PARALLELISM AND ERROR REDUCTION IN A HIGH PERFORMANCE ENVIRONMENT

APPROVED BY SUPERVISING COMMITTEE:

R. Clint Whaley, Ph.D., Chair

Anthony Chronopoulos, Ph.D.

Carola Wenk, Ph.D.

Hugh Maynard, Ph.D.

Lucio Tavernini, Ph.D.

Accepted: Dean, Graduate School
PARALLELISM AND ERROR REDUCTION IN A HIGH PERFORMANCE ENVIRONMENT

by

ANTHONY M. CASTALDO, M.S.

DISSERTATION
Presented to the Graduate Faculty of The University of Texas at San Antonio in Partial Fulfillment of the Requirements for the Degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT SAN ANTONIO
College of Sciences
Department of Computer Science
December 2010
Acknowledgements

I want to thank my advisor, Dr. R. Clint Whaley, for an enormous amount of work on my behalf, and for his excellent academic advice. I want to thank my wife for her support and enthusiasm for my academic pursuits. The research for this work was supported in part by National Science Foundation CRI grants CNS-0551504 and CCF-0833203.

The following two paragraphs are required text by UTSA:

“This Masters Thesis/Recital Document or Doctoral Dissertation was produced in accordance with guidelines which permit the inclusion as part of the Masters Thesis/Recital Document or Doctoral Dissertation the text of an original paper, or papers, submitted for publication. The Masters Thesis/Recital Document or Doctoral Dissertation must still conform to all other requirements explained in the Guide for the Preparation of a Masters Thesis/Recital Document or Doctoral Dissertation at The University of Texas at San Antonio. It must include a comprehensive abstract, a full introduction and literature review, and a final overall conclusion. Additional material (procedural and design data as well as descriptions of equipment) must be provided in sufficient detail to allow a clear and precise judgment to be made of the importance and originality of the research reported.

It is acceptable for this Masters Thesis/Recital Document or Doctoral Dissertation to include as chapters authentic copies of papers already published, provided these meet type size, margin, and legibility requirements. In such cases, connecting texts, which provide logical bridges between different manuscripts, are mandatory. Where the student is not the sole author of a manuscript, the student is required to make an explicit statement in the introductory material to that manuscript describing the students contribution to the work and acknowledging the contribution of the other author(s). The signatures of the Supervising Committee which precede all other material in the Masters Thesis/Recital Document or Doctoral Dissertation attest to the accuracy of this statement.”

December 2010
This dissertation details contributions made by the author to the field of computer science while working to improve the performance of a state-of-the-art linear algebra library intended for use on commodity platforms. These platforms and libraries have become a de facto standard for most scientific researchers. There are three significant contributions to the field. The superblock family of summation algorithms generalizes the dot product and its error analysis, and allows floating point error to be controlled with a desired target bound and performance loss and with workspace determined at compile time. It demonstrates empirically that theoretical error bounds are good predictors of actual errors. The opposite had been asserted by leading experts in the field. The Master Last algorithm identifies and corrects a heretofore unknown performance drain in multi-core parallelism, Parallel Management Overhead (PMO). The author reduced PMO by as much as two orders of magnitude, and sped up QR and LU matrix factorizations (heavily used by researchers) as much as 65%, and asymptotically as much as 45%. Finally, in the Parallel Cache Assignment algorithm, the author realized the goal of exploiting the “collective cache” of all the cores in a multi-core system for a tightly coupled algorithm with high data dependencies requiring extreme synchronization. This allowed parallelization of some heavily used (and heavily researched) operations in linear algebra, panel factorizations, without changing the flop counts or arithmetic, a feat that had defied parallel researchers for many years. The result was superlinear speedup (as much as 19 times faster for an eight core system) for the QR panel factorization over the previous state of the art serial implementation.
# TABLE OF CONTENTS

Acknowledgements ................................................................. iii  
Abstract ................................................................................ iv  
List of Figures ......................................................................... ix  

CHAPTER 1: Introduction .......................................................... 1  

CHAPTER 2: Overview of Presented Research ............................... 2  
2.1 Overview of Superblocking .................................................. 2  
2.2 Overview of the Master Last Algorithm ................................. 2  
2.3 Overview of Parallel Cache Assignment (PCA) ......................... 4  
2.4 Overview Conclusion ........................................................... 7  

CHAPTER 3: The Superblock Family of Algorithms ....................... 8  
3.1 Preface .............................................................................. 8  
3.2 Introduction ........................................................................ 8  
3.2.1 Background and Related work ......................................... 9  
3.2.2 Outline .......................................................................... 10  
3.3 Known Dot Products ............................................................ 11  
3.3.1 Canonical Dot Product ..................................................... 11  
3.3.2 Blocked Dot Product ....................................................... 12  
3.3.3 Stability of the Pairwise Dot Product ................................. 15  
3.4 Superblocked Dot Product .................................................... 17  
3.4.1 Fixed-$N_b$ Superblock ...................................................... 21  
3.5 Statistical Studies ............................................................... 21  
3.5.1 Experimental Methodology .............................................. 22  
3.5.2 Results ........................................................................... 24  
3.6 Summary and Conclusions ................................................... 31
# TABLE OF CONTENTS

## CHAPTER 4: The Master Last Algorithm

4.1 Preface ............................................. 34
4.2 Introduction ....................................... 34
  4.2.1 Terminology and Definitions ............... 35
  4.2.2 Experimental Methodology ................. 37
4.3 Thread Management Techniques ................. 38
4.4 Impact of these techniques on runtime ....... 41
4.5 Computational Expansion ....................... 46
4.6 Cache Perfection ................................ 47
4.7 LAPACK Performance ............................ 52
4.8 Related work ..................................... 54
4.9 Future work ...................................... 54
4.10 Summary and Conclusions ..................... 55

## CHAPTER 5: Parallel Cache Assignment (PCA)

5.1 Preface ............................................. 56
5.2 Overview .......................................... 57
  5.2.1 Basic Terminology ......................... 58
5.3 Introduction ...................................... 58
  5.3.1 Our Approach: Parallel Cache Assignment (PCA) .... 59
  5.3.2 Surveyed Libraries and Important Routines .. 61
  5.3.3 Outline .................................... 62
5.4 QR Matrix Factorization ....................... 63
  5.4.1 The LAPACK Implementation ................ 63
5.5 Applying PCA to QR Panel Factorization .... 65
  5.5.1 Implementation Note: Lightweight Memory Synchronization ... 68
5.6 Experimental Methodology ..................... 69
5.7 Impact of these techniques on runtime ....... 71
  5.7.1 Full Problem Performance ................. 75
5.8 Scaling ........................................... 77
5.9 Extensions to Original PCA Work ............. 78
5.9.1 Super-Linear Speedups ............................................. 81
5.9.2 Core2 Panel Speedups ............................................. 82
5.9.3 Core2 Full Problem Performance ................................. 84
5.9.4 Opt8 Panel Performance ............................................. 87
5.10 Hessenberg Reduction Introduction ................................. 92
5.11 Future work ............................................................. 100
5.12 Summary and Conclusions ............................................. 101

CHAPTER 6: Conclusions 102

Bibliography ................................................................. 103

VITA
LIST OF FIGURES

3.1 Pseudo-code for blocked dot products .................................................. 13
3.2 Pseudo-code and flop exposure of pairwise dot product .......................... 16
3.3 Pseudo-code for Superblock Algorithms .................................................. 18
3.4 “t”—Level Superblock Summation ............................................................ 20
3.5 Average Absolute Error of Tested Algorithms .......................................... 25
3.6 Ratio Showing Reduction in Average Absolute Error ................................. 26
3.7 Ratio of average absolute errors, small problems ...................................... 30
3.8 **Canonical**, Worst Actual Error Vs Upper Bound ..................................... 32
4.1 Parallel Management Overhead in rank-K update for launch and join and persistent worker on a 2.5Ghz **C2** ................................................................. 40
4.2 PMO of rank-K update as a % of runtime for surveyed techniques .............. 42
4.3 Contribution to rank-K speedup by technique .......................................... 44
4.4 Speedup Comparison ................................................................................. 45
4.5 Partitioned Computational Expansion ....................................................... 46
4.6 $P_2$, Probability Matrix for Cache State after Read of $L_2$ ....................... 50
4.7 $P$, Probability Matrix for 4-W ay Cache State after Full Cache-Sized Read . 51
4.8 QR, LU Results with Fully Master Last BLAS ........................................... 53
5.1 LAPACK Blocked QR Factorization ............................................................ 64
5.2 DGEQR2, DLARFT Operational Outline .................................................... 65
5.3 ATLAS QR and LU Panel Speedups ............................................................ 72
5.4 QR Panel Factorization MFLOPS on Core2, Opt8 ..................................... 74
5.5 QR Full Problem Speedups, Core2 and Opt8 ............................................ 76
5.6 QR Scaling, Core2 and Opt8 ..................................................................... 77
5.7 PCA vs. QR2(a), QL2(b), RQ2(c), LQ2(d) on Core2 ................................. 81
5.8 QR, Core2, Panel Speedups ..................................................................... 82
5.9 QL, Core2, Panel Speedups ..................................................................... 83
5.10 QR (a,b,c,d) & QL (a,b,c,d), Core2, Full Problem Performances ............... 85
5.11 RQ (a,b,c,d) & LQ (c,d,e,f), Core2, Full Problem Performances ............... 86
5.12 PCA vs. QR2(a), QL2(b), RQ2(c), LQ2(d) on Opt8 ................................. 88
5.13 QR (a,b,c,d) & QL (a,b,c,d), Opt8, Full Problem Performances ............... 89
5.14 RQ (a,b,c,d) & LQ (c,d,e,f), Opt8, Full Problem Performances .............. 90
5.15 The Dominant GEMV Bottleneck .......................................................... 94
5.16 Raw MFLOP Performance Comparisons ............................................... 97
5.17 Hessenberg Speed Ratios ................................................................. 98
CHAPTER 1: INTRODUCTION

This dissertation follows the arc of the author’s research as a graduate student. The goal of that research was to improve performance in the ATLAS[69] high-performance linear algebra library, and the original contributions grew out of discoveries made in pursuit of that goal. Although these contributions are proven to be useful in the ATLAS context, they are not limited to use in ATLAS, nor limited to use in high performance, for that matter. Rather these are generalized theories and algorithms useful in numerous applications and environments.

The three main chapters of this dissertation will address error reduction, parallel startup costs, and multi-core cache exploitation, in that order. Peer-reviewed papers have been published by the author on each topic[10, 8, 9], the first in the journal SISC and the last two in the conferences IPDPS 2009\(^1\) and PPOPP 2010\(^2\) An additional contribution was made by the author to a fourth paper on accurate performance timing[68]; that contribution allowed the elimination of a dynamically allocated array and more efficient bookkeeping of dynamically allocated aligned memory blocks.

\(^1\)IPDPS 2009 acceptance rate was 100/440 < 23%.
\(^2\)PPOPP 2010 acceptance rate was 29/173 < 17%.
CHAPTER 2: OVERVIEW OF PRESENTED RESEARCH

2.1 Overview of Superblocking

The error analysis of algorithms provides maximum error bounds, and there are cases in high performance computing for which the empirically fastest algorithm is not used due to higher error bounds that threaten the stability of the result. During an investigation of one such case, to determine whether the higher theoretical error bound for a particular algorithm would actually translate into higher actual error, the author realized a way to unify the error analysis of four common approaches to computing dot products, the foundation of much of linear algebra. This produced a family of algorithms we called “superblocking”. Superblocking allows an implementation to increase accuracy at the cost of increased storage space for intermediate results, but not at the cost of increased flops, and the necessary additional space is $O(1)$ and determined at compile time, not runtime. There are many algorithms for increasing accuracy at the expense of flops, but they are frowned upon in high performance computing because using them means the “effective” flop rate cannot ever reach the theoretical peak of the machine. Superblocking sidesteps that issue because it only changes the order of operations, it doesn’t change the number of them. The author proved both better theoretical limits for superblocking and showed empirically better results on average. These results were published in SISC[10] (the SIAM Journal on Scientific Computing).

The development of superblocking produced some wholesale changes in the way ATLAS generates code for matrix multiply (which is essentially a large number of dot products). In particular the superblocking analysis allowed some straightforward changes to the instruction order that reduce the error bounds.

2.2 Overview of the Master Last Algorithm

The Master Last algorithm[8] (IPDPS 2009) is the result of investigating a performance mystery in parallel processing. Certain problems in linear algebra are easily divided into parallel parts with equal work and no coordination or combines necessary other than a final fence. With $p$ cores in a multi-core system these problems should enjoy roughly $p$ speedup, at least asymptotically where

\footnote{Simple summation as well as blocked, two level, and binary tree summations.}
the $O(p)$ costs of partitioning the problem and launching threads are well amortized. However in empirical experiments we frequently saw speedups of only 3 or 4 when we had 8 cores working, even on asymptotically large problems.

To investigate this performance deficit we had to isolate and identify the exact nature of the overhead causing it. We did that with a simplified parallel job. Thousands of automated experiments were run using various job sizes, thread counts, core assignments, core orderings, forced delays and a variety of synchronization methods. One particular result stood out: According to theory, the startup costs for 8 threads should be $O(8)$, a constant. They were not. They were $O(N)$, meaning the amount of overhead was rising with the size of the job.

The problem turned out to be the operating system work scheduler, and how it scheduled threads for execution. This scheduling impediment was observed across multiple hardware platforms and architectures, and exhibited itself in several operating systems, including Linux and Windows. Although they differed, they all interfered with the operation of the Master code. The Master’s job is to partition the problem and then launch the working threads (or schedule work for them if they are permanent or persistent). In order to use all cores and thereby achieve maximum performance, the Master schedules a working thread on every core. Inevitably, work must be scheduled on the same core where the Master code itself is running. When that thread was started the OS would immediately transfer control to it, thereby taking control away from the Master code. Since the Master code assigns the work partitions to the threads and initiates their operation (either through launching them or signalling them), it could not complete this management operation until the worker thread gave up its time slice. That was unlikely to happen because in our experiments the work load was primarily compute bound. The larger the worker’s share (which is a function of $N$) the longer it would take for it to stall or finish its time slice, and the longer it would take for the Master to finally complete its thread management tasks.

Under the constraints of working in a library for use by other applications, we could not control which core of a multi-core system would execute the Master code\textsuperscript{4}. However our code could read the ID of the core it was executing on, and explicitly schedule the work for that core last.

\textsuperscript{4}Because the library code is called as a subroutine pinning the Master code to a specific core would pin the caller’s code too, which is not just “unfriendly” but could, in theory, cause a deadlock or failure in the caller’s algorithm if it were already pinned elsewhere, or reliant upon independent execution by another core, or had code dependencies related to its core ID.
Then transferring control to that thread creates no delays, because the Master is just waiting for its workers to finish anyway. The “Master Last” algorithm overcame the OS-induced problem. It produced a small constant for overhead, as theory demanded, and significantly improved the parallel performance of both matrix multiply and some heavily used applications that rely upon matrix multiply, such as LU and QR matrix factorizations. Although this parallel management overhead may sound like a minor consideration, it was actually a major performance impediment. With the Master Last algorithm implemented throughout ATLAS, performance on LU and QR factorizations were improved as much as 50% on the LU factorizations, and as much as 65% on the QR factorizations for mid-range problems ($N \approx 3000$). Asymptotically, for large problems ($N \approx 8000$) the improvements due to the Master Last algorithm were as much as 25% and 50% on LU and QR respectively.

As noted, the problem Master Last solves was observed on several operating systems, including multiple versions of Linux and Windows, to varying degrees. We believe the underlying issue is unlikely to be addressed by OS code engineers for various reasons, essentially because the applications they are targeting are not typically high performance computational problems. Thus we expect the Master Last algorithm to be a critical ingredient for parallel performance in high performance computing for the foreseeable future. These results were presented at IPDPS-2009 in Rome, Italy [8].

### 2.3 Overview of Parallel Cache Assignment (PCA)

The author’s advisor suggested developing a way to better exploit the “collective” cache of the common commodity multi-core architecture, in which each core has its own cache. These caches are well employed when problems are easily partitioned into largely independent parts with low levels of inter-dependency (and thus infrequent synchronization and combine costs). For example the central kernel in ATLAS is GEMM (General Matrix Multiply), which can be so partitioned for large enough problems, and large parallelized GEMMs can run within a few percent of the $p$–core peak. In such cases the overhead of OS synchronization function calls is easily paid for by the parallelism and cache re-use enjoyed.

However for problems that require frequent synchronization, the typical approach of using Operating System provided routines results in excessive overhead. The invocation of the synchroniz-
ing function typically requires building a stack frame and undergoing at least two context switches (to the OS and back to the caller). Once the OS has control it will often take the opportunity to terminate the time slice of the caller for hundreds of microseconds or even milliseconds, passing control to another OS task or a different user application, either of which can pollute the cache of the target application (i.e. causes cache evictions that produce later cache misses for the target application).

Although these costs are easily amortized when used sparingly, parallelizing a tightly coupled algorithm like LAPACK’s QR panel factorization requires synchronizing every few microseconds, or even every few hundreds of instructions. The QR panel factorization (the panel is a tall, thin matrix) has an ironclad data dependency between columns of the panel: The complete results of computation on column $i$ must be used to transform column $(i+1)$ before it can be processed. The processing of a single column by multiple cores demands six synchronization points. This can result in tens of thousands of synchronizations in a single time slice; and under such circumstances the OS provided synchronization methods (condition variables, semaphores, etc.) have such prohibitive overhead they would destroy any speedup enjoyed by parallelism or cache re-use.

The author’s original contribution in this area is synthesis: Putting together various previous techniques in order to produce a way forward in parallelizing a very tightly coupled algorithm. The work on Master Last provided the introduction to the problems in QR Panel factorization; while previous work experience in the embedded systems field provided the means to synchronize cores without any use of the operating system\footnote{Many embedded systems have no operating system, the boot code is the entire application. The author worked on several such systems, some with multiple discrete processors synchronizing through dual-ported memory.}. Finally, the author’s work on error analysis ensured that parallelizing the panel in the form we envisioned would produce a slightly lower error bound than the original serial method.

There are two central ideas that power Parallel Cache Assignment (PCA):

1. A cache-contained data ownership model of parallelization; i.e. each core “owns” a block of the data that fits in some level of its cache, and that core is responsible for all operations on that data.

2. A hardware-speed synchronization method (i.e. one that does not rely on the operating system) that makes synchronization inexpensive enough that, if necessary, it can be performed
several times in the innermost nested loops of the algorithm.

Multi-core systems share a global memory and the hardware ensures in-order writes and cache-coherence\(^6\). This lets us use a hardware-based synchronization algorithm that does not involve the OS and operates reliably at essentially the speed of the databus; which is on the order of nanoseconds per read. This allowed us to synchronize cores as often as desired in a few dozen nanoseconds using hand-coded spin locks with minimal overhead.

Our initial target for this operation was again matrix factorizations; the ones we have addressed are the most commonly used: LU, QR, and Hessenberg. We were interested in parallelizing the panel operations. The reason for this target is that the panel operations are a looming bottleneck in these factorizations, and are highly dependent on Level 1 \(O(N)\) and Level 2 \(O(N^2)\) linear algebra operations that are notoriously difficult to efficiently parallelize. All of these algorithms (LU, QR and Hessenberg) have the aforementioned ironclad column-by-column data dependencies.

PCA assigns roughly equal blocks of the panel to each core in a multi-core system, just enough to fit into each core’s Level 2 cache. The panel width is arbitrary, so depending on the number of rows in the matrix we choose a width that allows the panel to fit into the collective cache of the cores. Without changing either the mathematics of the factorization or the data dependencies (except for blocking), all cores cooperate to complete the factorization, synchronizing as often as the particular algorithm requires.

In all cases (QR, LU and Hessenberg), the entire panel factorization requires three nested loops, and we are synchronizing several times in the innermost loop of the operation.

To the experienced high-performance practitioner this is an extremely counter-intuitive approach; if OS calls were used to accomplish the synchronization it would indeed be prohibitively time expensive. However with a hardware-based synchronization, the total synchronization cost is far outweighed by the gains in both parallelization and cache re-use. We proved this empirically by coding both QR and LU panel factorizations, speeding up the panel factorizations almost 20 fold. We showed strong and weak scaling for the PCA version, an important consideration since the previous serialized panel factorization was already becoming an increasingly larger bottleneck as the number of cores available in commodity systems grew: What was once 4% of the total factorization

\(^6\)Cache coherence ensures that if core A has cached a global memory location M, and that memory is changed by core B, then the cached version of M in core A’s cache is invalidated, so that if core A does try to read it the core will be forced to fetch the updated version from actual memory.
time in serial had grown to 20% of the factorization time when the non-panel operations were done in parallel\textsuperscript{7}. We also showed that when applied to the full factorization problem, the PCA panel factorization made the \textit{complete} factorizations as much as 35% faster, on a relatively small scale (8-core) multi-core system. We expect this will have a larger impact as the number of cores in such systems increases, as current trends indicate.

This work was peer reviewed and presented at PPOPP-2010 in Bangalore, India [9]. We have extended PCA to all precisions (single, single complex, double and double complex) and four variants of QR, as well as the Hessenberg reduction in double precision. These extended results will be shown in this dissertation. The Hessenberg reduction is an important tool in linear algebra; it is the first major computational step in discovering eigenvalues and eigenvectors of a non-symmetric matrix. We are in the process of producing an extension paper to publish as a journal article, and parts of this dissertation will be used in that paper.

\section{2.4 Overview Conclusion}

The author has made three significant original contributions: In error analysis (with superblocking), in systems research (with Master Last), and in developing a new parallelization technique with PCA, to address highly data-dependent algorithms on multi-core systems. The following chapters discuss these contributions in detail.

\textsuperscript{7}Because in the latest LAPACK the non-panel operations are much faster Level 3 ($O(N^3)$) operations with a parallel BLAS, while the panel factorization is still done by a single core working alone.
CHAPTER 3: THE SUPERBLOCK FAMILY OF ALGORITHMS

3.1 Preface

A version of this work appeared as the author’s Master Thesis, and also in the SIAM Journal on Scientific Computing (SISC)[10]. The author was the sole inventor of the superblock algorithm and all the proofs accompanying it. The author wrote the necessary code for the empirical experiments, ran them all, and produced the graphs. The author also devised the appropriate statistical approach for the comparison of algorithms. Co-authors Drs. Whaley and Chronopolous provided focus discussions, writing and formatting aids, and verified all of the author’s mathematical proofs.

