c-1$ in steps of $n_r$, $\mathcal{J}_r = j_r : j_r + n_r - 1$
Loop 1          for $i_r = 0, \ldots, m_c-1$ in steps of $m_r$, $\mathcal{I}_r = i_r : i_r + m_r - 1$
Loop 0            for $k_r = 0, \ldots, k_c-1$       // Micro-kernel
                    $C_c(\mathcal{I}_r, \mathcal{J}_r)$ += $A_c(\mathcal{I}_r, k_r)$ $B_c(k_r, \mathcal{J}_r)$
                  endfor
                endfor
              endfor
            endfor
          endfor
        endfor
```

Figure 6.1: Loops implementing Goto's Algorithm in BLIS.

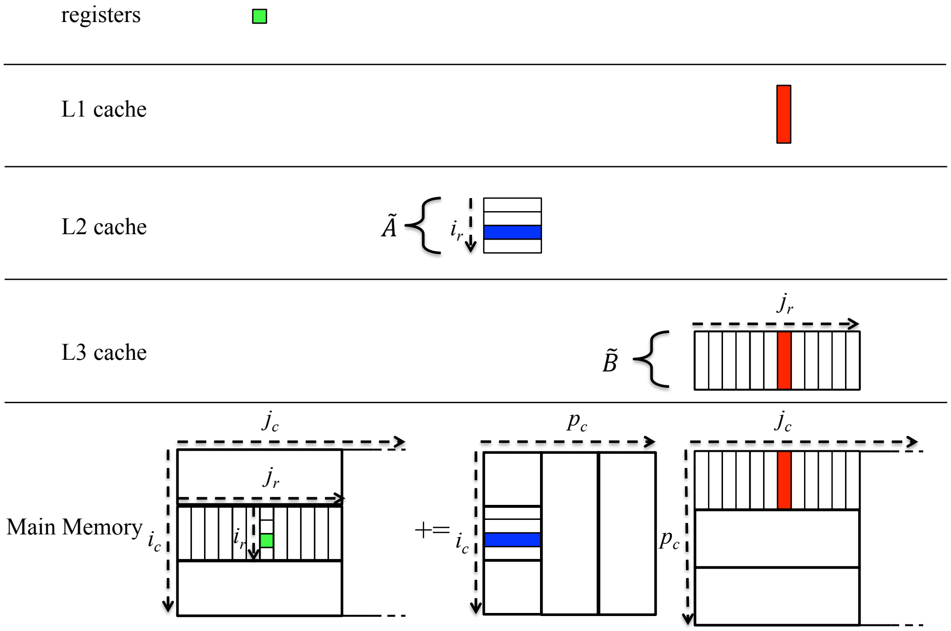


Figure 6.2: Illustration of which parts of the memory hierarchy each block of A and B reside in during the execution of the micro-kernel with BLIS.

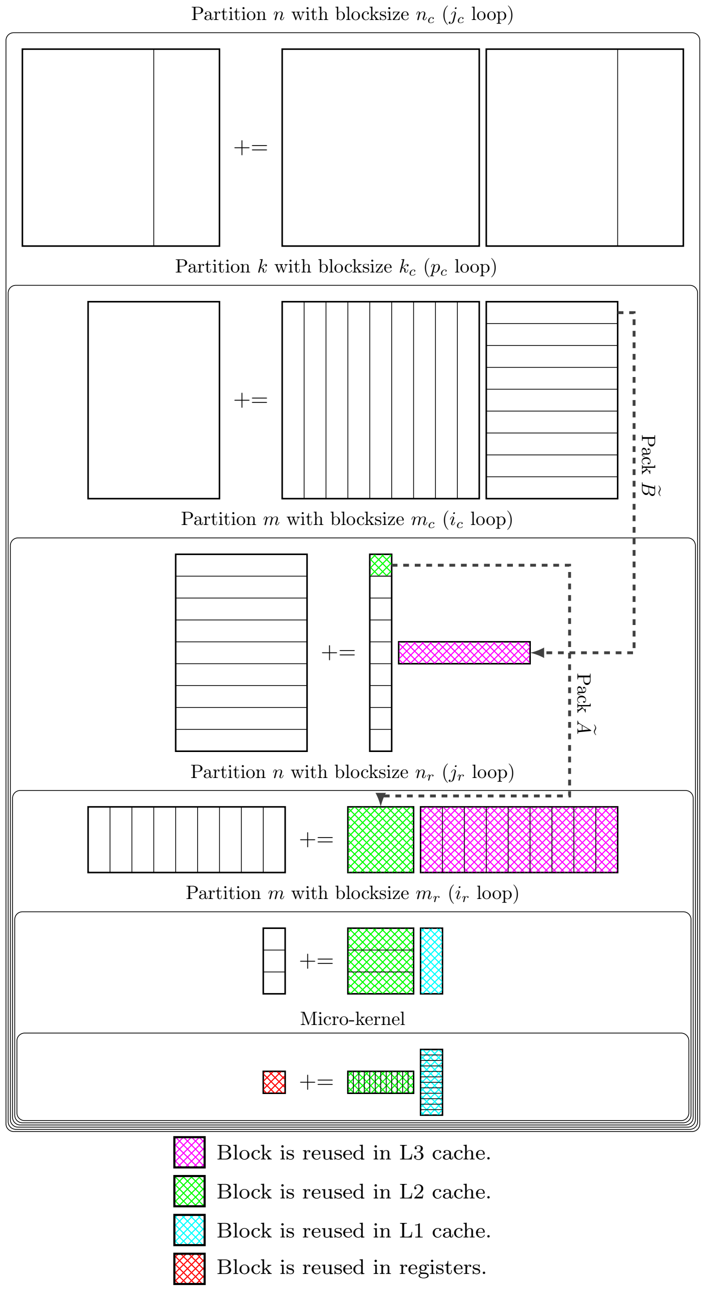

Partition n with blocksize n_c (j_c loop)
+=
Partition k with blocksize k_c (p_c loop)
+=
Pack $\widetilde{B}$
Partition m with blocksize m_c (i_c loop)
+=
Pack $\widetilde{A}$
Partition n with blocksize n_r (j_r loop)
+=
Partition m with blocksize m_r (i_r loop)
+=
Micro-kernel
+=
Block is reused in L3 cache.
Block is reused in L2 cache.
Block is reused in L1 cache.
Block is reused in registers.

Figure 6.3: Diagram of Goto's Algorithm implemented in BLIS.

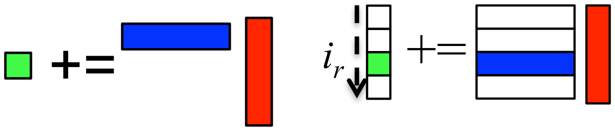


Figure 6.4: Left: the micro-kernel. Right: the first loop around the micro-kernel.

share the same micro-panel of $\widetilde{B}$, which resides in the L_1 cache.

Regardless of the size of the matrices on which we operate, this loop has a fixed number of iterations, $\lceil \frac{m_c}{m_r} \rceil$, since it loops over m_c in steps of m_r. Thus, the amount of parallelism that can be extracted from this loop is quite limited. Additionally, a micro-panel of $\widetilde{B}$ is brought from the L_3 cache into the L_1 cache and then used during each iteration of this loop. When parallelized, less time is spent in this loop and thus the cost of bringing that micro-panel of $\widetilde{B}$ into the L_1 cache is amortized over less computation. The cost of bringing $\widetilde{B}$ into the L_1 cache may be overlapped by computation, so it may be completely or partially hidden. In this case, there is a minimum amount of computation required to hide the cost of bringing $\widetilde{B}$ into the L_1 cache. Thus, parallelizing is acceptable only when this loop has a large number of iterations. These two factors mean that this loop should be parallelized only when the ratio of m_c to m_r is large. Unfortunately, this is not usually the case, as m_c is usually on the order of a few hundred elements.

6.2.2 The second loop around the micro-kernel

Now consider Loop 2 in Figure 6.1, illustrated in Figure 6.5. If one parallelizes the second loop around the micro-kernel (indexed by j_r), each thread will be assigned a different micro-panel of $\widetilde{B}$, which resides in the L_1

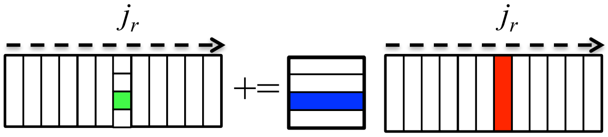


Figure 6.5: The second loop around the micro-kernel.

cache, and they will all share the same block of $\widetilde{A}$, which resides in the L_2 cache. Then, each thread will multiply the block of $\widetilde{A}$ with its own micro-panel of $\widetilde{B}$.

Similar to the first loop around the micro-kernel, this loop has a fixed number of iterations, as it iterates over n_c in steps of n_r. The time spent in this loop amortizes the cost of packing the block of $\widetilde{A}$ from main memory into the L_2 cache. Thus, for similar reasons as the first loop around the micro-kernel, this loop should be parallelized only if the ratio of n_c to n_r is large. Fortunately, this is almost always the case, as n_c is typically on the order of several thousand elements.

