8-2-2003

High-Performance Matrix Multiplication: Hierarchical Data Structures, Optimized Kernel Routines, and Qualitative Performance Modeling

Wenhao Wu

HIGH-PERFORMANCE MATRIX MULTIPLICATION: HIERARCHICAL DATA STRUCTURES, OPTIMIZED KERNEL ROUTINES, AND QUALITATIVE PERFORMANCE MODELING

By

Wenhao Wu

A Thesis
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Computer Science
in the Department of Computer Science and Engineering

Mississippi State, Mississippi

August 2003

HIGH-PERFORMANCE MATRIX MULTIPLICATION: HIERARCHICAL DATA STRUCTURES, OPTIMIZED KERNEL ROUTINES, AND QUALITATIVE PERFORMANCE MODELING

By

Wenhao Wu

Approved:

Anthony Skjellum Associate Professor of Computer Science and Engineering (Major Professor)	Edward A. Luke Assistant Professor of Computer Science and Engineering (Committee Member)
Donna S. Reese Associate Professor of Computer Science and Engineering (Committee Member)	Susan M. Bridges Professor and Graduate Coordinator of the Department of Computer Science and Engineering

A. Wayne Bennett
Dean of the College of Engineering

Name: Wenhao Wu

Date of Degree: August 2, 2003

Institution: Mississippi State University

Major Field: Computer Science

Major Professor: Dr. Anthony Skjellum

Title of Study: HIGH-PERFORMANCE MATRIX MULTIPLICATION: HIERARCHICAL DATA STRUCTURES, OPTIMIZED KERNEL ROUTINES, AND QUALITATIVE PERFORMANCE MODELING

Pages in Study: 69

Candidate for Degree of Master of Science

The optimal implementation of matrix multiplication on modern computer architectures is of great importance for scientific and engineering applications. However, achieving the optimal performance for matrix multiplication has been continuously challenged both by the ever-widening performance gap between the processor and memory hierarchy and the introduction of new architectural features in modern architectures. The conventional way of dealing with these challenges benefits significantly from the blocking algorithm, which improves the data locality in the cache memory, and from the highly tuned inner kernel routines, which in turn exploit the architectural aspects on the specific processor to deliver near peak performance. A state-of-art improvement of the blocking algorithm is the self-tuning approach that utilizes “heroic” combinatorial optimization of parameters spaces. Other recent research approaches include the approach that explicitly

blocks for the TLB (Translation Lookaside Buffer) and the hierarchical formulation that employs memory-friendly Morton Ordering (a space-filling curve methodology).

This thesis compares and contrasts the TLB-blocking-based and Morton-Order-based methods for dense matrix multiplication, and offers a qualitative model to explain the performance behavior. Comparisons to the performance of self-tuning library and the "vendor" library are also offered for the Alpha architecture. The practical benchmark experiments demonstrate that neither conventional blocking-based implementations nor the self-tuning libraries are optimal to achieve consistent high performance in dense matrix multiplication of relatively large square matrix size. Instead, architectural constraints and issues evidently restrict the critical path and options available for optimal performance, so that the relatively simple strategy and framework presented in this study offers higher and flatter overall performance. Interestingly, maximal inner kernel efficiency is not a guarantee of global minimal multiplication time. Also, efficient and flat performance is possible at all problem sizes that fit in main memory, rather than "jagged" performance curves often observed in blocking and self-tuned blocking libraries.

DEDICATION

To my parents.

ACKNOWLEDGMENTS

The author would like to express his sincere thanks to his major professor, Dr. Anthony Skjellum, for his invaluable support and guidance for this research work. Thanks are also due to his committee members Dr. Donna Reese and Dr. Edward Luke. The author would also like to thank for his collaborators, including Mr. Kazushige Goto, Mr. Vinod Valsalam, and Dr. Robert A. Van de Geijn at University of Texas at Austin. The author also owes special thanks to the colleagues in High Performance Computing Laboratory lab and MPI Software Technology, Inc for making both places enjoyable and exciting research environments, and MPI Software Technology, Inc and Hewlett-Packard for kindly providing the testing facilities. Finally, the author would like to thank Ms. Neli Fairfield and Mr. Brian Chase for helping him with the production of this document.

TABLE OF CONTENTS

Page

DEDICATION ii

ACKNOWLEDGMENTS iii

LIST OF TABLES vi

LIST OF FIGURES vii

CHAPTER

I. INTRODUCTION 1

1.1 Background 1
1.2 Hypothesis 2
1.3 Motivation 3
1.4 Contributions 4
1.5 Organization 5

II. LITERATURE REVIEW 6

2.1 Matrix Multiplication and Linear Algebra Libraries 7
2.1.1 Basic Concepts and Algorithms 7
2.1.2 Matrix Multiplication in the Linear Algebra Libraries 9
2.2 Architecture Study 9
2.2.1 Processor Microarchitecture and The Peak Performance 10
2.2.2 Memory Hierarchy and Delivered Performance 11
2.3 Previous Matrix Multiplication Work 13
2.3.1 Early Stage of the Blocking Algorithm 14
2.3.2 Self-Tuning Libraries 15
2.3.3 Cache-Oblivious Algorithm 17
2.3.4 Hierarchical Data Structures 18
2.3.5 Variants of New Blocking Approach 19
2.4 Lessons learned 20

CHAPTER Page

III. RESEARCH APPROACH 21

3.1 TLB-Obvious Architectural Modeling 21
3.1.1 Review of Memory Access Process 21
3.1.2 Modeling of Blocking Algorithm 23
3.1.3 Blocking for TLB 26
3.2 The Matrix Multiplication Framework 27
3.2.1 Hierarchical Data Structure 28
3.2.2 Iterative Algorithm 30
3.2.3 Optimized Kernel Strategy for Alpha 32

IV. PERFORMANCE INTEGRATION, TUNING, AND BENCHMARKING . . 37

4.1 Integration and Tuning Strategy 37
4.2 User Level Performance Metrics 39
4.2.1 In-Cache Performance 40
4.2.2 Out-of-Cache Performance of Matrix Multiplication 41

V. EXPERIMENTS, RESULTS, AND ANALYSIS 44

5.1 Notations and Experimental Configuration 44
5.2 Performance of Inner Kernels 47
5.2.1 Peak Kernel vs Aggregate Kernel 48
5.2.2 HPCL vs ATLAS 48
5.3 Performance of matrix multiplication 51
5.3.1 Explicit TLB blocking vs Implicit Memory Optimization 51
5.3.2 Delivered Performance 54

VI. CONCLUSIONS 60

6.1 Summary 60
6.2 Future Work 62

REFERENCES 63

APPENDIX

GLOSSARY 67

LIST OF TABLES

TABLE Page

2.1 The Characteristics of x86 Processor Microarchitecture 12

2.2 Memory Hierarchy Parameters of Current x86 Architecture 14

4.1 User Level Performance Metrics 40

5.1 Notations of Different Methods 45

5.2 Experimental Configurations (Adapted from [20]) 46

5.3 Degree of Flatness and Aggregate MFLOPS 54

LIST OF FIGURES

FIGURE Page

2.1 An Example of Matrix Multiplication 7

2.2 Pseudocode for Standard ijk Algorithm 8

2.3 The Pyramid of Memory Hierarchy 13

3.1 The Diagram of Memory Access Process 22

3.2 Pseudocode for the jki Blocking Algorithm 25

3.3 A Hierarchically Stored $10 * 18$ Matrix (Adopted from [43]) 29

3.4 Pseudocode for the ijk iterative algorithm (Adapted from [43]) 31

3.5 Sample Assembly Code I of Software Pipelining (Adopted from [8]) 35

3.6 Sample Assembly Code II of Software Pipelining (Adopted from [8]) 35

4.1 Performance Flatness Regarding to The Blocking Size 39

5.1 Performance of Peak Kernel and Aggregate Kernel 49

5.2 The In-Cache Performance of HPCL and ATLAS 50

5.3 The Out-of-Cache Performance of GOTO and HPCL 52

5.4 Performance of Various Approaches on EV67-Tru64 56

5.5 Power of 2 Performance of Various Approaches on EV67-Tru64 57

5.6 Performance of Various Approaches on EV6-Linux 58

5.7 Power of 2 Performance of Various Approaches on EV6-Linux 59

CHAPTER I

INTRODUCTION

1.1 Background

The optimal implementation of matrix multiplication on modern processors is of great importance for many scientific and engineering applications [26, 27, 45]. The primary reason for such importance is that, given a high-performance implementation of matrix multiplication, most dense linear algebra operations can be efficiently implemented on various platforms [22, 36]. In turn, linear algebra is central to many aspects of scientific computing. Another major reason for such importance is that matrix multiplication inherently has good data locality and algorithm features to be exploited to obtain the peak performance on these advanced processors [26, 27].

Achieving the optimal performance for matrix multiplication, however, has been continuously challenged both by the ever-widening performance gap between the processor and memory hierarchy and the introduction of new architectural features [32] in modern architectures. The conventional way of dealing with these challenges benefits significantly from the blocking algorithm [25, 38, 41, 48] and highly optimized inner kernels [30, 46]. By carefully dividing the matrix into the submatrices that can fit into the top level of the memory hierarchy, the blocking algorithm helps with bridging the performance gap be-

tween the processor and the memory hierarchy. On such submatrices, the inner kernel routines perform matrix-multiplication-type operations that exploit the specific architectural features of the processor. Such optimizations at the kernel level are often coded at the assembly level because of the demand for optimal performance and the inability of current compiler technology to deliver peak performance on the triply nested loops [29, 46].

In this thesis, several recent approaches in this field are studied and unified for pursuing the high-performance implementation of matrix multiplication.

1.2 Hypothesis

The hypothesis of this thesis is that strategies that are refined and synthesized from conventional methods and recent approaches offer better overall matrix multiplication performance than current state-of-art implementations. In this context, strategies refer to the ways that activities can be done and be exploited in parallel within the processor and the memory hierarchy. In particular, by carefully overlapping and sequencing data movement and computational activities, this study achieves flatter and higher performance than the vendor libraries and the libraries that are driven by the self-tuning methodology.

The following background introduction and quantitative descriptions of some important terms further accomplish this hypothesis:

Conventional methods: These refer to the well-known blocking type approaches [25, 38, 41, 48].

Recent approaches: These refer to the recent variants of blocking type approaches [28, 30] and the recent research on hierarchical data structures [43, 47].

Flatter performance: The degree of flatness of approach A's performance, $F(A)$, is defined in Section 4.2.2 over a certain range of matrix size. The performance of approach A is flatter than performance of approach B if and only if $F(A) < F(B)$ on this range of matrix size.

Higher performance: Over a certain range of matrix size, the performance $M(A)$ (measured in MFLOPS[1]) of approach A is higher than B if and only if $\sum[M(A) - M(B)] > 0$. A detailed derivation of this formula is also given in Section 4.2.2.

1.3 Motivation

The motivation of this thesis arises from the importance of high-performance implementation of matrix multiplication, the challenges posed by the complexity of modern architectures (especially cache architectures), and a few observations in previous experiments with different matrix multiplication implementations:

- The same number of cache misses in two competitive algorithms does not necessarily degrade the overall matrix multiplication performance equally;
- The maximal matrix multiplication kernel efficiency is not a guarantee of global minimal multiplication time; and,
- Current architecture models do not incorporate certain important architectural features keyed to achieving the peak performance, such as the Translation Lookaside Buffer (TLB), data prefetching techniques, and out-of-order execution units.

[1]Million floating-point operations per second

This study works to assimilate previous knowledge and aims to balance the interactions of kernel computations and memory accesses in the global picture, which play important roles for the delivered matrix multiplication performance.

1.4 Contributions

The contributions of this thesis include validation of the following corollaries related to the hypothesis:

Self tuning vs Global tuning

The self-tuning approach does not guarantee optimal overall performance because the balancing of kernel efficiency and memory access patterns must be done globally.

Hierarchical Data Structures vs TLB-Blocking Algorithm

For the relatively large matrix size, the approaches that use Morton Ordering as an intermediate format for native row majored or column majored matrices is not worthwhile compared to the TLB-blocking approach that exploits the intermediate storage buffer and does incremental copying that are explicitly designed for TLB optimization.

Performance Critical Path

TLB misses, inter-cache peak bandwidth, and register file size evidently reduce the number of choices available to achieve maximum performance, so that relatively simple strategies offer better overall performance compared to “heroic” combinatorial optimization of parameters spaces.

1.5 Organization

The organization of this thesis is as follows: Chapter II summarizes some background and related work in this field. The research methodology of this study is presented in Chapter III, followed by the introduction of performance metrics of this thesis and their measurements in Chapter IV. Chapter V discusses the experimental setup in detail, presents the measured results of the interested performance metrics, and gives a performance analysis to validate the hypothesis. Chapter VI concludes the thesis and suggests future work. A glossary that lists terms associated with this thesis work is given in the appendix.

CHAPTER II

LITERATURE REVIEW

Matrix multiplication is a fairly simple operation from a mathematical point of view and is also a fairly computationally rich operation, which makes it arguably the most important operation of most linear algebra libraries on high performance architectures [14, 22, 26, 27, 45]. Section 2.1 offers a brief introduction concerning matrix multiplication and its unique importance in linear algebra libraries.

