NTNU
Norwegian University of
Science and Technology

Techniques and Tools for Optimizing Codes on Modern Architectures:
A Low-Level Approach

Rune Erlend Jensen

Master of Science in Informatics
Submission date: May 2009
Supervisor: Anne Cathrine Elster, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

NTNU
Norwegian University of
Science and Technology

Techniques and Tools for Optimizing Codes on Modern Architectures:

A Low-Level Approach

Rune Erlend Jensen

Master of Science in Informatics
Submission Date: May 2009
Supervisor: Anne C. Elster

Norwegian University of Science and Technology
Department of Computer and Information Science
Section of Complex Computing Systems

ABSTRACT

This thesis describes novel techniques and test implementations for optimizing numerically intensive codes. Our main focus is on how given algorithms can be adapted to run efficiently on modern microprocessor exploring several architectural features including, instruction selection, and access patterns related to having several levels of cache.

Our approach is also shown to be relevant for multicore architectures. Our primary target applications are linear algebra routines in the form of matrix multiply with dense matrices. We analyze how current compilers, microprocessor and common optimization techniques (like loop tiling and date relocation) interact. A tunable assembly code generator is developed, built, and tested on a basic BLAS level-3 routine to side-step some of the performance issues of modern compilers.

Our generator has been test on both the Intel Pentium 4 and Intel's Core 2 processors. For the Pentium 4, a 10.8 % speed-up is achieved over ATLAS's rank2k, and a 17% speed-up is achieved over MKL's implementation for 4000-by-4032 matrices.

On the Core 2 we optimize our code for 2000-by-2000 matrices and achieved a 24% and 5% speed-up over ATLAS and MKL, respectively with our multi-threaded implementation. Also for other matrix sizes, descent speed-ups are shown. Considering that our implementation is far from fully tuned, we consider these result very respectable.

Acknowledgements

Numerous people have helped me with interesting and revealing problems, and my motivation during the course of this work. First, I would like to thank my supervisor Dr. Anne C. Elster, for her support and guidance, Jan Christian Meyer, for many very interesting discussions and good guidance on several theoretical issues, and Thorvald Natvig, for the framework used and support in many issues.

I also want to thank the HPC-lab, for hardware support and infrastructure, and my co-students there: Jérôme Dubois, Andreas Bach, Atle Rudshaug, Robin Eidissen, Eirik Ola Aksnes, Daniele Giuseppe Spampinato, Åsmund Herikstad, Rune Johan Hovland, and many others for their help.

Thanks to Sindre Berg Stene, for helping to clear my mind after long days, and providing insights on how to write my thesis, and finally, my family, especially my sister Tone Elisabeth Jensen, for reading and commenting on my thesis.

Contents

Abstract **iii**

Acknowledgements **v**

1 Introduction **1**
1.1 Goal . . . 3
1.2 Outline . . . 4

2 Background and Previous Work **7**
2.1 BLAS . . . 9
2.1.1 ATLAS . . . 10
2.1.2 MKL . . . 10
2.1.3 Goto BLAS . . . 11
2.2 Issues Not Considered . . . 11
2.2.1 Hardware Test Beds . . . 12

3 Technical Background & Concepts **15**
3.1 Operating systems - Virtual Memory and Paging . . . 15
3.2 Modern Architectures . . . 16
3.2.1 Cache System . . . 16
3.3 The Intel® x86 Processor Family . . . 21
3.4 Intel® Pentium 4 Architecture . . . 25
3.5 Intel Core 2 Architecture . . . 26
3.6 Current Optimization Techniques . . . 28
3.6.1 Loop Tiling . . . 28
3.7 Tools . . . 35
3.7.1 Compilers . . . 35
3.7.2 Valgrind . . . 36
3.7.3 Objdump . . . 36
3.8 Topics Not Covered . . . 36

4 Optimizing codes on the Pentium 4 **39**
4.1 Theoretical Issues . . . 39
4.1.1 Symmetrical Considerations . . . 40
4.1.2 Loop Tiling and Symmetry . . . 42
4.1.3 Optimal Loop Tiling and Prefetching . . . 44
4.1.4 Write Pattern . . . 46
4.1.5 Data relocation and Prefetching . . . 48
4.1.6 Calculation Order . . . 50
4.2 Implementation Issues . . . 55
4.3 Pentium 4 Issues . . . 55
4.3.1 Main Kernel . . . 56
4.3.2 Data Layout . . . 56
4.3.3 The Math Kernel . . . 60
4.3.4 Improved Calculation Order . . . 64
4.3.5 Calculation Order Design . . . 65
4.3.6 Calculation Order Generation . . . 69
4.4 Compiler Test Case . . . 70
4.5 Method . . . 71
4.5.1 Setup . . . 73
4.6 Results . . . 74
4.6.1 Compiler Test Case . . . 78
4.7 Evaluation of Pentium 4 Optimizations . . . 80
4.8 Modeling Using Virtual Multi-Threading . . . 80