Consider the case where this loop is parallelized and each thread shares a single L_2 cache. Here, one block $\widetilde{A}$ will be moved into the L_2 cache, and there will be several micro-panels of $\widetilde{B}$ which also require space in the cache. Thus, it is possible that either $\widetilde{A}$ or the micro-panels of $\widetilde{B}$ will have to be resized so that all fit into the cache simultaneously. However, micro-panels of $\widetilde{B}$ are small compared to the size of the L_2 cache, so this will likely not be an issue.

Now consider the case where the L_2 cache is not shared, and this loop over n_c is parallelized. Each thread will pack part of $\widetilde{A}$, and then use the entire block of $\widetilde{A}$ for its local computation. In the serial case of GEMM, the process of packing of $\widetilde{A}$ moves it into a single L_2 cache. In contrast, parallelizing this loop results in various parts of $\widetilde{A}$ being placed into *different* L_2 caches. This is due to the fact that the packing of $\widetilde{A}$ is parallelized. Within the parallelized packing routine, each thread will pack a different part of $\widetilde{A}$, and so that part of $\widetilde{A}$ will end up in that thread's private L_2 cache. A cache coherency protocol must then be relied upon to guarantee that the pieces of $\widetilde{A}$ are duplicated across the L_2 caches, as needed. This occurs during the execution of the microkernel and may be overlapped with computation. Because this results in extra memory movements and relies on cache coherency, this may or may not be desireable depending on the cost of duplication among the caches. If the architecture does not provide cache coherency, the duplication of the pieces of $\widetilde{A}$ must be done manually.

6.2.3 The third loop around the inner-kernel

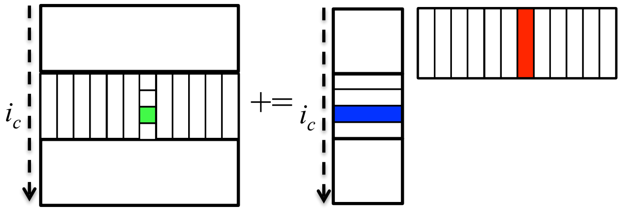


Figure 6.6: The third loop around the micro-kernel (first loop around Goto's inner kernel).

Next consider Loop 3 in Figure 6.1, illustrated in Figure 6.6. If one parallelizes this first loop around what we call the macro-kernel (indexed by i_c), which corresponds to Goto's inner kernel, each thread will be assigned a different block of $\widetilde{A}$, which resides in the L_2 cache, and they will all share the same row panel of $\widetilde{B}$, which resides in the L_3 cache or main memory. Subsequently, each thread will multiply its own block of $\widetilde{A}$ with the shared row panel of $\widetilde{B}$.

Unlike the inner-most two loops around the micro-kernel, the number of iterations of this loop is not limited by the blocking sizes; rather, the number of iterations of this loop depends on the size of m. When m is less than the product of m_c and the degree of parallelization of the loop, blocks of $\widetilde{A}$ will be smaller than optimal and performance will suffer.

Now consider the case where there is a single, shared L_2 cache. If this loop is parallelized, there must be multiple blocks of $\widetilde{A}$ in this cache. Thus, the size of each $\widetilde{A}$ must be reduced in size by a factor equal to the degree of parallelization of this loop. The size of $\widetilde{A}$ is $m_c \times k_c$, so either or both of these may be reduced. If we choose to reduce m_c, parallelizing this loop is equivalent to parallelizing the first loop around the micro-kernel. If instead each thread has its own L_2 cache, each block of $\widetilde{A}$ resides in its own cache, and thus it would not need to be resized.

Now consider the case where there are multiple L_3 caches. If this loop is parallelized, each thread will pack a different part of the row panel of $\widetilde{B}$ into its own L_3 cache. Then a cache coherency protocol must be relied upon to place every portion of $\widetilde{B}$ in each L_3 cache. As before, if the architecture does not provide cache coherency, this duplication of the pieces of $\widetilde{B}$ must be done manually.

6.2.4 The fourth loop around the inner-kernel

Consider Loop 4 in Figure 6.1, illustrated in Figure 6.7. If one parallelizes this second loop around the macro-kernel (indexed by p_c), each thread will be assigned a different block of $\widetilde{A}$ and a different block of $\widetilde{B}$. A problem when parallelizing this loop is that each thread will update the same block

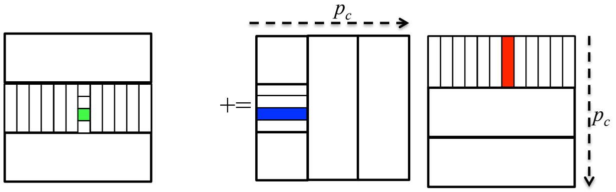


Figure 6.7: The fourth loop around the micro-kernel (second loop around Goto's inner kernel).

of C, potentially creating race conditions. Thus, parallelizing this loop either may requires a synchronization mechanism coordinating the threads updating the same parts of C. Another solution is for each thread to accumulate parts of AB in its own buffer, with partial accumulations of C summed together afterwards, as illustrated in Figure 6.8. Loop 4 should only be parallelized when m and n are small compared to the number of threads so that only by parallelizing this loop can a satisfactory level of parallelism be achieved. It is for these reasons that 2.5 or 3D distributed memory matrix multiplication algorithms [71, 2] parallelize the k dimension for parallelization (in addition to parallelizing the m and n dimensions).

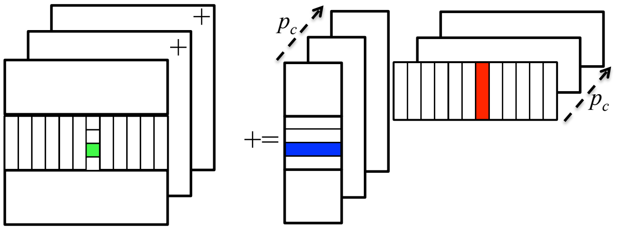


Figure 6.8: Parallelization of the p_c loop requires local copies of the block of C to be made, which are summed upon completion of the loop.

6.2.5 The outer-most loop

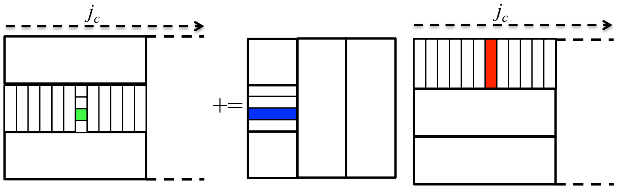


Figure 6.9: The fifth (outer) loop around the micro-kernel.

Finally, consider Loop 5 in Figure 6.1, the outermost loop in BLIS, and illustrated in Figure 6.9. If one parallelizes this loop, each thread will be assigned a different row panel of $\widetilde{B}$, and each thread will share the whole matrix A which resides in main memory.

Consider the case where there is a single L_3 cache. Then the size of a panel of $\widetilde{B}$ must be reduced so that multiple of $\widetilde{B}$ will fit in the L_3 cache. If n_c is reduced, then this is equivalent to parallelizing the 2nd loop around the micro-kernel, in terms of how the data is partitioned among threads. If instead each thread has its own L_3 cache, then the size of $\widetilde{B}$ will not have to be altered, as each panel of $\widetilde{B}$ will reside in its own cache.

Parallelizing this loop thus may be a good idea on multi-socket systems where each CPU has a separate L_3 cache. Threads parallelizing this loop do not share any packed buffers of $\widetilde{A}$ or $\widetilde{B}$, so parallelizing this loop is, from a data-sharing perspective, equivalent to gaining parallelism outside of BLIS.

6.2.6 Parallelism within the micro-kernel

In this section we have considered the micro-kernel to be a black-box basic unit of computation. We will now discuss parallelizing within it. The micro-kernel is a block dot-product implemented as a loop around rank-1 updates of an $m_r \times n_r$ block of C that is accumulated in registers. Parallelizing this loop around these rank-1 updates is ill-advised because the threads would accumulate contributions to the same block of C, and the operation is too small to amortize the synchronization overheads required to facilitate multiple threads updating the same data.

One could envision carefully parallezing the m or n dimension in the micro-kernel. Such parallelism could be described as some combination of parallelizing the first and second loop around the micro-kernel.

6.3 MMM Parallelization for Intel Xeon Phi KNC

We now discuss how BLIS supports high performance and scalability on the Xeon Phi architecture.

6.3.1 Architectural Details

The Xeon Phi has 60 cores, each of which has its own 512 KB L_2 cache and 32 KB L_1 data cache. Each core has four hardware threads, all of which share the same L_1 cache. A core is capable of dispatching two instructions per clock cycle, utilizing the core's two pipelines. One of these may be used to execute vector floating point instructions or vector memory instructions. The other may only be used to execute scalar instructions or prefetch instructions. If peak performance is to be achieved, the instruction pipeline that is capable of executing floating point operations should be executing a fused multiply accumulate instruction (FMA) as often as possible. One thread may only issue one instruction to each pipeline every other clock cycle. Thus, utilizing two hardware threads is the minimum necessary to fully occupy the floating point unit. Using four hardware threads further alleviates instruction latency and bandwidth issues [48].

Although these hardware threads may seem similar to the hyper-threading found on more conventional CPUs, the fact is that hyper-threading is not often used for high-performance computing applications, and these hardware threads must be used for peak performance.