Nevertheless, the optimal implementation of matrix multiplication on computer architectures, especially cache architectures, is not easily achieved because of the inherent complexity of these architectures. It is recognized that to be able to achieve the peak performance from software, the algorithm designer must have a good understanding of the underlying architecture, that is, what the different components of a computer system are, how these components work and interact with each other, and most importantly, how they determine the delivered performance of a computer system [12, 32]. An architecture study of cache architectures is consequently necessary and is given in Section 2.2.

There have been decades of research efforts devoted to the optimal implementation of matrix multiplication. Section 2.3 presents several conventional approaches and recent

approaches in this field and discusses the advantages and disadvantages of each approach wherever appropriate.

2.1 Matrix Multiplication and Linear Algebra Libraries

2.1.1 Basic Concepts and Algorithms

Generally, the matrix multiplication operation in scientific computing is in the form of $C \longleftarrow \alpha AB + \beta C$, where α and β are constants, A is a $M \times K$ matrix, B is a $K \times N$ matrix and C is a $M \times N$ matrix. Dense matricies are assumed throughout this thesis. Row or column major storage is assumed for matrices A, B, and C. Matrices A and B may also be transposed. Figure 2.1 shows an example of such matrix multiplication when $\alpha = 1$, $\beta = 0$, $M = 2$, $N = 2$ and $K = 2$.

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \times \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{pmatrix}$$

Figure 2.1 An Example of Matrix Multiplication

Strassen's algorithm is not considered in this thesis because of its extra memory requirements and poor data locality [27], although it has a lower arithmetic complexity and good performance is obtainable upon careful implementations [43]. Instead, this thesis

chooses the standard ijk algorithm representation for matrix multiplication, which is basically a triply nested loop as shown in Figure 2.2.

```
for(i = 0; i < M; i++) {
   for(j = 0; j < N; j++) {
      for(k = 0; k < K; k++) {
         C[i,j] += A[i,k] * B[k,j];
}  }  }
```

Figure 2.2 Pseudocode for Standard ijk Algorithm

Other formats of matrix multiplication update, such as Saxpy formulation, are available but are not good from the computational point of view [27] and thus not chosen for representing the matrix multiplication in the algorithm level. Another reason for choosing this ijk representation is to better reflect the changes in the algorithm level when adapting the block representation of matrix multiplication later.

The number of floating-point operations (flops) of matrix multiplication is in the order of $2MNK$, and $MN+NK+MK$ matrix entries are involved. This means that asymptotically there are $\Theta(S^3)$ floating-point arithmetic operations performed on $\Theta(S^2)$ memory operands for square matrix multiplication with matrix size S. The good data locality and high F:M ratio (the number of floating-point operations per memory operation) [45] pro-

motes matrix multiplication to be the preferred operation on high performance computer architectures [27, 45].

2.1.2 Matrix Multiplication in the Linear Algebra Libraries

The optimal implementation of linear algebra libraries goes back to the mid-1970s' LINPACK library [21] and level-1 Basic Linear Algebra Subprograms (BLAS) [39] on vector supercomputers. Later, the level-2 BLAS [23] were also introduced to obtain higher performance than level-1 BLAS on vector supercomputers. With the advent of cache memory hierarchy system in the late 1980s, level-3 BLAS [22] and LAPACK [13, 14] were standardized to make full use of this advanced memory system.

It is shown by these efforts that the high performance of matrix multiplication can be transformed to the high performance of most linear algebra operations on cache architectures [36]. Also, the computational richness of matrix multiplication provides many opportunities to achieve near peak performance on cache architectures.

2.2 Architecture Study

While other platforms, like vector computers [5] and VLIW (Very Long Instruction Word) machines [29], are available and active in some specific applications, the RISC/CISC (Reduce Instruction Set Computers and Complex Instruction Set Computers) families are dominant in the current high performance architectures [10]. Also, current advanced CISC processors (*e.g.*, Intel Pentium 4 and AMD Athlon) have copied the best attributes of RISC

processors, that is, employed a RISC-like strategy in their microcode design and implementation, to obtain the same level of performance as RISC machines [11, 26, 34].

Based on all these facts, the architectures of interest in this study are the modern RISC/CISC processors with a deep memory hierarchy, which encapsulate the interesting architectural features that determine the delivered performance of matrix multiplication. A good example is the Compaq Alpha platform [19], which is also the test bed of this thesis. Current x86 architectures, however, will be used as the reference architectures in this architecture study as there are more literature information available for this processor family.

2.2.1 Processor Microarchitecture and The Peak Performance

The modern processor's microarchitecture is designed to deliver the highest on-chip parallelism [32, 37]. Some important architectural features keyed to the performance of matrix multiplication are summarized below:

Superscalar pipelining: By paralleling multiple pipelines with the help of hardware and software, most modern processors can execute a combination of several operations in one cycle. For example, on an AMD Athlon processor, up to nine OPs (the RISC-type instructions that are decoded from x86 instructions) can be simultaneously issued to the three-way integer execution pipelines, three-way address generation pipelines and three-way floating-point execution pipelines [11].

Out-of-Order execution: To be able to better exploit the available hardware resources and resolve the dependencies that may stall the pipelines, modern processors often implement the out-of-order unit, which can re-order the instruction streams, execute them speculatively (out of the original program order) , and commit them in the original program order.

Data Prefetching: A cost-effective solution for bridging the performance of processor and memory hierarchy is the use of cache. However, the cache only fetches data when the processor explicitly asks for it. Most processors nowadays have data prefetching hardware that overlap the possible cache misses with the processor computation, increasing the utilization of memory hierarchy bandwidth, and hiding the memory accessing latency being seen by the program [44].

Data level parallelism: Except for squeezing out on-chip instruction level parallelism, modern processors seek to exploit data-level parallelism. A good example is the SIMD (Single Instruction Multiple data) unit being seen on current x86 architectures: Intel SSE/SSE2 and AMD 3DNow! [11, 34].

Table 2.1 shows the important characteristics of current x86 processor microarchitecture (adapted from [11, 33, 34]). More specific information can be found in [9].

2.2.2 Memory Hierarchy and Delivered Performance

The memory hierarchy on modern processors can be abstracted in a pyramid, as shown in Figure 2.3. The registers are at the top of the memory hierarchy. They are also the clos-

Table 2.1 The Characteristics of x86 Processor Microarchitecture

Feature	*Pentium III*	*Pentium 4*	*Athlon*
Core	P6 Microarchitecture	NetBurst Microarchitecture	QuantiSpedd Microarchitecture
Pipeline Length	12	20	10(Int), 15(FP)
Execution Units	2 ALU/FPU	2 ALU/FPU	3 ALU/FPU/AGU
Registers	8 GPR 8 x87 8 SSE	8 GPR 8 x87 8 SSE/SSE2	8 GPR 8 x87 8 SSE
Single Precision Speed up	x87: 1 SSE: 4	x87: 1 SSE: 4	x87: 2 3DNow!: 4
Double Precision Speed up	x87: 1 SSE2: 0	x87: 1 SSE2: 2	x87: 2 SSE2: 0

est storage device to the CPU core[1]. From the top down in this hierarchical structure, L1 cache, L2 cache, Translation Lookaside-Table Buffer (TLB), main memory, and disk storage are the common storage devices seen in modern architectures. The further away it is from the CPU core, the cheaper the storage device is and the slower the access speed. The important metrics for measuring the performance of memory hierarchy are the memory bandwidth and memory latency of each level of memory hierarchy. The width of the rows between the layers in Figure 2.3 illustrates qualitatively the amount of memory bandwidth is available in between, and the memory latency has the opposite trend of the memory bandwidth.

[1]It usually takes about one cycle to load the data from registers to execution units, but this latency can usually be hidden by pipelining. Thus, the register and CPU core are shown as one component in Figure 2.3.

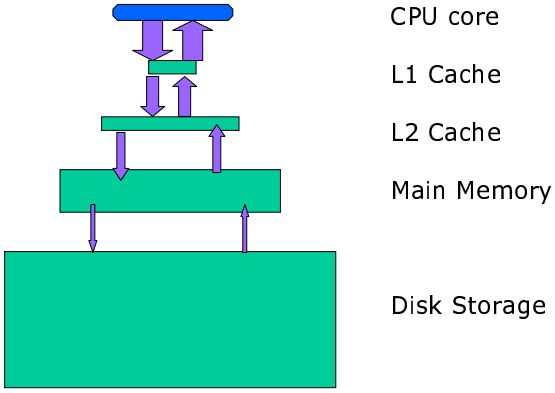


Figure 2.3 The Pyramid of Memory Hierarchy

The important memory hierarchy parameters for current x86 architectures are given in Table 2.2 (adapted from [9, 11, 34]). The detailed modeling of the memory hierarchy and the corresponding optimization strategy will be offered in Chapter III.

2.3 Previous Matrix Multiplication Work

An important lesson learned in the study of the underlying architecture is that the peak performance is only available at the top level of memory system because of the huge memory access overhead associated with accessing data in the lower level of the memory system. Thus, the fundamental optimization strategy for matrix multiplication is to keep data in the top level of the memory system and to reduce the data movement between the different levels of the memory hierarchy while keeping the pipeline as busy as possible.

Different approaches that possess this fundamental optimization principle have been developed to achieve the high-performance implementation of matrix multiplication on

Table 2.2 Memory Hierarchy Parameters of Current x86 Architecture

Feature	*Pentium III*	*Pentium 4*	*Athlon*
L1 I-Cache D-Cache	32 Bytes/Line 16 KB, 4-way 16 KB, 4-way	64 Bytes/Line 12K uops, 8-way 8KB, 4-way	64 Bytes/Line 64 KB, 2-way Exclusive with L2
L2 Cache	32 Bytes/Line 512 KB .. 2 MB 4-way	128 Bytes/Line 256/512 KB 8-way	64 Bytes/Line 256 KB 16-way
L1 to L2 Speed	0.5, 1× Clock Speed	1× Clock Speed	1× Clock Speed
L1 Latency	3 cycles	2/6 cycles	3 cycles
L1 + L2 Latency	7 - 27 cycles	9 - 13 cycles	11 - 20 cycles

various architectures. This section gives an overview of these approaches and discusses their advantages and limitations.

2.3.1 Early Stage of the Blocking Algorithm

The early exploration of blocking algorithms for the hierarchical memory system can be found in [25, 38, 41, 48]. Different strategies were employed for deriving the optimal blocking factors: Gallivan, et. al. [25] used a decoupling methodology to capture the important system parameters and model the blocking algorithm, Wolf and Lam [38, 48] improved the data locality based on the data reuse and loop transformation theory, and Schreiber and Dongarra [41] studied automatic transformation and blocking for nest loops.

Through these early studies of matrix multiplication and blocking algorithm, it is recognized that the high performance of matrix multiplication can be achieved by combining a high-level blocking strategy with a carefully tuned inner kernel for computing the submatrix product. However, the optimal blocking factors are determined by various machine parameters. Because of their inherent complexity, the performance optimization is relatively labor intensive for complex multi-level memories [46].

2.3.2 Self-Tuning Libraries

The PHiPAC project [15] demonstrated that automatic turning is achievable by using parameterized code generators for producing matrix multiplication codes in high level languages, with the expense of several days or weeks of optimization process. Next, the Automatically Tuned Linear Algebra Software (ATLAS) project [3, 46] reduced the search space for automatic tuning by generating and targeting only one specific kernel routine. Other tradeoffs are also made in ATLAS to further constrain the number of different blocking implementations being considered so that the total tuning time is shortened to within the order of several hours.

While the performance of self-tuning libraries is competitive with the vendor hand-tuned BLAS routines, the empirical philosophy behind these libraries and the strategies being employed in them have the following shortcomings:

Compiler dependency: Whaley, et. al. [46] state that ATLAS only requires an “Adequate ANSI C compiler” because the code generator can be trained to perform the complier’s

optimization work. However, ATLAS's performance optimization ability is strongly tied to the compiler's instruction scheduling ability. For example, compared with gcc[2] 2.95.x, ATLAS observes a 10% to 50% performance drop when using gcc 3.0. The reason for this performance degradation is that gcc 3.0 schedules the load operations slightly differently than gcc 2.95.x [1]. Hence, an implicit sensitivity to compiler versions exists.

Assembly code dependency: Whaley, et. al. [46] also state that "ATLAS is written entirely in ANSI/ISO C." However, in the latest version (3.5.0) of ATLAS , assembly coded kernels were used on many platforms, including the Intel Pentium 4 processor, AMD Athlon processor, PowerPC/Altivec architecture, and SUN UltraSparc architecture [2]. There are basically two reasons for this apparent contradiction:

- ATLAS is not adaptive to the new architectural features because of the compiler dependency. Such examples include the SSE/SSE2 optimized kernels for Intel platform, the 3DNow! optimized kernels for AMD Athlon, and the kernels that incorporated the machine-specific prefetching instructions for UltraSparc architecture.
- ATLAS's kernel strategy is not optimal. Thus it has to incorporate better assembly kernels for keeping the performance advantages. Such examples include the optimized assembly kernels for the AMD Athlon Processor [2].

Jagged performance curve: ATLAS's performance curve is not flat with respect to matrix size and this undermines the library's performance predictability[3].