5 Low-Level Optimization on the Intel® Core 2 **83**
5.1 Theoretical issues . . . 83
5.1.1 Low-Level Instruction Selection . . . 84
5.1.2 Cache Evaluation . . . 90
5.1.3 L1 Cache and Prefetching . . . 93
5.2 Implementation Details . . . 94
5.2.1 Extending to Full Rank2k (dsyr2k) . . . 94
5.3 Tools and Their Motivation . . . 96
5.3.1 Core Tester — Benchmarker . . . 96
5.4 Prefetching Access Pattern . . . 96
5.5 Assembly Code Design: Core Code Generator . . . 97
5.5.1 Core Code Generator . . . 98
5.5.2 Core Code Generator Tuning . . . 99
5.5.3 Using the Core Code Generator . . . 99
5.5.4 CCG Inner Working . . . 100
5.5.5 Linear Layout of AB . . . 100
5.5.6 Writing of C . . . 101

5.5.7 Prefetching . . . 101
5.6 Parallelization . . . 102
5.6.1 OpenMP . . . 102
5.7 Method . . . 103
5.7.1 Setup . . . 105
5.8 Results . . . 107
5.8.1 Tuning Test: nop . . . 114
5.9 Evaluation of Core 2 Results . . . 114
5.9.1 Code Issues from the Pentium 4 implementation . . . 115

6 Conclusions, Current & Future Work 117
6.1 Current and Future Work . . . 118
6.1.1 Future Scalability . . . 119
6.1.2 SSE5 . . . 119
6.1.3 AVX - Advanced Vector Extensions . . . 119
6.2 Remaining Issues . . . 120

Bibliography 124

Appendix List 125

A GCC In-line Assembly Code Sample 127

B Cache Simulator Description 133

C GCC In-line Reformatter 135

D Notur 08 Poster 137

List of Figures

3.1 4 way cache with N sets. 17
3.2 4 way cache with 8 sets. 18
3.3 Sample instruction formats. 24
3.4 A multiplication instruction, size 4 bytes. 24
3.5 Multiplication and memory load, size 5 bytes. 24
3.6 Multiplication and complex memory load, size 10 bytes. 24
3.7 Block division and cache illustrated. 30
3.8 Block division and cache illustrated. 31
3.9 Step 16, with an added illustration of another blocking layer. . 32
3.10 Step 17. Shows the completed part of the calculation, and how the algorithm continues. 32
3.11 Various block shapes and how cache efficient they might seem to be. 33
3.12 Illustration of data relocation, where selected parts of A and B' are moved into two different data arrays. 34

4.1 The two calculation parts merged at the right with the AB' part faded. The symmetry is also shown. 41
4.2 The symmetry illustrated. 41
4.3 Blocking and symmetry combined. 43
4.4 Combining blocking and streaming for better cache usage. . . 45
4.5 Illustration of a worm like access pattern. 46
4.6 Linear writing of C 47
4.7 The hardware prefetcher problems. 49
4.8 Hardware prefetcher with data relocation. 49
4.9 Out of Order Execution. The processor state after 4 clock cycles. 50
4.10 The processor state after 9 clock cycles. 51
4.11 The processor state after 12 clock cycles. 51
4.12 Real processor state after 12 clock cycles. 52
4.13 Data misses that do not carry any side effects. 53
4.14 The issuing of all loads that generate cache misses first. 53

4.15 The skipped work performed, after all the L1 cache misses are done. 54
4.16 The partially improved data relocation. 57
4.17 The improved data relocation pattern. 58
4.18 Blocking, symmetry, improved data relocation and improved data layout combined. 59
4.19 Interleaved cache misses and hits. 64
4.20 Balanced mix of hits and misses. 65
4.21 Median performance with N=2000 and K=2016, on Clustis2. . 75
4.22 Median performance with N=4000 and K=4032, on Clustis2. . 76
4.23 Performance Distribution with Random patterns. 77
4.24 Flow Graph, writing 1024 floats. 79