6.3.2 The BLIS implementation on the Intel Xeon Phi

Because of the highly parallel nature of the Intel Xeon Phi, the micro-kernel must be designed while keeping the parallelism gained from the core-sharing hardware threads in mind. On conventional architectures, micro-panels of $\widetilde{A}$ and $\widetilde{B}$ are sized such that $\widetilde{B}$ resides in the L_1 cache, and $\widetilde{B}$ is streamed from memory. However, this regime is not appropriate for the Xeon Phi. This is due to the fact that with four threads sharing an L_1 cache, parallelizing in the m and n dimensions means that there must be room for at least two micro-panels of $\widetilde{A}$ and two micro-panels of $\widetilde{B}$ in the L_1 cache. On the Xeon Phi, to fit so much data into the L_1 cache would mean reducing k_c to a point where the cost of updating the $m_r \times n_r$ block of C is not amortized by enough computation. Thus no data will reside in the L_1 cache.

We now discuss the register and cache blocksizes for the BLIS implementation of Xeon Phi, as they affect how much parallelism can be gained from each loop. Various pipeline restrictions for the Xeon Phi mean that its micro-kernel must either update a 30×8 or 8×30 block of C. For our study, we have choosen 30×8. Next, the block of $\widetilde{A}$ must fit into the 512 KB L_2 cache: m_c is chosen to be 120, and k_c is chosen to be 240. There is no L_3 cache, so n_c is only bounded by main memory, and by the amount of memory we want to use for the temporary buffer holding the panel of $\widetilde{B}$. For this reason we choose n_c to be 14400, which is the largest n dimension for any matrix we use for our experiments.[1]

6.3.3 Which loops to parallelize

The sheer number of threads (240) and the fact that hardware threads are organized in a hierarchical manner suggests parallelizing multiple loops. We use the fork-join model to parallelize multiple loops. When a thread encounters a loop with P-way parallelism, it will spawn P children, and those P threads parallelize that loop instance. The total number of threads is the product of the number of threads parallelizing each loop. We will now take the insights from the last section to determine which loops would be appropriate to parallelize, and to what degree. In this section we will use the name of the index variable to identify each loop, as shown in Figures 6.1 and 6.3.

- The i_r loop: With an m_c of 120 and m_r of 30, this loop only has four iterations, thus it does not present a favorable opportunity for parallelization.
- The j_r loop: Since n_c is 14400, and n_r is only 8, this loop provides an excellent opportunity for parallelism. This is especially true among the hardware threads. The four hardware threads share an L_2 cache, and if this loop is parallelized among those threads, they will also share a block of $\widetilde{A}$.

---

[1] If we instead choose n_c to be 7200, performance drops by approximately 2 percent of the peak of the machine.

- The i_c loop: Since this loop has steps of 120, and it iterates over all of m, this loop provides a good opportunity when m is large. Additionally, since each core of the Xeon Phi has its own L_2 cache, parallelizing this loop is beneficial because the size of $\widetilde{A}$ will not have to be changed as long as the loop is not parallelized by threads within a core. If the cores share an L_2 cache, parallelizing this loop would result in multiple blocks of $\widetilde{A}$, each of which would have to be reduced in size since they would all have to fit into one L_2 cache.

- The p_c loop: We do not consider this loop for reasons explained in Section 6.2.4 above.

- The j_c loop: Since the Xeon Phi lacks an L_3 cache, this loop provides no advantage over the j_r loop for parallelizing in the n dimension. It also offers worse spatial locality than the j_r loop, since there would be different buffers of $\widetilde{B}$.

We have now identified two loops as opportunities for parallelism on the Xeon Phi, the j_r and i_c loops.

6.3.4 Parallelism within cores

It is advantageous for hardware threads on a single core to parallelize the j_r loop. If this is done, then each hardware thread is assigned a different micro-panel of $\widetilde{B}$, and the four threads share the same block of $\widetilde{A}$. If the four hardware threads are synchronized, they will access the same micro-panel of $\widetilde{A}$ concurrently. Not only that, if all four threads operate on the same region of $\widetilde{A}$ at the same time, one of the threads will load an element of $\widetilde{A}$ into the L_1 cache, and all four threads will use it before it is evicted. Thus, parallelizing the j_r loop and synchronizing the four hardware threads will reduce bandwidth requirements of the micro-kernel. The synchronization of the four hardware threads is accomplished by periodically executing a barrier. Synchronizing threads may be important even when threads are located on different cores. For example, multiple cores conceptually will share a micro-panel of $\widetilde{B}$, which is read into their private L_2 caches. If they access the $\widetilde{B}$ micro-panel at the same time, the micro-panel will be read just once out of L_3 (or memory) and replicated using the cache coherence protocol. However, if cores fall out of synch, a micro-panel of $\widetilde{B}$ may be read from main memory multiple times. This may penalize performance or energy.

For our Xeon Phi experiments, the four threads on a core parallelize the j_r loop, and a barrier is executed every 8 instances of the micro-kernel. However, we do not enforce any synchronization between cores.

6.3.5 Parallelism between cores

As noted, it is particularly advantageous to parallelize the i_c loop between cores as each core has its own L_2 cache. However, if parallelism between cores is only attained by this loop, performance will be poor when m is small. Also, all cores will only work with an integer number of full blocks of $\widetilde{A}$ (where the size of the $\widetilde{A}$ is $m_c \times k_c$) when m is a multiple of 7200. For this reason, we seek to gain parallelism in both the m and n dimensions. Thus, we parallelize the j_r loop in addition to the i_c loop to gain parallelism between cores, even though this incurs the extra cost of the cache-coherency protocol to duplicate all of $\widetilde{A}$ to each L_2 cache.

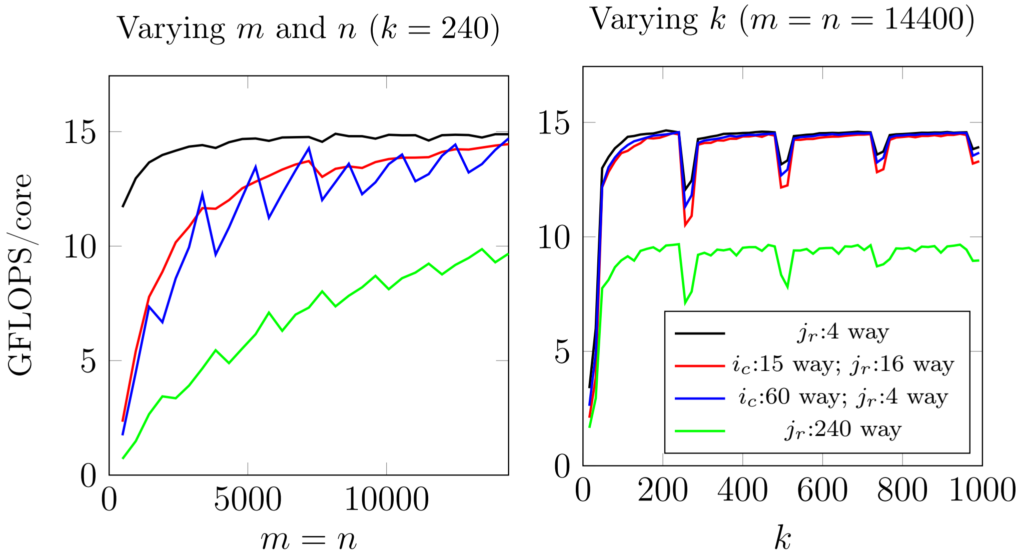


Figure 6.10: MMM performance using different parallelization schemes within BLIS on the Intel Xeon Phi. 'i_c:n way' indicates n-way parallelization of the third loop (indexed by i_c) around the micro-kernel, and 'j_r:n way' indicates n-way parallelization of the second loop (indexed by j_r) around the micro-kernel.

6.3.6 Performance results

Given that (1) each core can issue one floating point multiply-accumulate instruction per clock cycle, and (2) the SIMD vector length for double-precision real elements is 8, each core is capable of executing 16 floating point operations per cycle, where a floating point operation is either a floating point multiply or addition. At 1.1 GHz, this corresponds to a peak of 17.6 GFLOPS per core, or 1056 GFLOPS for 60 cores. In the performance results presented in this paper, the top of each graph represents the theoretical peak of that machine.

Figure 6.10 compares the performance of different parallelization schemes within BLIS on the Xeon Phi. There are four parallelization schemes presented. They are labeled with how much parallelism was gained from the i_c

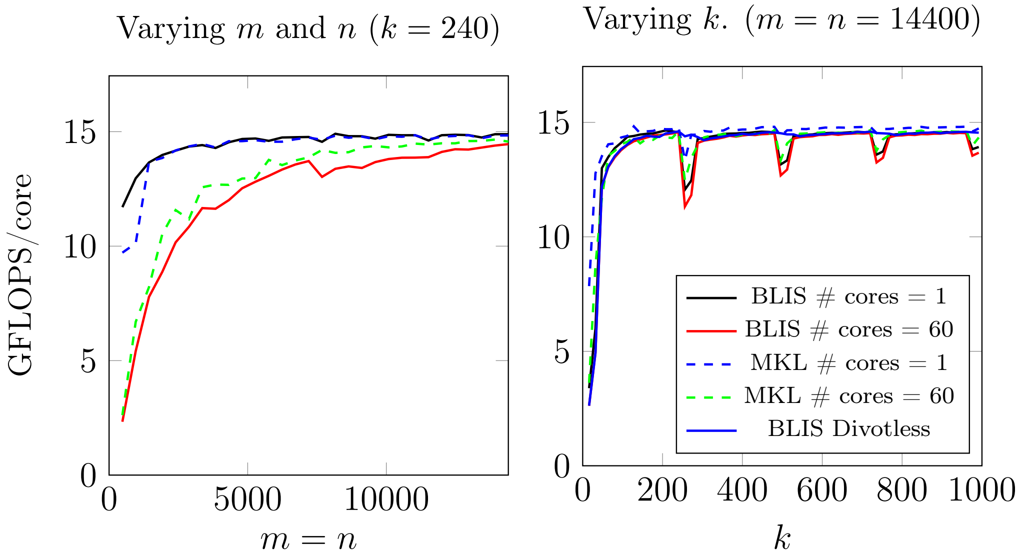


Figure 6.11: MMM performance comparision of BLIS and MKL on the Intel Xeon Phi KNC.