3.2 Introduction

A host of linear algebra methods derive their error behavior directly from dot product. In particular, most high performance dense systems derive their performance and error behavior overwhelmingly from matrix multiply, and matrix multiply’s error behavior is almost wholly attributable to the underlying dot product that it is built from (sparse problems usually have a similar relationship with matrix-vector multiply, which can also be built from the dot product). With the expansion of standard workstations to 64-bit memories and multicore processors, much larger calculations are possible on even simple desktop machines than ever before. Parallel machines built from these hugely expanded nodes can solve problems of almost unlimited size. The canonical dot product has a worst-case error bound that rises linearly with vector length. In the past this has not been deemed intolerable, but with problem sizes increasing it becomes important to examine the assumption that a linear rise in worst-case error is tolerable, and whether we can moderate it without a noticeable loss in performance.

Dot product is an important operation in its own right, but due to performance considerations linear algebra implementations only rarely call it directly. Instead, most large-scale linear algebra operations call matrix multiply (AKA: GEMM, for general matrix multiply) [2, 4], which can be made to run very near the theoretical peak of the architecture. High performance matrix multiply can in turn be implemented as a series of parallel dot products, and this is the case in our own ATLAS [72, 71] project, which uses GEMM as the building block of its high performance
BLAS \([32, 44, 17, 18, 16]\) implementation. Therefore, we are keenly interested in both the error bound of a given dot product algorithm, and whether that algorithm is likely to allow for a high performance GEMM implementation. The implementation and performance of GEMM are not the focus of this chapter, but we review them for each algorithm briefly, to explain why certain formulations seem more promising than others.

### 3.2.1 Background and Related work

Because dot product is so important to the error analysis of linear algebra, it has been well studied; Probably the main reference for linear algebra error analysis in general is Higham’s excellent book\([36]\), which extended the foundation provided by Stewart in \([60]\). We will therefore adopt and extend the notation from \([36]\) for representing floating point rounding errors:

\[
fl(x \circ y) = (x \circ y) \cdot (1 + \delta), \quad \text{with } |\delta| \leq u, \quad (3.2.1)
\]

where (i) \(\circ\) is \(\oplus, \ominus, \otimes, \oslash\) for floating-point (as opposed to exact) add, subtract, multiply or divide operations; (ii) \(u\) is the unit roundoff error, defined as \(u = \frac{1}{2}\beta^{1-t}\); (iii) \(\beta\) is the base of the numbers being used, and (iv) \(t\) is the number of digits stored. Also, we assume that \(|\delta| \leq u\) and that \(\delta_{\text{anything}}\) is reserved notation for values such that \(|\delta_{\text{anything}}| \leq u\). By the IEEE floating point standard, for single precision \(u = 2^{-24} \approx 5.96 \times 10^{-8}\), and for double precision \(u = 2^{-53} \approx 1.11 \times 10^{-16}\). This model presumes that a guard digit is used during subtraction, which is a required feature of IEEE floating point arithmetic.

The floating point computations in dot product are multiplication and addition. We will see that the multiplicative error does not compound in dot product except through accumulation, and hence the main algorithmic opportunity for error reduction comes in strategies for summing the individual elementwise products. Therefore, the most closely related work is on reducing error in summations, as in \([48, 27, 47, 55, 53, 23, 3, 14, 52, 28]\). In this chapter we use Stewart’s \(\langle k \rangle\) “error counter” notation \([60]\):

\[
\langle n \rangle := \prod_{i=1}^{n} (1 + \delta_i)^{\rho_i}, \quad \text{with } \rho_i = \pm 1, \quad |\delta_i| \leq u.
\]

Multiple \(\langle k \rangle\) may represent distinct sets of \(\delta_i\) and \(\rho_i\) and are not necessarily equal to each other (thus cannot be multiplicatively factored out). These terms can be combined and manipu-
lated as follows, where $x_1$ and $x_2$ are floating point numbers:

1. $x_1 \langle p \rangle \odot x_2 \langle m \rangle = (x_1 \cdot x_2) \langle p + m + 1 \rangle$, and
2. $x_1 \langle p \rangle \oplus x_2 \langle m \rangle = (x_1 + x_2) \langle \max(p, m) + 1 \rangle$

(i.e., given any $\langle p \rangle$ and $\langle m \rangle$ there exists a set of $\delta_i$ s.t. $|\delta_i| \leq u$ and $\rho_i = \pm 1$ forming a $\langle \max(p, m) + 1 \rangle$ that satisfies (2)). (2) is the key to algorithmic reduction in error: if we can evenly balance the size of $p$ and $m$ during addition, we can minimize the resulting error bound. Conversely, if $m = 1$ and $p = i$, as in canonical summation (where $i$ is the induction variable), then the error bound is maximized. We further show empirically that the algorithm with the least error bound usually produces the least error.

Higham (pg 69 of [36]) provides bounds for $\langle n \rangle$ (ignoring the possibility of overflow or underflow):

**Lemma 3.2.1.**

$$|\delta_i| \leq u, \rho_i = \pm 1, nu < 1, \prod_{i=1}^{n} (1 + \delta_i)^{\rho_i} = 1 + \theta_n; |\theta_n| \leq \frac{nu}{1 - nu} =: \gamma_n. \quad (3.2.2)$$

Noting that equality holds only for $|\theta_1|$.

This chapter is concerned only with algorithmic improvements for controlling error suitable to high-performance implementation. A related and orthogonal approach is using extra and/or mixed precision arithmetic to reduce error, as in [14, 15, 45], but such approaches often present additional computational demands, and/or architecture-specific coding techniques, and so they are not the focus of this chapter. The broader effects of floating point error in linear algebra is also too large a pool of literature to survey in detail, but more information can be found in the overview texts [60, 36], and in [12, 43].

### 3.2.2 Outline

The remainder of this chapter is organized in the following way: Section 3.3 surveys some known dot product implementations, including their error bounds, while Section 3.4 introduces a general class of dot product algorithms we call **superblock**, which we believe is new. Section 3.5 then shows some results from our statistical studies of these algorithms, which will allow us to draw some conclusions about these techniques, the most important of which are summarized in Section 3.6.
3.3 Known Dot Products

In this section we overview several dot products of interest. Note that we are primarily interested in dot products that could likely be extended into high performance GEMM implementations. Since GEMM has $\mathcal{O}(N^3)$ floating point operations (flops) and $\mathcal{O}(N^2)$ memory use, after tuning its performance is typically limited by the amount of computation to be done, and therefore we do not consider methods requiring any additional flops (e.g., compensated summation, or the algorithms described in [61]). For performance reasons, we also avoid sorting the vectors of each individual dot product as discussed in [53, 13, 52]. Finally, we do not consider using extra or mixed precision, as both the performance and accuracy of such algorithms is strongly influenced by the architecture and compiler, and our focus here is on general algorithmic strategies.

Therefore, we present and analyze three known methods in this section, including comments indicating their suitability as a building block for high performance GEMM implementation. Section 3.3.1 discusses canonical dot product, Section 3.3.2 surveys two versions of the blocked dot product, and Section 3.3.3 presents pairwise dot product.

3.3.1 Canonical Dot Product

The canonical algorithm is:

```
for (dot=0.0,i=0; i < N; i++) dot += X[i] * Y[i];
```

which calculates the dot product for two $n$-dimensional vectors, $x = \{x_i\}_{i=1}^n$ and $y = \{y_i\}_{i=1}^n$ in the order

$$((\ldots((x_1 \odot y_1) \odot x_2 \odot y_2) \odot x_3 \odot y_3) \odot \ldots) \odot x_n \odot y_n) \iff
(x_1 \cdot y_1)\langle n \rangle + (x_2 \cdot y_2)\langle n \rangle + (x_3 \cdot y_3)\langle n - 1 \rangle + \ldots + (x_n \cdot y_n)\langle 2 \rangle$$

In [36] and much of the summation literature, this method is called the recursive algorithm.

Since the pairwise algorithm (surveyed in Section 3.3.3) is naturally implemented using recursion

---

8 Strassen’s partitioning can reduce the order of the floating point operations to $\approx \mathcal{O}(N^{2.807})$ and various refinements can reduce the exponent even further, but at the cost of increasing the error bound significantly. The forward error bound for $N \times N$ matrix multiply using a canonical dot product is $\mathcal{O}(N)$. In Higham’s numerical stability analysis (pg 359 [35]), he arrives at a bound (depending upon the level where recursion stops) of $\mathcal{O}(3N^2 + 25N)$ (for one level of recursion) to $\approx \mathcal{O}(6N^{3.585} - 5N)$ (for full recursion). Thus Strassen’s and refinements thereof are not considered in this chapter, since our aim is reducing error. The superblock technique described herein may well reduce these error bounds to make Strassen’s more acceptable, but that is a line of inquiry we do not address here.
and this method is naturally implemented using a simple iterative loop, we avoid this name and refer to this algorithm, which is certainly the most widely used in practice, as canonical. The forward error bound is given on page 69 of [36] as: For \( s = x^T y \) and \( \hat{s} = x^T \odot y \),

\[
|s - \hat{s}| \leq \gamma_n |x^T y|.
\] (3.3.1)

For insight into this result, notice that the subscript on the \( \gamma \) term is the greatest number of flops to which any given input is exposed; this is the maximum error counter \( \langle \cdot \rangle \). This leads us to the observation that different algorithms distribute flops differently over their inputs, and thus the more uniformly an algorithm distributes flops over inputs the better it will be on worst-case error. Dot product forward error bounds are all of this general form; for expositional simplicity we shall abuse notation by referring to the error bound by the \( \gamma \) element that changes from algorithm to algorithm. Thus we shall say the canonical algorithm has a \( \gamma_n \) error bound, or in a further abuse of notation to ease exposition, an \( O(n) \) error bound.

**Implementation notes**  Canonical dot product is the usual starting point for optimized block products. A host of transformations can easily be performed on canonical product (unrolling, pipelining, peeling, vectorization, prefetching, etc.). SIMD vectorization (as is used in architectural extensions like SSE or 3DNow!) in particular is equivalent to a “post-load” blocked dot product, as discussed next in 3.3.2, and will approximately divide the error bound by the vector length, e.g., if \( n \) is a multiple of the vector length of 4, a typical vectorization will produce an error bound of \( \gamma(n/4+3) \).

Canonical dot product is almost never used to directly build a high performance GEMM, since it fails to efficiently use the memory hierarchy. Dot product has no opportunity for cache reuse, but GEMM does. Therefore, when parallel dot products are used to implement matrix multiply, they are typically blocked to encourage cache reuse, and this type of dot product is discussed in Section 3.3.2.

### 3.3.2 Blocked Dot Product

For some optimizations it is necessary to block operations into chunks that make good use of the various levels of local memory that exist on computers. For a dot product of two vectors of large dimension \( N \), this implies breaking up the vectors into \( N_b \)-sized subvector chunks that are computed separately, then added together. There are two obvious algorithms for blocked dot product, which
we call pre-load and post-load; we show that post-load is strongly preferable to the pre-load due to error growth. Figure 3.1 gives pseudo-code for both versions of the algorithm (we assume $N$ is a multiple of $N_b$ for expositional simplicity throughout this section).

\[ s = 0.0 \]
\[ blocks = \frac{N}{N_b} \]

\[
\text{for}(b = 0; b < \text{blocks}; b++)
\{
\begin{align*}
\text{for}(i = 0; i < N_b; i++) & \quad \text{sb} = (x[0] \odot y[0]) \\
\text{for}(i = 1; i < N_b; i++) & \quad sb = sb \oplus (x[i] \odot y[i]) \\
\text{for}(i = 1; i < N_b; i++) & \quad s = s \oplus sb \\
x += N_b; y += N_b & \quad x += N_b; y += N_b
\end{align*}
\}
\]

\[
\text{return}(s) \quad \text{return}(s)
\]

(a) Pre-load blocked dot product (b) Post-load blocked dot product

Figure 3.1: Pseudo-code for blocked dot products

The pre-load algorithm of Figure 3.1(a) is probably the most obvious implementation. However, it is not optimal error wise. The term $s$ is used in every intermediate computation, so the error term on $s$ will dominate the total error. The first add to $s$ is an add to zero that doesn’t cause error. There are $N - 1$ adds to $s$ producing $\gamma_{N-1}$ error bound, combined with the $\gamma_1$ error bound from the multiply, which produces for the pre-load blocked dot product a $\gamma_N$ error bound, the same error bound as canonical.

Now consider a slight alteration to this algorithm, as shown in Figure 3.1(b). Instead of accumulating on a single value throughout the computation, we accumulate the dot product for each block separately, and then add that result to $s$. So the blocked dot product consists of $\frac{N}{N_b}$ canonical dot products each of size $N_b$, each of which then adds to the total sum. In [36] (p70) Higham notes the forward error bound for this algorithm is now:

\[
|s - \hat{s}| \leq \gamma_{\frac{N}{N_b} + N_b - 1} |x|^T |y| \quad (3.3.2)
\]

If we assume a fixed $N_b$, post-load reduces the error bound by a constant factor which depends on $N_b$. The minimum value is found by looking at the $\gamma$ subscript as a function of $N_b$,
say \( f(N_b) = N_b + \frac{N}{N_b} - 1 \). As Higham notes the minimum occurs\(^9\) at \( N_b = \sqrt{N} \), yielding an error bound of \( \gamma_{(2\sqrt{N} - 1)} \); thus changing the order of the error. Extending this procedure to an arbitrary number of levels of blocking is the key idea behind superblocking.

**Implementation notes** Most high performance GEMM implementations use one of these blocked algorithms, where the \( N_b \) value is chosen based on the size of one of the caches and other architectural features (such as the number of floating point registers, instruction cache size, etc.). It is perhaps not obvious, but the post-load algorithm requires no extra storage when used in GEMM. When extended to GEMM, the scalar accumulators used by the dot product algorithm naturally become output matrices. However, these output matrices must be loaded to registers to be operated on, and thus the architecture provides a set of temporaries that are not present in storage. This is indeed where the algorithms get their names: in pre-load, the summation-so-far is loaded to the registers before beginning the loop indexing the common dimension of the input matrices, but in post-load the summation is not loaded until that loop is complete. Therefore, whether the pre-load or post-load algorithm is used varies by library (ATLAS mostly uses the pre-load algorithm at present); indirect experience with vendor-supplied BLAS seems to indicate that many use the pre-load version, but it is impossible to say for sure what a closed-source library does algorithmically. However, personal communication with Fred Gustavson (who is strongly involved in IBM’s computational libraries) indicates that at least some in the industry are aware of the error reductions from post-load, and have at least historically used it.

In a high performance library \( N_b \) is chosen to optimize performance and can be heavily architecture dependent, and so it cannot typically be varied exactly as called for to get the \( \gamma_{(2\sqrt{N} - 1)} \) bound. In our own ATLAS (and we suspect in other libraries as well), \( N_b \) is either completely fixed or selectable from a small set of values that obtained the best performance. However, even libraries with a fixed \( N_b \) can produce lower-order worst-case errors for a reasonable range of problems sizes, and those that can choose amongst several candidate \( N_b \)’s can do even better, as we outline next in Section 3.3.2.

\(^9\) \( f'(N_b) = 0 \Rightarrow -N \cdot N_b^{-2} + 1 = 0 \Rightarrow N \cdot N_b^{-2} = 1 \Rightarrow N_b = \sqrt{N} \)
Optimal and Near-Optimal Blocking for Post-Load Dot Product

Post-load blocked dot product has the error bound given in (3.3.2), but to achieve this bound \( N_b \) must be variable, and we believe in most implementations it is not. What we show here is that there is wide latitude in choosing \( N_b \), so a few fixed block sizes can achieve the \( \mathcal{O}(\sqrt{N}) \) error bounds across an expansive range of problem sizes. More formally, suppose for some small constant \( c \), 
\[
N_b = c \cdot \sqrt{N}.
\]
Then the error factor on each term will be
\[
\gamma(c \cdot \sqrt{N} + \frac{N}{c \cdot \sqrt{N}} - 1) = \gamma((c + \frac{1}{c})\sqrt{N} - 1).
\]

Implementation note The utility of this modified bound becomes clear with an example. Suppose we have a GEMM with an \( N_b = 60 \), a typical value used by ATLAS. This is the perfect block size (lowest possible error bound for one level of blocking) when \( N = 3600 \). But it achieves nearly the same error bound for all \( N \in [900, 14400] \): If \( N = 900 \) the optimal blocking factor is 30, so at this extremity, \( N_b = \frac{1}{2} \sqrt{N} \), so by 3.3.3 we have an error bound of \( \gamma(2.5\sqrt{N} - 1) \). At the other extremity, the optimal blocking factor for \( N = 14400 \) is 120. Then \( N_b = 2\sqrt{N} \), so by (3.3.3) we again have an error bound of \( \gamma(2.5\sqrt{N} - 1) \). This is only about 25% higher than if we were able to use the optimal values of 30 and 120, and no \( N \in [900, 14400] \) has a worse bound. If we wanted \( \mathcal{O}(\sqrt{N}) \) bounds on much larger \( N \), a second choice of \( N_b' = 240 \) would seamlessly cover the adjacent range \( N = 14400 \) to \( N = 230400 \) with the same bound, \( \gamma(2.5\sqrt{N} - 1) \). Any \( N_b' \in (60, 240] \) would overlap the range \( [900, 14400] \); in such cases \( N_b \) or \( N_b' \) could be chosen, either the one producing the lesser bound or by performance related criteria. Because of this wide latitude in choosing \( N_b \), one may imagine a GEMM implementation in which a small handful of block sizes with overlapping ranges are chosen in order to allow \( \mathcal{O}(\sqrt{N}) \) error bounds across the full range of practical problem sizes, with little to no impact on even the most highly tuned performance.

3.3.3 Stability of the Pairwise Dot Product

Finally, we consider the pairwise algorithm, which can be naturally implemented using recursion, as shown in Figure 3.2(a). This algorithm performs precisely as many flops as the canonical form, but instead of accumulating the products one by one, it constructs a \( \lceil \log_2(N) \rceil \) deep binary tree of them, as shown in Figure 3.2(b). Thus this algorithm has the property of distributing the flop load
Scalar dotProduct
(int n, Vector X, Vector Y)
{
    scalar sum;
    int n1 = n/2;
    int n2 = n - n1;
    if (n == 1) return (X[0] * Y[0]);
    sum = dotProduct(n1, X[0:n1-1], Y[0:n1-1]);
    sum += dotProduct(n2, X[n1:n-1], Y[n1:n-1]);
    return (sum);
} // END *** dotProduct ***

(a) Pairwise pseudo-code

(b) Pairwise summation diagram

Figure 3.2: Pseudo-code and flop exposure of pairwise dot product

Pairwise demonstrates an instance where recursion, by distributing usage of prior results uniformly, inherently improves the error bound of the result. In general, this is a powerful principle: The fewer flops elements are subjected to, the lower the worst-case error. We will demonstrate in Section 3.5 that these lower worst-case error algorithms do indeed produce lower actual errors on average than canonical.

Implementation notes: Despite its superior error bound, this algorithm has several drawbacks that prevent it from being the default dot product for performance-aware applications. First, the general recursive overhead can be too expensive for most applications. Second, the smaller sizes found towards the bottom of the recursion prevent effective use of optimizations such as unrolling, pipelining, and prefetch. These optimizations often must be amortized over reasonable length vectors, and for optimizations such as prefetch, we must be able to predict the future access pattern. Straightforward recursive implementation will limit or completely remove the freedom to perform these optimizations, and so it is generally much less optimizable than a loop-based implementation, even when the recursive overhead can be minimized. We note that the naive version of this algorithm requires $\frac{n}{2}$ workspaces (stored on the stack in the recursive formulation).
to store the partial results. Generally, extra workspace usage results in greater cache pollution, which tends to degrade performance even further. With smarter accumulator management, we may reduce the workspace requirements to \((1 + \log_2(n))\) (see [7] for details). We will derive a similar result using our \text{superblock} algorithm, in Section 3.4. However, since in matrix multiply these workspaces must be matrices (as opposed to scalars for dot product), \text{pairwise} is usually not practical due to memory usage, even if the performance considerations highlighted above do not discourage its use.

### 3.4 Superblocked Dot Product

The error bound on a dot product is primarily due to summation; the initial multiplication of elements only adds one to the final \(\gamma\) subscript. Thus we begin both our analysis and introduction with \text{superblock} summation: Given \(t\) temporaries we can accumulate a sum in \(t\) levels; so our initial idea was to find optimal blocking factors for each level to minimize the upper bound of the floating point error. We discovered the ideal blocking factor is identical for all levels, which also simplifies the error bound. The code for this generalized \text{superblock} dot product is given in Figure 3.3(a), and an illustration of the summation part of the algorithm is shown in Figure 3.4. In Figure 3.4 note that each level only needs one temporary, so all shown summations are using the same workspace. Therefore, after level \(t-1\) receives its \(N_b^{th}\) addition into the first running sum shown on the left of level \(t-1\) of Figure 3.4, the \(t-1\) sum is added into the current \(t-2\) sum, zeroed, and the next summation shown to the right is begun using the same workspace. The principle is straightforward; on each level the blocking factor \(N_b\) is identical\(^{10}\), and for \(t\)-level summation this requires \(N_b = N^{1/t}\).

When there is only one level \((t=1)\) this algorithm becomes \text{canonical}, for \(t=2\) it becomes \text{post-load blocked}, and for \(t=\log_2(N)\) it becomes space-optimal \text{pairwise} (as shown in Proposition 3.4.3). Therefore, all the algorithms surveyed in Section 3.3 may be viewed as special cases of the \text{superblock} class of dot products. In proposition 3.4.1 we first prove the \text{superblock} summation result and its error bound, then in proposition 3.4.2 we extend that result to the \text{superblock} dot product.