Non-optimal performance: There are two major reasons for the inability of ATLAS to deliver the optimal matrix multiplication performance.

---

[2]GNU Compiler Collection, see [6]

[3]The term *predictability* refers to "having a small upper bound on the variation in performance (MFLOPS) with respect to matrix size" [43].

- As already claimed, *ATLAS's kernel strategy is not optimal*.
- As will be shown later, ATLAS's blocking strategy is not optimal either.

This thesis offers models to explain the performance behavior of self tuning methods and simple strategies to overcome their performance drawbacks.

2.3.3 Cache-Oblivious Algorithm

Cache-oblivious algorithms specifically refer to algorithms that do not rely on hardware-dependent variables for the optimized performance. Specifically, by use of a divide-and-conquer algorithm and an "ideal-cache" model proposed in [16], Frigo, et. al. [24, 40] prove that such a cache-oblivious algorithm achieves the same theoretical bound on cache misses for the same amount of work load. Although currently there is no concrete high performance implementation of such a cache-oblivious algorithm for matrix multiplication, similar ideas were used in the hierarchical methods for pursuing the optimized matrix multiplication [43, 47].

Notice that the "ideal-cache" model that cache-oblivious algorithms are based on does not incorporate some important architectural features that are key to achieving optimal performance. The corresponding optimization techniques that exploit these features are then left out of the scope of this asymptotic optimal algorithm:

- Cache misses can be overlapped with the processor computational activities with the help of out-of-order execution, data prefetching and careful sequencing of memory operations and floating point operations.
- TLB misses have more dramatic effects on performance than cache misses. It remains questionable as to whether the cache-oblivious algorithm is naturally TLB-oblivious.

For the optimal implementation of the matrix multiplication on the modern architectures, one has to combine the recursion strategy of cache-oblivious algorithms with iteration methods that take advantage of the above mentioned architectural features. This study is built on and benefits from both strategies.

2.3.4 Hierarchical Data Structures

The quadtree representation of recursive matrix storage was used by Frens and Wise [47] with the demand of up to 78% extra memory. This extra memory requirement may or may not be acceptable on virtual memory systems, but is not acceptable on real memory systems. The same recursive algorithm is also employed by Chatterjee, et. al. [18] with improved storage format, but the performance was significantly degraded because of the computations performed on the padding elements for handling of arbitrary sized matrices. Gustavson also employed padding in the recursive data structure and developed the corresponding recursive algorithm, which produces good performance because the computations are carefully designed to not perform on the padded elements. The 4D storage format that is similar to what Chatterjee discussed in [17] and the usage of table were also suggested and described by Gustavson [31].

More recently, Valsalam and Skjellum [43] proposed a new hierarchical data structure for the storage of matrices. This hierarchical storage formulation handles arbitrary size matrices while avoiding the use of tables, extra memory, or computations. It also improves data locality by the utilization of Morton ordering. Benefits of using this hierarchical

data structure also include: high performance, flat performance curves, avoidance of data copying cost, and admission of polyalgorithms[4].

2.3.5 Variants of New Blocking Approach

Some recent treatments of blocking produce high performance on various architectures and are summarized below:

Multiple-level Blocking: By benefiting from a simple theoretical model of hierarchical memories, the ITXGEMM project [30] yields implementations that outperform self-tuning libraries and rivals. The theory defined by the ITXGEMM researchers suggests that different algorithms that are dependent on the matrix dimensions at each level of the memory hierarchy must be employed for achieving maximal performance. This theoretical corollary is consistent with the favor of the polyalgorithm approach in [43]. The theoretical formula of ITXGEMM could also be utilized by self-tuning methods to reduce the search space.

Optimization for TLB: It was recognized that the time spent for handling TLB misses can account for 40% to 80% of total run time for the worse-case scenario on a virtual memory system [35]. The blocking algorithm used by Strazdins [42] that explicitly considers TLB size when choosing the blocking factors achieves good performance. However, Strazdins' work is done on a physically tagged cache machine (Sun UltraSparc) that emphasizes the

[4]The term *polyalgorithms* was introduced by Professor John Rice and refers to "The choice of one suitable algorithm from a set of candidate algorithms, all designed to solve the same problem, with the goal of obtaining the best possible performance." [43]

utilization of TLB, and the machine clock cycle is relatively low (170 MHz) so that the optimization work is not challenging. Recently, Goto and van de Geijn [28] proposed a new optimization strategy that is driven by the minimization of TLB misses and their libraries achieve highly competitive performance on various kinds of high performance computing platforms [7], including Intel Pentium 4, Itanium, and AMD Opteron processors.

2.4 Lessons learned

This literature review emphasized several important points that are key to high-performance implementations of matrix multiplication. They are as follows:

1. Blocking is a fundamental optimization strategy for matrix multiplication;
2. The mechanical parameter searching is useful if the complexity of the optimization problem can not be clearly identified;
3. The recursive strategy can obtain good performance. But it has to be bound with the iteration methods that can exploit the underlying architecture; and
4. Not only the cache system, but also the TLB is a performance-critical constraint.

CHAPTER III

RESEARCH APPROACH

The research methodology of this thesis is to identify the architectural performance bottlenecks through advanced architecture study and modeling, then determine the optimization priorities and strategies accordingly, and attack these bottlenecks with the help of a hierarchical data structure and optimized kernel routines. This chapter provides a qualitative analysis of different optimization approaches, describes the research framework that is based on the combination of the recursive ordering and an iteration algorithm, and presents the integration and tuning strategies of this study.

3.1 TLB-Obvious Architectural Modeling

3.1.1 Review of Memory Access Process

Previous architectural models for matrix multiplication often dismissed the presence of TLB misses, which have been identified as the performance culprit of matrix multiplication by some recent research [7, 28]. Thus, it is necessary to examine the practical memory access procedures on modern architectures and incorporate the performance role of the TLB into the global processing picture.

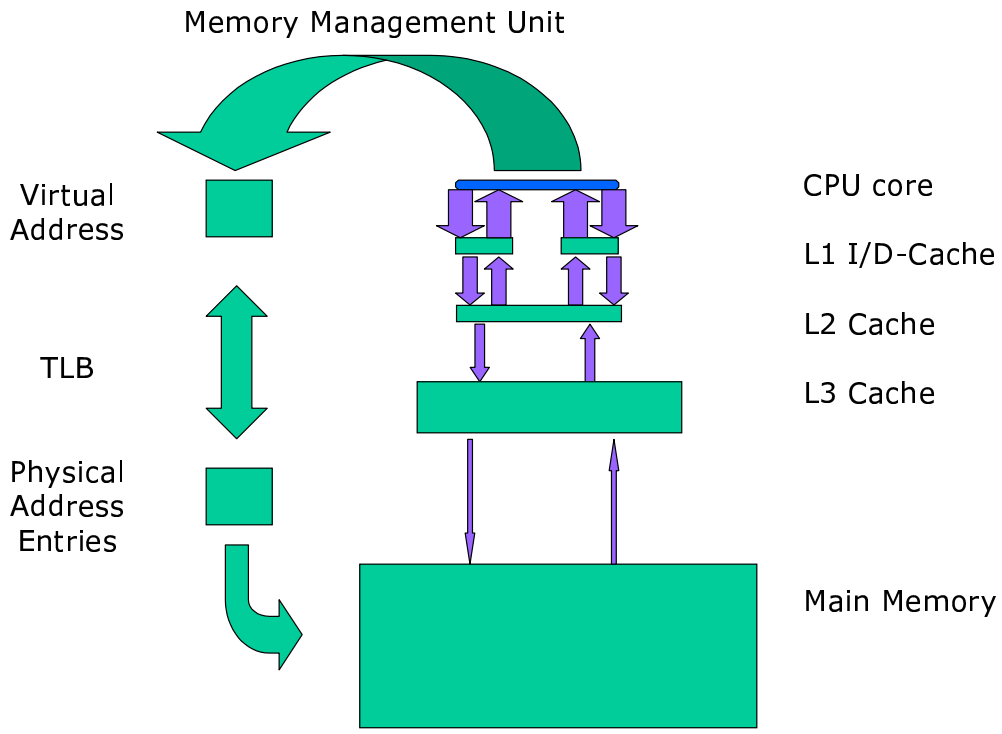


Figure 3.1 The Diagram of Memory Access Process

As shown in Figure 3.1, data fetching activities are done in parallel by both the MMU (memory management unit) and the cache system. The TLB deals with memory requests that cannot be accomplished in the cache system. The TLB can be deemed as a cache for the page table, which records the mapping from the virtual address to the physical page address and is usually stored in main memory. If an entry is found in the TLB, then the virtual to physical address translation can be done promptly. However, if the data is not found in the TLB, the TLB miss requires interrupt handling (for example, for software-managed TLB, Compaq Alpha processor and IBM POWER processors; see [35]), which in turn will flush the pipelines, fetch the page table from the main memory (at least two more memory references), update the TLB entry, and load the desired data entry. All these activities contribute to long pipeline stalling time. The cache misses, however, do not necessarily stall the pipelines. The processor can continue processing the instructions that are independent of the cache-missed entries.

3.1.2 Modeling of Blocking Algorithm

Blocking is a useful optimization strategy to improve data locality and reduce the access misses resulting from the memory hierarchy. Pseudocode for the jki variant blocking algorithm is given in Figure 3.2. Notice that the loop order of jki is often exchangeable and entails multiple levels of loops that may be employed for multiple-level cache blocking. Usually the copy process is also accompanied by possible transposition of the

submatrix for optimal kernel performance. Clean-up code for the blocking size that cannot be evenly divided by P, Q, and R are skipped for the simplicity of this pseudocode.

Based on Figure 3.1 and 3.2, the performance of this blocking algorithm on the virtual memory systems can be decomposed as follows:

$$T_B = T_K + T_{TLB} + T_{Cache} - T_O + T_{Copy} \tag{3.1}$$

where T_B is the overall execution time of the blocking algorithm, T_K is the execution time of the kernel routines (which may have different formats and assumptions of initial positions of matrix entries), T_{TLB} is the pipeline stalling time because of TLB misses, T_{Cache} is the total time of memory accesses (which includes all levels of cache accesses and main memory accesses but not the TLB accesses), T_O is the part of T_{Cache} that can be overlapped with the actual computation time T_K, (*e.g.*, the cache misses may be able to be overlapped with the computation by prefetching techniques), and T_{Copy} is the cost of copying the matrix into the global buffer, which occurs in most blocking algorithm implementations.

The blocking size is an important factor for the delivered performance of matrix multiplication. Kernel performance, measured by T_K, will only reach near peak performance when the matrix entries are stored in the top level cache of the memory hierarchy. Since the cache size is usually limited, the blocking size being seen in the conventional blocking algorithms is usually quite small, which means that more copy overheads T_{Copy} are involved and significant time is spent within the copy routine. The tradeoff here is that when the blocking size decreases, T_K, T_{TLB}, and T_{Cache} decrease, while T_{Copy} increases

```
for(sj = 0; sj < N; sj += Q) {
   for(sk = 0; sk < K; sk += R) {
      /* Copy part of B to the global buffer SB */
      COPY(SB, B, Q, R, IF_TRAN);
      for(si = 0; si < M; si += P) {
      /* Copy part of A to the global buffer SA */
      COPY(SA, A, P, R, IF_TRAN);
      /* Kernel computation, C <- SA' * SB */
      Kernel_MM(C, SA, SB, P, Q, R);
}  }  }
```

Figure 3.2 Pseudocode for the jki Blocking Algorithm

significantly. When the blocking size increases, T_K, T_{TLB}, and T_{Cache} increase, while T_{Copy} decreases significantly.

3.1.3 Blocking for TLB

Naturally, one would want to increase the blocking size to reduce the copy overheads T_{Copy} and try to still keep the pipeline full, that is, keep the sum of $(T_{TLB} + T_{Cache} - T_O)$ equal to the original processing time. Sustaining peak performance for a big submatrix demands consideration of the following three factors [8]. One requirement is that memory bandwidth from the lower level of memory hierarchy to the processor core be wide enough for streaming the data in. Another requirement is that the memory latency from the lower level of memory hierarchy to the processor core has to be well hidden so that the time for the cache misses can be overlapped with the kernel computation time. Prefetching techniques, including both hardware prefetching instructions and software prefetching, are helpful with this respect. These two requirements will minimize $(T_{Cache} - T_O)$ as much as possible. The third factor, which concerns the TLB, comes from the following facts presented on modern architectures:

1. TLB misses are more expensive than both L1 and L2 cache misses,
2. TLB misses do not occur if the memory entry is found in the cache system,
3. The TLB usually is bigger than L1 cache but smaller than or equal to L2 cache size, and
4. A few cache misses may be tolerated, but not TLB misses.

Components of the memory hierarchy usually follow a pyramid scheme (see Figure 2.3). This means that the larger and lower levels of cache are more expensive to access than higher elements, as illustrated in Table 2.2. Based on Item 1 and 2, the TLB could be regarded as one more extra "virtual cache" between L2 cache and the main memory at the algorithmic level. This is the view of the conventional blocking algorithm, which makes no distinctions between cache misses and TLB misses, reducing them with the same approach. But, based on Item 3, the TLB should reside between L1 cache and L2 cache in the memory hierarchy, not below the L2 cache, as indicated by items 1 and 2. This conflict and item 4 naturally leads to the following rules for performance optimization of matrix multiplication:

- The TLB has a stronger limitation for the blocking size than L2 cache, and
- The TLB misses should be handled in a different way as cache misses.