and j_r loops. In all cases, parallelization within a core is done by parallelizing the j_r loop. Single-thread results are not presented, as such results would be meaningless on the Xeon Phi.

The case labeled 'j_r: 240 way', where all parallelism is gained from the j_r loop, yields very poor performance. Even when $n = 14400$, which is the maximum tested (and a rather large problem size), each thread is only multiplying each $\widetilde{A}$ with seven or eight micro-panels of $\widetilde{B}$. In this case, not enough time is spent in computation to amortize the packing of $\widetilde{A}$. Additionally, $\widetilde{A}$ is packed by all threads and then the cache coherency protocol duplicates all micro-panels of $\widetilde{A}$ among the threads (albeit at some cost due to extra memory traffic). Finally, $\widetilde{A}$ is rather small compared to the number of threads, since it is only 240×120. A relatively small block of $\widetilde{A}$ means that there is less opportunity for parallelism in the packing routine. This makes load balancing more difficult, as some threads will finish packing before others and then sit idle.

Next consider the case labeled 'i_c:60 way; j_r:4 way'. This is the case where parallelism between cores is gained from the i_c loop, and it has good performance when m is large. However, load balancing issues arise when m_r multiplied by the number of threads parallelizing the i_c loop does not divide m (that is, when m is not divisible by 1800). This is rooted in the $m_r \times n_r$ micro-kernel's status as the basic unit of computation. Now consider the case labeled 'i_c: 15; j_r:16'. This case ramps up more smoothly, especially when m and n are small.

In Figure 6.11, we compare the best BLIS implementation against Intel's Math Kernel Library (MKL), which is a highly-tuned implementation of the BLAS. We note that MKL's performance has improved since these experiments were originally performed, in 2013. For the top graph, we compare the 'i_c:15 way; j_r:16 way' scheme against MKL when m and n vary. For the bottom graph, we use the 'i_c:60 way; j_r:4 way' scheme, since it performs slightly better when m is large. This case is particularly favorable for this parallelization scheme because each thread is given an integer number of blocks of $\widetilde{A}$. This only happens when m is divisible by 7200.

In the bottom graph of Figure 6.11, when both m and n are fixed to 14400, notice that there are 'divots' that occur when k is very slightly larger than k_c, which is 240 in the BLIS implementation of GEMM, and evidently in the MKL implementation of GEMM as well. When k is just slightly larger than a multiple of 240, an integer number of rank-k updates will be performed with the optimal blocksize k_c, and one rank-k update will be performed with a smaller rank. The rank-k update with a small value of k is expensive because in each micro-kernel call, an $m_r \times n_r$ block of C must be both read from and written to main memory. When k is small, this update of the $m_r \times n_r$ submatrix of C is not amortized by enough computation. It is more efficient to perform a single rank-k update with a k that is larger than the optimal k_c than to perform a rank-k update with the optimal k_c followed by a rank-k update with a very small value of k. This optimization is shown in the curve in Figure 6.11 labeled "BLIS Divotless".

Figure 6.11 shows BLIS attaining very similar performance to that of Intel's highly-tuned MKL, falling short by only one or two percentage points from the achieved performance of the Xeon Phi. We also demonstrate great scaling results when using all 60 cores of the machine. Additionally, we demonstrate that the performance 'divots' that occur in both MKL and BLIS when k is slightly larger than some multiple of 240 can be eliminated.

6.4 MMM Parallelization for IBM PowerPC A/2

We now discuss how BLIS supports high performance and scalability on the IBM Blue Gene/Q PowerPC A2 architecture [42].

6.4.1 Architectural Details

The Blue Gene/Q PowerPC A2 processor has 16 cores available for use by applications. Much like the Intel Xeon Phi, each core is capable of using up to four hardware threads, each with its own register file. The PowerPC A2 supports the QPX instruction set, which supports SIMD vectors of length four for double-precision real elements. QPX allows fused multiply-accumulate instructions, operating on SIMD vectors of length four. This lets the A2 execute 8 flops per cycle.

The 16 cores of BGQ that can be used for GEMM share a single 32 MB L_2 cache. This cache is divided into 2 MB slices. When multiple threads are simultaneously reading from the same slice, there is some contention between the threads. Thus there is a cost to having multiple threads access the same part of $\widetilde{A}$ at the same time. The L_2 cache has a latency of 82 clock cycles and 128 byte cache lines.

Each core has its own L_1 prefetch, L_1 instruction, and L_1 data cache. The L_1 prefetch cache contains the data prefetched by the stream prefetcher and has a capacity of 4 KB [23]. It has a latency of 24 clock cycles and a cache line size of 128 byes. The L_1 data cache has a capacity of 16 KB, and a cache line size of 64 bytes. It has a latency of 6 clock cycles [34].

The PowerPC A2 has two pipelines. The AXU pipeline is used to execute QPX floating point operations. The XU pipeline can be used to execute memory and scalar operations. Each clock cycle, a hardware thread is allowed to dispatch one instruction to one of these pipelines. In order for the A2 to be dispatching a floating point instruction each clock cycle, every instruction must either execute on the XU pipeline, or it must be a floating point instruction. Additionally, since there are inevitably some instructions that are executed on the XU pipeline, we use four hardware threads so that there will usually be an AXU instruction available to dispatch alongside each XU instruction.

6.4.2 The BLIS implementation on the IBM PowerPC A2

As on the Intel Xeon Phi, $\widetilde{A}$ and the sliver of $\widetilde{B}$ reside in the L_2 cache and no data resides in the L_1 cache. (This amount of L_1 cache per thread on the A2 is half that of the Xeon Phi.)

For the BLIS PowerPC A2 implementation, we have chosen m_r and

n_r to both be 8. The block of $\widetilde{A}$ takes up approximately half of the 32 MB L_2 cache, and in the BLIS implementation, m_c is 1024 and k_c is 2048. The PowerPC A2 does not have an L_3 cache and thus n_c is limited by the size of memory; therefore, we have choosen a rather large value of $n_c = 10240$.

6.4.3 Which loop to parallelize

While there are fewer threads to use on the PowerPC A2 than on the Xeon Phi, 64 hardware threads is still enough to require the parallelization of multiple loops. Again, we refer to each loop by the name of its indexing variable.

- The i_r loop: With an m_c of 1024 and m_r of 8, this loop has many iterations. Thus, unlike the Intel Xeon Phi, the first loop around the micro-kernel presents an excellent opportunity for parallelism.

- The j_r loop: Since n_c is large and n_r is only 8, this loop also provides an excellent opportunity for parallelism. However when threads parallelize this loop, they share the same $\widetilde{A}$, and may access the same portions of $\widetilde{A}$ concurrently. This poses problems when it causes too many threads to access the same 2 MB portion of the L_2 cache simultaneously.

- The i_c loop: Since all threads share the same L_2 cache, this loop has similar advantages as the i_r loop. If multiple threads parallelize this loop, $\widetilde{A}$ will have to be reduced in size. This reduction in size reduces the computation that amortizes the movement of each sliver of $\widetilde{B}$ into the L_2 cache. If we reduce m_c, then parallelizing the i_c loop reduces this cost by the same amount as parallelizing i_r.

- The p_c loop: Once again, we do not consider this loop for parallelization.
- The j_c loop: This loop has the same advantages and disadvantages as the j_r loop, except that this loop should not be parallelized among threads that share a core, since they will not then share a block of $\widetilde{A}$.

Since the L_2 cache is shared, and there is no L_3 cache, our choices for the PowerPC A2 is between parallelizing either the i_c or i_r loops, and either the j_c or j_r loops. In both of these cases, we prefer the inner loops to the outer loops. The reason for this is two-fold. Firstly, it is convenient to not change any of the cache blocking sizes from the serial implementation of BLIS when parallelizing. But more importantly, parallelizing the inner loops instead of the outer loops engenders better spatial locality, as there will be one contiguous block of memory, instead of several blocks of memory that may not be contiguous.

6.4.4 Performance results

Like the Xeon Phi, each core of the PowerPC A2 can issue one double-precision fused multiply-accumulate instrucion each clock cycle. The SIMD vector length for double-precision arithmetic is 4, so each core can execute 8 floating point operations per cycle. At 1.6 GHz, a single core has a double-precision peak performance of 12.8 GFLOPS. This becomes the top line in Figures 6.12 and 6.13. The theoretical peak with all 16 cores is 204.8 GFLOPS.