5.1 Final Core 2 instruction assembly code block. 91
5.2 The final L2 loop tilling access pattern. 98
5.3 Median single-thread performance on Clustis3. 108
5.4 Median multithread performance on Clustis3. 110
5.5 MKL multithread variance on Clustis3. 113

List of Tables

2.1 Clustis2 processor details. 12
2.2 Clustis3 processor details. 13

3.1 Pentium 4 (model 2) instruction details. 26
3.2 Enhanced Core 2 instruction details. 27

4.1 Statistics with N=2000 and K=2016, on Clustis2. 75
4.2 Statistics with N=4000 and K=4032, on Clustis2. 76
4.3 Random runs with N=4000 and K=4032, on Clustis2. 77

5.1 Pentium 4 instruction layout. 85
5.2 Extended Pentium 4 instruction layout. 87
5.3 Minimal register level loop tiling. 88
5.4 Final Core 2 instruction layout. 89
5.5 4 J columns and 12 I columns without overlap. 92
5.6 4 J columns and 12 I columns with overlap. 93
5.7 Single-thread statistics with N=1000 and K=1000, on Clustis3. 109
5.8 Single-thread statistics with N=2000 and K=2000, on Clustis3. 109
5.9 Single-thread statistics with N=4000 and K=4000, on Clustis3. 109
5.10 Single-thread statistics with N=8000 and K=8000, on Clustis3. 110
5.11 Multithread statistics with N=1000 and K=1000, on Clustis3. 111
5.12 Multithread statistics with N=2000 and K=2000, on Clustis3. 112
5.13 Multithread statistics with N=4000 and K=4000, on Clustis3. 112
5.14 Multithread statistics with N=8000 and K=8000, on Clustis3. 112
5.15 *nop* instruction test with the core tester. 114

List of Programs

1 Basic code for the matrix multiplication $Z = Z + XY$. 28
2 Loop tiled (blocked) code for the matrix multiplication $Z = Z + XY$. 28
3 Illustration code for the optimal base case function. 60
4 Illustration code for the main section. 61
5 The math kernel code. 63
6 Excerpt of the Pentium 4 innermost loop. 67
7 The linear access pattern, enumerated in C. 68
8 Generation of C enumerations. 68
9 Generation of a miss/hit sequence. 69
10 A simple C program, writing 1024 floats. 70
11 A simple implementation of dsyr2k. 71

Chapter 1

Introduction

A large part of the workload on present day computers involves solving various linear algebra problems, both within HPC and consumer class products.

> "... In modern treatments of linear algebra, matrices are considered first. Matrices provide a theoretically and practically useful way of approaching many types of problems including: Solution of Systems of Linear Equations, Equilibrium of Rigid Bodies (in physics), Graph Theory, Theory of Games, The Leontief Model in Economics, Forest Management, Computer Graphics, and Computer assisted Tomography, Genetics, Cryptography, Electrical Networks, and Fractals[1]" — Joseph Khoury, University of Ottawa.

Optimizing this subset of problems will not only give faster solutions, but might enable solving larger problems or saving money on power usage.

Modern CPU's are so complex that a simple model is lacking. The documentation from the manufacturer is often incomplete, and does not deal or explain precisely how various subsystems interact. This makes a complete or accurate model hard or practically impossible to design. Also, since different computer systems have many different sub parts, like those associated with memory speed and technology, the motherboard and chipset used, processor type and features includes, model, stepping, frequency and

[1] `http://aix1.uottawa.ca/~jkhoury/matrices.html`

cache sizes. Finally, the operating system can have a major impact on performance. All of these parts makes the design of optimal code very hard.

For compilers, the task of instruction generation becomes even harder. They currently lack any information on how to generate code with respect to the sub parts in the target computer. In order to ensure compatibility, they can not freely select instructions or optimizations, unless target architecture is specified. As compilers often are updated after the CPU architecture arrives at the market, the most modern CPU may not even have compilers optimized for their features.

In order to avoid these problems, an idea on how to make a simple code generator that will be capable of coming around some of these problems, was formulated. By making a low level code generator system, that use empirically gathered information from dynamically designed benchmarks, allows the code generator to build/evolve code that is near optimal.

Because of their importance, most of the underlying functionality have been standardised into a set of key libraries. This have been done to ensure that stable and error free implementations of these important functions are easily accessible. One of these libraries is the *Basic Linear Algebra Subprograms* (BLAS) library. The BLAS library performs fundamental operations such as multiplication on vectors and matrices, and therefore implements the basis for more advanced linear algebra tools such as MATLAB.