\(^{10}\)For clarity we assume the computation produces an integer blocking factor.
scalar dotProd(Vec X, Vec Y, int t)
{
    int n = X.length;
    int nb = pow(n, 1/t);
    scalar tmp[t] = {0.0};
    int cnt[t] = {0};

    for (i=0; i < n; i++)
    {
        tmp[t-1] += X[i] * Y[i];
        if (++cnt[t-1] == nb)
        {
            for (j=t-2; j; j--)
            {
                tmp[j] += tmp[j+1];
                tmp[j+1] = 0.0;
                cnt[j+1] = 0;
                if (++cnt[j] < nb) break;
            }
        }
    }
    return(tmp[0]);
}

scalar dotProd(Vec X, Vec Y, int nb)
{
    int n = X.length;
    int nblks = n/nb;
    int nsblks = sqrt(nblks);
    int blksInSblk = nblks/nsblks;
    scalar dot=0.0, sdot, cdot;

    for (s=0; s < nsblks; s++)
    {
        sdot = 0.0;
        for (b=0; b < blksInSblk; b++)
        {
            cdot = X[0] * Y[0];
            for (i=1; i < nb; i++)
            {
                cdot += X[i] * Y[i];
                sdot += cdot;
                X += nb; Y += nb;
            }
            dot += sdot;
        }
    }
    return(dot);
}

(a) t-level superblock

(b) 3-Level Fixed-$N_b$ superblock

Figure 3.3: Pseudo-code for Superblock Algorithms
Proposition 3.4.1. For a \( t \) temporary superblock summation, the blocking factor that minimizes the worst-case error counter is \( N^{\frac{1}{t}} \), which produces a worst-case error counter of \( \left\langle t(N^{\frac{1}{t}} - 1) \right\rangle \), corresponding to an error bound of \( \gamma_{(t(N^{\frac{1}{t}} - 1))} \).

Proof. The proof is inductive on \( t \). The proposition is trivially true for \( t = 1 \), producing a worst case error counter of \( \left\langle N - 1 \right\rangle = \langle N - 1 \rangle \).

Therefore assume the proposition is true for \( t \). For \( t + 1 \) level blocking we choose \( N_b \) as the lowest level blocking factor. This will produce a worst case error counter of \( \left\langle N_b - 1 \right\rangle \) on each block summation, with \( \frac{N}{N_b} \) elements (the block sums) to be added in the subsequent \( t \) levels of the addition. By the proposition, the ideal blocking factor for these remaining \( t \) levels is \( \left( \frac{N}{N_b} \right)^{\frac{1}{t}} \), and will produce a worst case error counter of \( \left\langle t \left( \left( \frac{N}{N_b} \right)^{\frac{1}{t}} - 1 \right) \right\rangle \), which shall be added to \( \langle N_b - 1 \rangle \) to find the worst case error counter of a \( t + 1 \)-level superblock summation. This can be expressed as \( \langle f(N_b) \rangle \) with \( f(N_b) \) shown in (3.4.1a). We minimize this and solve for \( N_b \), as shown in (3.4.1b).

\[
f(N_b) = N_b - 1 + t \left( \left( \frac{N}{N_b} \right)^{\frac{1}{t}} - 1 \right)
\]  

(3.4.1a)

\[
f'(N_b) = 1 + t \left( N^{\frac{1}{t}} \times \frac{1}{t} \times N_b^{-\frac{1}{t} - 1} \right) = 0 \Rightarrow 1 = N^{\frac{1}{t}} \times N_b^{-\frac{t+1}{t}} \Rightarrow N_b = N^{\frac{1}{t+1}}
\]  

(3.4.1b)

Thus the optimal \( N_b \) for level \( t + 1 \) is \( N^{\frac{1}{t+1}} \), leaving \( \frac{N}{N_b} = \frac{N}{N^{\frac{1}{t+1}}} = N^{\frac{t}{t+1}} \) elements to be summed in \( t \) levels. By our assumption the optimal blocking factor for these elements is \( \left( \frac{N}{N^{\frac{1}{t+1}}} \right)^{\frac{1}{t}} = N^{\frac{t}{t+1}} \), so the same blocking factor is used on all \( t + 1 \) levels. Substituting this \( N_b \) in (3.4.1a) gives the worst case error counter of

\[
\left\langle N^{\frac{1}{t+1}} - 1 + t \left( \left( \frac{N^{\frac{1}{t+1}}} \right)^{\frac{1}{t}} - 1 \right) \right\rangle \iff \left\langle (t + 1)(N^{\frac{1}{t+1}} - 1) \right\rangle
\]

which implies an error bound of \( \gamma_{(t+1)(N^{\frac{1}{t+1}} - 1)} \), as desired.

\[\Box\]

Proposition 3.4.2. The following bound holds true on the forward error for the \( t \) temporary superblock dot product computation:

\[
|s_n - \hat{s}_n| \leq \left( \gamma_{t(\sqrt{N} - 1) + 1} \right) |x|^T |y|
\]  

(3.4.2)
Figure 3.4: “t”–Level Superblock Summation

**Proof.** Superblock dot product differs from summation only in that it has 1 additional error factor of $(1 + \delta)$ due to the multiply, and we note that adding 1 to the subscript of the result proven in Proposition 3.4.1 yields (3.4.2).

\[ \tilde{\mathbf{s}} = \mathbf{s} + 1 \]

**Proposition 3.4.3.** An $N = 2^t$, $t$-level superblock dot product is equivalent to the space-efficient pairwise dot product.

**Proof.** Replacing $N$ with $2^t$ initially, and later $t$ with $\log_2(N)$, and applying Proposition 3.4.2 tells us the worst case error counter must be

\[ \langle t(2^t)^{\frac{t}{2}} - 1 + 1 \rangle = \langle t(2^t) - 1 + 1 \rangle = \langle t + 1 \rangle = \langle \log_2(N) + 1 \rangle \]

Which implies the error bound

\[ |s - \tilde{s}| \leq \tau_{(\log_2(N)+1)} |\mathbf{x}^T| \mathbf{y} | \]

identical to the pairwise result (3.3.4), accomplished in $t = \log_2(N)$ workspaces.

\[ \square \]

**Storage Note:** Caprani [7] showed space-efficient pairwise takes $[\log_2(N)] + 1$ storage locations, which disagrees with our count of $t = \log_2(N)$ (where we assume $[\log_2(N)] = [\log_2(N)]$) as just
shown in (3.4.3). Caprani uses a stack-based scheme almost identical to ours, but his algorithm assumes a separate temporary storage area for the final result; while we assume the result is being computed in place. This accounts for the one unit difference, and thus we do not claim to require less storage despite the differing counts.

3.4.1 Fixed-\(N_b\) Superblock

As so far presented, superblock is interesting mainly from a theoretical standpoint, since its implementation would probably be only a little more practical than pairwise. However, we can make a straightforward adaptation to this algorithm which makes it a practical algorithm for building a high performance GEMM (at least in the way we perform GEMM in ATLAS) in those cases where the problem size is too great for post-load blocked GEMM alone to give the lower-order worst-case error term. As previously mentioned, in implementation \(N_b\) is either fixed, or at most variable across a relatively narrow range. Therefore, we assume \(N_b\) is not variable when deriving our practical superblock algorithm. The second choice is how many temporaries to require. Depending on the types and levels of cache blocking applied by ATLAS’s GEMM, each additional temporary beyond the problem’s output \(N \times N\) matrix and the machine’s registers (which handle the post-load dot product in the innermost loop) would require either an \(N_b \times N_b\) temporary in the best case, or an \(N \times N_b\) in the worst. Also, additional storage locations will tend to depress performance due to added cache pollution. Therefore, we choose to add only one additional workspace beyond the problem’s output and the machine’s registers, leading to the \(t = 3\) algorithm shown in Figure 3.3(b) (for clarity we again assume that all blocksize calculations produce non-zero integral answers). This produces an error bound of \(\gamma\left(N_b+2(\sqrt{N_b^{-1}})\right)\) (note that, as expected, this is the same as (3.4.2) when \(N_b = \sqrt[N]{N}\)). We believe this algorithm, requiring only one additional buffer, will provide reasonable error reduction on pretty much all problem sizes that are practical in the near and medium term (Section 3.5 puts some statistics behind this belief), without insupportable workspace requirements or sharp performance reductions.

3.5 Statistical Studies

It is widely known that worst-case errors are almost never seen in practice. This is mostly due to the fact that a prior over-estimation is often balanced by a later under-estimation, so that the worst-case
bound is loose indeed. Many practitioners believe that with these extremely loose bounds and the self-cancelling nature of floating point error, all the algorithms perform fairly indistinguishably for most data. This idea is endorsed in a limited way by Higham [53], which demonstrates that there exist particular data and orderings which will make any of these ‘better’ dot product algorithms produce worse results than the others (e.g., a case can be constructed in which pairwise gets worse error than canonical). The conclusion in Higham’s paper [53] goes further (remember that the ‘recursive summation’ of Higham is our ‘canonical’):

However, since there appears to be no straightforward way to predict which summation method will be the best for a given linear system, there is little reason to use anything other than the recursive summation in the natural order when evaluating inner products within a general linear equations solver.

This section provides results of statistical studies we have undertaken, which show that there is indeed a benefit on average to using the lower worst-case error algorithms, and that this benefit grows with length (though not at anything like the rate suggested by the worst-case analysis).

### 3.5.1 Experimental Methodology

A frequently used approach for experimental exploration of algorithmic improvements is to select vectors with known properties that will behave in known ways. In contrast to this approach, we wish to understand how much error reduction a user can expect on more typical data. To get a feel for this, we chose a more statistical approach, where we contrast the behavior of various algorithms using the same randomly generated vectors. For each vector length \( n \) we randomly generate 10,000 different vector pairs, and each algorithm (including canonical) is run on each of these vector pairs (so algorithms are always compared using the same data) using IEEE single precision. The amount of error is found by comparison to an “exact” answer for the same pair computed with IEEE double precision and compensated addition (originally described by Kahan [40], various error analyses reported by Higham (pg 85 [37])), which is theoretically accurate to \( \pm u \) for IEEE single precision\(^{11} \). We consider two cases of interest for unstructured data, and so we have separate charts for when the elements are generated continuously in the range \([-1,1]\) and \([0,1]\). All of these

\(^{11}\text{Keeping in mind the error bound is for an accumulation in double precision and then translated to single precision for comparison.}\)
experiments were run on an x86 machine using single precision SSE instructions (so that true 32 bit precision is enforced).

Because we wish to make a statistical argument, it is important that we are measuring something that can be meaningfully averaged over our 10,000 experiments. Relative error is a fairly standard metric of error, so we originally averaged the individual relative errors of each algorithm (i.e. for each input \(X\) and \(Y\), compute \(\frac{|X \cdot Y - \hat{X} \cdot Y|}{|X \cdot Y|}\)). Unfortunately, this approach is fundamentally flawed for mixed sign data: over the 10,000 trial vector pairs it is not uncommon to find a few outliers for which the relative error denominator, \(|X \cdot Y|\), is very small, making the individual ratio so large that these outliers dominate the overall average (in some cases eliminating a single outlier would reduce the average by an order of magnitude). Therefore, we see that averaging ratios where the denominator can be arbitrarily near zero makes the average too sensitive to outliers and is therefore unlikely to represent the “typical” case. We also considered averaging the ratios of the relative errors produced by the two algorithms, but of course on identical input vectors the denominators \(|X \cdot Y|\) would cancel, leaving the ratio of two absolute errors in the form \(\frac{|e_1|}{|e_2|}\); and this suffers from the same sensitivity drawback: If on a particular vector pair the second algorithm produces an \(|e_2|\) very near zero, a huge individual ratio can overwhelm the overall average ratio.

We chose instead to average the absolute errors for each algorithm first and then find the ratio of these two averages. It is important to point out that “average absolute error” is a relatively meaningless measure of algorithmic error in isolation; but the ratio of these averages, when both algorithms are processing identical vector pairs, does have statistical value: For a given \(n\) this ratio is proportional to the ratio of the statistical expected value of their forward errors\(^{12}\), and proportional to the ratio of their expected relative errors (assuming the dot product is never zero).

\(^{12}\)Consider the form of the upper bound on error for an individual dot product of \(n\) length vectors \(X_i\) and \(Y_i\), \(|e_i| \leq c \cdot |X_i| |Y_i|\), where \(c\) is a constant (usually computed as \(\gamma_{f(n)}\), e.g. \(\gamma_n\) or \(\gamma_{\log_2(n)}\), but with \(n\) given this reduces to a constant). The constant \(c\) is statistically independent of the values of \(|X_i| |Y_i|\). We adopt this form as the model for the expected amount of error; so for algorithm 1 we assume a linear model for the error on each vector pair of \(|e_{1i}| = c_{1i} \cdot |X_i| |Y_i|\), and for algorithm 2 (on the same vector pair) \(|e_{2i}| = c_{2i} \cdot |X_i| |Y_i|\). Using appropriate random variables, taking expected values on both sides, using the independence argument, and assuming the errors are normally distributed, we find \(E(c_{1i})/E(c_{2i}) = |e_{1i}|/|e_{2i}|\); i.e. the ratio of the average absolute errors is proportional to the ratio of forward errors. A similar argument holds for relative errors, thus we believe this ratio is a reasonable general measure for comparing algorithmic error.
Further, if on a particular dot product one algorithm produces error \(|e_1|\) and a second algorithm \(|e_2| = \alpha \cdot |e_1|\), with \(\alpha > 1\), then in general it requires \(\log_2(\alpha)\) more bits to represent \(|e_2|\) than to represent \(|e_1|\), implying \(\log_2(\alpha)\) fewer bits of accuracy in \(|e_2|\), regardless of the magnitude of \(|e_2|\). The same formula does not hold for the averages\(^{13}\) \(\frac{|e_1|}{|e_2|}\), but in our experiments we did calculate the average bits of accuracy and found over all algorithms tested a greater than 99% correlation\(^{14}\) between our ratio, \(\log_2(\frac{|e_1|}{|e_2|})\), and the average bits of significance lost (or added) when it is positive (or negative, due to \(\frac{|e_1|}{|e_2|}\) being less than 1). In fact for our algorithms \(\log_2(\frac{|e_1|}{|e_2|})\) by itself predicted the average advantage in bits of significance within 0.9 bits for the same sign vectors and within 0.5 bits for mixed sign vectors.

### 3.5.2 Results

For each of the charts shown here, we compare the surveyed algorithms against canonical. The algorithms are pairwise, autol3superblock (superblock with \(t=3\) and \(N_b=\sqrt{N}\)), 13superblock60 (superblock with \(t=3\) and a fixed lowest-level blocking of \(N_b=60\)), autoblock (post-load blocked with \(N_b=\sqrt{N}\)), and block60 (post-load blocked with \(N_b=60\)). These parameter values were chosen due to practical reasons: \(N_b=60\) is a typical midrange blocking factor (when \(N_b\) is selected for performance), and \(t=3\) requires only one additional workspace in GEMM.

Average absolute errors for the various algorithms are charted in Figure 3.5.2(a) and (b), scaled by the constant \(\epsilon_M=2^{-23}\) (\(\epsilon_M\) is the smallest power of 2 which can be added to 1 to get a different number) for display purposes. In both charts, for the majority of the span the ranking of the curves is (from top (worst) to bottom (best)): canonical, block60, autoblock, autol3superblock, 13superblock60 and pairwise (although autol3superblock and 13superblock60 are nearly indistinguishable). We see that the improved algorithms perform notably better than canonical. However, if we take the mixed sign data as the more typical case, we can perhaps understand better the above quote from Higham: despite the obvious win experienced on average by the improved methods, the total error is quite low, and it seems unlikely many problems would be so sensitive to error that an improved algorithm would be critical.

The error buildup is much more appreciable on same-sign data, but of course here we

\[^{13}\]The average of the logs of a set of values does not equal the log of the set's average value.

\[^{14}\]With the exception of block60 on same sign data, which had only a 95% correlation.
Legend: Black hourglasses: Pairwise. Red dashed line: auto\textsuperscript{3}superblock. Dark blue solid line: \textsuperscript{3}superblock60. Light green point-down triangle: autoblock. Light blue squares: block60.

Figure 3.5: Average Absolute Error of Tested Algorithms
Legend: Black hourglasses: Pairwise. Red dashed line: auto\texttt{lsuperblock}. Dark blue solid line: \texttt{lsuperblock60}. Light green point-down triangle: \texttt{autoblock}. Light blue squares: \texttt{block60}.

Figure 3.6: Ratio Showing Reduction in Average Absolute Error
might expect the fact that the answer is also large to help hide the increased error. In this case, however, if the vectors are sufficiently long, new elements cease to change the result due to alignment error, and thus for long enough vectors, this should eventually prove intolerable, even in relative error. Therefore areas in which an improved algorithm might be critical include extremely long same-sign vectors, vectors which have long-running patterns of same-sign results, but whose answer is nonetheless small (e.g., imagine multiplying two vectors which produce positive results for the first $\frac{n}{2}$ elements, and negative for the remainder), highly iterative algorithms which accumulate error, or applications processing inherently low resolution data (such as 8-bit sensor readings). However, even absent these conditions, our results show that these improved methods substantially reduce the average error, and therefore it makes sense to employ at least those methods which do not negatively impact performance whenever possible.

These graphs are revealing, but the fast-rising canonical error makes all other algorithms almost indistinguishable due to scale, as well as making all errors appear somewhat linear. Therefore, in our remaining error charts, we track the ratio of canonical’s average absolute error divided by the average absolute error achieved by the improved method (as discussed in Section 3.5.1). Thus, an algorithm with a plotted ratio of 10 achieved an average absolute error 10 times smaller than canonical on the vectors of that size. Note this inverts the order of the charts relative to Figure 3.5.2, so Canonical will always be “1” and Pairwise, with the least error, will always be the curve on top. The average is over the 10,000 trial vector pairs; each algorithm using the same unique 10,000 vectors for each vector length $n$).

Figure 3.5.2 shows the average (over the 10,000 trials) absolute error of the canonical algorithm divided by the average error of the surveyed algorithms on the range $N = [1000, 100000]$ in steps of 1000, with the problem sizes along the X-axis and error ratio along the Y. Figure 3.5.2(a) shows this chart for mixed-sign vectors, and Figure 3.5.2(b) shows the same for all-positive vectors. In Figure 3.5.2(a) the order of the curves from top (best) to bottom (worst) is pairwise, autol3superblock, l3superblock60, autoblock and block60. The canonical algorithm is not shown, it would be the horizontal line of “1” (and thus the most error prone algorithm in the chart). In Figure 3.5.2(b) the algorithm l3superblock60 is above autol3superblock, although they are both still quite close.

The first thing to note is that the error ratios for the all-positive data is much larger than for the mixed sign. This may at first be counter-intuitive, as all-positive vectors have a condition
number of 1. However, one of the main ways these algorithms reduce error is by minimizing alignment error (i.e., bits lost when mantissas with differing exponents must be aligned prior to adding) by tending to add elements that are likely to be in the same basic range (since they have been exposed to a more balanced number of flops). Alignment error is less of an issue for mixed-sign data, as the dot product accumulator does not necessarily grow at every step as it can with same-sign data.

The worst performer of the improved algorithms is always block60, which nonetheless produces 7 (8) times less error on average than canonical for long vectors (mixed and same sign, respectively). It may seem surprising that block60 is competitive with autoblock, since block60 has $O(N)$ error bound where autoblock has $O(\sqrt{N})$. However, our analysis in 3.3.2 shows block60 has an $O(\sqrt{N})$ error bound for much of this range. Since actual error builds up much slower in practice, block60 maintains this lower-order behavior longer as well (note that the range where block60 is almost the same as autoblock, $N<10000$, is well within the lower-order range proven in Section 3.3.2). Since this form of GEMM requires no extra storage, this is a strong suggestion that even absent other measures, it makes sense to utilize post-load blocking in modern GEMM algorithms, and that it produces lower average errors on most problem sizes in use today.

Another surprising result is how much difference separates the 13superblock algorithms from the autoblock algorithm, since they both have an $O(\sqrt{N})$ error bound. The different 3-level superblock algorithms, as expected, behave almost the same in practice, which indicates that a fixed-$N_b$ superblock, which allows for much greater tuning freedom than autoblocked, will be adequate for error control. In mixed sign data, the 3-level superblock algorithms behave as expected: autol3superblock is generally slightly better, but since the lines are so close, superblock60 occasionally wins. For same-sign data, superblock60 actually wins across the entire range, though again the difference is minor. It is difficult to say why this might be the case, but we note that the optimal block factor is based on a worst-case analysis which doesn’t happen in practice, so using larger $N_b$ should not cause a problem. It may be that $N_b=60$ results in less alignment error on average when adding the higher level blocks (e.g., the summation totals for $N_b=60$ are more uniform than for $N_b=\sqrt[3]{N}$), but this is pure speculation on our part. We note that 13superblock is substantially better error-wise (with an error almost 23 (55) times better than canonical, respectively) than post-load blocked for all but the very beginning of this range, which suggests that error-sensitive algorithms may want to employ superblock even for reasonably sized vectors.
if the performance effect can be made negligible.

Finally, we notice that pairwise is decidedly better on average across the range. The clear superiority of pairwise over the next best tested algorithm, 13superblock60, indicates it may be interesting to study higher level superblocks to see how the performance/error win tradeoff plays out in practice. We note that the pairwise algorithm displays a sawtooth pattern for the same-sign data, with the least average error (maximum ratio) found at powers of two. Again, the reason is probably due to alignment error: when pairwise’s binary tree becomes unbalanced because $N$ is slightly above a given power of 2, each branch of the tree will yield markedly different-sized values, thus increasing the probability of alignment error. It seems likely that changing the algorithm to better balance the addition tree could moderate the drops, but since pairwise is not our main focus, we did not investigate this further.

Since standard methods work fine in practice for smaller problems, we have concentrated primarily on these large problem sizes. However, as many people are interested in small-case behavior, Figure 3.5.2 gives the same information for $N = [10, 1000]$ in steps of 10. Here we see less algorithmic difference, as one would expect. For instance, for all $N \leq 60$, canonical, 13superblock60 and block60 are the same algorithm. These fixed-size block algorithms do not strongly distinguish themselves from canonical until any $N \mod N_b$ is overwhelmed by problem size, as we see. Therefore, the fixed-$N_b$ algorithms are mainly interesting for decreasing error for large problems; since these are precisely the cases in which we most need to ameliorate the build up of error, this is not a drawback in practice.