In short, the TLB plays a decisive role for in the performance of matrix multiplication. Specifically optimizing for TLB will allow big blocking sizes to be used, which in turn reduces the copy overheads significantly. This conclusion is already shown by [7, 28]. Their implementations that explicitly considered the effects of TLB achieved highly competitive performance for row-majored and column-majored matrix multiplication.

3.2 The Matrix Multiplication Framework

A combination of recursion and iteration strategy is employed as the framework of this thesis. The actual components include the hierarchical data structure, the iterative

algorithm, and the optimized kernel routines for the Alpha processor. The matrix is first hierarchically decomposed to submatrices. Recursive Morton Ordering is then applied in one hierarchy to organize the submatrices for better two-dimensional data locality and better exploitation of the memory hierarchy. It is on such submatrices that an iterative algorithm is carried out for computing the inner product, during which special consideration is given to optimization of the processor computation resource, and the possible overlapping of memory access activities and floating-point computation activities.

3.2.1 Hierarchical Data Structure

The hierarchical data structure of this thesis is inherited from variant 1 of HERO [43], which stands for "hierarchical extension of recursive ordering." This hierarchical storage format is carefully designed to utilize Morton Ordering to improve data locality in two dimensions and handle matrices of arbitrary size by taking advantage of the following fact: any integer can be expressed as a sum of powers of two.

Four levels of division and decomposition are required to transform the original matrix into the new hierarchically stored matrix, as shown in Figure 3.3. Padding is first applied to ensure that the matrix size is evenly divided by the blocking size, which will be determined later by considering the cache, TLB, and pipeline optimization.

Level 1 and 2 decomposition accomplishes the recursion process. In the first level, the padded matrix is divided into blocks. The matrix entries inside these individual blocks are still in row-major order. The original matrix then can be deemed as a small matrix that

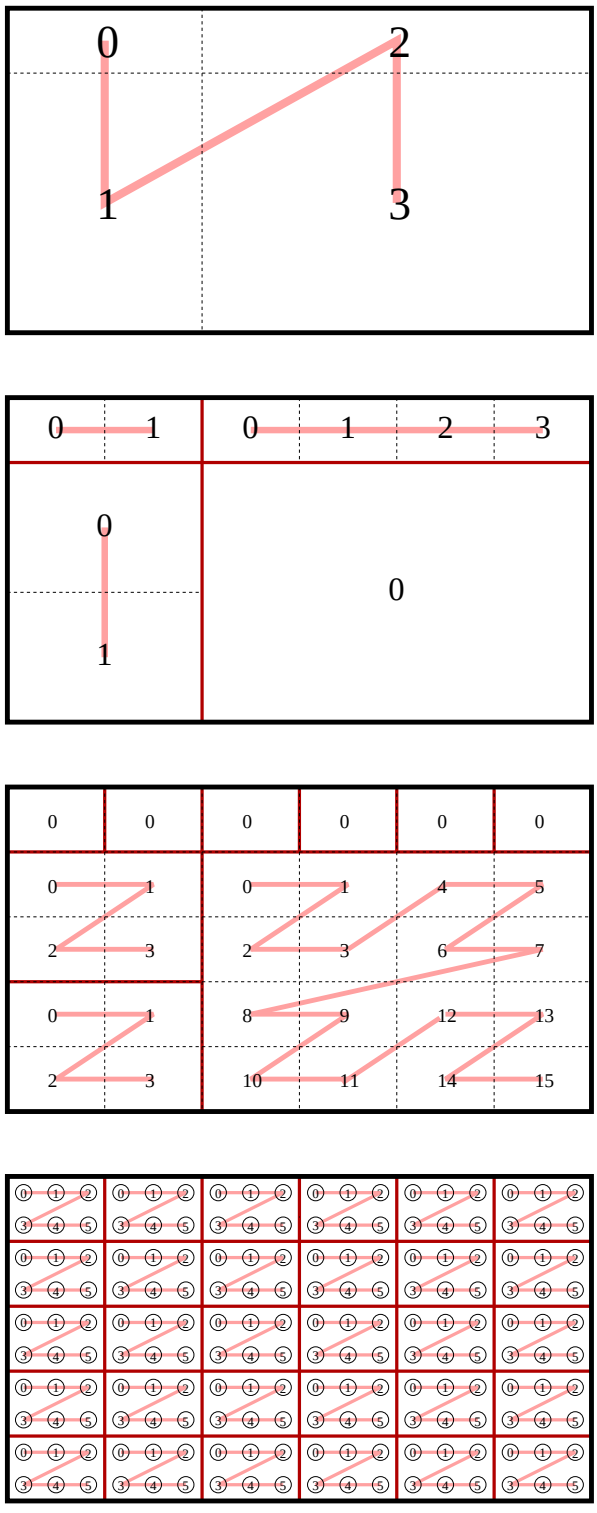


Figure 3.3 A Hierarchically Stored $10 * 18$ Matrix (Adopted from [43])

is composed by the level 1 blocks. This small matrix is easily decomposed to a series of level-2 matrices, whose size are power of 2 and arranged such that the biggest one stays in the right-bottom corner and the smallest one stays in the left-top corner. Inside these level-2 matrices, the level-1 blocks are recursively organized by Morton Ordering.

The third and fourth level are only designed to uniquely specify the order of the level-2 matrices. The level-2 matrices are tiled in the third level and level-3 tiles are arranged in column-major order (or row-major order) at the fourth level.

3.2.2 Iterative Algorithm

Although different algorithms may be employed for this hierarchical storage format to exploit the advantages of the polyalgorithmic approach, it is suggested by several previous researchers [30, 43] that the recursion strategy must be combined with iterative methods that can utilize the computation resource at a fine granularity.

This study chooses an ijk variant iterative algorithm for the multiplication of the recursively stored matrix. The pseudocode of this algorithm is shown in Figure 3.4. Notice that the indexing for the third and the fourth level is trivial and is thus not given. The Morton-Ordered address calculation for the first and the second level is rather complex. An algorithm that is based on fast integer dilation method is employed here for reducing such indexing overheads [43].

The performance of such an iterative algorithm can be modeled as:

$$T_I = T_K + T_{TLB} + T_{Cache} - T_O + T_{INDEX} \tag{3.2}$$

```
for(Each Level-3 Tile) {
  for(Each Level-2 Square Matrix) { /* Size is b*b */
    /* Find the first matrix entry of each Level-2 Matrix */
    FIND(SA2, SB2, SC2);
    for(si = 0; si < b; si++) {
      for(sj = 0; sj < b; sj++) {
        /* Find the first entry of each Level-1 C Block */
        SC1 = C + (SC2 + INDEX_MOR_ORI(si, sj)) * P * Q;
        for(sk = 0; sk < b; sk++) {
        /* Find the first entry of each Level-1 A and B Blocks */
        SA1 = A + (SA2 + INDEX_MOR_ORI(si, sk)) * P * R;
        SB1 = B + (SB2 + INDEX_MOR_ORI(sk, sj)) * R * Q;
        /* Kernel computation, C <- A * B */
        Kernel_MM(SC1, SA1, SB1, P, Q, R);
}  }  }  }  }
```

Figure 3.4 Pseudocode for the ijk iterative algorithm (Adapted from [43])

where T_I is the overall execution time of the iterative algorithm, T_{INDEX} is the cost of finding the correct Morton-Ordered indexes of matrix entries based on their original column-major order indexes, and the other factors possess the same meaning as in Equation 3.1.

Inspection of Equations 3.1 and 3.2 implies that the combination of the hierarchical data structure and the iterative algorithm leads to the following:

1. Good two-dimensional data locality is achieved such that the performance of the memory hierarchy is automatically optimized;
2. The time consuming copy cost associated with blocking algorithm is avoided;
3. The Morton Order indexing is done by integer operations, which are cheaper than memory operations involved in the copy process;
4. The fine-grained control over the processor pipeline is achievable with the iterative algorithm; and,
5. It is possible to reassemble the indexing process (integer operations) and the computation process (floating point operations) in a proper way such that the superscalar architecture is exploited.

3.2.3 Optimized Kernel Strategy for Alpha

Until now, this thesis has only discussed high-level modeling, the hierarchical data structure, and algorithms. This section presents the lowest level kernel design strategy as well as the specific optimization details for the chosen Alpha 21264 processor.

It was identified by Valsalam and Skjellum [43] that the "fat" interface of DGEMM $C \longleftarrow \alpha AB + \beta C$ is not suitable for the kernel optimization. First, this interface is designed for general purpose matrix multiplication and involves some parameters that may not be necessary for kernel operation, such as the accumulation parameter β and the scaling factor α. Also, the kernel performance is indeed sensitive to the various parameters of

the kernel routine, including the blocking size for A, B, and C and the α and β parameters as well as the loop unrolling factors. The optimized code for one case may be largely suboptimal for another case. If too many parameters are involved in the kernel routine and one composes different kernel subroutines for each possible case, the combination of these parameters would lead to code explosion in the kernel code. Even self-tuning libraries that train the computer to generate the C code for different kernel cases would need to make some compromises so that only a subset of all possible cases are generated.

Therefore, this thesis advocates a simplified version of the DGEMM interface: $C \longleftarrow A * B$. The handling of α of β can be accomplished by complementary subroutines. Also, instead of coding a kernel toward a fixed blocking size and unrolling factors, the kernel C interface is designed to work with different blocking sizes and unrolling factors. The performance of such a kernel strategy may not be optimal. However, as will be shown later, better overall performance is still achievable with the help of a memory-friendly matrix storage format and yields the following advantages from using a simple yet adaptive interface:

- flexible tuning and integration in the later phase,
- less development cost because of the reduced kernel cases, and
- only ϵ slower kernel performance.

The optimized kernel of this thesis has been provided by Mr. Kazushige Goto at University of Texas at Austin and JPTO [8] and is completely coded in Alpha assembly instructions. The main advantage of using assembly instructions other than high level pro-

gramming languages is the delicate handling of the processor pipeline and computation resources, which facilitates the following techniques keyed to optimized performance:

Flexible Resource Allocation

The computation resources available on the Alpha processors are the 32 integer registers for integer/load/store instructions and the 32 floating-point registers. Directly manipulating them allows better implementation of most inner kernels and avoids running out of registers because of the excessive loop unrolling, which is possible if one depends on the compiler for the loop optimization.

Explicit Software pipelining

The Alpha processor 21264 has a 4-way out-of-order superscalar pipeline. It can issue two integer instructions (including both load and save instructions) and two floating-point instructions in the same cycle. Therefore, grouping these instructions that can be executed together (software pipelining) will explicitly expose the instruction level parallelism and boost performance. Although the out-of-order unit on the Alpha processor will help with this respect, it is still beneficial to follow these coding principles. In Figure 3.5, four instructions (two load instructions, one floating point add instruction and one floating-point multiply instruction) can be executed in one cycle and thus grouped together in the kernel routines.

Such code groups are largely used in the kernel assembly code. Sometimes when resource conflicts are expected, the “unop” instruction is inserted to maintain this coding pattern while waiting for the resource to be available again, as shown in Figure 3.6

```
addt $f8, $f16, $f8      # $f8 = $f8 + $f16
lda $3,  SIZE($3)        # $3 = *($3 + SIZE)
mult $f23, $24, $16      # $f16 = $f23 * $f24
ldt $f23,  SIZE($4)      # $f23 =   *($4 + SIZE)
```

Figure 3.5 Sample Assembly Code I of Software Pipelining (Adopted from [8])

```
addt $f1, $17, $f1       # $f1 = $f1 + $f17
unop                     #
mult $f20, $f25, $f17    # $f17 = $f20 * $f25
ldt $f25,  SIZE($6)      # $f25 = *($6 + SIZE)
```

Figure 3.6 Sample Assembly Code II of Software Pipelining (Adopted from [8])

Overlapping of Memory Access and Computation

The idea of increasing the blocking size to reduce overhead will fail if peak performance cannot be preserved for big blocks. The order of memory access operations, including both load and save instructions, therefore needs to be scheduled very carefully along with the floating point operations to hide the higher memory latency implied by the bigger blocking size. Prefetching techniques are also helpful for keeping the pipeline full. The prefetching distance depends on the latency of the memory hierarchy , which is given in Table 5.2. If a matrix entry is to be used at cycle n and it is estimated to be stored in the i level of memory hierarchy, then it would need to be loaded at cycle $[n - \sum^{i} L(i)]$, where L(i) is the access latency from level i to level $i - 1$ of the memory hierarchy. This

can be done by either the hardware prefetching instructions or an explicit arrangement and scheduling in the assembly language level.

Of course, the ability to prefetch data entries is not infinite, and memory bandwidth is always limited no matter how cleverly one schedules the instructions. The kernel performance and blocking size that allows one to sustain peak performance is actually largely dependent on the programmer's skill. Fortunately, the kernel created by Mr. Kazushige Goto obtained around 96% of peak performance on a Alpha 21264 processor in measurements done as part of this thesis work. For fairly big blocks that are larger than the L1 cache of the Alpha processor, this kernel still reached about 93% of peak performance.