Note that the BLAS library standards main goal is to describe a platform independent interface for these routines. It is left up to library implementations, including computer vendors, to implement optimal versions of these routines.

Much work has been done to improve performance both from a low level computer aspect and from a high level algorithmic position. Current implementations are often carefully tuned by the microprocessor vendors, in order to extract every drop of performance.

How this work started: My Personal Motivation

Because these key libraries are highly optimized, they are good candidates for student competitions when they learn about how to optimize code for

performance. This task is hard since there are many aspects that must be successfully addressed. This was also the outline for an exercise in parallel programing course taught by Dr. Elster that the author took in 2007. The challenge was to achieve at least a certain fraction of the speed of a good implementation (e.g. ATLAS [20]), on a single given BLAS function.

I have always considered myself somewhat capable of performing optimization, however, this turned out to be a challenge like never before. All my attempts seemed to only marginally improve speed, and it was less than half of what was obviously possible, so continually increasing systematic analysis was done and tests performed. This culminated in a version of rank2k that managed to beat the best implementation available by several percent, most of these percents was gained on the last day before delivery. The competition was won, without reading any papers or research on what others had done before. Only the processor documentation and analysis on how the given processor worked was used.

Beating ATLAS was considered to be important, and Dr. Anne C. Elster (later my supervisor) suggested a master thesis was appropriate for analysing and explaining how this performance was achieved, and for further research.

1.1 Goal

As mentioned this thesis will look at techniques for optimizing memory and numerically intensive code. The main focus is not on designing optimal algorithms, but on how algorithms can be adapted to run efficiently on modern microprocessor. The thesis will primarily focus on linear algebra algorithms, in the form of matrix multiply with dense matrices. It also looks at how current compilers, microprocessor and common optimization techniques (like loop tiling) interact with each other.

Personal Goal

Some of the issues I had notice early on, was that utilising only official guidelines and known techniques did not give an implementation that approached the performance of ATLAS archives. While writing the original

implementation, several of the improvements was found only by going against the recommendations in the processor documentation. Moreover, several parts of that documentation was contradicting itself. I therefore felt that if decades of research could not beat a month's work, by someone that never even heard about BLAS before, then it was less important to learn precisely how they implemented their code. Even the very detailed processor documentation was inconsistent with the real-world processor behaviour I was seeing. When working on speeding up my code, novel techniques were found that seemed contrary to the processor documentations. This thesis will attempt to document some of these findings.

I chose to deliberately avoid looking at the implementation details of what other BLAS library developers had specifically done, to avoid falling into their pattern of thought — at least until I had a good understanding of what made my implementation fast. In the end, this turned out to be both a blessing and a curse, as expected but not quite in the way I envisioned.

1.2 Outline

The rest of this thesis is organized as follows:

Chapter 2 includes some more background details, as well as related work on this topic. The competing implementations are also introduced.

Chapter 3 describes the key features impacting performance in modern microprocessors. Several known optimization techniques will also be presented. Some of the tools that are needed or used are also described here.

Chapter 4 consists of the first of two parts. Each part contains a model, implementation, benchmarks and a short evaluation. The first part looks on the old implementation made before the thesis, and the early work done with it during the thesis. Fundamental concepts and framework is made and tested, so that important findings can be included in the later implementation designs.

Chapter 5 contains the second part of our work, which looks at a new implementation done for a newer processor (Intel® Core 2), using what was found in the first part (Intel® Pentium 4). Additional key concepts and theories are also explored.

Chapter 6 presents an overall summary of the results we achieved. It also discusses further improvements of our implementation and looks at possible future work, with known microprocessor changes that are approaching in the next few years.

Appendix A Code Example showing the assembly code for one minimal core function.

Appendix B A description of the cache simulator developed.

Appendix C the Perl script that reformats the assembly code from the core code generator, into the GCC assembly in-line style.

Appendix D NOTUR 08 Poster, showing more issues with compilers.

In addition, a zipped file with additional code and benchmarks is available upon request.

Chapter 2

Background and Previous Work

Numerous articles concerning almost every aspect of matrix multiplication have been published, so any attempt at a complete overview is impossible within the scope of this thesis. However, we discuss a small number of selected articles we feel are central to our work and which are some of the better ones we are aware of.

Micro Processors and x86 Programing

A good basis of processor design and inner working can be found in *Computer Organization and Design, the hardware/software interface*[17], this book go into the really low level aspects and all the way up to compilers. Appendix A have an excellent overview on assembly writing and the assembler, although its for the MIPS processor.