Having considered average performance, we next consider exceptions to the average. For each algorithm we tallied its “Win/Lose/Tie” record against canonical, on the same vectors. Here we discuss 13superblock60, but very similar conclusions apply to all the algorithms in keeping with the ranking of their error bounds. “Win” means the alternative algorithm had less error, “Loss” means it had more error, and “Tie” means the error was identical within the roundoff error. For mixed sign data, 13superblock60 wins or ties against canonical 95% of the time, and for same sign data 99% of the time, becoming increasingly superior as the vector length increases.

For the small percentage of cases in which 13superblock60 loses to canonical, it is never because 13superblock60 produced an excessively large error. In fact, the largest error it produces in such circumstances is still only 52% of canonical’s average error: The reason 13superblock60 loses is because canonical produced an unusually small error on that particular
Figure 3.7: Ratio of average absolute errors, small problems

vector: On average, when canonical wins against l3superblock60, the canonical error is just 10% of the overall average of canonical error and the l3superblock60 error is 20% of that overall average.

**Results as a percentage of worst-case error bound**

It is well known that the theoretical worst-case error bound is extremely loose, and almost never achieved for long vectors. Our statistical studies agree strongly with this common wisdom, as shown in Figure 3.8, where we plot the worst error achieved over 10,000 randomly generated vector pairs for each problem size. As expected, mixed sign data shows lower actual error, and the percentage of the worst-case error goes down strongly with vector length. What may be less well-known is just how small the actual error is, at least on this type of data: we see that even for short vectors of same-sign data (the worst case for error), the worst error ever achieved over 10,000 trials was less than 3% of the theoretical worst-case error bound, and that for long vectors this drops off to a paltry 0.35% (which is nonetheless more than two orders of magnitude greater than the percentage encountered for long-vector mixed-sign data).

### 3.6 Summary and Conclusions

In summary, we have presented a survey of several of the most important known dot products along with their error properties (Section 3.3), and we showed that the lower order error achieved by post-load blocking is not overly sensitive to block size (Section 3.3.2), so a relatively small selection of hand optimized block sizes can accommodate an extremely wide range of vector sizes. Further, we have presented a new class of dot product which subsumes these known algorithms, including a modification which is suitable for high performance GEMM (Section 3.4). Finally, in Section 3.5 we demonstrated that despite the very large difference between the theoretical worst case bounds and the errors observed in practice, which make the worst case bounds of the algorithms extremely poor estimates of actual error, the worst case bounds do provide a fairly reliable guide to sorting out which algorithms will provide less error in practice. That may be useful in applications where precision is paramount; such as working with very poorly conditioned matrices near the edge of singularity.

Our main conclusion is two-fold. The first is that contrary to some thought, algorithms
Figure 3.8: Canonical, Worst Actual Error Vs Upper Bound

(a) For random data in range [-1,1]

(b) For random data in range [0,1]
with lower worst-case error bounds behave noticeably better in practice. The second is that, with the strategies we have outlined, using such algorithms should be possible with little to no performance loss in high performance libraries such as ATLAS.
CHAPTER 4: THE MASTER LAST ALGORITHM

4.1 Preface

This preface is required by Dissertation guidelines. Portions of this work were presented in conference at IPDPS 2009[8].

The author decided upon all measurements to be made in determining the cause of parallel under-performance, wrote all of the code to make such measurements, ran them, analyzed them, discovered the anomalies and devised and implemented the corrective (The Master Last algorithm).

Dr. Whaley, the co-author of the paper, in discussions with the author helped set the focus of the research to be done, outlined the experimental problem for the author to implement, and helped interpret the intermediate results. Once the problem and corrective were devised by the author, Dr. Whaley independently implemented the algorithm for verification, and collaborated with the author to determine the experiments that would best demonstrate the value of the new algorithm in a paper and presentation.

4.2 Introduction

Long-running architectural trends have signaled the end of sustained increases in serial performance due to increasing clock rate and instruction level parallelism (ILP), but have not changed Moore’s law [34]. Therefore, since architects are faced with an ever-increasing circuit budget which can no longer be leveraged for meaningful serial performance improvements, they have increasingly turned to supplying additional cores within a single physical package. Today it is difficult to buy even a laptop chip that has less than two cores, and 4 cores is common on the desktop. This trend is expected to continue, with some architects predicting even desktop chips with huge numbers of simplified cores, as in the IBM Cell [41] and Intel Larrabee [58] architectures.

As commodity operating systems (i.e. those not written specifically for HPC) are used on increasingly parallel machines, previously reasonable assumptions may become untenable. In particular, we can no longer assume the hardware, OS or compilers are highly tuned to exploit multiple cores or efficiently share common resources. Such assumptions were built into our own ATLAS [72, 71, 70] (Automatically Tuned Linear Algebra Package) software (eg., we assumed that
the OS would schedule threads to separate cores whenever possible). However, on several eight core systems running a standard Linux OS, ATLAS produced alarmingly poor parallel performance even on compute bound, highly parallelizable problems such as matrix multiply.

Dense linear algebra libraries like ATLAS and LAPACK [2] are almost ideal targets for parallelism: the problems are regular and often easily decomposed into subproblems of equal complexity, minimizing any need for dynamic task scheduling, load balancing or coordination. Many have high data reuse and therefore require relatively modest data movement. Until recently, ATLAS achieved good parallel speedup using simple distribution and straightforward threading approaches. This simple threading approach failed to deliver good performance on commodity eight core systems, and thus it became necessary to investigate what had gone wrong.

We developed two new measurements that help characterize parallel behavior. The first of these is Parallel Management Overhead (PMO). Because our problems are statically partitioned and compute intensive ($O(N^2)$ or $O(N^3)$), PMO should grow only with $O(t)$ (the number of threads). For eight threads it should be a constant. It is not. Not only is PMO a major factor in our lack of parallel efficiency, it grows with problem size, even on very large problems.

Outline: Section 4.2.1 introduces necessary terminology and defines our timing measurements, while Section 4.2.2 discusses our timing methodology. In Section 4.3 we survey techniques for managing thread startup and shutdown, show that PMO is a significant cost that can grow with problem size rather than $t$, and introduce our technique for reducing PMO to a small constant on $t$. In Section 4.4 we provide a quantitative comparison of these techniques, and show they are important even in very large operations that can be perfectly partitioned statically. In Section 4.7 we show how these relatively simple changes to a kernel library such as the BLAS [16] (Basic Linear Algebra Subprograms) can deliver substantial parallel speedup for higher level applications such as the QR and LU factorizations found in LAPACK. Finally, in Section 4.9 we discuss future work, and offer our summary and conclusions in Section 4.10.

4.2.1 Terminology and Definitions

We refer to one serial execution engine/CPU as a core, with multiple cores sharing a physical package (or just package). A physical package is the component plugged into a motherboard socket, whether comprised of one actual chip (as with recent AMD systems) or multiple chips wired together (as with recent Intel systems).
When discussing the problem, we may refer to the *full problem*, which is the size of the entire problem to be solved. The *partitioned problem* is the problem size given to each core after decomposition for parallel computation (in this chapter we consider only problems that can be simply divided so that each core has a static partition of equal size).

We directly measure several important times. The *Full Serial Time* (FST) is the elapsed time when solving the full problem in serial. The *Partitioned Serial Time* (PST) is elapsed time when solving the partitioned problem serially.

The *Full Parallel Time* (FPT) is the elapsed time when solving the full problem using the parallel algorithm and multiple cores.

Finally, the per-core time (PCT) is the elapsed time each core spends computing on its section of the partitioned problem. Thus PCT does not include any thread management overhead (eg., signalling other threads or waiting on mutex or condition variables). On a problem requiring no shared resources, therefore, PCT would always equal PST, but we will see that it does not for our present parallel implementations.

Using these directly measured times we define two quantities we believe illuminate the major causes of slowdown in our parallel algorithms. These indirect measures led us to the improved algorithms provided here, and so we believe this is a contribution that may benefit other researchers as well. Recall PMO is the full parallel time minus the maximum PCT.

In an ideal system PMO would be zero, meaning the parallel algorithm ran only as long as required to solve the partitioned problem on the slowest processor. A non-zero PMO represents time spent on non-computational activities, such as starting/killing threads, waiting on mutexes, etc. Therefore, the major goal of this chapter on $t$, which is its expected value for statically distributed problems with minimal interactions.

Low PMO by itself is not enough to ensure efficient parallel performance, since it says nothing about how long the computation itself takes. We capture this information with a measure we call *Partitioned Computational Expansion* (PCE), which is the average of the per-core compute time on a given problem divided by the average partitioned serial time for that same problem.

In an ideal parallel machine with no shared resources, this value would be one. In practice cores share some resources (caches, buses, TLB, etc) and interfere with each other. The most obvious example is when multiple cores attempt to access memory on a shared bus, causing additional
stalls and making $\text{PCE} > 1$.

### 4.2.2 Experimental Methodology

When performing serial optimization on a kernel with no system calls, we often report the best achieved performance over several trials [68]. This will not work for parallel times: parallel times are strongly affected by system states during the call and vary widely based on unknown starting conditions. If we select the best result out of a large pool of results, we cannot distinguish between an algorithm that achieves the optimal startup time by chance once in a thousand trials from one that achieves it every time by design. An average of many samples will make that distinction.

In our timings, the sample count varies to keep experiment runtimes reasonable: for the rank-K experiments, we used 200 trials. For the $O(N^3)$ factorizations, we used 200 trials for $N \leq 3000$, and 50 trials for larger problems. Since the BLAS are typically called with cold caches, we flush all processors’ caches between timing invocations, as described in [68].

All timings used ATLAS3.9.4. We changed the threaded routines as discussed and modified our timers to more thoroughly flush all core caches. We timed on two commodity platforms, both of which have 8 cores in two physical packages. The OS is critically important, in that it determines scheduling and degree of threading support. Linux is a system where the programmer can manually control the thread affinity, and thus avoid having the system schedule competing threads on the same processor despite having unloaded processors (without affinity, this occurs on both Linux and OS X). OS X possesses no way to restrict the set of cores that a thread can run on. Even within a given OS, scheduler differences may change timing significantly, so we provide kernel version information here. Our two platforms were:

1. 2.1Ghz AMD Opteron 2352 running Fedora 8 Linux 2.6.25.14-69 and gcc 4.2.1 (this system is abbreviated as $\text{Opt}$),

2. 2.5Ghz Intel E5420 Core2 Xeon running Fedora 9 Linux 2.6.25.11-97 and gcc 4.3.0 (C2).

Each physical package on the $\text{Opt}$ consists of one chip, which has a built-in memory controller. The physical packages of the Intel processors contain two chips, and all cores share an off-chip memory controller.

We will survey several operations in order to show the generality of these techniques. Our main operation will be the rank-K update, which is the main performance kernel of the LAPACK
library. The rank-K update is a matrix multiply where the dimension common to both the input matrices (the K dimension) has been restricted to some small value for cache-blocking purposes. On the Core2-based platforms, this value will be 56, and is 40 for the AMD machine (these values come from ATLAS’s tuned GEMM, which is the BLAS routine providing GEneral Matrix Multiply).

4.3 Thread Management Techniques

We are aware of two basic approaches for handling threading. We call the simplest approach Launch and Join (abbreviated LJ), and in pthreads this is accomplished by having the master process create t worker threads to perform the computation, and then make t calls to pthread_join to wait on all thread completion (a slight variant of this approach is to launch t − 1 threads, and have the master process perform one unit of computation).

In the simplest LJ approach, the master process creates all t threads, and then joins them, for an O(t) theoretical cost. This simplest algorithm is what ATLAS (and quite a few software packages) use. A more complicated LJ approach is to have the created threads spawn threads as well, which can reduce the theoretical cost to O(log₂(t)).

We implemented both, but because the cost of initiating thread creation is much lower than the time it takes to actually begin executing a created thread, the simpler O(t) algorithm outperforms the O(log₂(t)) algorithm for t = 8. Therefore, our LJ times use the simple O(t) algorithm.

Note that launch and join is also the paradigm that OpenMP presents to the programmer; we will see, however, that it is not necessarily the paradigm actually used by particular OpenMP implementations.

If we assume that creating and destroying a thread is relatively expensive, a second approach suggests itself: create the t threads just one time (for a library, on the first call), and keep the threads around for the entire execution (they can be killed using the ANSI C standard atexit function for a library). We call this approach Persistent Worker threads, (abbreviated as PW). There are many ways to implement PW; our workers each have an individual interface area that includes a condition variable which is signalled by the master to have that thread do the unit of work specified by its interface area.

Both of these approaches can be augmented with processor affinity. Many operating
systems (including Linux and Solaris, but not OS X) allow the user to restrict the range of processors a thread can be scheduled on using OS-specific calls. This can make a big difference: when left alone, the OS often schedules two or more threads to a single processor, so that some cores go unused while some delay the computation due to competing threads.

The OS may eventually notice this bad scheduling and move threads around, but this has a cost beyond the initial lack of parallelism: any cache that was warm (and much of linear algebra is rich in cache reuse) is now cold, thus causing increasing bus traffic as threads are migrated across cores. Therefore, the obvious extension is to create $t$ threads, each of which can be scheduled only on a unique core.

To indicate that a given approach has affinity, we suffix it with an ‘A’.

These techniques are the ones we found in the literature, and it was assumed they would suffice to reduce our overhead to a constant on $t$, but this did not prove to be the case, as shown in Figure 4.1(a). This figure shows the PMO for rank-K update in microseconds as a function of problem size on the C2 platform for launch and join both with and without affinity. Since all of these problem sizes use $t = 8$, and the cost is $O(t)$, we expect the overhead to be flat across the graph, but instead it essentially rises linearly (the rank-K computation is $O(N^2)$) with $N$. This linear rise in cost with problem size holds true for both architectures and for persistent worker threads as well.

We examined detailed logs of individual runs, and found a large variance in overheads, depending on where the master process was in relation to the spawn.

Essentially, when the master was on the same processor as one of the early-started threads, that thread would compete with the master for execution, and seriously extend the time it took to get all threads working (Linux perplexingly seems to preferentially launch new threads on the master core). The same problem was observed with PW: if we start a worker thread on the master core, it competes with the master for memory and time slices, delaying the startup.

This led us to a new technique, which we call Master Last (suffix: ML), which ensures that, whether using LJ or PW, you call for threads to begin executing on all other cores before starting the computation on the core that the master is executing on. This is shown in Figure 4.1(b) for both launch and join and persistent workers, and we see that the PMO is now independent of problem size, and orders of magnitude less.

We can also see that it is much quicker to signal $t$ condition variables (as in PWAML) than to
Figure 4.1: Parallel Management Overhead in rank-K update for launch and join and persistent worker on a 2.5Ghz C2
create and join $t$ threads from scratch (as in LJAML). In Section 4.4 we will quantify how important these overhead savings are in terms of total runtime, so that a judgment can be made as to whether PWAML is worth the extra complexity.

Note that master last is important because of the relatively poor scheduling job being done by the OS: an OS that scheduled threads to non-master cores first would achieve master last implicitly. As a practical matter, OS X appears to almost never achieve implicit master last, but an older version of Linux which we were running on the Opt before a system update achieved master last most of the time. Therefore, master last appears to have become important due to OS scheduling algorithms that are probably more geared to desktop than HPC use.

In the remaining PMO timings, we wish to quantify the contribution of each of these techniques, which we do with the easy-to-understand launch and join paradigm, giving us three different LJ timings: LJ, LJA (LJ with affinity only) and LJAML (LJ with affinity and master last). Finally, we will show the performance for the lowest-overhead technique we have yet discovered, persistent worker threads with affinity and master last (PWAML).

### 4.4 Impact of these techniques on runtime

We have shown that the master last technique can reduce the PMO to a small constant on $t$, which only matters if PMO is an important component of the total runtime. Obviously this will be strongly affected by the problem size, since work done is $O(N^2)$, while PMO either remains constant, or appears to rise with $N$. Figure 4.2 shows the PMO of rank-K update as a percent of the total runtime on both the C2 and Opt platforms.

On both platforms, we see that for small problems, the overhead is actually the dominant cost of the algorithm, but for the master last algorithms PMO cost drops precipitously. Algorithms without ML decline much less quickly, as the $O(N^2)$ cost slowly dominates the startup time.

From this, we would expect something similar on full matrix multiply, but with the $O(N^3)$ computation dominating more quickly. We see that LJAML is almost as good as PWAML towards the end of the curve, but that the PMO advantage of PWAML is still fairly important in the medium-range problems (eg., $N \leq 2000$).

So far, we have been focusing completely on overhead, but this can be misleading, in that a given technique might reduce PMO and yet make PCE worse, thus making the algorithm slower.
Figure 4.2: PMO of rank-K update as a % of runtime for surveyed techniques
despite a lower PMO. Our goal is actually to increase parallel speedup, and we need to show that these methods do that. This is shown in Figure 4.3, which also quantifies how much each technique contributes to final speedup.

In this chart, the average speed of LJ is taken as 100%, and we plot the speed of all others relative to it. Specifically, the Y axis is computed as \( \frac{LJ\text{time}}{method\text{time}} \times 100.0 \).

The first thing to notice is the sheer magnitude of the speedups from these techniques: for small problems the best technique is over 220% of the speed of the naïve technique, and even for very large problems, the advantage is still over 5%!

The second lesson from these graphs concerns the relative importance of each technique: we see that all techniques have the greatest impact for small and midrange problem sizes, where the \( O(N^2) \) computation is not so dominant. Affinity is important on both machines, and while its impact initially goes down with problem size, it then levels off – we therefore conclude that affinity is important regardless of problem size (this is confirmed even with full GEMM timings, where we have \( O(N^3) \) computations).

PW is particularly helpful on the C2, but on both machines its advantage looks likely to go away asymptotically. Finally, master last is extremely important on both systems, particularly for midrange problems. It is worth noting that what we are calling “midrange” problems are actually quite large; for problems roughly in the range of \( 2000 \leq N \leq 4000 \), master last accounts for over half of the available advantage over LJ!

We did test the performance of PWA, i.e. persistent worker threads with affinity but not master last. For clarity we left it off the charts. PWA performance is about halfway between LJAML and PWAML. PWA is about 25% slower than PWAML when \( N \leq 2000 \). On larger problems the deficit narrows to about 4% asymptotically. It is comparable to the difference between LJA and LJAML. Thus in both PW and LJ implementations, master last delivers significant improvement.

However, there remains a mystery to solve. In Figure 4.4 we examine absolute speedup of our best algorithm (PWAML) and the simple LJ on each machine. We do indeed do much better, particularly on smaller problems where PMO is a big part of overall runtime.

Nevertheless, having made PMO negligible, and given our perfectly partitioned problem division, with eight cores we’d expect to have an asymptotic peak speedup near the maximum of 8. We do not observe that; in fact on the Opt8 our maximum speedup is 4.74 (\( N=5120 \)) and on the Core2 we max out at 5.15 (\( N=1344 \)). The source of this discrepancy is addressed next.
Figure 4.3: Contribution to rank-K speedup by technique
Figure 4.4: Speedup Comparison

(a) Core2 PWAML vs. LJ

(b) Opt8 PWAML vs. LJ
4.5 Computational Expansion

In Figure 4.5 we show some details on the concept discussed earlier, PCE (PartitionedComputational Expansion). A typical kernel in linear algebra must fetch its arguments from memory; and in a multi-core system the kernel may be competing with other kernels for system-wide resources, such as memory access. We show results for the Core 2 system; the Opt8 system was similar.

![Bad Partitioned Computational Expansion](image)

(a) Core2 Example of Bad Core Order Assignment

![Good Partitioned Computational Expansion](image)

(b) Core2 Example of Good Core Order Assignment

Figure 4.5: Partitioned Computational Expansion

What we see in Figure 4.5(a) is, on the X-axis, the total number of cores used to solve a job. Think of each such column set as a separate run\(^\text{15}\), and within each run, imagine that all cores utilized are performing the same standard unit equal work, such as a kernel-sized square matrix.

\(^{15}\)Actually these are the averages of 200 runs per column set.
multiply. That same unit of work is timed for a single core working alone, so when two cores are both doing the same thing, ideally their PCT (per-core-time, which does not include any overhead) would be identical.

It is not. Even when we just have two cores working, they take slightly longer to run, and when we have eight cores running, they take almost 80% longer to complete the same amount of work. The difference between Figure 4.5(a) and Figure 4.5(b) is the selection of which cores are used, when something less than all the cores are used. For the most dramatic example, compare the PCE when four cores are used in Figure 4.5(a) to the four core usage in Figure 4.5(b): The cores in the former take 45% longer to run than the serial version, while the cores in the latter take only 5% longer. On the Core 2, the least PCE occurs when cores are chosen to alternate both packages and sides within a package; so if only four cores are to be used, they should be cores 0, 2, 4, and 6.

In our example problems for this chapter we found it is always faster to use all 8 cores than to use any lesser amount. Thus there is no core assignment trick we can use to reduce PCE. PCE does explain why we are not getting linear speedup.

Our particular form of PCE turns out to be almost 100% memory access contention. If experiments of this same length and flop count are run with all of the data pre-loaded into the local, private cache of each core and “perfected,” (see 4.6), we see virtually zero PCE. The slowdown here is caused by several cores competing for memory service or hardware. This is an important point for library authors in tuning their code: When a kernel is tuned in isolation (the equivalent of our serial code) it does not have to share system-wide resources with copies of itself running on other cores. When other copies of the kernel are running on other cores, the net bandwidth of the kernels to any given device (such as a memory controller) may be multiplied by \( p \), the number of cores in use. In our case, the best kernel for serial operation is not the best kernel for parallel operation, because somewhere around \( p = 5 \) our kernels are demanding more than the memory subsystems can deliver. Thus our kernels are stalling, and that is the source of the PCE we see in Figure 4.5.

### 4.6 Cache Perfection

A cache typically has a number of sets, each of which correspond to some common part of a memory address. Because it is only a part of the address, many addresses in the actual memory will be
mapped to the same cache set; i.e. they share it. The set consists of ways, and each way holds one cache line as well as some accounting information. A line is a contiguous chunk of memory, and 32 or 64 bytes per line is quite common. There can be just one way, but to prevent the cache from “thrashing” (due to set conflicts) the cache typically has several ways (2, 4, 8 and 16 ways are common). When memory is read, finding a line resident in the cache is called a hit, failing to find it is called a miss. Not all of the lines that map to the same set can be retained in the cache, so if a new one arrives, an old one must be evicted (and written back to memory if changed). The “Cache Replacement Policy” is the algorithm for determining which way is to be evicted.