CHAPTER IV

PERFORMANCE INTEGRATION, TUNING, AND BENCHMARKING

The performance integration, tuning, and benchmarking strategies of this thesis are aimed at balancing memory access and kernel computations for better overall matrix multiplication performance. They are interconnected with each other and introduced together in this chapter. The following section explains the integration and tuning strategy, which is employed at the final phase of this study to put the framework together. In order to ensure the fair comparison of different optimization methods and validation of the hypothesis, the performance metrics have to be well defined from the perspective of the application users. These metrics are specified in Section 4.2 and described along with the practical measurement procedures for obtaining their trustworthy values.

4.1 Integration and Tuning Strategy

The integration of this matrix multiplication framework is relatively straightforward. The hierarchical storage format that this study is based on grants a flexible polyalgorithmic approach for matrix multiplication. In turn, the selection of the iterative algorithm allows the delicate handling of processor resources. Also, the reduced kernel interface enables the seamless embodiment of optimized assembly kernels. However, optimal performance is

not obtained by simply putting the optimal components together. A careful tuning strategy is required to reach the peak performance of the processor.

The basic tuning strategy involves mechanical parameter searching, which is similar to the idea behind the self-tuning strategy, except that the tuning of this study is done at coding time instead of run time. The reduced kernel interface also helps here since there are only a few factors that need to be specifically considered: the blocking size P, Q, and R that correspond to the original matrix size M, N, and K, respectively.

Two tuning stages are employed to ensure better overall performance achievement. The first stage tunes the kernel performance. In this stage, the range of P, Q, and R can be estimated by performance modeling as described in Section 3.1. Then a special benchmark program is created to measure the kernel performance for this range of P, Q and R, within which the best cases were picked. In the second stage, the global matrix multiplication performance is used as the performance evaluation criteria. Another specially designed benchmark program is applied to monitor the overall performance for matrix multiplication with size $M = N = K = 1000$. The blocking sizes in the second searching process were setup to be the ones found in the previous tuning phase.

An important performance feature of the hierarchical data structure is the extremely flat performance curve. The overall performance flatness is not only presented for different matrix sizes, but also shown for different blocking size. This attractive feature indicates that the tuning procedure will choose the optimal blocking size for better overall performance without being affected by the concrete tuning parameters, such as the matrix size

1000 used in the final tuning stage. Figure 4.1 illustrates such a performance flatness regarding to the blocking size P. In later chapters, the flatness of the performance curve will also be mathematically defined, measured, and compared.

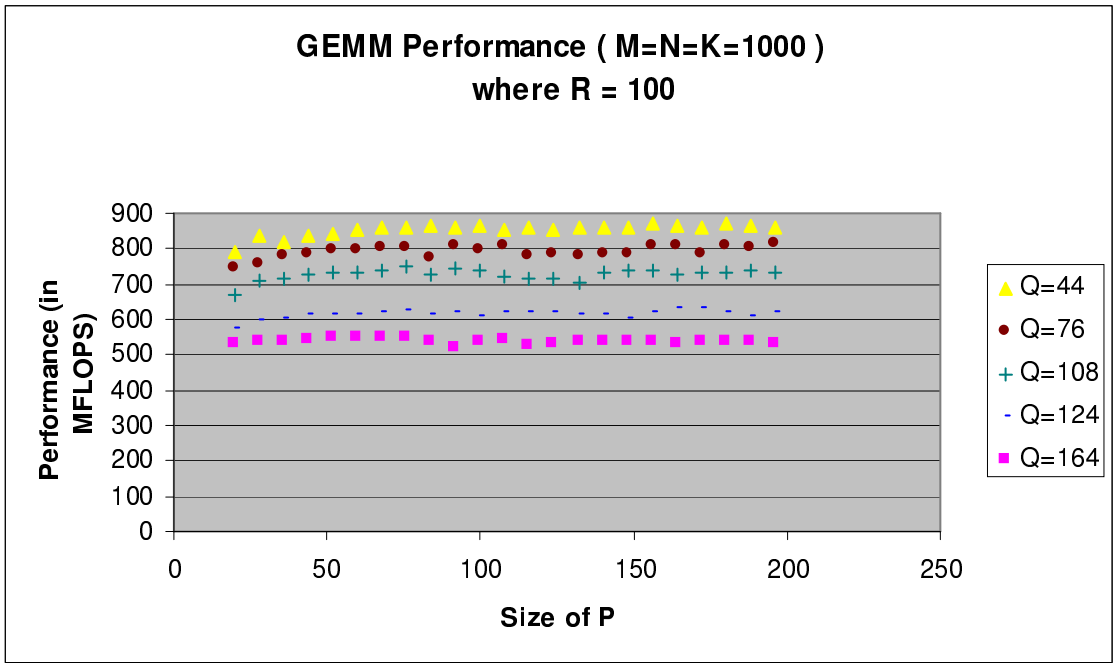


Figure 4.1 Performance Flatness Regarding to The Blocking Size

4.2 User Level Performance Metrics

A basic principle of determining the proper metrics for performance measurement is that the performance of each architectural component is not an accurate indication of the overall performance. Since each approach has made their own tradeoffs between different

performance components in Equation 3.1 and 3.2, comparing the performance components alone, (*e.g.*, T_{Cache} will mean little for the global performance), which is what end users desire. Thus, only the performance that is concerned by the end users is measured, compared and contrasted in this study. Table 4.1 shows an overview of such user level performance metrics and their mathematical formulas. The double precision floating point matrix multiplication is assumed throughout this thesis.

Table 4.1 User Level Performance Metrics

Performance metrics	*Measured Performance*	Formula
In-cache MFLOPS	Kernel Performance	$K(A) = \frac{2MNK}{Clock(A) * ClockSpeed}$
Out-cache MFLOPS	Global Performance	$M(A) = \frac{2MNK}{T(A)}$
Degree of Flatness	Global Performance	$F(A) = \frac{\max[M(A)] - \min[M(A)]}{\max[M(A)]}$

4.2.1 In-Cache Performance

The in-cache performance is the performance of matrix multiplication whose operands can well fit into the top level of the memory hierarchy. It isolates the influences of memory hierarchy and indicates the exact extent of the processor pipeline being exploited. While most application users are more concerned about the problem of big matrix multiplication, this metric reflects the available kernel performance in a certain degree and also facilitates the first stage of the tuning process.

To ensure accurate measurement of the in-cache performance, it is assumed that all the matrix entries involved in such kernel routines are already stored in cache. Practically, this can be done by repeating the kernel operations several times on one specific set of matrices until all the matrix entries are fetched into the L1 cache. Then the timing results of matrix multiplication kernel routine A are averaged and the mean value $T(A)$ is obtained. However, since the in-cache computation is quite fast and the matrix size is relatively small, the standard C timing utility may not be sufficiently precise. A little bit assembly instruction that can measure the cycles, $Clock(A)$, passed during the execution will better suit the required accuracy.

Notice that there is no standard format and size specification for the kernel routine. Different approaches have their own choices of these parameters for the optimal performance. For the fair comparison of different approaches, it is necessary to employ a metric that eliminates the impacts of different problem size. MFLOPS is suitable for such measurement, and the In-cache performance of approach A is then defined as follows:

$$K(A) = \frac{2MNK}{Clock(A) * ClockSpeed} \tag{4.1}$$

4.2.2 Out-of-Cache Performance of Matrix Multiplication

An important metric for evaluating the the actually delivered performance of matrix multiplication is the number of out-of-cache MFLOPS being obtained. There are two major differences when measuring the out-of-cache MFLOPS and the in-cache MFLOPS.

One difference is the initial position of the matrix entries. While the matrix entries are assumed already in cache in the case of measuring in-cache performance, measuring out-of-cache performance only assumes that the matrix entries are stored in main memory. Another difference is that the range of interested matrix size for two experiments is selected not to overlap with each other. Since the matrix size for out-of-cache problem is usually relatively large, the standard C timing routing can be used to obtain the average execution time $T(A)$ of approach A.

The out-of-cache MFLOPS can be calculated by:

$$M(A, M, N, K) = \frac{2MNK}{T(A, M, N, k)} \tag{4.2}$$

where $k \in [200, 5000]$. The minimum value 200 is calculated on the size of L1 cache, and $5,000$ is calculated on the size of main memory.

Degree of flatness is another important metric for evaluating the overall performance of matrix multiplication. It simplifies the tuning procedure and provides the performance predictability that is critical for real time computing and parallel computing.

Given the out-of-cache MFLOPS $M(A, M, N, K)$ of approach A, the degree of flatness of approach A is defined as follows:

$$F(A) = \frac{\max[M(A, M, N, K)] - \min[M(A, M, N, K)]}{\max[M(A, M, N, K)]} \tag{4.3}$$

The performance of approach A is aggregately higher than B if and only if:

$$\sum^{ijk}[M(A, M, N, K) - M(B, M, N, K)] > 0 \tag{4.4}$$

The following formulas define the performance criteria for the square matrix multiplication with size S that this thesis will experiment:

$$F(A) < F(B), \text{where } F(A) = \frac{\max[M(A,S)] - \min[M(A,S)]}{\max[M(A,S)]} \tag{4.5}$$

$$\sum^{S}[M(A,S) - M(B,S)] > 0 \tag{4.6}$$

Notice that neither F(A) nor M(A) is meant to be an accurate indication of better performance by itself. However, the combination of generally larger MFLOPS and smaller F(A) evidently shows the better performance of one approach over another one. Also, these metrics are not meant to be used for comparing the kernel performance because that the kernel format and assumption of initialization are different for different approaches.

CHAPTER V

EXPERIMENTS, RESULTS, AND ANALYSIS

This chapter presents the experimental setup of this study, reports the measured performance of different methods on the same test bed, and compares and contrasts them within a detailed analysis to validate the hypothesis. The experimental environment described in the next section is specifically designed such that an objective and precise performance measurement of various approaches is achievable. The performance metrics defined in Section 4.2 are employed for ensuring unbiased performance evaluation. The benchmark results are then presented in Section 5.2 and 5.3. The performance analysis are given wherever appropriate to justify the correctness of the hypothesis. Then, certain amendments of the hypothesis are made.

5.1 Notations and Experimental Configuration

This thesis studies five different implementations of matrix multiplication. The TLB-blocking algorithm (GOTO library) and the hierarchical methods (including the original HERO and the optimized HPCL[1] implementation) are implemented by the author and the author's collaborators. Reference libraries include the self-tuning library (ATLAS) and the

[1]HPCL is specifically the label for the new work of this thesis

vendor library (CXML). For the sake of convenience, the framework of the approaches being studied, their name and concrete version, as well as the label that will be used in the performance table and graphs are all listed in Table 5.1.

Table 5.1 Notations of Different Methods

Framework	*Library Name and Version*	*Label*
Blocking for TLB	Goto	Triangle
Hierarchical DS(Data Structure)	HERO	Multiply
Hierarchical DS + Optimized Kernel	HPCL	Diamond
Conventional blocking algorithm	CXML 5.1.0 and CXML 5.2.0	Circle
Self-Tuning Methodology	ATLAS 3.2.1 and ATLAS 3.4.1	Square and Plus

Based on the reasons specified in Section 2.2, this thesis chooses the Alpha 21264 processor based platforms to verify the proposed hypothesis. Both Linux and Tru64 Unix based systems are used to set aside the possible performance interference produced by the operating system. The compilers used are usually the highest versions of GCC tools that are available for that particular machine. One of the reference libraries, ATLAS, has the compiler dependency, so that an old version of GCC needs to be used to obtain the highest performance. The approach introduced in this study does not suffer from this issue. The author made sure that appropriate compilers were used to produce the highest performance whenever possible, as well as other significant settings for the delivered performance.

Table 5.2 Experimental Configurations (Adapted from [20])

Machine	*Compaq Alpha DS20 Server*	*HP AlphaServer SC V2.5 UK1 system*
Processor	Alpha 21264/EV6	Alpha 21264A/EV67
Clock Rate	500 MHz	667 MHz
Pipeline Length	7(Int), 9(FP)	7(Int), 9(FP)
L1 cache size	64 Kbytes	64 Kbytes
L1 latency	2 cycles	2 cycles
L2 cache	4 Mbytes	8 Mbytes
L2 latency	8 cycles	8 cycles
TLB size	128 Entries	128 Entries
TLB misses	Several hundreds cycles	Several hundreds cycles
Page size	8 Kbytes	8 Kbytes
Main Memory	768 Mbytes	4 Gbytes
Memory latency	About 150 cycles	About 150 cycles
FP Speed up	Multiply by 2	Multiply by 2
Peak MFLOPS	1000	1333
Compiler Used	GNU GCC and Compaq C Compiler	GNU GCC and Compaq C compiler
Operating system	Linux 2.2.14-6.0 and 2.4.9-32.5smp	Tru64 Unix version V5.1A
Notation	EV6-Linux-6.2	EV67-Tru64

Details regarding the machine configuration, including both hardware and software aspects, are shown in Table 5.2. Notice that from the perspective of computer architecture, the Alpha EV6 and EV67 processors being used belong to the same processor generation [20]. The only difference that may affect the performance behavior is the difference of the clock speed, which can be eliminated by employing the proper performance metric. This way it is made sure that the performance properties being observed comes purely from the software layers, not the hardware side. Unless explicitly specified, the default platform is the Compaq Alpha DS20 server with Linux kernel version 2.2.14-6.0. The Linux kernel 2.4.9-32.5smp is only being employed because the CXML was not available on the default platform. Another default setting is that only the performance of double precision square matrix multiplication (of size S) will be studied unless otherwise stated.