An guide for using SIMD (MMX and SSE) in C and C++ is the *Intel® C++ Intrinsic Reference*[3], good as a start if one are new to programing with this technology.

The *Intel® 64 and IA-32 Architectures Software Developer's Manual*[5] contain documentation on instructions and general processor functionality, Volume 1 have a detailed guide on how to program x86 processors. Recommends in order to get the intermediate understanding into place.

The *Intel® 64 and IA-32 Architectures Optimization Reference Manual*[4] contains massive amounts of details and suggestions for how to achieve good performance on their processors. Because of the proprietary nature of the inner processor design much information is missing. Chapter 2.1 give nice overview of the Core 2, and 2.3 the same for Pentium 4.

Chapter 3, 4, 6 and 7 contains a huge list of optimizations and rules that have been used, recommended reading (although contradictions exists). This is advanced level, for compilers and assembly writers.

Appendix B.5 contains an explanation on performance counters while B.6 and B.7 points out how to understand the values obtained and which exists. This a good source of complex information on the inner workings of the Core 2, even while it has not been studied nearly enough it seems imperative to understand what is written between the lines. This hidden information was found too late for being useful unfortunately.

A source of what the future x86 architecture and instruction set brings is the *Intel® Advanced Vector Extensions Programming Reference*[11], good reading in order to be prepared for the change.

Memory and Locality

Some insight into DRAM and memory bottlenecks beyond simple bandwidth issues can be found in *A Case for Studying DRAM Issues at the System Level*[12].

Cache reuse by loop transformations are described by *The Cache Performance and Optimizations of Blocked Algorithms*[15], good insight to conflict misses and how to deal with them.

Virtual memory plays tricks behind the scene, as shown in *Page Placement Algorithms for Large Real-Indexed Caches*[13]. Although it deals with caches with less associativity than currently available it highlights the fundamental problem. *The Effect Of Page Allocation On Caches*[16] looks the same effect on somewhat smaller caches, useful reference for the small L2 caches in new processors.

Data layout transformations are beneficial and might (finally) also be done guided by the compiler as shown in *Refactoring for Data Locality*[1].

2.1 BLAS

The standard framework for fundamental linear algebra is *Basic Linear Algebra Subprograms* (BLAS[1]). With a stable reference implementation, it is often used as the main library for many projects. However it is not designed for being very fast or optimal. In order to come around this, a number of high speed implementations exist. Some are highly specialized (and costly), made by the maker of the CPU itself, other are general and free. There have been, and is continual research into how to achieve best performance on numerous hardware platforms. Only a few will be looked into here.

Rank2k

The first problem considered was optimization of the BLAS function *cblas_dsyr2k*, or Double precision Symmetrical Rank2k shown in Equation 2.1.

$$C = xAB' + xBA' + yC \tag{2.1}$$

The A, B, C are matrices and x and y are scalar values. The size of the matrices can be described by two parameters, N and K. With A and B of size N*K, and C being an N*N matrix. In order to make the initial exercise somewhat easier, the scalar values x and y were fixed to 1.0 and 0.0. This reduced the problem to the slightly simpler Equation 2.2 (note that ATLAS tests for this special case as well).

$$C = AB' + BA' \tag{2.2}$$

This thesis will tune on this simpler version. However, how to extend this work back to the full Rank2k Equation 2.1 will be included in the discussion.

One free library implementation for doing BLAS in an efficient way is *Automatically Tuned Linear Algebra Software* (ATLAS) [20]. Intel's *Math Kernel Library* MKL [6] math library is an commercial implementation that is made specially for Intel© processors. We will test our generated code against these two libraries. A discussion of Goto's [9] more hand-optimized approach is also discussed.

[1] `http://www.netlib.org/blas/`

2.1.1 ATLAS

A recent efficient implementation of the BLAS library is ATLAS [20]. Its capable of finding near optimal implementations of the BLAS functions, by testing a large number of implementations and/or condition parameters. This approach is supplemented with handwritten high performance assembly code, contributed by many people. It is used on many systems, and often beats many commercial vendor-specific implementations.

An analysis on how ATLAS works, and tests replacing the searching with an analytical model is done in *Is Search Really Necessary to Generate High-Performance BLAS?* [21]. The same authors also go into more details *An Experimental Study of Self-Optimizing Dense Linear Algebra Software* [14], where multiple low level issues are evaluated.

The project started as an small unfunded project, maintained by a few people. Later it have been funded several governmental agencies. The latest stable version[2] (3.8.3) have received major speed improvements on the Core 2 processor.