The ideal policy is usually considered to be “Least Recently Used” (LRU), which assumes temporal locality of references: Something recently used is most likely to be used again. However, the hardware necessary to implement LRU is prohibitive once the ways exceed four; because LRU requires, on every read, something like an insertion sort operation (e.g. if the seventh most used way is read, it must move to the top of the list and the previous top six entries must move down a rung.) This hardware would have to be present for every set of ways, and requires a prohibitive amount of circuitry and time, since it would be executed after every memory read.

Instead many architectures\(^\text{16}\) use a “Pseudo-Random Replacement” policy: If a way is needed to store a new line, then the machine chooses one of the ways in the set at random to evict. It may even be the most recently used. In normal operation this is not as good as LRU, but is about half as good: Presuming the choice is truly random, the average selection is of middle rank in terms of how recently it was used.

Thus, if one wants to test how a kernel runs when all of its data is expected to be in some level of cache, it is not enough just to read a block of memory that we know fits in the target cache: With a random replacement policy, some of the later reads in the target block will evict the early reads of target block elements.

For an example, consider a cache with 1024 sets and 8 ways per set. The line size does not make a difference here, but let us say it is 64 bytes, so this hypothetical cache can hold 512K bytes.

The sets are independent, so we can consider a single set of the cache: The first read that maps to this set will be stored in the cache, evicting some current occupant of a way, because a new

\(^{16}\text{We infer this from cache behavior and public statements; actual engineering specs at this level of detail are not generally available.}\)
read is *always* cached. However, the second read that maps to this set has a $\frac{1}{8}$ chance of evicting the *first* read. If it does not, the third read that maps to this line has a $\frac{2}{8}$ chance of evicting one of the first two reads, and so on. If way selection is truly random, the chance that all 8 reads that map to this set will select different ways is $\frac{8!}{8^8}$, or about 1 in 416. In fact, after 8 reads that map to the same set, the statistically expected number of ways occupied by the target data is only 5.251129 (We will show the analytical calculation below).

In a 4-way cache we expect 2.734375 ways to be occupied on average after one read of a cache-sized data set, in a 16-way cache 10.302814 ways are occupied. In percentages this is (68.36%, 65.64%, 64.39%) of the ways for (4, 8, 16) way caches, respectively, which we will characterize as roughly $\frac{2}{3}$ of the ways.

The number of ways occupied is a monotonic function (ignoring reads by the operating system, or code corrupting the cache) as long as we only read the target data. Once a way is occupied, another read of a line of target data that maps to that set will either be cached or not. If it is cached, it leaves the number of target-occupied ways the same. If it is not cached, the randomly selected way to be ejected will either be occupied by a line of target data (leaving the number of target-occupied ways the same) or it will contain non-target data. The latter condition means we will have increased by one the number of ways that contain target data. Thus no read of a line of target data can reduce the number of ways in a set that are occupied by target data.

If we do enough additional reads of the target data that map to this set, eventually all of the ways of the set will be occupied by the target data, and at that point all reads of the target data will result in a cache hit. Thus none of the ways would ever be evicted to make way for another, and we call this a “perfected” set in the cache.

If the selection of ways is random, then eventually reading target data must perfect the set. Perfecting all of the sets is called perfecting the cache.

The object of cache perfection is to read the target data sufficient times to ensure to some level of statistical certainty (such as 99%) that it is contained in the cache.

The proper statistical analysis tool is a Markov Chain. For a four-way set intended to hold four distinct lines \(\{L_1, L_2, L_3, L_4\}\), we can describe the state of that set as which of the four lines are currently cached in the set. Each of the lines is either cached or not, so there are \(2^4 = 16\) possible states. When a particular line \(L_i\) is read, each possible state of the set will evolve to a new state, with some probability, that new state must represent \(L_i\) as being cached.
We use a binary string to represent the presence or absence of each line in the cache, so for example 1000 means $L_1$ is cached and $L_2, L_3, L_4$ are not. When we read $L_2$, there are two possibilities: There is a 25% chance that the way holding $L_1$ will be selected, so it will be ejected and the new state will be 0100, and there is a 75% chance some other way will be selected to hold $L_2$, and the new state will be 1100.

On the other hand, if the state is 1011 and we read $L_2$, there is a 25% chance of each of the following states: 0111, 1101, 1110, 1111.

We can construct a $16 \times 16$ state transition matrix for each possible read. For example, in Figure 4.6 we show the transition probability matrix $P_2$ for reading $L_2$. Each column represents the starting state, and each row represents a resultant state. If we have a starting vector $S \in \mathbb{R}^{16 \times 1}$ that contains the probability of a set being in each of the possible states, then multiplying $P_2 \cdot S$ gives the probability of each possible state after reading $L_2$. Note that half of the probabilities are zero; for example we cannot read $L_2$ and enter a state where $L_2$ is not cached (which is half of the states).

![Figure 4.6: $P_2$, Probability Matrix for Cache State after Read of $L_2$](image)

We can construct similar transition probability matrices $P_1, P_3,$ and $P_4$ for reading $L_1, L_3,$ and $L_4$, respectively. If we multiply $P_4 \cdot P_3 \cdot P_2 \cdot P_1$ we obtain the transition matrix $P$, for one read of all four lines. To link to our discussion, this is equivalent (for each set) of one read of a cache-sized target dataset. That matrix is shown in Figure 4.7.

Our starting state probabilities with zero reads is $S_0 = \{1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0\}^T$, and $S_n = (P^n) \cdot S_0$ is the cache state probability
Figure 4.7: $P$, Probability Matrix for 4-Way Cache State after Full Cache-Sized Read

vector after $n$ reads of a cache-sized data set. The number of lines cached in each state is $C = \{0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 3, 4\}$, and the weighted sum of probable lines cached is just the dot product $S_n \cdot C$. That is the expected number of lines cached in a set after $n$ reads. This model matches the simulation-derived probabilities, to at least six decimal points after ten million trials.

After 8 full reads of a data set we expect 97.97% of the sets are perfected, after 16 full reads we expect 99.977%, and after 32 full reads we expect 99.9999972% are perfected. To compute the probability of all sets being perfected ($p$) by the number of sets ($s$): $p^s$. So, for example, if a cache has 1024 sets and 4 ways; after 16 reads the probability of all sets being perfected is $0.99977^{1024} = 0.79353$.

Similar Markov state transition matrices can be constructed for any number of ways $W$, they are $\in \mathbb{R}^{2W \times 2W}$ but easy to construct programmatically.

Although some modern hardware can report actual cache hits and misses, we used a simulation to verify the mathematics instead of a real machine because a real machine will inevitably produce noise (due to about a 2% overhead from an operating system that is reading data and “corrupting” the cache). Also, we do not know the actual details of how pseudo-random replacement might be implemented in commercial hardware or if it is truly random replacement. Some systems use true LRU on one level of cache, or treat a cache as a victim cache, or may have implemented proprietary schemes for circumventing some of the problems described here that are not publicly documented. Since we cannot model pseudo-random replacement without knowing all these details, we modelled true random replacement as a proxy in the expectation that the actual implementation
is not worse than true random replacement.

Reading a target data set dozens of times is not a prescription for high performance. Our primary motivation for investigating cache perfection was to use *imperfection* as an explanation for **PCE**. That goal was met. As an experiment, we read target data several dozen times to perfect the caches on the cores before we began timing our parallel operations, with the result that over 95% of **PCE** vanished. Thus at least 95% of the **PCE** was due to contention between cores for memory access, alleviated by the perfecting step.

One important takeaway of this research is the concept that caches *need* perfecting. It helps us explain one of our contention mysteries. Kernels like matrix multiply with a high reuse factor are essentially perfecting the cache as they run. Even if they copy their data (or block fetch it [66]) only about $\frac{2}{3}$ of it survives in the cache after the first read. Thus they will suffer cache misses, which require memory access, which in turn usually employs hardware resources shared by some or all of the cores, so cores must contend for access to those resources. In general, the caching of data is only about $\frac{2}{3}$ effective after a single read (for 4, 8, and 16 ways). This means the $p$ cores running will still require some memory access bandwidth to complete their operations, and this can be significantly more than the bandwidth required by a single kernel.

Since bandwidth restrictions are implicitly taken into account by automatic tuning, this suggests kernels for parallel use should be tuned differently than kernels intended for single kernel use. Finally, one method of improving the cache effectiveness is to simply use less of it. The fewer ways per set required to cache the target data, the more likely it is to remain cached. In a hypothetical 8-way cache, if only four of the eight ways per set are needed, 88% of the sets are perfected after the second pass through the data (in the sense that all of the target data has been cached), versus 6.7% if all of the ways are required. For algorithms with low or moderate reuse, blocking the data to use half of the available cache may improve performance.

### 4.7 LAPACK Performance

Note: This section has been re-written from the original paper, to include improved results and development since that publication.

It is possible to speed up a compute kernel, and yet have little effect on application performance. That is not the case here; Figure 4.8 shows two charts documenting the speedups
Figure 4.8: QR, LU Results with Fully Master Last BLAS

(a) QR Opt8 Post-Paper Performance Improvement

(b) LU Core2 Post-Paper Performance Improvement
achieved for QR and LU by a library-wide implementation of Master Last, on two platforms. These show the usual percentage speedups over LJ, e.g. 145% implies the Master Last routine is 45% faster than the identical LJ routine. However, these timings are for a threaded BLAS implementations with Master Last implemented throughout\footnote{The original paper implemented Master Last only for GEMM, the most heavily used BLAS routine.}. Both QR and LU factorization results shown are for double precision only.

4.8 Related work

We have found no work in our area that directly addresses overhead issues due to threading, but \cite{1} discusses using process (rather than thread) affinity for performance improvement. There are numerous papers that address these issues in some way outside our field; we cite only a representative handful of these here. There are several papers that discuss the advantage of process affinity in at least a cursory way, including \cite{11, 49}. The mention of persistent worker threads which is most closely allied to our area can be found in papers on optimizing OpenMP libraries, as in \cite{54, 42}. We have seen worker threads discussed in online papers for Intel’s OpenMP implementation as well, so this idea is clearly well-known to the community.

Threading overhead issues are widely discussed in research on web servers and services \cite{38, 56, 46}. They have standardized some terminology: our launch and join paradigm roughly corresponds to their thread-per-request, while our persistent worker threads is roughly their thread pool. We have kept our idiosyncratic naming strategy, since the web services terminology has built-in assumptions that are not true for HPC (e.g., that you have multiple requests at once, or that the thread pool consists of far more threads than the number of cores). We have found no mention of master last in any publication.

4.9 Future work

Operating systems like OS X lack the ability to fix processor affinity and the scheduler seems to migrate threads frequently; other means must therefore be developed to capture at least some of the benefit of affinity and master last. If such techniques are not found, then operating systems without affinity can expect substantial performance loss when compared to operating systems that
support affinity. This loss is presently around 40%, but it should rise as we address PCE (several PCE techniques will need affinity for optimal application).

### 4.10 Summary and Conclusions

We have introduced two measures, PMO and PCE, which can help illuminate the major causes of parallel slowdown. We have discussed a variety of techniques for reducing thread management overhead which should be usable for almost any threading application, the most efficient of which (master last) we have not seen in the literature.

We have shown that the conventional wisdom that overhead is unimportant when doing large compute-intensive operations needs to be re-examined. We have quantified the relative contribution of these techniques to the performance of one of the most widely used kernels in Linear Algebra; GEMM. We have also shown that these techniques provide substantial speedups for LAPACK factorizations on the high-end systems of today, and are therefore likely to be critical for even the desktop machines of tomorrow. We conclude that all HPC threading should employ affinity and master last.

Choosing between PW and LJ is less clear-cut, and depends on the problem, the sophistication of PCE control, and whether PW and LJ threads must share the machine. Our experiments (In particular with OpenMP) suggest that persistent workers/thread pools (which are widely used) do cause problems when more than one library is independently threading. This is quite common for large applications, for example they may employ libraries for linear algebra, graphics and signal processing simultaneously. If the environment for a library might include other libraries for other purposes running simultaneously, the performance advantage of PW seen by the library developers while testing their library in isolation may be transformed into performance degradation in the field.
CHAPTER 5: PARALLEL CACHE ASSIGNMENT (PCA)

5.1 Preface

This preface is required by Dissertation guidelines. Portions of this work were presented in conference at PPOPP 2010[9], with Dr. Whaley as the co-author.

Dr. Whaley had previously suggested the idea of using all of the caches of the cores in concert to parallelize the QR panel factorization, which we knew to be a series of BLAS Level-2 operations (and therefore notoriously difficult to parallelize). The large overheads and preemption difficulties that arise when using the synchronization methods provided by the Operating System (OS) would seem to confound this approach. The QR panel factorization is extremely data dependent and when parallelized using PCA requires lockstep synchronization every few microseconds.

The author’s contribution to this research was devising a fast hardware-based method of synchronization for multi-cored systems with shared memory that relied upon cache-coherence instead of the OS, and thus could accomplish a synchronization within a few dozen machine cycles (instead of tens of thousands of cycles). Without this synchronization technique, PCA would not be effective. The author also devised the arithmetic necessary to reblock the LAPACK versions of finding the 2-norm (which is a rather involved iterative process to avoid a loss of precision by squaring) and blocked the other QR panel operations for parallelization. The author implemented all of the code to implement PCA for the QR matrix factorization, wrote the testers, ran all the experiments and timings and produced all of the graphs and wrote most of the paper. Dr. Whaley simultaneously implemented PCA for the LU matrix factorization and tested and timed it, and directed the writing of the paper through several drafts.

Subsequent to the PPOPP 2010 conference, the author performed the research for applying PCA to the Hessenberg reduction, a new algorithmic area. The author identified the issues here, wrote the tester and several versions of code in order to prototype various approaches. This chapter also presents results obtained by other students implementing the author’s approaches with technical supervision by the author. Using the author’s code as a template, graduate student Siju Samuel implemented PCA for the variants of QR (QL, RQ, LQ) in multiple precisions as part of his Master’s thesis.
5.2 Overview

This chapter extends our PPOPP 2010 [9] conference paper, with a significant amount of new material. In that paper we presented a parallelization technique we named PCA (Parallel Cache Assignment) that we believe is generally applicable to a variety of problems. To demonstrate the PCA technique we applied it to two LAPACK matrix factorization techniques; the QR decomposition and the LU decomposition, in double precision real numbers, and on two platforms. Our extension covers both single and double precision in both real and complex (four types/precisions). We also cover two more LAPACK matrix decomposition techniques; in addition to QR we provide QL, which is similar (and given QR and QL we can, by transposition, compute LQ and RQ, respectively). We also implemented PCA for the Hessenberg reduction, in double precision (real). The Hessenberg reduction is significantly different from our previous cases (for reasons that will be described in that section), and although we still achieved superlinear speedups in our implementation, the Hessenberg reduction demonstrates that there are cases where PCA can have sharp limits if the algorithm being parallelized cannot be fully blocked.

Finally, we also implemented on the GPU platform\textsuperscript{18} a proof-of-concept for the single precision QR panel factorization that produced excellent speedups over both the most frequently used LAPACK alternative and our own PCA implemented on a typical GPU host platform.

The parallelization technique we describe here is quite general, even though our chosen demonstration cases involve a particular set of routines in dense linear algebra (where it solves a long-standing problem). The central lesson for parallel practitioners in general is that, given a bus-bound operation with the potential for memory reuse, and a system with hardware-enforced cache coherence, it can be highly profitable to block for cache reuse even if this introduces parallel overheads into the innermost loop. We demonstrate here superlinear speedup for an operation that has resisted significant parallel speedup for years; we accomplish this feat by enabling cache reuse at the cost of increasing the number of parallel synchronization points by more than two orders of magnitude. This approach leads to superlinear speedups because it changes the computational speed from the speed of memory to the speed of the cache, and the additional parallel overhead only slightly reduces this performance bonanza, because in a multicore system with shared memory

\textsuperscript{18}GPU for “Graphics Processing Unit”, in our case the equivalent of a system with 30 single-precision vector processors.
the additional parallel synchronizations can run at essentially the speed of shared-memory access.

5.2.1 Basic Terminology

LAPACK [2, 29] (Linear Algebra PACKage) is one of the most widely-used computational APIs in the world. Since linear algebra is computationally intensive, it is important that these operations, which are inherently very optimizable, run as near to machine peak as possible. In order to allow for very high performance with a minimum of LAPACK-level tuning, LAPACK does the majority of its computation by calling a lower-level API, the BLAS (Basic Linear Algebra Subprograms). The BLAS are in turn split into three “levels” based on how much cache reuse they enjoy, and thus how computationally efficient they can be made to be. In order of efficiency, the BLAS levels are: Level 3 BLAS [16], which involve matrix-matrix operations that can run near machine peak, Level 2 BLAS [17, 18] which involve matrix-vector operations and Level 1 BLAS [32, 44], which involve vector-vector operations. The Level 1 and 2 BLAS have the same order of memory references as floating point operations (FLOPS), and so will run at roughly the speed of memory for out-of-cache operation.

5.3 Introduction

In LAPACK, many matrix operations are cast as block algorithms which iteratively process a panel using an unblocked Level 2 BLAS-based algorithm and then update a trailing matrix using the Level 3 BLAS. Level 2 BLAS perform $O(N^2)$ flops on $O(N^2)$ data, making them largely bus-bound. On the other hand, Level 3 BLAS routines perform $O(N^3)$ flops on $O(N^2)$ data, allowing for extensive cache and register reuse. This strongly reduces their dependence on bus speed, so that the Level 3 BLAS can often be tuned to run near the peak flop rate of the machine (i.e. the Level 3 BLAS tend to be compute bound rather than bus bound). LAPACK QR, LU and Hessenberg factorizations exemplify this general BLAS/blocking strategy, and for large problems running in serial (i.e. with all operations running on a single core) on our test systems, this tuned LAPACK approach results in the serial QR and LU spending over 95% of their runtime in the Level 3 BLAS, resulting in extremely high performance. (This is not true for the LAPACK Hessenberg reduction, as we will describe.)

However, as commodity platforms trend increasingly to multi-core systems that offer high
degrees of parallelization, this approach is faltering. Level 3 BLAS routines typically scale well with 
$p$ (the number of cores) but the bus bound Level 2 BLAS routines do not; once the bus is saturated 
they cannot get any faster, and bus speeds are not growing as fast as $p$. Therefore Amdahl’s law 
tells us that as $p$ grows, the Level 2 BLAS-based panel factorizations will increasingly dominate the 
runtime. We can see this happening already on current architectures: On our 8-core test systems, 
QR panel factorization is only 4% of the runtime on a large problem when a single core is used, but 
over 20% of the runtime when the Level 3 BLAS routines are parallelized. This is going to be the 
case any time massive parallelization is employed; for example when the factorization is done on 
a GPU, the GPU typically relies upon the CPU to perform the panel factorizations[64, 65], which 
are typically done at bus speed, thus creating a serious bottleneck.

Clearly there is a need to further parallelize the panel factorization. Prior work[63, 31, 20] 
has shown that recursion can help do that by employing the Level 3 BLAS within the panel 
factorization. However as the recursion deepens and the panel thins, the Level 3 BLAS get less 
and less benefit from parallelism, cache and register blocking, and are ultimately bus bound again. 
This is the problem we address with our new approach, which we call Parallel Cache Assignment 
(PCA).

5.3.1 Our Approach: Parallel Cache Assignment (PCA)

The key to PCA is a data-ownership model of parallelism where data is assigned to the cores in order 
to enable cache reuse. One side effect of this assignment is that we introduce parallel synchronization 
into the innermost loop in order to maximize cache reuse: since without this transformation the 
performance is limited to the speed of memory, this counter-intuitive optimization turns out to 
more than pay for itself.

The central insight here is that a QR or LU panel factorization takes $O(N \cdot N_b^2)$ flops for 
$N \cdot N_b$ data, and thus, with $O(N_b)$ flops per data element, there is the potential of cache reuse. In 
other words we should be able to find some way to escape the bus bound limitation of the Level 2 
BLAS routines being used. However, the reuse is tied to the outer loop that indexes the columns; by 
this we mean that if we split that loop into $p$ equal parts we reduce the cache reuse by a factor of $p$. 
Instead, we break the inner loop that indexes the rows (in addition to enabling greater cache reuse, 
this allows for parallelism when operating on the very long columns seen in these panel operations). 
In short, we partition the panel horizontally, assigning a roughly equal number of rows to each core,
and have them all loop over the \( N_b \) columns of the panel. Then, each core can keep its share of the panel in its own cache (we target the L2 cache here), and loop over all \( N_b \) columns. However, this creates a new constraint: If we want to preserve the current factorization arithmetic, both QR and LU must proceed one full column at a time, from left to right. Thus all the cores cooperate and synchronize on every column, and on the subsequent trailing matrix updates within the panel.

In practice every core processes their share of the column, and at the necessary sync points we perform a \( \log_2(p) \) combine of their results. Once that is complete the cores use the global result to proceed independently to the next sync point. LU requires only one synchronization point; QR requires several.

Note that we must introduce parallel synchronizations into the innermost loop. Driving the parallel synchronization into the innermost loop is counter-intuitive as it will obviously increase parallel overhead, and if it required expensive communications it would represent an intolerable drag on performance. However, on multi-core systems with shared memory, we can produce a lightweight and efficient cache-coherence-based sync scheme (see Section 5.5.1), and in this multi-core environment we found the cost of frequent synchronization is far outweighed by the savings we gain from cache reuse and increased parallelism. The cores continue to use the serial Level 2 BLAS routines, but instead of being bus bound from memory, they can run at L2 cache speeds.

We will call the aggregate of all the caches the cores can access (both private and shared) the “collective cache”. If the panel is too large to fit into the collective cache, we employ column-based recursion \([63, 31, 20]\) until the problem does fit. For extremely large problems, even one column could overflow the cache; in such cases we end the recursion when the number of columns grows too small for the recursion’s Level 3 BLAS usage to extract meaningful parallelism, and then call PCA. In our implementation each core performs a data copy of its share of the panel into local storage at the beginning of the panel factorization, and copies the computed result back out at the end. This allows us to optimize memory alignment, minimize TLB usage, false sharing, and cache line conflict misses. Due to its more effective use of the memory hierarchy, this copy operation usually improves performance significantly. However, if the number of rows causes recursion to severely restrict the panel width, and/or the panel overflows the collective cache, we get better performance by omitting the copy steps. The crossover point between copy and no copy can be empirically tuned for improved performance.