Figure 6.12 compares the performance of different parallelization schemes within BLIS on the PowerPC A2. It is labeled similarly to Figure 6.10, described in the previous section. All parallelization schemes have good performance when m and n are large, but the schemes that only parallelize in

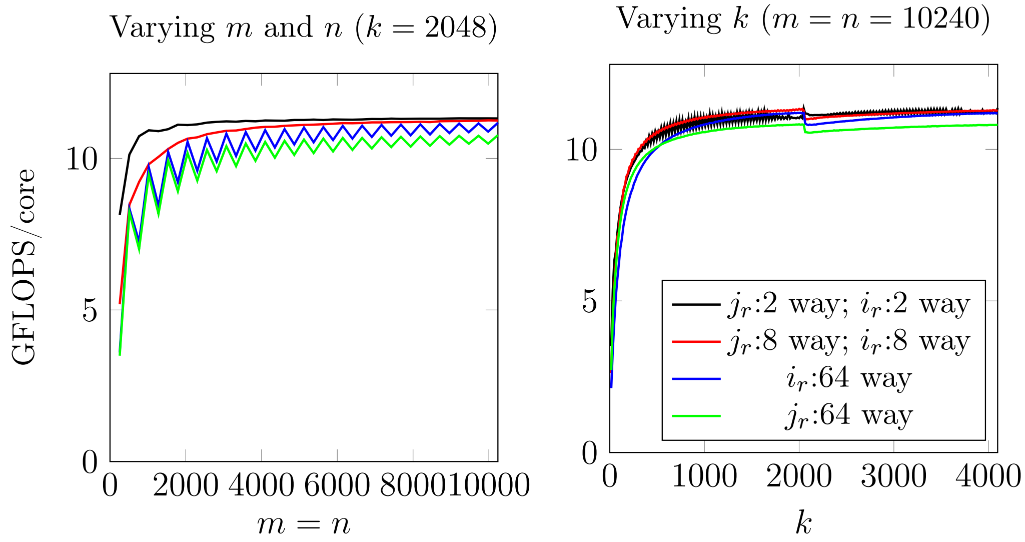


Figure 6.12: MMM with different parallelization schemes for the IBM PowerPC A2. 'j_r:n way' indicates n-way parallelization of the second loop (indexed by j_r) around the micro-kernel, and 'i_r:n way' indicates n-way parallelization of the first loop (indexed by i_r) around the micro-kernel.

either the m or the n dimensions have performance that varies according to the amount of load balancing. Proper load balancing for the 'i_r:64 way' case is only achieved when m is divisible by 512, and similarly, proper load balancing for the 'j_r:64 way' case is only achieved when n is divisible by 512.

The performance of BLIS is compared with that of ESSL in Figure 6.13. The parallelization scheme used for this comparision is the one labeled 'j_r:8 way; i_r:8 way'. Parallel performance scales perfectly to 16 cores for large m and n.

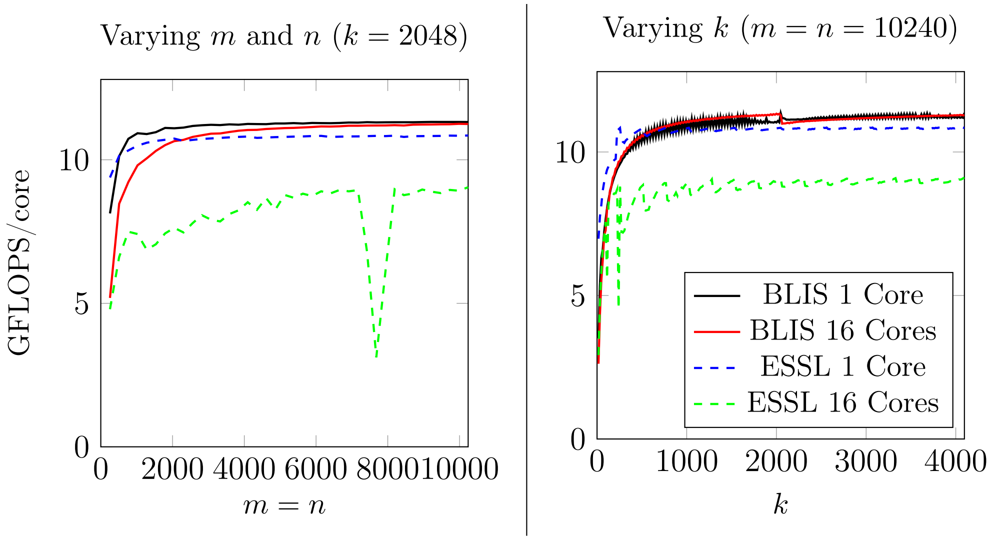


Figure 6.13: MMM performance comparision of BLIS and ESSL on the PowerPC A2 Blue Gene Q.

6.5 Paralleling the family of algorithms

We now turn our attention to how parallelism can be incorporated into the family of algorithms introduced in Chapter 5 in a cache-aware manner. We will start by thinking about the algorithm that is encountered at the L_h cache, we will then consider hazards to I/O costs that can arise during parallelism (as we saw when parallelizing Goto's Algorithm within BLIS).

We will suppose there are N threads that are cooperatively executing the L_h subproblem. Recall from Chapter 5 that the L_h subproblem has two small dimensions and one large, with the small and roughly square matrix residing in the L_h cache and the other two operands are streamed in from lower levels of the memory hierarchy. Then two loops partition the L_h subproblem, exposing the L_{h-1} subproblem.

We will now identify several hazards that arise when parallelizing the

L_h subproblem. Some of these hazards come from different threads working on different data. Threads working on different data can mean that more data must fit into cache, leading to reduced blocksizes and higher I/O costs. Other hazards can come from different threads working on the same data.

6.5.1 Parallelizing outside the L_{h-1} subproblem

We now consider hazards that can arize when parallelizing the L_h subproblem such that each thread works on a distinct L_{h-1} resident block. This happens if either the L_{h-1} inner and outer loops are parallelized. This can be thought of as a more coarse grained way of parallelizing the L_h subproblem.

Shared L_{h-1} cache. Suppose that the N threads share an L_{h-1} cache. Then, the size of the L_{h-1} resident blocks must be adjusted so that N blocks can fit into the L_{h-1} cache. This could increase the I/O cost into the L_{h-1} cache by factor $\sqrt{N}$. This suggests that parallelizing outside of the L_{h-1} subproblem be done only if threads have independent L_{h-1} caches.

This hazard happens for instance within Goto's Algorithm if all threads share an L_2 cache, and they parallelize the third loop around the micro-kernel. Then the size of $\widetilde{A}$ must be modified so that N of them must fit into the L_2 cache.

Exacerbating L_h versus L_{h-1} I/O tradeoffs. Remember that when optimizing for the I/O cost for both the L_h and L_{h-1} caches, tradeoffs occur. In particular, for LRU caches, the panels of the L_h streamed matrices that are exposed by the L_{h-1} outer loop must fit into the L_h cache along with the L_h resident matrix.

If N threads parallelize the L_{h-1} outer loop, and the N threads all share an L_h cache, then N such panels of each of the streamed matrices must fit into the L_h cache. One can parallelize the L_{h-1} inner loop to mitigate this issue, but note that there is generally less parallelism available in the L_{h-1} inner loop.

This hazard happens for instance within Goto's Algorithm if all threads share an L_2 cache, and they parallelize the second loop around the micro-kernel. Then the size of $\widetilde{A}$ and the size of the micro-panels of $\widetilde{B}$ must be modified so that N of the micro-panels must fit into cache along with $\widetilde{A}$. Note that this hazard was not thoroughly explored within the context of Goto's Algorithm in this chapter.

6.5.2 Parallelizing within the L_{h-1} subproblem

We now consider hazards that can arise when parallelizing loops within the L_{h-1} subproblem. In this case, each thread is working on the same L_{h-1} resident block. This can be thought of as a more fine grained way of parallelizing the L_h subproblem.

Unshared L_{h-1} cache. If different threads each have their own L_{h-1} cache, parallelizing inside of the L_{h-1} subproblem can lead to worse amortization of the I/O cost assocated with moving the L_{h-1} resident block into the L_{h-1} cache. In the serial case, the L_{h-1} block is moved into cache, and this data movement is amortized by the computation that happens during the L_{h-1} subproblem. In the parallel case, the L_{h-1} block is moved into multiple caches, and this is amortized over the same amount of computation. This movement of L_{h-1} into multiple caches intstead of a single cache represents a greater I/O cost.

This hazard happens for instance within Goto's Algorithm if each thread has its own L_2 cache, and they parallelize the second loop around the micro-kernel. Then one $\widetilde{A}$ must be moved into multiple caches in order to execute the macro-kernel instead of just into one of those caches.

Increased synchronization costs. Parallelizing at a finer granularity can result in more frequent synchronization, and barriers are expensive.

6.5.3 Summary

As one might have noticed while reading the potential hazards above and how to avoid them, there is no way to avoid all of the hazards. An engineer must carefully weigh their options and decide which loops to parallelize and therefore how much data should be shared between threads.