2.1.2 MKL

MKL is an proprietary math library made by Intel® containing extensive mathematical functionality, and one important part of this functionality is BLAS compatibility.

> "The Flagship for High-Performance Computing Math Software"[3]
> — Intel®

MKL is highly optimized for Intel® processors, and actively attempts to be the fastest implementation available for their own processors. It appears to be in an constant armsrace against ATLAS regarding performance, where they have alternated on having the fastest code. Its inner workings is not known.

[2] As of May 6 2009.

[3] `http://software.intel.com/en-us/intel-mkl/` contains in depth information and comparisons with ATLAS. As of April 22 2009 the ATLAS version used there is not the newest, and the newer version include heavy speedups on the platforms used in the comparisons.

2.1.3 Goto BLAS

A well-known more hand-optimzed versions of the BLAS have been developed by Goto [9] who focuses a lot on optimizing TLB (translation look-ahead buffer) misses. On the Pentium 4, we experienced that we also got more speed-up when added optimizations for TLB misses, using huge pages. Doing these optimizations, required extra operating system kernal support. Note, however, that several of the TLB issues are indirectly addresses by our data reordering schemes. Some additional TLB comments are added at the end of Chapter 3. Note also that the Intel Core 2 can handle many more TLB misses, so these issues are less relevant on this newer architecture.

2.2 Issues Not Considered

There are a couple of important issues that are considered outside the scope of this thesis. However, both these topics may imply speedups and correctness of the answer.

Algorithmic Improvements

There exist several way to calculate matrix multiplications faster than the basic triple for loop. Finding such an algorithm is therefor of great importance to performance. The Strassen matrix multiplication algorithm [2], and other algorithms with better asymptotic complexity are not considered for two reasons: The first reason is that they might give numerical stability problems (or precision loss), and handling this in a good way might require extensive mathematical knowledge. The other is that code optimization techniques rather than finding the algorithm itself, is the focus in this thesis, so selecting a simpler algorithm is advantageous.

Numerical Stability

Numerical stability is an issue related to the way processors implement floating point arithmetic. Limited precision imposed by what can be

encoded in the available data size is normally the only point taken into consideration, however precision loss can be amplified based on the sequence math operations are ordered. This is of great concern with algorithms like Strassen's, while for the algorithm implemented here there are no such issues. An evaluation of precision loss for both algorithms, and other similar ones can be found in [10]. While it is possible to enhance the precision by careful analysis, it's not taken into consideration in this thesis.

2.2.1 Hardware Test Beds

Custom hardware that will solve this kind of problems in a known optimal way (from a software/algorithmic perspective) often cost too much to be justified. The trend of using clusters of low cost 'home' computers, typically with an x86 processor, have been going on for several years. The older Intel Pentium 4® and the new Intel Core 2® series were chosen as our main testbeds due to both availability and popularity (and thus importance).

Clustis2

Clustis2 was the old Pentium 4 cluster at Department of Computer and Information Science at NTNU. The Clustis2 nodes used for testing consists of a single Xeon®Pentium 4® processor, with hyper threading disabled. Its processor details is shown in Table 2.1.

Table 2.1: Clustis2 processor details.

Cpu family	15
Model	2
Model name	Intel® Pentium® 4 CPU 3.40GHz
Stepping	9
L1 cache size	8 KB
L2 cache size	512 KB

Clustis3

Clustis3 is the new Core 2 based cluster at Department of Computer and Information Science at NTNU. It is used primarily for teaching parallel

computing, and was installed in January 2009. The Clustis3 nodes used for testing consists of two quad core Intel® Xeon® E5405 processors running at 2 GHz. This allows scalability testing up to 8 threads. Each node have 9 GB of main memory, physically configured in a 1 GB + 8 GB layout. Its processor details is shown in Table 2.2.

Table 2.2: Clustis3 processor details.

Cpu family	6
Model	23
Model name	Intel® Xeon® CPU, E5405 @ 2.00GHz
Stepping	10
L1 cache size	32 KB
L2 cache size	6144 KB

Chapter 3

Technical Background & Concepts

In order to highlight the details of the implementations later on, a number of key concepts are presented in this chapter. First, a number of well known features related to how new processors works will be presented. This is important to include since some of these parts contain details that are often overlooked, while potentially representing major performance issues. Details of the processor family used in the implementation and the specific processors used will also be presented, as several of their particular abilities are relatively unknown. A number of known programing techniques for optimizing performance are then presented. These are included as they are often used to address various processor features, and are necessary for the best performance. Finally, a few essential tools will be mentioned for completeness.