We will show superlinear speedup in both QR and LU panel factorizations on 8-core
systems (e.g. 16-fold speedups over the corresponding serial performance). We also show speedups of 2–3 over the fastest known parallel panel factorizations, and we show our approach scales well with $p$. As $p$ continues to grow, the collective cache size is likely to increase as well, so ever larger problems can be made to run at cache speed rather than memory speed.

Other work done in this area includes [30], [22], and [6]. However, some of this work significantly increases the number of flops beyond the LAPACK blocked algorithm, while our approach requires no extra flops at all. Our approach also has the advantage of making no changes to the arithmetic of the LAPACK blocked algorithms, and is thus guaranteed to preserve both its stability and error bound\textsuperscript{19}.

We believe PCA is a general approach for all or most LAPACK panel operations, and we have implemented it as described for QR and LU Factorizations\textsuperscript{20} and for Hessenberg factorizations\textsuperscript{21}. We will concentrate primarily on the QR factorization, but we will show results for LU and Hessenberg as well.

### 5.3.2 Surveyed Libraries and Important Routines

Both the LAPACK and BLAS APIs have a reference implementation available on netlib, but many vendors and academic groups provide optimized alternatives. In general, the reference BLAS (particularly the Level 3 BLAS) are orders of magnitude slower than optimized BLAS. Optimized LAPACK implementations can show notable improvements, but this is not nearly as marked as for the Level 3 BLAS.

In this chapter we discuss two optimized LAPACK and BLAS frameworks. ATLAS [72, 71, 70] (Automatically Tuned Linear Algebra Software) is our package which uses empirical tuning to auto-adapt the entire BLAS and a subset of LAPACK to arbitrary cache-based systems. ATLAS uses the netlib reference LAPACK to provide those LAPACK routines that are not currently supported. The GotoBLAS [24, 25] provides meticulously hand-tuned implementations (written mostly in carefully crafted assembly) for many current machines, with performance that is almost always near or at the best known for the machine. The GotoBLAS provide a complete BLAS, and a subset of LAPACK.

\textsuperscript{19}Indeed, due to parallel blocking the error bound constant is reduced.
\textsuperscript{20}In our original conference paper.
\textsuperscript{21}In our extension work for a journal paper.
We also compare against a new linear algebra framework, which is planned as a complete rewrite of LAPACK in order to exploit massive parallelism. This new research project is called PLASMA [21] (Parallel Linear Algebra for Scalable Multi-core Architectures), and currently implements new, explicitly parallel, approaches to a subset of LAPACK.

In this chapter we will often refer to an algorithm by its LAPACK or BLAS name. For both APIs, the first character provides the type or precision the routine operates on; for the first part of this chapter all routines will start with D, indicating they operate on double precision real data. In our extension work, the symbols S, C and Z correspond to single precision real, single precision complex and double precision complex, respectively. New names will be introduced as needed, but there are a few of enough importance to introduce here. DGEQRF is the LAPACK routine which performs the QR factorization (see Section 5.4) on an entire matrix. In LAPACK this algorithm is parallelized implicitly by the Level 3 BLAS, which also provide the foundation for its serial performance. The performance of DGEQRF rests primarily on three routines. In order of typical importance these are DGEMM, DTRMM and DGEQR2. DGEMM and DTRMM are Level 3 BLAS routines handling general rectangular and triangular matrix multiplication, respectively. The third such routine, DGEQR2, is an LAPACK subprogram called at each step of DGEQRF’s iteration to factor a column panel (i.e. a block of columns). DGEQR2 in turn primarily relies on the performance of two Level-2 BLAS routines for its performance: DGER performs a rank-1 update (update a matrix using the outer product of two vectors), and DGEMV performs a matrix-vector product. In DGEQR2, DGEMV is always called with a transposed matrix, which we will denote as DGEMV^T.

5.3.3 Outline

In Section 5.4 we review the routines involved in the QR panel factorization, and Section 5.5 provides a detailed description of our technique and its application to QR. Section 5.6 discusses our timing methodology. In Section 5.7 we provide a quantitative comparison of this technique on two representative commodity architectures; show they have excellent scaling with p and produce significant measured speedup versus ATLAS, the GotoBLAS [24, 25], and the PLASMA library (which implements QR as described in [6]). Finally, in Section 5.11 we discuss future work, and offer our summary and conclusions in Section 5.12.
5.4 QR Matrix Factorization

A QR factorization of \( A \in \mathbb{R}^{M \times N} \) with \( M \geq N \) yields

\[
A = Q \cdot R = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix} \begin{bmatrix} R_1 \\ 0 \end{bmatrix} = Q_1 \cdot R_1
\]  

(5.4.1)

Where \( Q \in \mathbb{R}^{M \times M} \) and \( R_1 \in \mathbb{R}^{N \times N} \) is upper triangular. There are multiple ways to accomplish this, for examples see [57]. In LAPACK, \( Q \) is found as a product of Householder transforms, one per column. For our example we need \( N \). Each transform is of the form \( H_i = I - \tau_i \cdot v_i \cdot v_i^T \), where \( v_i \) has a unit-2 norm. Note that Householder transforms are involutary, i.e. \( H_i = H_i^T \) and \( H_i \cdot H_i = I \) (\( H_i \) is its own inverse). Each \( H_i \) is computed in order to zero the elements beneath the diagonal of \( A \) on column \( i \), thus

\[
(H_n \cdots H_1) \cdot A = R, \\
A = (H_1^{-1} \cdot H_2^{-1} \cdots H_N^{-1}) \cdot R, \\
Q = H_1 \cdot H_2 \cdots \cdot H_N.
\]

\( Q \) is orthogonal, so \( Q^{-1} = Q^T \), i.e. inversion has no loss of precision. This is useful in solving systems of linear equations \( A \cdot X = B \): QR produces the result \( R \cdot X = Q^T \cdot B \), which is then solved to high precision using back substitution.

5.4.1 The LAPACK Implementation

We address the LAPACK formulation of a blocked QR factorization, as coded in \texttt{DGEQRF}, which relies heavily on [57] in forming the compact WY representation which saves storage space (versus the WY representation of [62]). The compact WY representation of \( Q \) is \( Q = (I - Y \cdot T \cdot Y^T) \), where the columns of \( Y \) represent the vectors \( v \) computed for each Householder transform, and \( T \) is upper triangular.

Consider factoring a double precision matrix \( A \in \mathbb{R}^{M \times N} \) using a blocking factor of \( N_b \). This is done iteratively; we first factor the column panel \( A[1..M, 1..N_b] \) using the routine \texttt{DGEQR2}. Thus at the end of the panel factorization, \( Y \) is \( M \times N_b \), and because the vector length decreases
by one for each column, it is lower trapezoidal and stored beneath the diagonal of the original $A$ matrix$^{22}$.

Figure 5.1 illustrates the basic operation. The panel is factored by the routine DGEQR2 into the upper triangular $R_p$ and lower trapezoidal $Y_p$. Then the routine DLARFT uses $Y_p$ to compute $T$ in a work area that is $N_b \times N_b$ elements, and finally, DLARFB is used to compute $Q$ times the trailing matrix portion of $A$, using the identity $Q = (I - Y_p \cdot T \cdot Y_p^T)$, producing $R_t$ (the upper portion of the final $R$) and $A_T$ (gray area, and the new matrix to factor). The dotted lines indicate the next iteration of the algorithm. Note that building $T$ is not necessary on the final iteration.

Figure 5.1: LAPACK Blocked QR Factorization

$^{22}$Unlike the textbook QR factorization, LAPACK forces $v$ to have 1.0 as its first element; thus this element is not stored. $R$ is stored on and above the diagonal of $A$, thus the factored $A$ replaces the original $A$. 

64
5.5 Applying PCA to QR Panel Factorization

To describe our work we need to further detail the implementation. In the default ATLAS multicore implementation of this algorithm, the DGEQR2 and DLARFT operations are performed by a single core (these routines use the Level 1 and 2 BLAS, which are not parallelized in ATLAS). DGEQR2 consists of Level 1 and Level 2 BLAS routines, and as discussed these are bus-bound and thus often scale poorly with \( p \) even when they are threaded (threading these routines on some platforms can actually cause slowdowns due to bus contention). The DLARFB operation uses two Level 3 BLAS routines, DTRMM and DGEMM, which have parallel implementations, which we did not change.

Figure 5.2 provides an outline of the DGEQR2 process, which is described more fully in the following paragraphs. In our original paper we parallelized DLARFG as found in LAPACK 3.1.1; however DLARFG was replaced by DLARFP in LAPACK 3.2 so in our extension work we parallelized DLARFP. No impact on either the parallelization or performance was observed (see [39]).

PCA uses a data ownership model of parallelism along the rows; so in Figure 5.1 we divide the column panel (the first \( N_b \) columns of \( A \), shown as \( R_p \) and \( Y_p \)) horizontally into a stack of \( p \) roughly equal-height blocks, one block per core. We assign cores multiples of 8 rows at a time because 8 doubles fills one cache line on both of our test machines. Any “leftover” rows are assigned to the top block, on the theory it has less work to do by virtue of owning the triangular portion of \( Y_p \). There are three operations that require the full panel height to produce a global result, and two additional operations on the full panel height that are parallelized. We will take each of these five
operations in turn, referring to Figure 5.2. All the cores loop over all $N_b$ columns. The asterisks in Figure 5.2 indicate synchronization points within the loop.

1. **DNRM2**: To compute the Householder vector we need the 2–norm of the full column. The 2–norm requires the square root of the sum of the squares. This is actually complicated in LAPACK, because it is coded to preserve precision and avoid unnecessary overflow. The idea is simple: as it traverses the vector it separately keeps track of the largest magnitude number found thus far and scales all numbers to it; i.e. before squaring, it multiplies new numbers by the inverse of the scaling number, and then adds the result to the sum of squares. However, occasionally a new largest magnitude must become the new scale, so it must rescale the sum-so-far to the new scale. When complete, DNRM2 can return $(\text{scale} \cdot \sqrt{\text{sum of squares}})$ as the final answer, thus avoiding the possible overflow of having ever squared the largest magnitude number, and also the potential loss of precision created by summing smaller squares to the largest squared number.

To parallelize this and preserve the behavior, each core must return for its portion of the column both the scale it found and the sum of squares. We combine these two at a time in a $\log_2(p)$ operation, using the obvious arithmetic\textsuperscript{23}. Core 0 then produces the final answer and signals the other cores that it is available.

2. **DSCAL**: All cores do the math of producing the Householder scalar values for TAU, Beta, etc. “Problem” vectors must be re-scaled and the 2–norm recomputed; we do this as well, looping back to the DNRM2* step as necessary. Note the decision to rescale and recompute is based on the final global value, so if one core rescales, they will all make the same decision. Problem vectors are extremely rare; normally each core proceeds to scaling its portion of the column and then immediately to the next step. This scaling produces the Householder vector $v_i$ (for column $i$); the Householder matrix is $H_i = I - \tau_i \cdot v_i \cdot v_i^T$.

3. **DGEMV\textsuperscript{T}***: Having computed the Householder vector $v_i$, we must now update the trailing matrix of the panel by multiplying it by $H_i$. In Figure 5.1 imagine $Y_p$ is divided vertically on

\textsuperscript{23}Given $\text{scale}_1$, $\text{sum}_1$, $\text{scale}_2$ and $\text{sum}_2$, and presuming $\text{scale}_1 > \text{scale}_2$, the proper combined sum of squares would be $\text{scale}_1^2 \cdot \text{sum}_1 + \text{scale}_2^2 \cdot \text{sum}_2$, but we want $\text{scale}_1^2$ factored out, and we want to avoid computing the squares of either $\text{scale}_1$ or $\text{scale}_2$. Thus we compute the pair $\left[ \text{scale}_1, \left( \text{sum}_1 + \left( \frac{\text{scale}_2}{\text{scale}_1} \right)^2 \cdot \text{sum}_2 \right) \right]$. 

66
some column $i$; and we will call $Y_L$ the columns $1 \ldots i$ and $Y_R$ the columns $(i + 1) \ldots N_b$. We need a new $Y_R$, which we will call $Y'_R$. We compute that as $Y'_R = H_i \cdot Y_R$.

Replacing $H_i$ with $(I - \tau_i \cdot v_i \cdot v_i^T)$, we must compute $Y'_R = (I - \tau_i \cdot v_i \cdot v_i^T)(Y_R)$, and this expands to $Y'_R = Y_R - \tau_i \cdot v_i \cdot (v_i^T \cdot Y_R)$. Working from the right we can first compute the vector $W = v_i^T \cdot Y_R$. It is easier to compute $W^T$, and since $W$ is a vector this requires no actual transpose operation. So this step will compute $W^T = (Y_R)^T \cdot v_i$.

Because we divide $Y_{pR}$ on rows, our cores each compute a vector that is $(N_b - i)$ elements long; and we add them in a $\log_2(p)$ combine. This is a sync point because in the next step, all cores will need the combined global $W$ vector to complete this operation.

4. **DGER:** This is the second step of applying the Householder matrix. In the previous step we computed $W = v_i^T \cdot Y_R$, now we complete the application of $H_i$ to $Y_R$. By substitution we have $Y'_R = Y_R - \tau_i \cdot v_i \cdot W$, which we recognize as a simple rank-1 update of $Y_R$. **DGER** accomplishes that. No synchronization is necessary after **DGER**, so once each core completes its **DGER** it proceeds to the next column and begins work on its **DNRM2**. After the last column, they can proceed to the **DLARFT** computation (or the threads can exit if **DLARFT** is not needed).

5. **DLARFT, DGEMV$^T$**:* After completing **DGEQR2**, we must build the $T$ matrix with a call to **DLARFT**, where we primarily parallelize the work in **DGEMV$^T$**. The **DGEMV$^T$** portion of this is quite similar to the previous **DGEMV$^T$** of **DLARF**. We are using the Schreiber/van Loan[57] method of computing $T$, so we only update one column at a time. But unlike the previous **DGEMV$^T$***, this time we are using the left portion of the $Y_p$ matrix at each column, and computing the vector $Y_p^T \cdot v$.

All cores compute their share of this vector and we sum them ($\log_2(p)$) to produce a final vector. Core 0 alone completes the update of $T$ with a **DTRMV**. We require synchronization because the computation of the new $T$ must be completed before the core can begin the next **DGEMV$^T$** (so technically Core 0 does not signal completion until it completes the **DTRMV**). Currently this **DTRMV** is too small $(N_b \times N_b)$ to gain any performance from parallelization, and is done by Core 0 alone, but if future $N_b$ grow large enough then parallelizing this step should be straightforward.

To minimize synchronization overhead we use a hardware-based synchronization that relies
on cache coherence instead of OS calls. We also tweaked the ATLAS tuning framework to tune
\texttt{DGEMV} and \texttt{DGER} for separate kernels to use when operands are pre-loaded to the L2-cache, which
PCA almost guarantees, and we provided new vectorized kernels within the ATLAS framework.
Both the parallel and serial timings reported in this chapter use the improved kernels. Cache-
specific kernel tuning is an area we hope to explore further in future work. We will now outline our
approach for lightweight shared-memory synchronization, and then present timings to demonstrate
the benefits of PCA for both LU and QR.

5.5.1 Implementation Note: Lightweight Memory Synchronization

Our PCA algorithm depends on low-overhead synchronization, which we can implement cheaply
on cache-coherent systems through memory (therefore running at the speed of hardware, rather
than the OS, as in the case of condition variables). Since all participating cores are dedicated to
the panel factorization, we can use spinlocks without loss of performance. Because these memory
synchronizations are prone to subtle programming bugs, we outline the approach we use here (this
approach is well known in several areas and we are not claiming any originality; we discuss it here
for implementors who may be unaware of the details of this long-standing technique).

In our code we assign one thread per core and the thread ranks start at zero and ascend.
Each thread knows its rank, and the rank is used to index two critical arrays (declared as \texttt{volatile}
to avoid register assignment). Each array is $P_m$ long, where $P_m$ is the maximum number of threads
that can be used. The first such array (\texttt{active}) is a simple boolean array; before the master
program launches any threads, it sets $\texttt{active}[\text{rank}]$ to true if the thread of rank $\text{rank}$ will be
participating in the panel factorization; otherwise it is set to false. This allows active threads to
check which partners are participating in the operation; this information will be needed when local
information is combined to produce global results.

The \texttt{sync[]} array has one integer per thread, and the master program initializes all of
these to $-1$. In our panel operations two types of synchronization are necessary; one for the $\log_2$
combine and the other to signal that a global result is finished and the threads depending on it
may proceed. All our synchronizations have the effect of a barrier, so the first part of the idea here
is that when a thread finishes its work up to a sync point, it signals interested partner threads by
incrementing its sync array entry ($\texttt{sync[i\_am]}++$).

If a thread $T_i$ is responsible for combining a result, say $T_4$ must wait on $T_5$ to finish, then
$T_4$ must wait until ‘sync[5] > sync[4]’. We do this by spinning on memory; e.g. if $i_{\text{am}} = 4$ and $\text{him} = 5$:

$$\text{if (active[\text{him}]) while (sync[i_{\text{am}}] \geq sync[\text{him}]);}$$

When this loop exits, the fact that $T_5$ incremented $\text{sync}[5]$ implies it is done with its work. After $T_4$ has done its combine (more generally, all of the combines for which it is responsible) then it increments $\text{sync}[4]$.

To wait on a global result, after each thread completes its work and has incremented its own sync array entry, it can wait for $\text{sync}[0]$ to match its own entry; e.g. ‘while ($\text{sync}[0] < \text{sync}[i_{\text{am}}]$)’, assuming rank 0 is the destination of the combine.

An easy example is the LU pivot operation. Each thread must compute a local maximum, and these must be combined to produce a global maximum; the row that holds that maximal value is then exchanged with the active row. $T_0$ does the final combine to discover the global maximum, and then performs the exchange of the rows (which isn’t worth parallelizing). Once that exchange is complete, $T_0$ increments $\text{sync}[0]$, which allows all the other threads to proceed.

This general technique is easily used in any $\log_2$ (or linear) combine operation.

### 5.6 Experimental Methodology

When performing serial optimization on a kernel with no system calls, researchers often report the best achieved performance over several trials\[68\]. This will not work for parallel times: parallel times are strongly affected by system states during the call and vary widely based on unknown starting conditions. If we select the best result out of a large pool of results, we cannot distinguish between an algorithm that achieves the optimal startup time by chance once in a thousand trials from one that achieves it every time by design. An average of many samples will make that distinction.

In our timings, the sample count varies to keep experiment runtimes reasonable: for all panel timings, we take the average of 100 trials; for full square problems, we use 50 trials to factor matrices under $4000 \times 4000$ elements, and at least 20 trials for larger matrices. We flush all processors’ caches between timing invocations, as described in \[68\].

The libraries used were ATLAS (v-3.9.11), GotoBLAS (v-r1.26), PLASMA (v-1.0.0), and LAPACK (v-3.1.1). All timings used the ATLAS timers and ATLAS’s LAPACK autotuner[67] was
used to empirically tune the LAPACK blocking parameter to the GotoBLAS, and to autotune both blocking factors used in PLASMA. Goto uses the LAPACK outer routines (DGEQRF, DGEQR2, etc); PLASMA uses its own versions of the outer routines and the ATLAS serial BLAS. ATLAS uses C equivalents of the LAPACK routines or a newly written DGEQRR and DGEQR2 that are drop-in replacements for DGEQR2. Both ATLAS and Goto libs are tuned separately for each test platform to provide high performance.

Unlike ATLAS, the GotoBLAS parallelize DGEMV and DGER and thus provide us with a ready-made implementation of the traditional route to panel factorization parallelization. Goto, ATLAS and PLASMA, properly tuned, are the state-of-the-art for comparison purposes.

The ATLAS BLAS create and destroy threads for each BLAS call, using the master last algorithm [8]. Our PCA factorizations creates and joins $p$ threads for each call to the panel factorization (in addition to the threads that the Level 3 BLAS call will create). Both PLASMA and the GotoBLAS use pthreads and affinity, and both employ “persistent” threads\(^{24}\).

We timed on two commodity platforms, both of which have 8 cores in two physical packages. We use Linux, but even different minor releases of Linux can have scheduler differences that change timing significantly, so we provide kernel version information here. Our two platforms were:

(1) **Opt8, O8**: 2.1Ghz AMD Opteron 2352 running Fedora 8 Linux 2.6.25.14-69 and gcc 4.2.1, GotoBLAS r1.26, PLASMA 1.0.0, LAPACK 3.1.1 and ATLAS 3.9.11 (on this platform, we did not use ATLAS’s architectural defaults, since the full ATLAS search yielded noticeably faster BLAS),

(2) **Core2, C2**: 2.5Ghz Intel E5420 Core2 Xeon running Fedora 9 Linux 2.6.25.11-97 and gcc 4.3.0, ATLAS 3.9.11, Goto r1.26, PLASMA 1.0.0, LAPACK 3.1.1 and ATLAS 3.9.11.

Each physical package on the Opt8 consists of one chip, which has a built-in memory controller. The physical packages of the Core2 contain two chips, and all cores share an off-chip memory controller.

ATLAS tunes GEMM (the Level 3 BLAS routine providing GEneral Matrix Multiply) differently on these two platforms. On the Core2 platform the ATLAS blocking factor is 56, and on the Opt8 we use 72. Our best performing QR blocking factors will turn out to be multiples of

\(^{24}\)Technically, the GotoBLAS delay thread exits for a fraction of a millisecond, so it can reuse threads if the BLAS is called again quickly enough. In the context of our factorizations this is always true so this is the equivalent of creating persistent worker threads.
these when ATLAS is performing the GEMM.