6.6 Summary

In this chapter, we described the five loops around the BLIS micro-kernel that implement Goto's Algorithm within the BLIS framework. We discussed where, during the execution of the micro-kernel, data resides and used this to motivate insights about opportunities for parallelizing the various loops. We discussed how parallelizing the different loops affects data sharing sharing and amortization of data movement. These insights were then applied to the parallelization of this operation on two architectures that require many threads to achieve peak performance: The IBM Blue Gene/Q and the Intel Xeon Phi. We also showed that parallelizing multiple loops is a key to high performance and scalability.

At the time that these experiments were performed (in 2013), it was a curiosity that on both of these architectures the L_1 cache is too small to support the multiple hardware threads that are required to attain near-peak performance. Now, within the family of algorithms introduced in Chapter 5, we that it is simply not convenient for the L_2 guest matrix to be placed within the L_1 cache for these architectures.

The parallelism of the loops that implement Goto's Algorithm has been extended to all of the level-3 BLAS operations within BLIS. The same lessons apply to those operations, but one must keep in mind that for the operations TRSM and TRMM, parallelism in the m dimension is tricky, as there are dependencies along it.

Chapter 7

Conclusion

In this work, we have identified three algorithms for MMM that when considering a single layer of memory in a hierarchical memory architecture, are optimal in terms of the number of reads from slower layers of the memory hierarchy. Each of these algorithms is associated with a particular shape of MMM. Applying these algorithms at different levels of the memory hierarchy has allowed us to derive practical algorithms for sequential and parallel MMM on architectures with hierarchies with multiple layers.

7.1 Results

In this dissertation several novel contributions have been reported.

Lower bounds for MMM. We have proven new theoretical I/O lower bounds for MMM for machines with a single level of fast memory. According to this lower bound, any classical MMM operation must incur an I/O cost of at least $\frac{2mnk}{\sqrt{S}}$, where S is the size of fast memory. We use the same general strategy for proving these I/O lower bounds as many other papers, but we have made tactical improvements to the lower bound proofs that have allowed us to find the correct coefficient on the leading term of the I/O lower bound. The first of these tactical improvements is that we assumed that computation is performed via FMA instructions, allowing us to account for the cost associated with reading elements of the matrix C from slow memory. The second of these tactical improvements is a generalization of the proof strategy. The commonly used I/O lower bound proof strategy, first introduced in [46], breaks computation down into phases, where each phase has the same I/O cost. Then, a lower bound on the number of phases gives an I/O lower bound. We showed that by allowing the I/O cost of each phase to be a variable, instead of being fixed to the size of fast memory, one can obtain an improved I/O lower bound.

A family of algorithms for MMM. We described and analyzed a previously known algorithm, called Resident C, showing that it attains the I/O lower bound and thus is optimal. This also proves that the I/O lower bounds are tight. We also described to other algorithms, Resident A and B, that are optimal with respect to the number of reads from slow memory. We then derived a family of algorithms for a hierarchical memory architecture with multiple levels of fast memory. By composing two loops per level of cache, one of Resident A, B, and C can be encountered at each level of the memory hierarchy. We analyzed the tradeoffs between the I/O costs of adjacent levels of cache that arise from this family of algorithms, showing that it not always benefitial to optimize for the I/O cost at each level of the memory hierarchy. From this, we expanded the family of algorithms to allow more flexibility in rectifying these tradeoffs, and in the process the state-of-the-art Goto's Algorithm became a part of this family of algorithms. Then, we developed practical algorithms from this family that improved greatly upon the state-of-the-art in terms of I/O cost and that greatly outperform state-of-the-art algorithms when the bandwidth to main memory is low.

Cache-aware loop-based parallelism. We devised a loop-based parallelization scheme for our family of algorithms and applied this scheme to the BLIS implementation of Goto's Algorithm. The key observation is that the parallelization must be aware of how caches are shared by threads, and thus the threads must share or not share data accordingly. We analyzed the properties of the loops within BLIS and the effects of parallelizing each of these loops. We then applied this loop-based parallelism to obtain practical performance and good scalability on the manycore architectures the Intel Xeon Phi Knight's Corner and the IBM Blue Gene/Q.

7.2 Future work

There are several avenues of research suggested by this dissertation.

Lower bounds for other operations. The new techniques introduced by this work can be applied to other operations as well. With these techniques, the lower bound strategy we have used can be used to improve the coefficient on the I/O lower bounds for a wide variety of operations.

New lower bounds proof strategies. While it is true that we have found the best possible coefficient for I/O lower bounds for MMM when it is performed with FMA instructions, we have come across limitations of this proof strategy. In our family of algorithms, there are tradeoffs when optimizing for the I/O cost at adjacent levels of cache. We would like to prove theoretically whether or not such tradeoffs are necessary. We want to obtain I/O lower bounds for MMM when it is not performed with FMAs. We also want to obtain tight lower bounds for DLA operations where one operand appears twice in the operation. For example, the BLAS operation triangular matrix-matrix multiplication is essentially an MMM operation, however because the operation is $B \mathrel{+}= AB$, with B appearing twice, we cannot prove that the coefficient on the leading term is two.

Real-world application of algorithms For our family of algorithms, in order to demonstrate a practical benefit over state-of-the-art algorithms, we artificially lowered the amount of bandwidth to main memory in order to show a benefit for algorithms that better utilize the L_3 and L_4 caches. Identifying a computer architecture where a member of our family of algorithms outperforms Goto's Algorithm (be it by better utilizing the L_1, L_3, or some other cache) remains future work. Then, while we have described algorithms that optimize for the aggregate fast memory of several processors, we have not shown a practical benefit to doing so.

Appendices

Appendix A

Table of Symbols

$\alpha, \beta, \ldots$	Scalar variables
$a, b, \ldots$	Vector variables
$A, B, \ldots$	Matrix variables. A, B, and C are the canonical operands of MMM.
m, n, k	Matrix dimensions. A is always $m \times k$. B is always $k \times n$. C is always $m \times n$.
$L_1, L_2, \ldots$	Caches. L_{h+1} is smaller and faster than L_h
S	Capacity of fast memory in elements

Appendix B

Constrained Global Maximum of $\sqrt{xyz}$

In this appendix, we give details on how the optimal F is determined. The problem to be solved is

$$\text{maximize } F \text{ under the constraints } \left\{ \begin{array}{r} F \leq \sqrt{xyz} \\ 0 \leq x, y, z \\ x + y + z \leq S + M \end{array} \right. .$$

We first observe that if any of x, y, or z is zero, then so is F and hence will only consider the case where $0 < x, y, z$. Also, if $x + y + z$ are strictly less than $S + M$, then one of x, y, or z can be increased, thereby increasing F, and hence we only need to consider $x + y + z = S + M$. Finally, given these constraints we can optimize $F = \sqrt{xyz}$, as long as we check that the result is a maximum. The constrained problem thus becomes

$$\text{maximize } F = \sqrt{xyz} \text{ under the constraints } \left\{ \begin{array}{r} 0 < x, y, z \\ x + y + z = S + M \end{array} \right. .$$

We can use the Lagrange Multiplier method to solve $\nabla F = \lambda \nabla (x + y + z - (S + M))$ for x, y, z. Hence

$$\frac{yz}{2\sqrt{xyz}} = \lambda, \quad \frac{xy}{2\sqrt{xyz}} = \lambda, \quad \frac{xz}{2\sqrt{xyz}} = \lambda, \quad \text{and } S + M = x + y + z.$$

Since then $yz = xy = xz$ and we know that x, y and z are nonzero, we deduce that $x = y = z$ and hence $S + M = 3x$. As a result, the solution is $x = y = z = (S + M)/3$. To show that this is a global maximum, we can find the second derivative of F at this point, or we can evaluate F at this point and any point on the boundary of our region to show that any value on the boundary is smaller.

We conclude that the global maximum of F is:

$$F = \frac{S+M}{3}\sqrt{\frac{S+M}{3}} = \frac{(S+M)\sqrt{S+M}}{3\sqrt{3}}$$

Appendix C

Bandwidth requirements for different datatypes

In this section, we analyze the bandwidth requirement for MMM for different datatypes, where the bandwidth is the rate that reads and writes happen. According to the lower bound, MMM must have an I/O cost of at least $\frac{2mnk}{\sqrt{S}}$. For simplicity, we assume that the MMM must have $\frac{2mnk}{\sqrt{S}}$ elements read, as in the Resident C algorithm. Suppose we wish to compute at a rate of R floating point operations per second. Since MMM requires $2mnk$ floating point operations, achieving this rate of computation means that the MMM must be executed in $\frac{2mnk}{R}$ seconds. This imposes a bandwidth requirement of $\frac{R}{\sqrt{S}}$ elements per second.

Different datatypes have different sizes. To compare the bandwidth requirements of different datatypes, we need to express this bandwidth requirement in terms of bytes per second. Suppose a datatype has W bytes per element. Then the bandwidth requirement for that datatype is $\frac{RW}{\sqrt{S}}$ bytes per second. We must also take into account the fact that S is the capacity of fast memory in terms of the number of elements that it can hold. Suppose fast memory can hold S_B bytes. Then it can hold S_B/W elements, and the bandwidth requirement becomes $\frac{WR}{\sqrt{S_B/W}}$ bytes per second.