5.2 Performance of Inner Kernels

The measured kernel performance of HPCL and ATLAS are presented and compared in this section. Other approaches' kernel performance are not benchmarked because that they are using either the similar kernel with HPCL (*e.g.*, CXML also uses the kernel made by Mr. Kazushige Goto [4]), or a much slower suboptimal kernel (as the case of the original hierarchical framework).

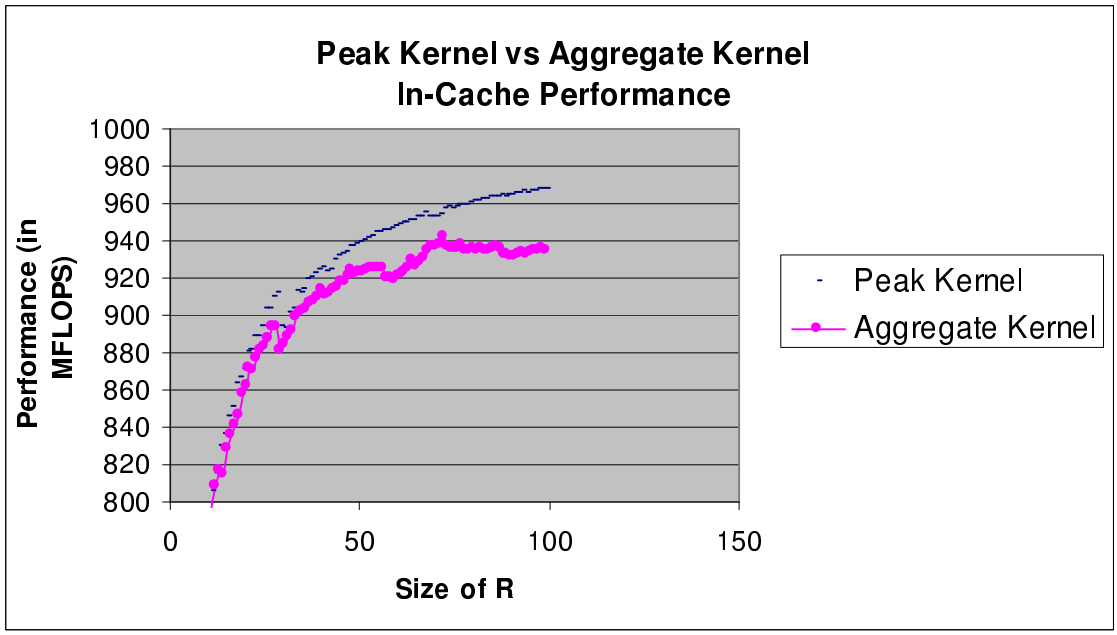
5.2.1 Peak Kernel vs Aggregate Kernel

First, the relationship of the kernel performance and the global performance is evaluated. The kernel of HPCL reaches the maximal performance , 97% of peak performance, when $P = 12$, $Q = 36$, and $R = 100$ (hereinafter called the "Peak Kernel"). However, the blocking size for the optimal overall performance is $P = 128$, $Q = 40$, and $R = 100$ (hereinafter called the "Aggregate Kernel"). Figure 5.1 reports the in-cache performance of the Peak Kernel and the Aggregate Kernel and the out-of-cache performance being obtained by using Peak Kernel and Aggregate Kernel respectively.

The Aggregate Kernel can only reach 93% of peak performance, which is about 4% slower than the performance of the Peak Kernel. However, the overall performance of using the Aggregate Kernel outperforms the performance of using the Peak Kernel on almost each significantly large matrix size. *These two figures show that maximum kernel performance is not a guarantee of the optimal global performance.*

5.2.2 HPCL vs ATLAS

This section shows a comparison of ATLAS's kernel performance and HPCL's kernel performance. A coincidence being observed is that ATLAS also reaches the maximum kernel performance at the same size of HPCL's Peak Kernel. Since ATLAS generates thousands of inner kernels, one only needs to compare with its best kernel performance. Such a performance comparison is presented in Figure 5.2. The HPCL Aggregate Kernel is about 3% higher than ATLAS's fastest kernel, which runs around 90% of peak performance.



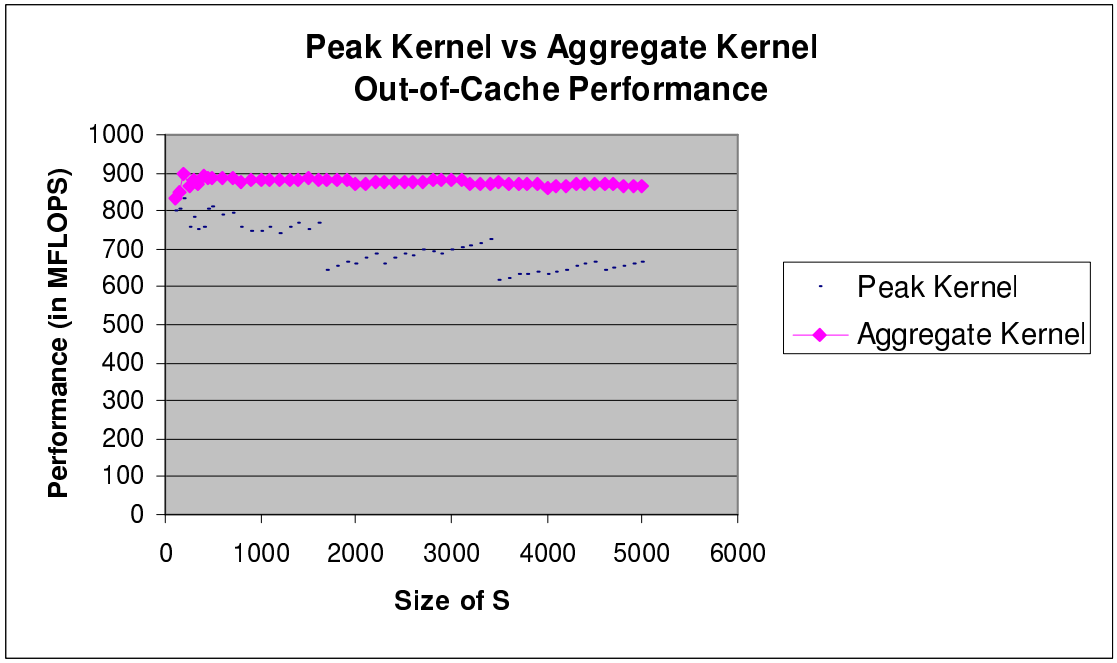


Figure 5.1 Performance of Peak Kernel and Aggregate Kernel

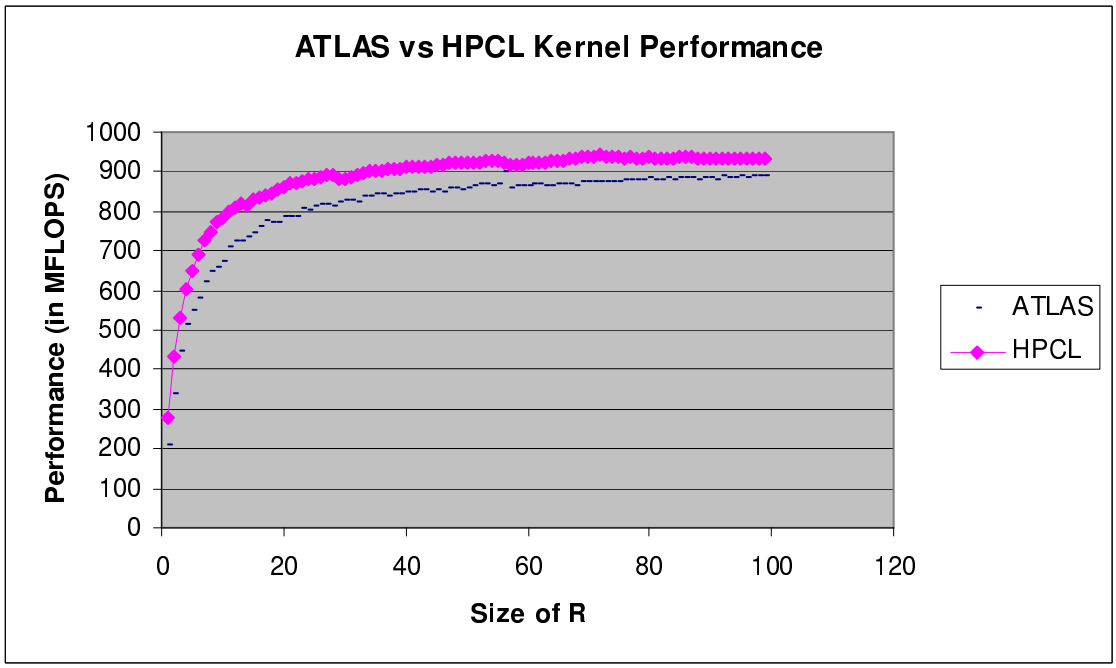


Figure 5.2 The In-Cache Performance of HPCL and ATLAS

This performance superiority demonstrates that the kernel strategy being utilized in this study is more adaptive and more robust than the self-tuning approach. There are two major reasons for this performance advantage. One is the assembly coding that allows the direct manipulation and exploitation of the processor resources. Another reason is the reduced kernel interface that avoids some unnecessary parameters (and possible overheads) of the kernel routine.

5.3 Performance of matrix multiplication

The section offers a detailed performance comparison of all five implementations that include GOTO, ATLAS, CXML, HERO, and HPCL (see Table 5.1 for more information on these libraries). The out-of-cache performance of these approaches on both Tru64 Unix and Linux based platforms are related, compared, and analyzed in Section 5.3.1 and Section 5.3.2.

5.3.1 Explicit TLB blocking vs Implicit Memory Optimization

The out-of-cache performance of GOTO library that explicitly blocks for the size of TLB and HPCL implementation that implicitly optimized for both cache and TLB layers is presented in Figure 5.3

These performance curves exhibit the following two properties that all promote a polyalgorithm approach for the optimized implementation of matrix multiplication.:

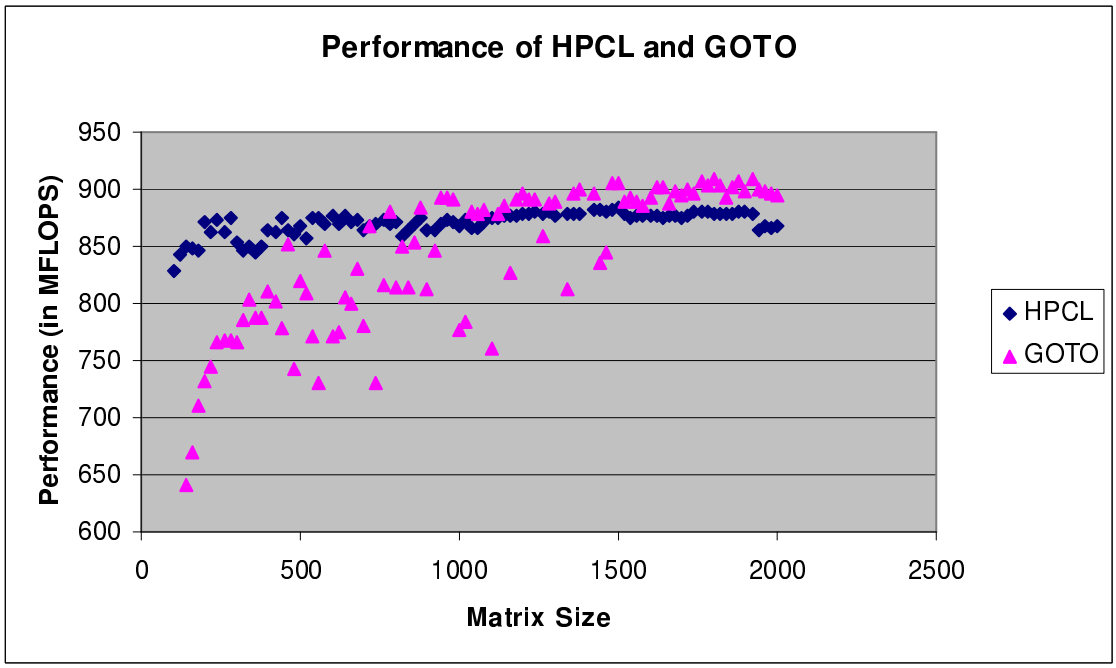


Figure 5.3 The Out-of-Cache Performance of GOTO and HPCL

Flat Performance of Hierarchical Framework: The performance of HPCL reaches the peak (about 870 MFLOPS) for a relatively small matrix size and then stays on a flat plateau for arbitrary large matrix size. In contrast, the GOTO library does not reach the peak until a matrix size of around $1,000$, and its performance waved around the peak performance for certain matrix sizes, especially when matrix size is near a power-of-two integer. Obviously, the utilization of the hierarchical data structure makes the HPCL implementation more favorable to the optimization of the memory hierarchy.