3.1 Operating systems - Virtual Memory and Paging

Virtual Memory is a central aspect of operating systems. Most of the features associated with this topic is beyond the scope of this thesis, one feature (or lack of) must be pointed out however. In order for cache the techniques used the physical memory layout must be similar to the virtual layout,

namely linear arrays must be linear physically in the memory for the cache system. If this is not the case one might get conflict misses in the cache, on seemingly linear data. When a program asks for memory the operating system *opportunistically* allocate only the pages written too. This can cause *fragmentation* of physical memory, if its not already fragmented. Kessler [13] shows that this effect can give up to 30% unnecessary cache misses in large caches with low associativity.

In order to avoid unexpected conflict misses and maintain cache effectiveness some (few) operating systems use a technique called *Page Coloring*[1]. Page Coloring keeps track of physical layout, and selects pages such that conflict misses are avoided. The operating systems used for testing do not have this feature. This might lead to lower performance and potentially random speed variations between otherwise equal runs. Finding out if the physical layout is bad or good is impossible without low level operating system support, none of the systems used for testing seem to have this support either.

3.2 Modern Architectures

The modern era of microprocessor design, with nearly a billion transistors, have lead to many complex processor designs. At the same time, several of the most used processor families are based on very old designs. This has lead to many solutions that can be considered to be suboptimal, and this is very true for the *x86 processor family* that is used in this thesis. Although many of these solutions only play a minor practical role, some of the most relevant issues will be mentioned. First, some of the key concepts of modern processor architectures will be briefly explained, with less focus on well known concepts and more on some less known issues. Among the concepts explained are the Cache and some of its features, pipelining and *Out of Order Execution* (OoOE). Then a number of the features of the used microprocessors are presented.

3.2.1 Cache System

Caches are designed to reduce the impact of today's main memory issues of long latency and limited bandwidth. The two fundamental ideas used in

[1] `http://www.freebsd.org/doc/en/articles/vm-design/page-coloring-optimizations.html` verified on April 27. 2009.

caches is that most memory accesses has close *temporal* and *spatial* locality. An memory location used once is often used again after a relatively short time, so caches exploit this temporal pattern. Also, if one use a memory location, then it is likely that the nearby memory locations will be used as well. This spatial pattern is also captured by the cache.

Today's caches are organized into several layers. The first are the two *Level 1* caches, one for instructions and one for data. They are designed for high speed, minimizing latency and having very good bandwidth. This design limits their size to somewhere in the low KB range, normally around 8–64 KB.

In order to handle larger data sets, an *Level 2* cache is used. The L2 cache is typically designed for being large, preventing as many memory accesses as possible from hitting the slow main memory. This gives a cache optimized primarily for large size first and only somewhat for bandwidth and latency, with sizes currently around 1–6 MB. Finally, *Level 3* caches are starting to become normal[2], with sizes in the multi-megabyte range. It normally takes the place of L2 cache as a large but slow cache, shared between several processor cores. In this case, the L2 cache will be optimized more towards a balance of size, latency and bandwidth, often being dedicated to a single processor core.

Cache Organization

Internally caches are organized into *ways* and *sets*. In Figure 3.1 a cache with 4 ways and N sets are shown. Each set have 4 *ways* where data can be stored, and each *way* can hold a small amount of data called a *cache line*. The cache line is the minimum block size that the cache operates on, irrespective of the actual data size used. Moreover, for any given memory location data can only be cached in a *single* set.

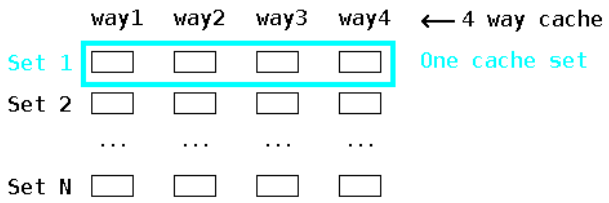


Figure 3.1: 4 way cache with N sets.

[2]For the more common processor versions in the x86 family.

For a simple cache with 4 ways, 8 sets and a cache line of 1 element this can be illustrated as in Figure 3.2. Every 8'th memory addresses can only be stored into a single set. If data is read sequentially with a stride of 8, this will fill one set while the others are unused. An cache miss generated in this fashion is called an *conflict misses*. They are a consequence of the *set-associative* design, where data competes for the space inside a given set. These stand in sharp contrast to the normal *capacity misses*, that can be expected to occur when the cache have been overfilled with data. More details can be found in [17, Chapter 7], if desired.