With this, we can compare bandwidth costs of different data types. Suppose that R_s, R_d, R_c and R_z are the rate of computation for the single precision, double precision, single precision complex, and double precision

Datatype	Element Size (bytes)	Rate of computation (relative to single)	BW requirement (relative to single)
single	4	1	1
double	8	1/2	$\sqrt{2}$
complex	8	1/4	$\sqrt{2}/2$
double complex	16	1/4	1

Table C.1: Typical element sizes, computation rate, and bandwidth requirements for MMM relative to single precision.

complex datatypes, respectively. When comparing real datatypes, doubling the size of the datatype often means halving the rate of computation. The change to the WR term cancels in this case, and the bandwidth requirement is multiplied by factor $\sqrt{2}$ because the cache can then hold half as many elements. When comparing real and complex datatypes for the same precision, often the size of the datatype doubles and the computation rate decreases by factor four. In this case, the WR term changes by factor $1/2$, and the size of the cache is halved, and the bandwidth requirement is multiplied by factor $\sqrt{2}/2$. Accordingly, we compare bandwidth requirements for single, double, single precision complex, and double precision complex in Table C.1.

Bibliography

[1] R. C. Agarwal, F. Gustavson, and M. Zubair. Exploiting functional parallelism on Power2 to design high-performance numerical algorithms. *IBM Journal of Research and Development*, 38(5), 1994.

[2] R.C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar. A three-dimensional approach to parallel matrix multiplication. *IBM Journal of Research and Development*, 39, 1995.

[3] Alok Aggarwal, Bowen Alpern, Ashok Chandra, and Marc Snir. A model for hierarchical memory. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 305–314. ACM, 1987.

[4] Bowen Alpern, Larry Carter, and Ephraim Feig. Uniform memory hierarchies. In *Foundations of Computer Science, 1990. Proceedings., 31st Annual Symposium on*, pages 600–608. IEEE, 1990.

[5] Edward Anderson, Zhaojun Bai, Christian Bischof, L Susan Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, Alan McKenney, et al. *LAPACK Users' guide.* SIAM, 1999.

[6] Edward Anderson, Zhaojun Bai, Jack Dongarra, Anne Greenbaum, Alan McKenney, Jeremy Du Croz, Sven Hammerling, James Demmel, C Bischof, and Danny Sorensen. LAPACK: A portable linear algebra library for high-performance computers. In *Proceedings of the 1990 ACM/IEEE*

conference on Supercomputing, pages 2–11. IEEE Computer Society Press, 1990.

[7] Greg Baker, John Gunnels, Greg Morrow, Beatrice Riviere, and Robert Van De Geijn. PLAPACK: High performance through high-level abstraction. In *Parallel Processing, 1998. Proceedings. 1998 International Conference on*, pages 414–422. IEEE, 1998.

[8] Grey Ballard, E Carson, J Demmel, M Hoemmen, Nicholas Knight, and Oded Schwartz. Communication lower bounds and optimal algorithms for numerical linear algebra. *Acta Numerica*, 23:1–155, 2014.

[9] Grey Ballard, Nicholas Knight, and Kathryn Rouse. Communication lower bounds for matricized tensor times khatri-rao product. *arXiv preprint arXiv:1708.07401*, 2017.

[10] Rodney J Bartlett and Monika Musiał. Coupled-cluster theory in quantum chemistry. *Reviews of Modern Physics*, 79(1):291, 2007.

[11] Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Jon Hiller, Sherman Karp, Stephen Keckler, Dean Klein, Robert Lucas, Mark Richards, Al Scarpelli, Steven Scott, Allan Snavely, Thomas Sterling, R. Stanley Williams, Katherine Yelick, Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Jon Hiller, Stephen Keckler, Dean Klein, Peter Kogge, R. Stanley Williams, and Katherine Yelick. Exascale computing study: Technology challenges in achieving exascale systems, 2008. Peter Kogge, Editor and Study Lead.

[12] Paolo Bientinesi, John A Gunnels, Margaret E Myers, Enrique S Quintana-Ortí, and Robert A Geijn. The science of deriving dense linear algebra algorithms. *ACM Transactions on Mathematical Software (TOMS)*, 31(1):1–26, 2005.

[13] Paolo Bientinesi, John A. Gunnels, Margaret E. Myers, Enrique S. Quintana-Ortí, Tyler Rhodes, Robert A. Geijn, and Field G. Van Zee. Deriving dense linear algebra libraries. *Formal Aspects of Computing*, 25(6):933–945, 2012.

[14] Jeff Bilmes, Krste Asanović, Cheewhye 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.

[15] Jeff Bilmes, Krste Asanović, Chee whye Chin, and Jim Demmel. The PHiPACv1.0 matrix-multiply distribution. Technical Report 98-35, Int'l Computer Science Institute, October 1998.

[16] George Bosilca, Aurelien Bouteiller, Franck Cappello, Samir Djilali, Gilles Fedak, Cecile Germain, Thomas Herault, Pierre Lemarinier, Oleg Lodygensky, Frederic Magniette, et al. MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes. In *Supercomputing, ACM/IEEE 2002 Conference*, pages 29–29. IEEE, 2002.

[17] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1):38–53, 2009.

[18] Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert Van De Geijn. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience*, 19(13):1749–1783, 2007.

[19] Ernie Chan, Enrique S Quintana-Ortí, Gregorio Quintana-Ortí, and Robert Van De Geijn. Supermatrix out-of-order scheduling of matrix operations for smp and multi-core architectures. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 116–125. ACM, 2007.

[20] Fannie Chen, Loring Craymer, Jeff Deifik, Alvin J Fogel, Daniel S Katz, Alfred G Silliman Jr, Raphael R Some, Sean Upchurch, Keith Whisnant, et al. Demonstration of the remote exploration and experimentation (ree) fault-tolerant parallel-processing supercomputer for spacecraft onboard scientific data processing. In *Dependable Systems and Networks, 2000. DSN 2000. Proceedings International Conference on*, pages 367–372. IEEE, 2000.

[21] Zizhong Chen and Jack Dongarra. Algorithm-based fault tolerance for fail-stop failures. *Parallel and Distributed Systems, IEEE Transactions on*, 19(12):1628–1641, 2008.

[22] Jaeyoung Choi, Jack J Dongarra, Roldan Pozo, and David W Walker. ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. In *Frontiers of Massively Parallel Computation, 1992., Fourth Symposium on the*, pages 120–127. IEEE, 1992.

[23] I-Hsin Chung, Changhoan Kim, Hui-Fang Wen, and Guojing Cong. Application data prefetching on the IBM Blue Gene/Q supercomputer. In

Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12, pages 88:1–88:8, Los Alamitos, CA, USA, 2012.

[24] Jack Dongarra, Jean-François Pineau, Yves Robert, Zhiao Shi, and Frédéric Vivien. Revisiting matrix product on master-worker platforms. *International Journal of Foundations of Computer Science*, 19(06):1317–1336, 2008.

[25] Jack J Dongarra, James R Bunch, Cleve B Moler, and Gilbert W Stewart. *LINPACK users' guide*. Siam, 1979.

[26] Jack J Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)*, 16(1):1–17, 1990.

[27] 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, 1988.

[28] Graham E Fagg and Jack J Dongarra. FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. In *Recent advances in parallel virtual machine and message passing interface*, pages 346–353. Springer, 2000.

[29] Matteo Frigo, Charles E Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 285–297. IEEE, 1999.

[30] Kyle Gallivan, William Jalby, and Ulrike Meier. The use of BLAS3 in linear algebra on a parallel processor with a hierarchical memory. *SIAM Journal on Scientific and Statistical Computing*, 8(6):1079–1084, 1987.

[31] Kyle Gallivan, William Jalby, Ulrike Meier, and Ahmed H Sameh. Impact of hierarchical memory systems on linear algebra algorithm design. *International Journal of High Performance Computing Applications*, 2(1):12–48, 1988.

[32] Marc Gamell, Daniel S Katz, Hemanth Kolla, Jacqueline Chen, Scott Klasky, and Manish Parashar. Exploring automatic, online failure recovery for scientific applications at extreme scales. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 895–906. IEEE Press, 2014.

[33] Burton S Garbow. EISPACK – A package of matrix eigensystem routines. *Computer Physics Communications*, 7(4):179–184, 1974.

[34] Megan Gilge. *IBM system Blue Gene solution: Blue Gene/Q application development*. IBM, June 2013.

[35] Kazushige Goto and Robert van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software*, 34(3), May 2008.

[36] Kazushige Goto and Robert Van De Geijn. High-performance implementation of the level-3 blas. *ACM Transactions on Mathematical Software (TOMS)*, 35(1):4, 2008.

[37] J.A. Gunnels, D.S. Katz, E.S. Quintana-Ortí, and R.A. van de Geijn. Fault-tolerant high-performance matrix multiplication: theory and practice. In *DSN 2001*, 2001.

[38] John A. Gunnels. A systematic approach to the design and analysis of linear algebra algorithms. Ph.D. dissertation. FLAME Working Note #6. Technical Report TR-2001-44, The University of Texas at Austin, Department of Computer Sciences, November 2001.