Performance Switching Point: On average, the performance of HPCL implementation is higher than GOTO library for matrix size from 100 to around $1,000$, and the performance of GOTO library is higher than HPCL implementation for matrix size from around $1,000$ to $2,000$. This is because that the memory access operation is usually several orders slower than the floating point operations. When the matrix size is small, the copy cost that has a complexity of $\Theta(S^2)$ will still be considerably larger than the computational cost that has a higher order complexity of $\Theta(S^3)$. Thus the hierarchal framework that avoids the copy cost will have a performance advantage. When the matrix size increases to a relatively large size that the memory copy cost can be well amortized over the computation time, the GOTO approach that explicitly copies and blocks for TLB will achieve higher performance. This derivation is also consistent with the qualitative architectural modeling done in Section 3.1.

5.3.2 Delivered Performance

This section offers a comprehensive performance analysis of the five approaches being studied. The performance metrics defined in Section 4.2.2 are measured and the performance results are given. Based on these results, the hypothesis of this thesis is validated.

The measured Degree of Flatness and Aggregate MFLOPS are provided in Table 5.3. These values are calculated based on the corresponding definition for matrix size ranging from $1,000$ to $5,000$. The selection of area is because, except the HPCL implementation, other libraries do not reach the peak performance until the matrix size increases to $1,000$. Also, it is more interesting to study the multiplication of big matrices from both theoretical and empirical perspectives. However, many applications are likely to appreciate higher performance for smaller problem sizes including those that do many small instances, rather than one huge instance.

Table 5.3 Degree of Flatness and Aggregate MFLOPS

Approach	*HERO*	*HPCL*	*GOTO*	*ATLAS3.2.1*	*ATLAS3.4.1*	*CXML*
EV6-Linux Degree of Flatness	2.7 %	2.6 %	4.4 %	59.4 %	14.8 %	6.2 %
EV67-Tru64 Degree of Flatness	1.8 %	2.4 %	2.7 %	27 %	6.8 %	3.7 %
EV6-Linux Aggregate MFLOPS	36.08 $\times 1000$	45.47 $\times 1000$	46.07 $\times 1000$	37.33 $\times 1000$	42.86 $\times 1000$	44.38 $\times 1000$
EV67-Tru64 Aggregate MFLOPS	50.58 $\times 1000$	62.77 $\times 1000$	63.42 $\times 1000$	57.37 $\times 1000$	60.94 $\times 1000$	61.24 $\times 1000$

Combining with the predefined performance evaluation criteria (Equations 4.5 and 4.6), this table clearly shows that the approach of this study (HERO, HPCL and GOTO) achieves better performance than the self-tuning library (ATLAS) and the vendor library (CXML). Notice that HERO implementation has the best degree of flatness, however, the aggregate performance of HERO is not good because it builds the kernel using C-language macros. HPCL implementation integrates a more optimized kernel into the hierarchical framework, and achieves both higher and flatter performance than all the reference libraries.

A complete performance comparison of all five approached on both Tru64 Unix and Linux system are shown in Figure 5.4 through Figure 5.7. The matrix multiplication are benchmarked for size from 200 to $5,000$ as well as the power-of-two sizes, which are specifically selected because that the performance of matrix multiplication is known to be sensitive for such matrix size or leading dimensions.

The figures explicitly shows that the hierarchical and the TLB-blocking based methods beat the "state-of-art" libraries (ATLAS and CXML) on all the platforms being studied. The GOTO library is always the best for big matrix size, expect HPCL outperforms it for the matrix size less than $1,000$. The HPCL library is the second best implementation. It has a performance advantage over both ATLAS and CXML until the matrix size increases to around $4,000$, after that the performance of HPCL is still competitive with ATLAS and CXML. A performance drop near $4,096$ is visible on all the performance curves except the hierarchical formulations, including both HPCL and HERO. By examining the power-of-two performance graphs, it is more clear that the blocking based implementa-

tions, including GOTO, ATLAS, and CXML, have to suffer at near power-of-two matrix sizes because of the cache thrashing effects. The usage of hierarchical work avoids such performance anomalies and provides the good performance predictability to the end users.

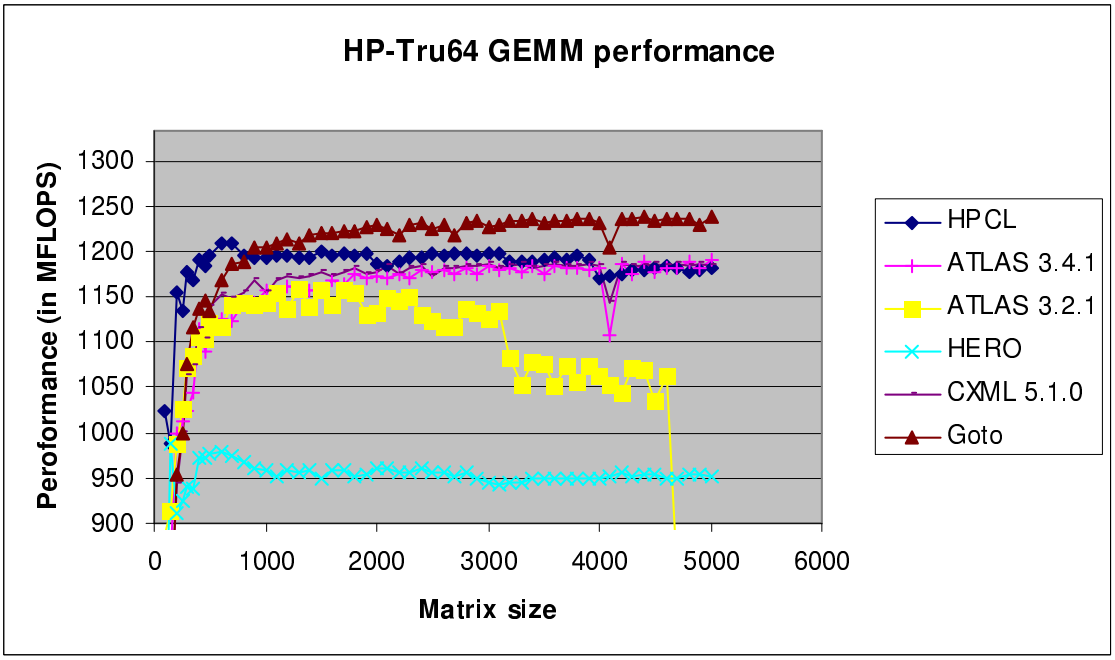


Figure 5.4 Performance of Various Approaches on EV67-Tru64

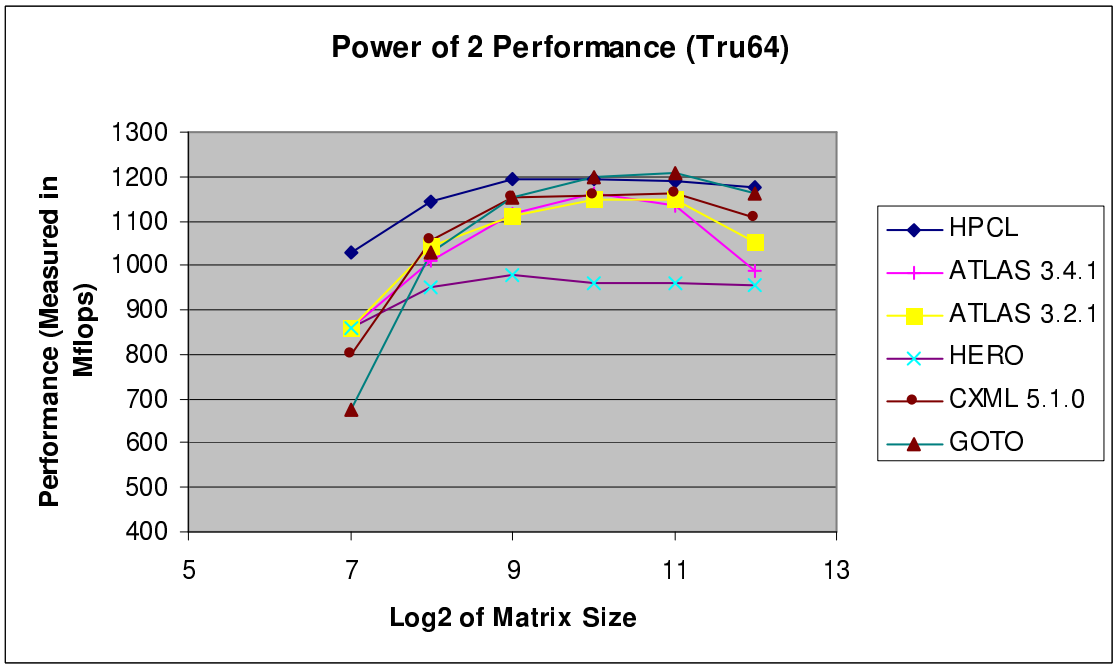


Figure 5.5 Power of 2 Performance of Various Approaches on EV67-Tru64

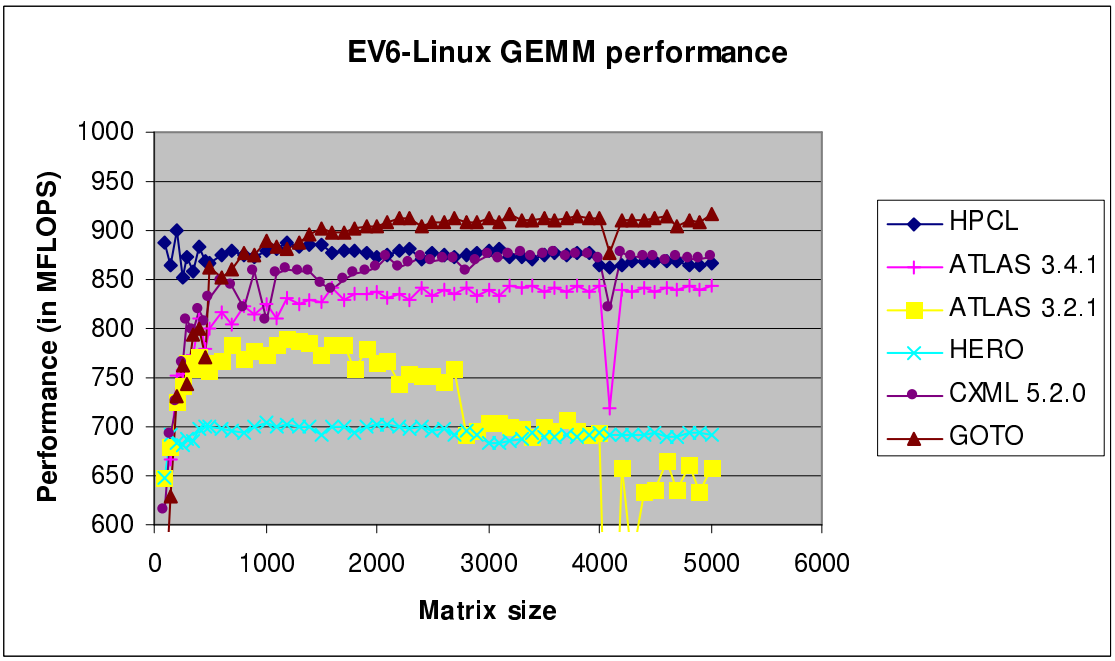


Figure 5.6 Performance of Various Approaches on EV6-Linux

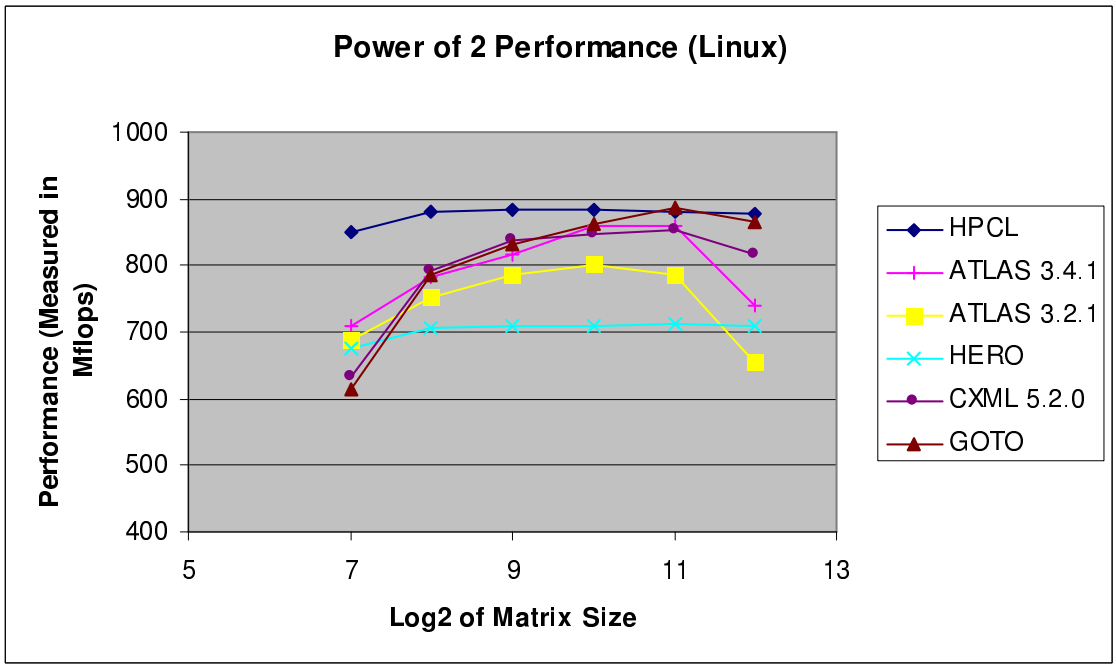


Figure 5.7 Power of 2 Performance of Various Approaches on EV6-Linux

CHAPTER VI

CONCLUSIONS

6.1 Summary

This thesis proved the hypothesis that higher and flatter matrix multiplication performance than the state-of-art implementations is offered by the relatively simple strategies that are refined and synthesized from several recent research approaches. This was demonstrated by the practical benchmark experiments on a modern processor family with well defined performance metrics.