To compare our panel factorization performance to the Goto and PLASMA libraries, we choose a panel width of 128. This is not ideal for ATLAS or PLASMA. The GotoBLAS does well with multiples of 64. The reason for choosing 128 is to make a fair head-to-head comparison of the methods with a realistic panel width that doesn’t explicitly favor our PCA or the ATLAS-tuned BLAS we are using. PLASMA does not possess a routine specialized for panel factorization, but since panel factorization is merely a factorization of a heavily non-square matrix, we report its best performance on that task.

5.7 Impact of these techniques on runtime

The measurements that interest us most are

- **Panel Speedup:** How much faster our panel factorization is than the alternatives, such as serial panel factorization or parallelizing using the parallel Level 2 BLAS routines (we show superlinear speedups versus serial asymptotically);

- **Overall Speedup:** The impact of faster panel factorization on the overall QR performance (we show up to 35% speedup over ATLAS/LAPACK’s previous method);

- **Scaling:** Whether this panel factorization speedup can be expected to scale with $p$ (we show excellent scaling);

The blocked version of QR requires more flops than the unblocked version. On square problems, the extra flops required range from about 9% for 1000, declining quickly to about 1.5% for 8000. Note, however, that all MFLOP rates reported in this chapter always use the unblocked flop count, as discussed below.

In measuring panel performance, we note that in the LAPACK blocked algorithm DGEQRF, DGEQR2 is followed by DLARFT, the two routines we have parallelized. When we report Panel MFLOPS, we are reporting the MFLOP rate achieved on both of these operations, for just the first panel of the factorization. We report the rate based on the effective flops, meaning we count the time of both DGEQR2 and DLARFT, but do not count any flops consumed by DLARFT. The PLASMA approach does not provide separate routines for a DGEQR2 and DLARFT; thus we time how long it
takes to factor a panel. Although technically this is only the \texttt{DGEQR2} equivalent time, we believe PLASMA will require extra flops roughly equal to those consumed by the canonical \texttt{DLARFT}.

For panel timings all of the panels are 128 columns wide (i.e. $N_b = 128$) and we use the average of 100 trials per problem size. We use \texttt{DGEQR2}'s flop count for all panel operations, which is given in equation 5.7.1 for an $M \times N_b$ factorization, where $M \geq N_b$ (this equation comes from LAPACK’s \texttt{dplas.f}).

\[
f_c = \frac{1}{6} \cdot (12 \cdot M \cdot N_b^2 + 12 \cdot M \cdot N_b + 6 \cdot N_b^2 + 28 \cdot N_b - 4 \cdot N_b^3)
\]  

(5.7.1)

Figure 5.3: ATLAS QR and LU Panel Speedups

The most critical result is the speedup of our approach over the best known algorithm (Figures 5.3(a) and (c)), and over the original LAPACK serial algorithm (Figures 5.3(b) and (d)).

72
In Figure 5.3 the X-axis is always the panel height (the panel width is 128 elements); and the Y-axis is the speedup we obtained. Thus in Figure 5.3(d), our QR PCA factorization of a 13000 × 128 element panel is 15 times faster than the serial version on the Core-2 platform. Both platforms have 8 cores, so notice we achieve superlinear speedup over serial for both QR and LU on both test platforms. The superlinear speedups are due to cache reuse: when the problem exceeds one core’s cache, the operations run at memory speed, but when the panel is cache-contained, it can run roughly at cache speed. Our approach uses all caches in parallel, and when even this roughly $p$-fold increase in cache space is not enough to contain the panel, we recur until the problem can be cache partitioned. For best performance, it is necessary to discover empirically how much cache space can be effectively used by a core (remember that caches are typically combined instruction/data, sometimes shared amongst processors, and often use non-LRU replacement, all of which tend to lower the effective cache size). For the Core2 (Opt8) we found a cache threshold of 1.2MB (512KB). The asymptotic speedup over serial for QR panel factorization on the Core2 (Opt8) speedup was 19.4 (11.2). The asymptotic speedup over serial for LU panel factorization on the Core2 (Opt8) was 17.9 (15.9).

Recursive formulations represent the current state-of-the-art in panel factorization; these formulations improve performance through increased Level 3 BLAS use, which provides decent parallel speedups for reasonable sized problems. The LU recursion stops at two columns; the QR recursion stops at 16 columns (the highest-performance stopping point on both machines). Both QR and LU recurse only on the column dimension.

Comparing against the recursive formulations, our PCA approach asymptotically gets speedups of 2.5 (2.0) on the Core2, for QR (LU). We get similar improvement on the Opt8, where we see a speedup of 2.25 (2.0) asymptotically for QR (LU). If we look at the collective L2 cache sizes, we can fit panels of 4096 × 128 or 9800 × 128 in the Opt8 and Core2 respectively. As core count increases, the size of the problem that fits in the collective cache should grow as well, but even when it is exceeded recursion can ameliorate the performance loss. Initial experiments on the Core2 show a roughly 20% decline in absolute performance for panels with 48,000 rows, yet this remains almost 2.5 times faster than the state-of-the-art recursive QR.

We next compare PCA for the QR panel factorization to the most credible alternatives in HPC (High Performance Computing) libraries that are freely available today. By combining LAPACK with the GotoBLAS (the GotoBLAS do not provide a native QR implementation) we
can achieve a state-of-the-art traditional panel factorization, where all of the parallelism comes from the BLAS (we cannot do this with the ATLAS BLAS, because ATLAS does not parallelize the Level 2 BLAS). This method of parallelism is labeled PGOTO in Figure 5.4. We also time the PLASMA library [21] which aims to completely redesign LAPACK for large-scale parallelization. It parallelizes the panel using a tiling approach and dynamic assignment of operations to cores through the use of a DAG (Directed Acyclic Graph) of computational dependencies. PLASMA increases parallelism at the cost of performing significant extra flops.

**Figure 5.4: QR Panel Factorization MFLOPS on Core2, Opt8**

In Figure 5.4 all libraries are factoring the same size panel ($M \times 128$). Each library has been tuned separately to find its best performance on each problem size (i.e. we allowed different
blocking factors for every $M$). We did an exhaustive search for each library, and took the blocking factors that produced the best average speed.

PCA and PGOTO are both building the $T \in \mathbb{R}^{128 \times 128}$ matrix required for the trailing matrix update (by `DLARFB`). PCA is the first bar (clear with diagonal lines) in each set; PGOTO is the second bar (black) and PLASMA is the third bar (gray). Note the scales are different; so PLASMA is getting roughly the same flop rate on both platforms. PGOTO begins to decline in performance for large problems, probably due to cache effects.

PCA achieves by far the fastest performance on the panel for both platforms. The speedups for the Core2 (Figure 5.4(a)) are easy to characterize, because the Parallel GotoBLAS and PLASMA plateau at nearly identical speeds of 6000 MFLOPS. PCA ramps up from a speedup (over both) of 1.5–1.6 at 1000 to a speedup of 2.3–2.4 at 3000. The dip in PCA performance at $M = 9000/10,000$ is where the benefit of copying the data has faded; for $M = 11,000$ and above, PCA is processing the data in-place. The asymptotic speedup is about 2.0 against PLASMA, and 2.42 against PGOTO.

On the Opt8 (Figure 5.4(b)) PLASMA and PGOTO differ markedly. PLASMA displays admirable consistency very quickly, settling in at 5600 MFLOPS, while PGOTO is declining throughout, finally reaching a speed of 775 MFLOPS. We suspect it is struggling with cache overflow problems. PCA shows a clear dip at 4500, the last panel size it can factor without recursion, and another dip at 9000, the last panel for which copying provides a speedup. It recovers to an asymptotic speed of 8000 MFLOPS. Our speedup over PGOTO climbs fairly steadily from 1.3 to about 2.9 at 8000, and from there to 10.2 at 16000. Against PLASMA, PCA starts out with a speedup of 1.15 at 1000, but then varies between 1.3 and 1.5 for the rest of the graph.

In summary, these experiments demonstrate that our new panel factorization produces superlinear speedups over the original LAPACK serial algorithm. We also achieve double or triple the performance of the recursive parallel approaches. We are 1.5 to 2.5 times faster asymptotically than the traditional parallel BLAS-based panel factorization using the GotoBLAS, more than double the performance of tuned PLASMA on the Core2 and 30–50% faster on the Opt8.

### 5.7.1 Full Problem Performance

We have shown our approach to be the fastest panel factorization by a wide margin. Obviously, this means huge performance improvements if you are solving strongly overdetermined systems, but will
it have a noticable effect on the full algorithm running on large square problems? Figure 5.5 answers that question with a decided yes. This chart shows results for full QR on large square problems; the Y-axis charts the speedup of the ATLAS-tuned full QR (DGEQRF) using our PCA panel factorization over the ATLAS-tuned DGEQRF which uses the default LAPACK panel factorization. The X-axis runs to 8000 by 8000 problems, in steps of 500.

Figure 5.5 shows that our approach improves the asymptotic perfomance of the full problem by almost 30% (25%) on the Core2 (Opt8) on even the 8-node systems that are readily available today. There are two reasons for this boost; the first is simply that the panel factorizations have been reduced from 20% of the original runtime to less than 4% of the improved runtime. The second reason is more subtle: when the panel factorization is made to run faster, we can use it to factor a wider panel without a large performance penalty, which allows us to use a larger $N_b$ when doing our update of the trailing matrix. These new PCA-tuned $N_b$'s are 2–4 times the $N_b$'s selected when tuning for the serial panel factorization. Using larger $N_b$ allows the Level 3 BLAS to get greater cache reuse, which improves both serial and parallel performance. Unlike in LU, however, the optimal QR $N_b$ is still strongly constrained, since the number of extra flops rises with $N_b$. 

Figure 5.5: QR Full Problem Speedups, Core2 and Opt8
5.8 Scaling

We now would like to address scaling, more specifically weak scaling. Perfect weak scaling is experiencing the same performance/core as \( p \) is increased when the work per core is kept constant. Strong scaling is not what we aim for: we have shown superlinear speedups for fixed problem sizes, so for large problem sizes we have better than perfect strong scaling. Weak scaling, where each core has the same amount of work to do, and thus roughly the same amount of potential cache reuse to employ, is a much more honest measure of scalability for dense linear algebra. Figure 5.6 plots the weak scaling of our PCA algorithm. We keep the panel width constant (128), and the panel height for each core is 1000 (so when \( p = 8 \), we are factoring an \( 8000 \times 128 \) QR).

![Figure 5.6: QR Scaling, Core2 and Opt8](image)

In Figure 5.6(a), the X-axis is the number of cores used, the Y-axis is the MFLOP rate achieved. We can see that our scaling is quite good, as the performance goes up pretty much linearly with the number of cores. However, we are not achieving perfect weak scaling, as Figure 5.6(b) shows. In this figure, we have divided achieved MFLOPS from (a) by the number of cores to compute the MFLOPS per core. Here we can see the MFLOP rate per core is decreasing modestly. Almost all of this loss comes from the need to bring the panel into the cache, and the fact that the caches do not use true LRU line replacement. Section 4.6 on Cache Perfection explains in detail how non-LRU replacement can impact performance. In this case the issue is that not all of the panel is retained in the cache by the copy operation, thus there is some performance loss due to
bus contention. We performed an experiment (not shown) in which we traversed the data several times; until statistically 99% of it could be expected to reside in the L2 cache (again, ignoring interrupts and context switches); and we compared the execution times (excluding copy times). The algorithm then achieved an almost flat performance per core curve; in fact this experiment eliminated roughly 90% of the decline shown in Figure 5.6(b). The remaining 10% is probably mainly threading overheads and cache misses due to interrupts and context switches.

5.9 Extensions to Original PCA Work

We have expanded upon the original QR PCA work in the previous section, completing the work for new precisions and variants of the QR factorization. In the QR factorization, the “R” is an upper-triangular matrix. The first variant is called “QL”, which factors a matrix \( A \) into an orthogonal matrix \( Q \) multiplied by a Lower Triangular matrix \( L \). Two other variants are the transposed versions of these: \( RQ \) is \( (Q \cdot L)^T \) and \( LQ \) is \( (Q \cdot R)^T \), i.e.:

\[
A^T = Q \cdot R \Rightarrow A = (Q \cdot R)^T = R^T \cdot Q^T = L \cdot Q^T \quad (5.9.1)
\]

\[
A^T = Q \cdot L \Rightarrow A = (Q \cdot L)^T = L^T \cdot Q^T = R \cdot Q^T \quad (5.9.2)
\]

And since \( Q^T \) is also orthogonal, and the “Q” in the LQ and RQ factorizations refer to any orthogonal matrix, we can obtain an LQ or RQ factorization of \( A \) by transposing the result of a QR or QL factorization of \( A^T \). This is how ATLAS accomplishes these factorizations, for performance reasons: The QL and QR factorizations access data in contiguous columns (the data is column-major) and proceed column-by-column, while the RQ and LQ factorizations, as written for LAPACK, access data in non-contiguous rows and proceed row-by-row. The cost of the transpose copy is small compared to the performance gained by using the more cache and TLB friendly memory access pattern employed by QR and QL.

To save space we will not present all possible timings; there are four variants, in four precisions each, and multiple ways of comparing them.

The serial version of PCA in the (QR, QL, RQ, LQ) factorizations is, respectively, the routines (QR2, QL2, RQ2, LQ2), the Level-2 BLAS-based routines that LAPACK uses to factor a

\[\text{Translation Lookaside Buffer.}\]
panel. The generic label Q2 refers to any of these four. We will show super-linear speedups for all of these variants on 128-column panels.

However to demonstrate clearly the true benefit of PCA we will examine the best tuned performance of PCA, versus the default LAPACK implementation, the best tuned performance of the LAPACK implementation (as per [67]), and the best tuned performance of the ATLAS recursive panel factorization.\footnote{\text{The ATLAS recursive panel factorization is called (\texttt{GEQRR}) and due to [19]; the QL version (\texttt{GEQLR}) is a very straightforward adaptation of that approach.}}

The comparisons will be done for both the panel-shaped matrices and square matrices, and in each case we will separately tune each routine for the best possible performance (with the ATLAS and LAPACK tuning framework) on that shape. We believe this is the fairest possible comparison. Most users of LAPACK do not do this tuning, thus use of PCA would provide them with much more dramatic improvements. But to be perfectly fair we want to show that when all methods are tuned by the same ATLAS tuning framework, PCA provides a speedup over the best performance the LAPACK or pre-PCA ATLAS code can deliver on the same problem.

We illustrate five comparisons for the panel, and four for the full problem:

1. Q2: For panels only, this is the LAPACK Fortran routine that PCA is parallelizing. It is shown primarily to demonstrate parallel speedup of PCA over the same algorithm executed serially.

2. Default LAPACK: This is the performance of the default installation of LAPACK. One significant feature of this installation is the default width of all panels for these factorizations is 32, for all precisions.

3. Tuned LAPACK: The panel-width is tuned for different problem sizes for each precision, according to [67].

4. Recursive ATLAS: Users of ATLAS (without PCA) gain some performance using a recursive panel factorization (due to Elmroth and Gustavson [19]).

5. PCA ATLAS: The algorithm for parallelizing a panel discussed in this chapter. The ATLAS implementation of PCA is recursive; the recursion stopping point is when the panel (or sub-panel) will fit into the aggregate cache.
The precisions are S,C,D,Z for Single Precision Real, Single Precision Complex, Double Precision Real, and Double Precision Complex. Following the LAPACK convention, the names of our methods will consist of the precision followed by the two-letter algorithm identifier; e.g. “ZQL” refers to the QL factorization for double-precision complex elements.

Note that the complex arithmetic versions have approximately six times as many flops to execute for the same data load as the real versions. This makes them inherently less bus-bound, and we would expect a priori they would exhibit a higher flop rate and benefit less from PCA, which derives much of its benefit from reducing memory access overhead by exploiting caches.

ATLAS’s automatic tuning algorithm determines an effective cache threshold for the machine during installation, geared toward matrix multiplication. We use \( \frac{7}{8} \) of that threshold throughout the PCA algorithm to determine our panel recursion threshold because we found that using the full value caused performance drops of up to 30% for problems very close to the full boundary\(^{27}\).

The performance numbers here differ in detail (but not in character) from the original work. The original work highlighted a need to specifically tune the Level-2 BLAS routines (e.g. GEMV, GER) to improve performance on cache-contained problems, and that initiated a larger project to tune the Level-2 BLAS in general for both in-cache and out-of-cache problems. This improved performance for both the non-PCA and PCA versions of all variants, but the non-PCA versions benefitted more due to having greater room for improvement. That is ongoing work in the ATLAS group, and some of the performance variance we show here is due to imperfect selection or tuning of code variants on the test machines. Note that PCA, because it’s cache is effectively 8 times larger (or generally \( p \times \) for a \( p \) core system) can exploit in-cache tuned kernels on much larger problems than can a single-core algorithm.

\(^{27}\)Ultimately the PCA panel recursion threshold should be empirically tuned for each machine, and possibly each precision, but that is an implementation detail beyond the scope of this dissertation.
5.9.1 Super-Linear Speedups

We show the numbers first for the Core2. In Figure 5.7 we show super-linear speedup, on the Core2 hardware, for our QR, QL, RQ and LQ variants, in all precisions. The graph’s X-axis shows the variable panel dimension (height for QR and QL, width for RQ and LQ), the other dimension is fixed at 128. The Y-Axis shows the speedup achieved. This is an 8-core system, and we have shaded (light green) the area of the graph that shows the region of super-linear speedup.

Recall that RQ and LQ are accomplished simply by transposing the copy for LQ and RQ, respectively, so we should expect similar performances. We do not because of a performance defect in the process of the transposition. Although the results are accurate, this is a correctable flaw in the implementation that makes the data copy overhead approximately twice as large as necessary, and reduces the performance correspondingly. As a result our PCA operations on the 128-row panels (for RQ and LQ) are nearly identical to our recursive implementation. The necessary repair is...
straightforward but time-consuming; it is essentially a strategy of delaying the transpose operation until the PCA routine itself can accomplish it.

However, these routines do provide speedups on the full problems, because PCA allows for a larger blocking factor than our best alternative, recursive panel processing.

5.9.2 Core2 Panel Speedups

We now look at the performance of QR when factoring panel-sized matrices. In Figure 5.8 we chart, for all types and precisions, the Default LAPACK performance (blue bar) and the best performance the ATLAS framework finds (by changing blocking factors) for the LAPACK provided code (maroon), the ATLAS pre-PCA Recursive code (yellow) and the new ATLAS PCA code.
The X-axis is the panel-height, the Y-Axis is MFLOPS achieved. Note that all of the performance charts in this section follow this scheme. The inflection points in the performance curve are generally thresholds where an additional recursion takes place. ATLAS does not currently tune recursion points separately; if it did it might lessen the depth of these inflection points.

Each problem size compares the four methods at that problem size. All of these QR panels are 128 columns wide; which by design for this experiment does not favor any of the methods shown. In particular, it can happen that the recursive method does slightly worse than the tuned LAPACK method on these panels, but still performs better than the tuned LAPACK method on full problems. This contradiction is explained by the fact that when ATLAS tunes the block factors for full square matrices, the best performing block factor for the recursive panel factorization is

![Graphs of panel performance](image-url)
much larger (3 to 4 times larger) than the best performing block factor for LAPACK. In essence, 128 columns is too narrow a panel for recursive factorization to perform near its peak.

The SQR code 5.8(a) on the Core2 PCA is generally about 2.5× faster than the best previous alternative (which is usually tuned LAPACK). The same is true for the DQR code 5.8(b). The CQR code 5.8(c) on the Core2 is the best performer for PCA, reaching over 3× faster than the best alternative (a different result than our a priori expectation that complex arithmetic would benefit less from PCA) and the ZQL code 5.8(d) is the worst performer, but still on average about twice as fast as the best alternative (matching our a priori expectations for complex arithmetic benefitting less from PCA).

We turn next to the QL performances we achieve factoring panel-sized matrices, in Figure 5.9. The SQL code 5.9(a) on the Core2 PCA is generally about 2.5× faster than the best previous alternative (which is usually tuned LAPACK). The DQL code 5.9(b) averages about 2.3× faster.

Figure 5.9(c,d) shows the complex arithmetic versions. The CQL code 5.8(c) on the Core2 also averages 2.5× faster (like the SQL code), while the ZQL code 5.9(d) is the worst performer, but still averages 2.1× the speed of the best alternative, over all problem sizes.

As mentioned earlier, we have left off the RQ and LQ panel performance charts due to a correctable performance defect in implementation. However, we will include them in the full performance charts.

5.9.3 Core2 Full Problem Performance

In Figure 5.10 we report full problem performance for both QR and QL. In this graph we see perhaps the most common and realistic tuning scenario, tuning for square matrix factorizations. Each variant (except the Default LAPACK) has been tuned to find the panel width that produces the maximum overall speed by problem size; so we see here both the overall impact of tuning and the superior performance of PCA. In all but the smallest problem sizes, the algorithms stack up in performance in the same order listed on the chart: Default LAPACK is the worst performer, Tuned LAPACK is an improvement, Recursive ATLAS adds more, and PCA performance is the best. The remainder of the full problem charts are fairly uniform so we will present them without individual discussion; and discuss them collectively afterward.
Figure 5.10: QR (a,b,c,d) & QL (a,b,c,d), Core2, Full Problem Performances
Figure 5.11: RQ (a,b,c,d) & LQ (c,d,e,f), Core2, Full Problem Performances
On our two main implemented factorizations, QR and QL, both have average speedups on the Core2 of (19%, 15%, 16%, 12%) for (S,D,C,Z), respectively. Their peak speedups (which occur for \( N < 4000 \)) are slightly different; for QR (31%, 23%, 28%, 22%) for (S,D,C,Z), and for QL (28%, 24%, 34%, 25%) for (S,D,C,Z), respectively. These speedups are over the fastest alternative: the state-of-the-art recursive ATLAS panel factorization.

These charts are remarkably uniform, suggesting PCA is a general enough approach that it translate across these 16 combinations of algorithms, types and precisions with roughly equal (and impressive) improvements: Recall that PCA is applied only to the panel factorizations, which are typically around 20% of the full problem run time (with a parallelized Level 3 BLAS, which we use here). With panel speedups of \( \approx 2.5 \) over the fastest alternative, just the parallelization part of PCA typically reduces the full-problem run time by about 12%. PCA also typically allows for wider panels than the alternatives (sometimes up to four times as wide), and this translates into better performance by the Level 3 BLAS in the non-PCA portion of the factorization.