[39] John A Gunnels, Fred G Gustavson, Greg M Henry, and Robert A Van De Geijn. FLAME: Formal linear algebra methods environment. *ACM Transactions on Mathematical Software (TOMS)*, 27(4):422–455, 2001.

[40] John A. Gunnels, Greg M. Henry, and Robert A. van de Geijn. A family of high-performance matrix multiplication algorithms. In *ICCS '01*, 2001.

[41] Fred G Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM Journal of Research and Development*, 41(6):737–755, 1997.

[42] Ruud A. Haring, Martin Ohmacht, Thomas W. Fox, Michael K. Gschwind, David L. Satterfield, Krishnan Sugavanam, Paul W. Coteus, Philip Heidelberger, Matthias A. Blumrich, Robert W. Wisniewski, Alan Gara, George L.-T. Chiu, Peter A. Boyle, Norman H. Chist, and Changhoan Kim. The IBM Blue Gene/Q compute chip. *Micro, IEEE*, 32(2):48 –60, March-April 2012.

[43] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. Libxsmm: accelerating small matrix multiplications by runtime code generation. In *Proceedings of the International Conference for High Per-*

formance Computing, Networking, Storage and Analysis, page 84. IEEE Press, 2016.

[44] Alexander Heinecke, Karthikeyan Vaidyanathan, Mikhail Smelyanskiy, Alexander Kobotov, Roman Dubtsov, Greg Henry, Aniruddha G. Shet, George Chrysos, and Pradeep Dubey. Design and implementation of the Linpack benchmark for single and multi-node systems based on Intel(r) Xeon Phi(tm) coprocessor. In *27th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2013)*, 2013.

[45] Greg Henry. BLAS based on block data structures. Technical report, Cornell University, 1992.

[46] Jia-Wei Hong and Hsiang-Tsung Kung. I/O complexity: The red-blue pebble game. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 326–333. ACM, 1981.

[47] Kuang-Hua Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Trans. Comput.*, 33(6), 1984.

[48] Intel. *Intel Xeon Phi Coprocessor System Software Developers Guide*, June 2013.

[49] 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.

[50] Bo Kågström, Per Ling, and Charles Van Loan. GEMM-based level 3 BLAS: High performance model implementations and performance evaluation benchmark. *ACM Transactions on Mathematical Software*, 24(3):268–302, 1998.

[51] HT Kung. Memory requirements for balanced computer architectures. In *ACM SIGARCH Computer Architecture News*, volume 14, pages 49–54. IEEE Computer Society Press, 1986.

[52] Monica S Lam, Edward E Rothberg, and Michael E Wolf. The cache performance and optimizations of blocked algorithms. In *ACM SIGARCH Computer Architecture News*, volume 19, pages 63–74. ACM, 1991.

[53] Chuck L Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323, 1979.

[54] Jin Li, Anthony Skjellum, and Robert D Falgout. A poly-algorithm for parallel dense matrix multiplication on two- dimensional process grid topologies. Master's thesis, Mississippi State University. Department of Computer Science., 1996.

[55] Lynn H. Loomis and Hassler Whitney. An inequality related to the isoperimetric inequality. *Bulletin of the American Mathematical Society*, 55:961–962, 1949.

[56] Tze Meng Low, Francisco D. Igual, Tyler M. Smith, and Enrique S. Quintana-Ortí. Analytical modeling is enough for high performance BLIS. Technical report, Technical report, Department of Computer Sciences, The University of Texas at Austin. Manuscript in progress, 2014.

[57] Tze Meng Low, Francisco D Igual, Tyler M Smith, and Enrique S Quintana-Ortí. Analytical modeling is enough for high-performance BLIS. *ACM Transactions on Mathematical Software*, 43(2):12, 2016.

[58] Bryan A. Marker, Field G. Van Zee, Kazushige Goto, Gregorio Quintana-Ortí, and Robert A. van de Geijn. Toward scalable matrix multiply on multithreaded architectures. In *European Conference on Parallel and Distributed Computing*, pages 748–757, February 2007.

[59] Nicholas D Matsakis and Felix S Klock II. The rust language. In *ACM SIGAda Ada Letters*, volume 34, pages 103–104. ACM, 2014.

[60] Devin A Matthews. High-performance tensor contraction without blas. *arXiv preprint arXiv:1607.00291*, 2016.

[61] Devin A Matthews and John F Stanton. Non-orthogonal spin-adaptation of coupled cluster methods: A new implementation of methods including quadruple excitations. *The Journal of chemical physics*, 142(6):064108, 2015.

[62] Elmar Peise and Paolo Bientinesi. Recursive algorithms for dense linear algebra: The relapack collection. *arXiv preprint arXiv:1602.06763*, 2016.

[63] 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 Transactions on Mathematical Software (TOMS)*, 39(2):13, 2013.

[64] John E Savage. Extending the hong-kung model to memory hierarchies. In *Computing and Combinatorics*, pages 270–281. Springer, 1995.

[65] Martin D Schatz, Jack Poulson, and Robert A van de Geijn. Scalable universal matrix multiplication algorithms: 2d and 3d variations on a theme. *submitted to ACM Transactions on Mathematical Software*, pages 1–30, 2012.

[66] John Shalf, Sudip Dosanjh, and John Morrison. Exascale computing technology challenges. In *International Conference on High Performance Computing for Computational Science*, pages 1–25. Springer, 2010.

[67] Tyler M. Smith, Robert 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 & Distributed Processing Symposium (IPDPS 2014)*, pages 1049–1059, 2014.

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

[69] Tyler M Smith, Robert A van de Geijn, Mikhail Smelyanskiy, and Enrique S Quintana-Ortí. Towards ABFT for BLIS GEMM. Technical Report TR-15-05, The University of Texas at Austin., 2015.

[70] Edgar Solomonik and James Demmel. Communication-optimal parallel 2.5 d matrix multiplication and lu factorization algorithms. In *European Conference on Parallel Processing*, pages 90–109. Springer, 2011.

[71] Edgar Solomonik and James Demmel. Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms. In *Proceedings of the 17th international conference on Parallel processing - Volume Part II*, Euro-Par'11, Berlin, Heidelberg, 2011. Springer-Verlag.

[72] Michael Turmon, Robert Granat, and Daniel S. Katz. Software-implemented fault detection for high-performance space applications. In *DSN 2000*, 2000.

[73] Michael Turmon, Robert Granat, and Daniel S Katz. Software-implemented fault detection for high-performance space applications. In *Dependable*

Systems and Networks, 2000. DSN 2000. Proceedings International Conference on, pages 107–116. IEEE, 2000.

[74] Michael Turmon, Robert Granat, Daniel S Katz, and John Z Lou. Tests and tolerances for high-performance software-implemehted fault detection. *Computers, IEEE Transactions on*, 52(5):579–591, 2003.

[75] Robert A. van de Geijn and Enrique S. Quintana-Ortí. *The Science of Programming Matrix Computations*. `lulu.com`, 2008.

[76] Field Van Zee, Ernie Chan, Robert van de Geijn, Enrique Quintana, and Gregorio Quintana-Orti. Introducing: The libflame library for dense matrix computations. *Computing in science & engineering*, 2009.

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

[78] Field G Van Zee and Tyler M Smith. Inducing complex matrix multiplication via the 3M and 4M methods FLAME Working Note# 81. 2016. Submitted to ACM Trans. Math. Softw.

[79] Field G. Van Zee, Tyler M. Smith, Bryan Marker Bryan, Tze Meng Low, Robert van de Geijn, Francisco D. Igual, Mikhail Smelyanskiy, Xianyi Zhang, Michael Kistler, Vernon Austel, John Gunnels, and Lee Killough. The BLIS framework: Experiments in portability. *ACM Transactions on Mathematical Software*, 42(2):12, 2016.

[80] Field G. Van Zee and Robert A. van de Geijn. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Transactions on Mathematical Software*, 41(3), 2015.

[81] Qian Wang, Xianyi Zhang, Yunquan Zhang, and Qing Yi. Augem: automatically generate high performance dense linear algebra kernels on x86 cpus. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 25. ACM, 2013.

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

[83] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[84] Panruo Wu and Zizhong Chen. Ft-scalapack: Correcting soft errors online for scalapack Cholesky, QR, and LU factorization routines. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 49–60. ACM, 2014.

[85] Panruo Wu, Chong Ding, Longxiang Chen, Feng Gao, Teresa Davies, Christer Karlsson, and Zizhong Chen. Fault tolerant matrix-matrix multiplication: correcting soft errors on-line. In *Proceedings of the second workshop on Scalable algorithms for large-scale systems*, pages 25–28. ACM, 2011.

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

[87] 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.

[88] Field G Van Zee, Paolo Bientinesi, Tze Meng Low, and Robert A Geijn. Scalable parallelization of flame code via the workqueuing model. *ACM Transactions on Mathematical Software (TOMS)*, 34(2):10, 2008.

Vita

Tyler Michael Smith was born in Silvis, Illinois on 12 April 1989, the son of Thomas Michael Smith and Sherri Lee Smith. He received Bachelor of Science degrees in Computer Science, Mathematics, and Statistics from Purdue University in May 2011. He began his graduate studies at the University of Texas at Austin in September 2011.

Permanent address: 507 W 33rd St
Austin, Texas 78705

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.