This thesis recognized that the architectural constraints and issues restrict the critical path and options available for optimal performance. Specifically, TLB misses, inter-cache peak bandwidth, and processor computation resources evidently reduce the number of choices available to achieve the optimal performance, This thesis also recognized that the peak kernel efficiency does not necessarily lead to the best overall performance. Also, efficient, flat performance is possible at all problem sizes that fit in main performance, rather than "jagged" performance curves observed in the conventional approaches.

The first chapter presented the hypothesis, motivations and the presumptive contributions of this study. The motivation of this thesis arose mainly from the previous obser-

vations regarding the performance of different architectural components and the overall performance of matrix multiplication.

The literature review was done in the second chapter. The background of matrix multiplication and its importance related to the linear algebra libraries were refreshed. The high performance implementation of dense matrix multiplication is challenged by the complexity of the computer architecture, thus the architectural features that are fundamental to the delivered performance were also studied in details. At last, the second chapter walked through the various related research work, which include the conventional blocking algorithm, the self-tuning strategy, the cache-oblivious algorithm, the hierarchical data structure based framework, and the new variants of the blocking algorithm.

It was concluded from the related work that a combination of recursion strategy and iteration strategy is necessary for better overall performance. The third chapter described such a framework with a qualitatively analysis of the performance tradeoffs made by different optimization methods. This framework consisted of a hierarchical storage format that utilizes the memory-friendly Morton Ordering, an iterative algorithm, and an optimized inner kernel for the Alpha 21264 processor.

In order to carry out an an unbiased performance evaluation of the interested implementations, user-level performance metrics were mathematically defined in the fourth chapter, in which the integration and tuning methodology of this study was also described. This study finally compared and contrasted the hierarchical framework based methods and TLB-blocking based methods that employ the similar inner kernel routines in the fifth

chapter. A performance comparison with the reference libraries (ATLAS and CXML) were also offered on the Alpha 21264 platform. The experimental environment was specifically constructed to obtain trustworthy values of the predefined performance metrics. The benchmarked results clearly showed that the approach of this study achieved higher and flatter performance than the self-tuning library and the vendor library and validated the hypothesis.

6.2 Future Work

This thesis has employed a qualitative model to explain the relationship of performance components and the overall performance and demonstrate the performance tradeoff being made by different approaches. It would be more interesting to explore the possible quantitative modeling of the performance-critical constraints for discovering potential performance improvements and polyalgorithmic tradeoffs. Also, it remains valuable if the knowledges learned in this study can be transferred to the useful pointers for the processor vendors. For example, the specific design strategies for building the next generation of high performance architectures can benefit from such a study on the fundamental architectural constraints and how they affect the performance of critical applications.

Understanding how these methodologies impact rectangular matrix multiplications, parallel matrix multiplication, and algorithmic storage tradeoff (filling the matrices vs. multiplying them) are all valid, interesting next steps.

REFERENCES

[1] "ATLAS Errata," http://math-atlas.sourceforge.net/errata.html#gcc3.0 Date of access: February 20, 2003.

[2] "ATLAS FAQ," http://math-atlas.sourceforge.net/faq.html#auth Date of access: February 20, 2003.

[3] "Automatically Tuned Linear Algebra Software (ATLAS)," http://www.netlib.org/atlas/ Date of access: February 20, 2003.

[4] "Compaq math library," http://h18000.www1.hp.com/math/new/. Date of access: June 15, 2003.

[5] "Earth Simulator," http://www.es.jamstec.go.jp/esc/eng/index.html. Date of access: February 20, 2003.

[6] "GCC Homepage," http://gcc.gnu.org/ Date of access: February 20, 2003.

[7] "Kazhusige Goto's Math Library," http://www.cs.utexas.edu/users/flame/goto/. Date of access: February 20, 2003.

[8] "Personal Communication," Kazushige Goto, March, 2003.

[9] "Sandpile.org, The world's leading source for pure technical x86 processor information," http://www.sandpile.org/ Date of access: February 20, 2003.

[10] "Top 500 Supercomputer Sites," http://www.top500.org/ Date of access: February 20, 2003.

[11] *AMD Athlon Processor x86 Code Optimization Guide*, Advanced Micro Devices, Inc, 2002.

[12] R. Agarwal, F. Gustavson, and M. Zubair, "Exploiting Functional Parallelism of POWER2 to Design High-Performance Numerical Algorithms," *IBM Journal of Research and Development*, vol. 38, no. 5, September 1994, pp. 563–576.

[13] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, "LAPACK: A Portable Linear Algebra Library for High-Performance Computers," *Proceedings Supercomputing '90*, IEEE Computer Society Press, Los Alamitos, California, 1990, pp. 2–11.

[14] E. Anderson, Z. Bai, J. Demmel, J. E. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. E. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK Users' Guide*, SIAM, Philadelphia, 1992.

[15] J. Bilmes, K. Asanović, C.-W. Chin, and J. Demmel, "Optimizing Matrix Multiply Using PHiPAC: A Portable, High-Performance, ANSI C coding methodology," *Proceedings of the International Conference on Supercomputing*, July 1997.

[16] R. D. Blumofe, M. Frigo, C. Joerg, C. E. Leiserson, and K. H. Randall, "An analysis of dag-consistent distributed shared-memory algorithms," *In Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, Padua, Italy, June 1996, pp. 297–308.

[17] S. Chatterjee, V. Jain, A. Lebeck, S. Mundhra, and M. Thottethodi, "Nonlinear Array Layouts for Hierarchical Memory Systems," *Proceedings of the 13th ACM International Conference on Supercomputing (ICS '99)*, June 1999.

[18] S. Chatterjee, A. Lebeck, P. Patnala, and M. Thottethodi, "Recursive Array Layouts and Fast Parallel Matrix Multiplication," *Proceedings of the 11th ACM Symposium on Parallel Algorithms and Architectures (SPAA '99)*, June 1999.

[19] *Alpha 21164 Microprocessor Hardware Reference Manual*, Compaq Computer Corporation, 1999.

[20] *Alpha 21264/EV6 Microprocessor Hardware Reference Manual*, Compaq Computer Corporation, 1999.

[21] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart, *LINPACK Users' Guide*, SIAM, Philadelphia, 1979.

[22] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff, "A Set of Level 3 Basic Linear Algebra Subprograms," *ACM Transactions on Mathematical Software*, vol. 16, no. 1, Mar. 1990, pp. 1–17.

[23] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, "An Extended Set of FORTRAN Basic Linear Algebra Subprograms," *ACM Transactions on Mathematical Software*, vol. 14, no. 1, Mar. 1988, pp. 1–17.

[24] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-Oblivious Algorithms," *40th Annual Symposium on Foundations of Computer Science*, New York, NY, October.

[25] K. A. Gallivan, W. Jalby, U. Meier, and A. H. Sameh, "Impact of Hierarchical Memory Systems on Linear Algebra Algorithm Design," *The International Journal of Supercomputer Applications*, vol. 2, no. 1, 1988, pp. 12–48.

[26] S. Goedecker and A. Hoisie, *Performance Optimization of Numerically Intensive Codes*, Society for Industrial and Applied Mathematics, Philadelphia, 2001.

[27] G. Golub and C. V. Loan, *Matrix Computations*, Third edition, The Johns Hopkins University Press, Baltimore, Maryland, 1996.

[28] K. Goto and R. van de Geijn, *On Reducing TLB Misses in Matrix Multiplication*, Tech. Rep. TR-2002-55, Department of Computer Science, The University of Texas at Austin, November 2002.

[29] B. Greer, J. Harrison, G. Henry, W. Li, and P. Tang, "Scientific Computing on the Itanium Processor," *SC2001*, Denver, Colorado, November, IEEE and ACM.

[30] J. A. Gunnels, G. M. Henry, and R. A. van de Geijn, "A Family of High-Performance Matrix Multiplication Algorithms," *Computational Science - ICCS 2001, Part I*, V. N. Alexandrov, J. J. Dongarra, B. A. Juliano, R. S. Renner, and C. K. Tan, eds. 2001, Lecture Notes in Computer Science 2073, pp. 51–60, Springer-Verlag.

[31] F. G. Gustavson, "New Generalized Data Structures for Matrices Lead to a Variety of High Performance Algorithms," *Proceedings of the IFIP TC2/WG2.5 Working Conference on the Architecture of Scientific Software*, October 2000.

[32] J. L. Hennessy and D. A. Patterson, *Computer Architecture A Quantitative Approach*, Second edition, Morgan Kaufmann Publishess, Inc, San Francisco, California, 1996.

[33] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, "The Microarchitecture of the Pentium 4 Processor," *Intel Technology Journal*, 2001 Q1.

[34] *IA-32 Intel Architecture Software Developer's Manual*, Intel Corporation, 2002.

[35] B. L. Jacob and T. N. Mudge, "A Look at Several Memory Management Units, TLB-Refill Machanisms, and Page Table Organizations," *In the Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, California, 1998.

[36] B. Kågström, P. Ling, and C. V. Loan, *GEMM-Based Level 3 BLAS: High-Performance Model Implementations and Performance Evaluation Benchmark*, Tech. Rep. UMINF-95.18, Department of Computing Science, Umeå University, Sweden, 1995.

[37] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing*, Second edition, Benjamin Cummings/Addison Wesley, 2001.

[38] M. S. Lam, E. E. Rothberg, and M. E. Wolf, "The Cache Performance and Optimizations of Blocked Algorithms," *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, April 1991.

[39] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic Linear Algebra Subprograms for Fortran Usage," *ACM Transactions on Mathematical Software*, vol. 5, no. 3, Sept. 1979, pp. 308–323.

[40] H. Prokop, *Cache-Oblivious Algorithms*, master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institution of Technology, Cambridge, Massachusetts, May 1999.

[41] R. Schreiber and J. Dongarra, *Automatic Blocking of Nested Loops*, Tech. Rep. CS-90-108, Department of Computer Science, University of Tennessee, May 1990.

[42] P. Strazdins, "Transporting Distributed BLAS to the Fujitsu AP3000 and VPP-300," *Proceedings of the Eighth Parallel Computing Workshop (PCW'98)*, Singapore, September 1998, School of Computing, National University of Singapore, pp. 69–76.

[43] V. Valsalam and A. Skjellum, "A Framework for High-performance Matrix Multiplication Based on Hierarchical Abstractions, Algorithms and Optimized Low-level Kernels," *Concurrency and Computation: Practice and Experience*, vol. 14, no. 10, 2002, pp. 805–840.

[44] S. P. VanderWiel and D. J. Lilja, "When Caches aren't Enough: Data Prefetching Techniques," *IEEE Computer*, vol. 30, no. 7, 1998, pp. 23–30.

[45] K. R. Wadleigh and I. L. Crawford, *Software Optimization for High-Performance Computing*, Prentice hall, Upper Saddle River, New Jersey, 2000.

[46] R. C. Whaley, A. Petitet, and J. J. Dongarra, *Automated Empirical Optimization of Software and the ATLAS project*, Tech. Rep. UT-CS-00-448, Department of Computer Science, University of Tennessee, Knoxville, Tennessee, September 2000.

[47] D. Wise and J. Frens, *Morton-order Matrices Deserve Compilers' Support*, Tech. Rep. 533, Department of Computer Science, Indiana University, Bloomington, Indiana 47405-4101, U.S.A., November 1999.

[48] M. E. Wolf and M. S. Lam, "A Data Locality Optimizing Algorithm," *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, June 1991.

APPENDIX

GLOSSARY

ATLAS

Automatically Tuned Linear Algebra Software

BLAS

Basic Linear Algebra Subprograms

CISC

Complex Instruction Set Computers

CXML

Compaq eXtended Math library

DGEMM

Double precision GEneral Matrix Multiplication

DLP

Data Level Parallelism

DoF

Degree of Flatness

FLOPS

FLoating-point Operations Per Second

GCC

GNU Compiler Collection

GEMM

GEneral Matrix Multiplication

HERO

Hierarchical Extension of Recursive Ordering

HDS

Hierarchical Data Structure

ILP

Instruction Level Parallelism

LAPACK

Linear Algebra PACKage

MFLOPS

Millions FLoating-point Operations Per Second

OoO

Out-of-Order execution

OS

Operating System

RISC

Reduce Instruction Set Computers

SIMD

Single Instruction Multiple data

SSE

Streaming SIMD Extensions

TLB

Translation Lookaside Buffer

VLIW

Very Long Instruction Word