Although the performance defect of the transpose cases (RQ and LQ) is an implementation detail, once it is corrected we believe PCA performance on the transpose cases will closely approximate the PCA performance on QL and QR, which will provide an even larger margin of improvement over the recursive panel factorizations.

5.9.4 Opt8 Panel Performance

The key difference for the Opteron 8 platform is a smaller cache per core than the Core2 (512K vs. 3072K), and a slower clock (2.1 MHz vs. 2.5Mhz). In Figure 5.12 we look first at the PCA performance versus Q2, its serial counterpart. Unlike the Core2, notice we do not achieve super-linearity in all precisions; particularly single-precision real and complex in the QR and RQ variants (although the speedups are still significant).

As before, these speedups translate into improvements in the full problems, as shown in Figures 5.13 and 5.14. As on the Core2 the charts are fairly uniform. The average full-problem improvement for the QR and QL variants are (25%, 24%, 20%, 13%) for (S,D,C,Z), respectively. The peak full-problem speedups are achieved on smaller problems (\( N < 4000 \)) and for QR are (34%, 44%, 40%, 25%) for (S,D,C,Z), respectively. For QL the peak full-problem speedups are (31%, 34%, 30%, 25%) for (S,D,C,Z), respectively.

We also see in these figures instances for which PCA on the largest problems is barely
Figure 5.12: PCA vs. QR2(a), QL2(b), RQ2(c), LQ2(d) on Opt8
Figure 5.13: QR (a,b,c,d) & QL (a,b,c,d), Opt8, Full Problem Performances
Figure 5.14: RQ (a,b,c,d) & LQ (c,d,e,f), Opt8, Full Problem Performances
better than the recursive factorization method, or (in one case, DRQ) is very slightly worse than the recursive factorization method: see 5.13(g,f) and 5.14(b,c) (corresponding to CQL, DQL, DRQ and CRQ respectively).

The particular instances are for larger problems ($N \geq 6000$). These are not a problem with PCA, the lesser performance is a side effect of a known tuning tradeoff ATLAS employs. Namely, in order to have the tuning process complete in a reasonable amount of time (e.g. a few hours), the ATLAS tuning algorithm assumes blocking factors will be monotonically increasing with problem size, or at least that enforcing monotonicity won’t impair performance by much. This allows a binary search with a restricted number of values, a necessity when every tested instance may require 1-3 minutes of compute time to complete. On these particular problems an exhaustive search for the best-performing blocking factor continues the trend of the smaller problems with performances 10% to 20% faster than the recursive version. The issue is that for PCA, the best performing blocking factor is not monotonically increasing, it rises and then declines with $N$, because the aggregate cache-size is fixed, so increasing the rows each core must hold decreases the columns it can accommodate, and the latter is directly tied to the blocking factor.

It is conceivable a faster tuning algorithm could be designed specifically for PCA that incorporates more direct knowledge of this cache-size constraint, but that is an ATLAS-specific issue beyond the scope of this dissertation. Thus we report the actual results of the ATLAS tuning, since at the time of this writing these are the performances we are certain ATLAS can deliver.
5.10 Hessenberg Reduction Introduction

The Hessenberg Reduction is the major first step in finding the Schur Decomposition. In the Hessenberg Reduction of a square matrix $A \in \mathbb{R}^{N \times N}$, we compute $Q$ and $H$ s.t. $Q$ is orthogonal (i.e. $Q^{-1} = Q^T$, like in the QR factorization) but $A = Q \cdot H \cdot Q^T$, where $H$ is in Hessenberg form\(^{28}\).

Notice $Q$ appears as a multiplier on both sides of $H$, for this reason the Hessenberg is called a "two-sided" factorization, while the QR and LU factorizations (where we compute $A = Q \cdot R$ or $A = L \cdot U$, respectively) are called "one-sided" factorizations.

In the Hessenberg, because $A = Q \cdot H \cdot Q^T$ and $Q^T = Q^{-1}$, $H$ is similar\(^{29}\) to $A$.

The Schur Decomposition, which we do not discuss in this dissertation, is a method of finding the eigenvalues and eigenvectors of the upper Hessenberg matrix $H$. With those in hand and given $Q$, we can find the eigenvectors of $A$ with just a matrix multiply\(^ {30}\).

There is more than one mathematical approach to accomplishing the Hessenberg reduction, but the most recent\(^ {31}\) LAPACK approach forms $Q$ as the product of Householder matrices, one per column of $A$ (except the last). In the simplest formulation of this approach, this is a three step process executed on each column of $A$, with data dependencies between each step. For a given column $i$:

1. Compute $Q_i$, the Householder matrix that annihilates all elements below the first sub-diagonal in column $i$ of $A$.
2. Update $A$ on the left, s.t. $A' = Q_i \cdot A$.
3. Update $A$ on the right, s.t. $A'' = A' \cdot Q_i$.

To improve performance, LAPACK implements this using a panelized approach, meaning it partitions the matrix on the column dimension. We call each partition of $Nb$ columns a "panel". A panel is factored in place, with updates to the rest of the matrix (that portion to the right of the panel) accumulated and applied using Level-3 BLAS routines after the panel has been factored.

---

\(^{28}\)Upper Triangular plus a the first sub-diagonal.

\(^{29}\)I.e. $H$ has the same eigenvalues as $A$, but not necessarily the same eigenvectors.

\(^{30}\)If $(\lambda, v)$ is an (eigenvalue, eigenvector) pair of $A$ s.t. $A \cdot v = \lambda \cdot v$, then $Q \cdot H \cdot (Q^T \cdot v) = \lambda \cdot v \Rightarrow H \cdot (Q^T \cdot v) = \lambda \cdot (Q^T \cdot v)$, so $(Q^T \cdot v)$ is the corresponding eigenvector of $H$. But $Q \cdot (Q^T \cdot v) = v$, so multiplying the eigenvectors of $H$ on the left by $Q$ produces the eigenvectors of $A$.

\(^{31}\)LAPACK version 3.2.1, [29]
This is almost identical to the approach of the \(QR\) factorization discussed earlier, with the added requirement of multiplying \(A\) by \(Q_i\) on both sides before proceeding to the next column.

The panel operation is similar to the column operation in steps, and the routine that factors a panel returns work-space matrices that are used to complete the left-side multiply and right-side multiply required to transform the \(A\) matrix to the right of that panel, then the next adjacent panel is processed, and so on. However, unlike the \(QR\) and LU factorizations in which data access can be entirely restricted to the panel, in the Hessenberg the entire trailing matrix must be read for each column. This fact will prove to be a major stumbling block in implementing PCA for the Hessenberg factorization, as we detail below.

In our experiments, we measure both how long we spend processing a panel and how much time an entire reduction takes. We call these the ”panel time” and ”problem time.” The reason for measuring both is that our approach was originally intended to improve performance on the panel time, and we wanted to measure how effective we were on that. We also measured problem time to report whether a performance boost on the panel component of the problem would have an overall impact on the full problem performance.

Just as in the \(QR\) factorization, LAPACK saves space by storing only a vector for each \(Q_i\) (see [57]), so within a panel, the two last two above steps involve two Level-2 BLAS operations\(^{32}\) for each step. LAPACK also implements the optimizations in [26], which permits some of the updates within the panel to be delayed and grouped into Level-3 BLAS operations, which further improves performance over previous LAPACK implementations. Despite these improvements, on every column LAPACK must execute a large Level-2 operation in step 3, a matrix-vector multiply (\textit{GEMV}) which dominates the time required to complete the Hessenberg reduction of matrices significantly larger than a single core’s cache.

We call this the “\textit{Dominant GEMV}”: a matrix-vector multiply in which the matrix is \(\mathbb{R}^{N \times (N-i)}\). The Dominant \textit{GEMV} requires the entire trailing matrix to be streamed through memory (meaning all of \(A\) to the right of the current column \(i\)). This is an out-of-cache bus-bound operation during most of the reduction process, and it is the single most expensive and performance limiting step of the entire Hessenberg reduction as performed by LAPACK. Figure 5.15 illustrates this performance bottleneck for one of our test systems, the Opt8. It shows how much of the problem time (and how much of the panel time, which is the natural target of PCA) is spent on just this one

\(^{32}\text{GEMV, a matrix vector multiply, and GER, a rank-1 update.}\)
operation. These are average timings of the existing LAPACK routine without any modifications. Notice the scale begins at 60% of total run-time, for a relatively small problem, and rises quickly to about 85%. We also chart how much of the panel processing time is consumed by the same Dominant GEMV: It begins around 82% and rises to over 98%.

We’d expect a similar chart on any system with caches and a much faster compute bandwidth (flop rate) than memory bandwidth. As \( N \) grows the Hessenberg is essentially bus-bound by the Dominant GEMV. This makes the prospect of gaining any significant performance improvement by parallelizing the cacheable parts of the panel quite bleak. Clearly, even a super-linear speedup on 2% of the problem will produce less than a 2% speedup.

As a cautionary tale, we implemented a form of this approach for comparison purposes, with unexpected (and distressing) results. The key concern is that if all the cores did a share of the Dominant GEMV, it would replace their cached data in the process of the computation. If only one core did the work, it would replace everything in that core’s cache. Whichever core suffered this fate would not be using in-cache data when it processed a panel, then due to our pervasive synchronization that core would drag all the other cores to its out-of-cache speed.

To counter this, we had the idea of dedicating one of our eight cores to just the Dominant GEMV operation, on the theory that (because we had shown good scaling with PCA in our QR work)
the other seven cores could still speed up the rest of the panel factorization. They did do that, but this pinned\textsuperscript{33} out-of-cache \texttt{GEMV} ran significantly slower (on average about 25\% slower) versus the identical code and data running unpinned. In further experiments, this impairment persisted to some degree any time the \texttt{GEMV} was restricted to run on a subset of the cores, although it is almost completely gone once a choice of four cores is allowed. Even on relatively small problems, the speedup on the cached panel failed to offset this “pinned \texttt{GEMV}“ impairment. Since the \texttt{GEMV} is so dominant (see Figure 5.15) this should not surprise us.

It is possible that a better understanding of this impairment may lead to new insights into parallelization performance improvement, especially with pinned code. However, that investigation was beyond the scope of our research goal, which was to expand upon our previous PCA contribution. Since parallel \texttt{GEMV} is not related to PCA, we do not discuss it further here (although we are investigating it now as a separate research project).

Our conclusion from the early experiments was that for PCA to be effective it must be matched to the data dependency pattern of the algorithm. The Hessenberg as implemented in LAPACK\textsuperscript{34} does not employ an independently processed panel: the data dependency pattern still requires access to the full matrix to complete each column. In contrast, the QR and LU panelized factorizations have no data dependencies outside the current panel, so PCA can be applied to just the panel.

It is possible to use alternative factorizations producing a blocked-Hessenberg (i.e. more than one sub-diagonal) form, such as using Givens matrices or the PLASMA\textsuperscript{21} approach, but these rely on different mathematics, produce different error bounds, require different stability proofs and all new code. Both of these examples (a block Givens approach and the blocked PLASMA approach) would benefit greatly if adapted for PCA, but a key claim we wish to make for PCA is that it isn’t necessary to change the math. In the case of the LAPACK approach to Hessenberg reduction, this limits the scope of PCA to matrices that can fit in the collective cache (or nearly so).

When the full matrix fits in the collective cache (or nearly so) our Hessenberg PCA delivered superlinear speedup versus the serial alternative (\texttt{DGEHD2}) and over 3× speedup versus

\textsuperscript{33}We call code “pinned” when affinity is employed to restrict it a single core; “unpinned” code can be moved by the OS to any available core.

\textsuperscript{34}The LAPACK routine is \texttt{DGEHRD2}. 
the best LAPACK alternative with a parallelized Level-3 BLAS (DGEHRD2), and this was true on both of our test platforms. However, as we shall see, once the matrix exceeds the collective cache our performance drops below the LAPACK alternatives.

**Routine Names and Descriptions:** LAPACK provides more than one routine to accomplish the Hessenberg reduction. For example, the routine DGGHRD will perform the task using Givens rotations to introduce the necessary zeros, while the routine DGEHRD performs the task using Householder transformations to introduce zeros. (The leading 'D' indicates the double-precision version of these routines). Because Givens rotations require more flops per element zeroed than do Householder rotations, and therefore take more time, the research community has devoted more effort to speeding up the Householder version. Currently the best optimized version of the LAPACK routines is DGEHRD2, which calls as a subroutine DLAHR2 that reduces a panel of the matrix in place. DLAHR2 also builds and returns (in a workspace) other matrices which can be used to update the right-hand-side; i.e. the portion of the full matrix to the right of the current panel. (In normal operation all processing to the left of the current panel is already completed.) A portion of the upper part of the panel and all of the right-hand-side updates can be completed using Level-3 BLAS routines, which enjoy much greater reuse and thus performance. In our comparisons, DGEHRD2 uses the most recent ATLAS optimized and parallelized Level-3 BLAS routines. The “cleanup” routine for this process, used when there are insufficient columns remaining to gain any advantage by using DLAHR2, is DGEHD2: This routine uses all Level-2 BLAS routines to reduce a matrix in place, and does not return any right-hand update matrices. DGEHD2 can actually be used for an entire matrix of any size. We use it that way as a point of comparison for reasons we will explain further below.

For matrices of appropriate size, PCA achieves super-linear speedup on the Hessenberg reduction. In Figure 5.16 we see these can be fairly large problems; the peak rates occur around \( N = 840 \) on the Opt8 and \( N = 920 \) on the Core2. In these charts DGEHRD2 is the fair comparison point; it represents the speed of the best LAPACK routine for the job, using parallelized Level-3 BLAS. The routine DGEHD2 only uses Level-2 BLAS routines, and in our test systems these are not parallelized. Thus DGEHD2 is our serial version against which we can measure parallel speedup, and that is done in Figure 5.17.

In Figure 5.17, we see that PCA achieves superlinear speedup for 8 cores for problems sizes between \( N = 520 \) and \( N = 1000 \) on the Opt8, and for \( N = 840 \) to \( N = 1240 \) on the Core2 (the comparison is PCA v. HD2). For the improvement over the best LAPACK routine, the comparions
Figure 5.16: Raw MFLOP Performance Comparisons

(a) O8 Hessenberg Routine Performances

(b) C2 Hessenberg Routine Performances

Figure 5.16: Raw MFLOP Performance Comparisons
Figure 5.17: Hessenberg Speed Ratios

(a) Opt8 Hessenberg Performance Ratios

(b) Core2 Hessenberg Performance Ratios

Figure 5.17: Hessenberg Speed Ratios
is PCA v. HR2. On the Opt8 PCA is about $3 \times$ faster for problem sizes between $N = 200$ and $N = 520$, declining slowly after that until $N = 1080$, the last point at which PCA is superior to LAPACK’s DGEHRD2. This is simply because we are well out of the cache for problems bigger than $N = 1080$. On the Core2, performance improves until $N = 1080$, reaching about $3.5 \times$ faster than DGEHRD2, and declines rapidly as the problem size grows beyond the collective cache. At $N = 1480$ we reach the last size for which PCA is the faster algorithm.

A key difference between the Hessenberg and QR (or LU) factorizations lies in the fact that a full matrix is being processed. All three of these algorithms follow the diagonal (or sub-diagonal) of their processing area, but in QR and LU the panel width is typically much less than the panel height, and in fact in our implementations of QR and LU, the entire diagonal is owned by a single core. Therefore in QR and LU all cores are participating throughout the factorization. In the Hessenberg full matrix factorization, this wouldn’t be as true if the data distribution was just to divide the matrix into $p$ horizontal blocks. To maximize our parallelism and keep all cores engaged throughout the factorization, we used a cyclic distribution of rows instead\(^{35}\).

For the Hessenberg, this distribution ensures the maximum number of cores that can be working will be working throughout the factorization. We believe it also shows how the PCA algorithm can be adapted to problem specifics. We aren’t changing the fundamental math of the factorization, but we can increase parallelism by remembering that we want to maintain an equal distribution of work throughout a process, not just an equal distribution of cached data.

Even though larger problems do progress to a size where the trailing matrix would fit in the collective cache, a hybrid algorithm that switches to PCA at that point offers little improvement. The Hessenberg reduction has $\approx \frac{10}{3} N^3$ operations, so the percentage of work that can be done using PCA declines very rapidly as $N$ increases; thus the share of work we would be improving would quickly approach insignificance. Nevertheless, we believe the problem sizes PCA can address are sufficiently large for many modern problems. As hardware trends toward more cores with larger local caches, the size of problem PCA can dramatically improve will increase even further.

We also note that a tiled approach to Hessenberg such as that implemented by [50, 33] builds large Hessenberg reductions by combining smaller “kernel” Hessenberg reductions. This is a very promising approach that could be further enhanced by using PCA to greatly increase their

\(^{35}\)Thus, with 8 cores, Core 0 owned every eighth row starting with row index 0 (0, 8, 16, . . . , $\lceil \frac{rows}{8} \rceil$). In general for a $p$ core system Core $i$ (for $i = 0 \ldots (p - 1)$) owns all row indices in the matrix that are $i \mod p$. 99
kernel size. Our understanding of that approach suggests PCA can triple the speed of the kernel processing on an eight core system like ours, and that simultaneously the larger kernels would reduce the overhead of the operations required to combine kernel results and would also improve the performance of those operations.

5.11 Future work

We should be able to use the same parallel cache assignment approach for Level 3 BLAS-based operations, including the original recursive panel factorizations. This should provide us with the same scalability, but with a higher peak performance.

When the higher-level algorithm is rewritten explicitly to facilitate cache reuse as we do here, it makes sense to have an autotuning framework such as ATLAS tune the kernels being used for in-cache usage, since prior work [68, 73] has indicated that tuning for cache state can drastically change the best optimization set. The ATLAS framework has recently added support for this differential cache-based tuning, motivated primarily by the results of this research.

The ideal number of cores to use with PCA is not necessarily all of them. Even if the operands are all contained in cache, Level-2 BLAS operations still demand $O(N^2)$ reads for $O(N^2)$ operations, and this is inherently inferior to Level-3 BLAS calls. As $p$ continues to increase and the size of the collective cache increases with it, the ideal number of cores to devote to a panel operation may become less than $p$: The ideal panel width is just enough to let the Level-3 BLAS operations approach their peak performance. If that width can be obtained with fewer than $p$ cores, the remaining cores might be devoted to other tasks.

For example, with sufficient cores this approach should fit nicely with current DAG-based approaches [6, 5, 59, 50, 51, 74]. The same idea could be adapted for use in GPU-based\textsuperscript{36} factorizations: Graduate student Kyung Min Su implemented a proof-of-concept for QR Panel factorization on the GPU\textsuperscript{37}, also using the author’s code as a template and under the author’s technical supervision. The prototype was for a single panel size, $5760 \times 64$ single-precision elements. The result was over $9 \times$ faster than the existing LAPACK implementation, and twice as fast as our best CPU-based PCA implementation. This is the fastest panel factorization (for this problem

\textsuperscript{36}Graphics Processing Unit.

\textsuperscript{37}An Nvidia Tesla C1060 GPU, equivalent to a system with 30 single-precision vector processors.
size) we are aware of for a commodity desktop machine, and our future work includes generalizing this prototype.

5.12 Summary and Conclusions

We have presented a new approach to parallelizing panel operations in LAPACK. We believe this approach is truly general, and can be applied to pretty much any LAPACK-like panel operation; we have shown impressive results for the factorizations LU and Hessenberg in double precision, and the factorizations QR, QL, RQ and LQ in four precisions (single, double, single complex, and double complex). These are the most widely-used factorizations in dense linear algebra and critical components in the most widely used applications of linear algebra, solving simultaneous equations and discovering eigenvalues and eigenvectors. For these operations, we achieved superlinear speedups over the original LAPACK serial algorithm (almost 20-fold asymptotic speedup for the best case), and impressive speedups over the prior state-of-the-art (asymptotically 2–2.5 times faster).

We have shown that speeding up the panel factorization provides impressive speedups (as high as 44% faster) for the full QR-type factorizations on today’s commodity desktop machines; and we have shown that this improvement should grow with core count. For the Hessenberg full problem in double precision we showed peaks of over $3\times$ faster than the best LAPACK alternative on two test machines. We have demonstrated excellent scaling for this algorithm and that it is the most scalable panel factorization of which we are aware. This addresses a serious concern in HPC today, and a critical one for tomorrow. We conclude that the presence of a fast hardware-based parallel synchronization between processing units requires a re-analysis of the conventional message-avoiding parallel design paradigm. In particular fast hardware-based parallel synchronization makes it cost-effective to parallelize very highly data dependent algorithms that have heretofore proven resistant to parallelization.
CHAPTER 6: CONCLUSIONS

The author has made three significant original contributions to the field of computer science.

As a theorist he generalized and unified the error analysis of blocked dot products, and the generalization produced a new variety of algorithms for the dot product that permits a new and superior level of control over dot product error without any additional computations. This was superblocking.

As a systems researcher, he isolated a previously unidentified but prevalent and significant performance impediment on modern commodity multicore platforms. He then invented a corrective for it; called Master Last. Master Last boosted the speed of some widely used linear algebra routines, LU and QR, 50% and 65% (respectively) in the midrange and over 25% and 45% (respectively) for the largest problems tested.

As a synthesist, the author repurposed tools used in the embedded systems arena to enable a novel parallelization solution to a growing problem in a heavily researched field. The result was PCA, Parallel Cache Assignment, which is both novel and delivers speedups of 35% on QR (full midrange problems), and over 300% on some midrange Hessenberg factorizations, one of the most important and difficult to parallelize factorizations in the linear algebra toolbox.

All of these contributions have been peer-reviewed and published, implemented, and are attracting the attention of both practitioners and fellow researchers.
BIBLIOGRAPHY


VITA

Anthony M. Castaldo graduated from high school in 1975. He served as an airman in the US Air Force from 1975 to 1977, and upon completion of his military service earned an Associate’s Degree in Computer Programming in 1979. He returned to college in 2002, entering as a sophomore. He completed his bachelor’s degree in Mathematics at UTSA in 18 months, then a Master’s degree in Mathematics at Texas A&M 15 months after that. He completed a second Master’s degree in Computer Science at UTSA in August of 2007. He has been happily married to Mary Ellen Royce for 21 years.