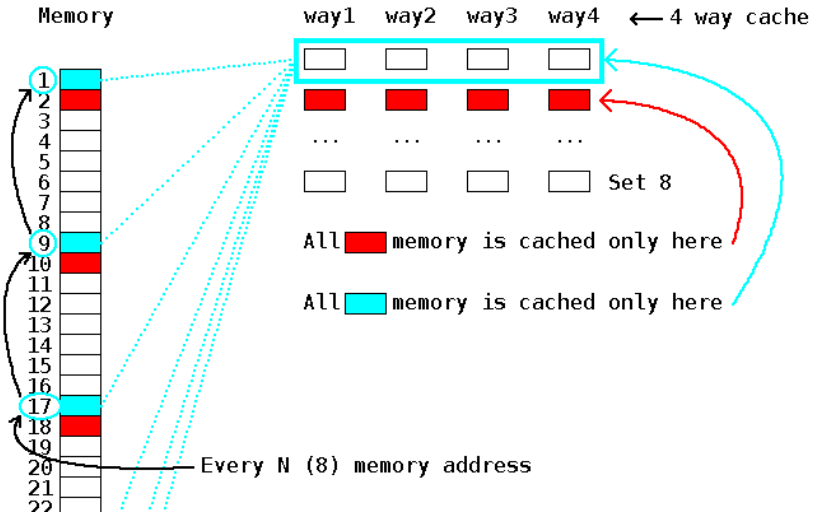


Figure 3.2: 4 way cache with 8 sets.

For a *physically addressed cache* the memory location determining which set to use is based on the physical memory location of the data. This is normally the method used in L2 and L3 caches. For a *virtually addressed cache*, one uses the memory location after the address have been translated into the virtual address space of the running application. This is normal in L1 caches, avoiding the translation cost.

Cache Replacement Policies

When a cache line is replaced by a new, one of the cache lines in the set must be replaced. In order to select which one to remove one useful Cache

replacement policy is *Least recently used* (LRU). It replaces the line that have remained unused for the longest time frame. An simpler scheme is just to select a *random* one. Because of high hardware costs, even LRU might be both too costly and too slow.

Nonblocking Cache

When a memory request results in a cache miss this will halt all further memory requests, unless the cache supports *hit under miss* or similar techniques. An cache with this support can continue to accept memory requests, while the first miss is being loaded. In order to achieve this, the cache system has a number of *Miss Status Holding Registers* (MSHR) that handles outstanding misses. Typically, one MSHR is needed for every miss that needs to be handled at the same time. This technique is often used to create more memory level parallelism, that is needed by out of order processors.

Pipelining and Throughput

Pipelining is a well known technique [17, Chapter 6] where the processor divide the execution of every instruction into several smaller stages. This means that there is a possibly long *latency* from the instruction starts until it is completed. When many instructions that depend on each other are stacked up in a long sequence, the execution time will be limited by latency, also called pipeline *stalls*.

The other effect of pipelining is that many instructions can be executed at the same time, each in different pipe stages. This gives a high *throughput*, even while each take a long time to complete. For sequence of instructions that do not closely depend on each other the execution time will depend on throughput. Most newer processors are also *super-scalar*, a technique where several instructions can be started at the same time.

Out of Order Execution

Large and more complex processors use *Out-of-order Execution* (OoOE) in order to reduce the effects of long latency. When an instruction that either depend on data from instructions that are not available (or on memory not in L1 cache), it is put into a queue waiting for the data to become available. The processor then continues with the next instructions, queuing instructions until it finds an instruction where all the needed data is available. Instructions will therefor be executed in a different sequence than intended, using possibly incorrect data or overwriting later needed data. In order to do this the processor use *rename registers* to create several versions of the same register, each containing data from different time frames. Several other techniques are also used to avoid these data hazards (false dependencies), and keeping the impression that all instructions are executed in a sequential manner. Doing this improves effective throughput, at the cost of greatly increased design complexity. Out-of-order execution is normally combined with nonblocking caches.

Branch Prediction

Branch prediction is used in modern processors in order to try to guess which way an branch will take. In order to mask the long latencies associated with pipelining, which instructions to execute after a branch are guessed. For numerically intensive HPC code, this is of less importance. However, it is extremely important for most other code. A related feature that is more useful here is *loop prediction*, where the processor try to guess the number of times a loop will be iterated. This avoids the same latencies associated with branches.

System Memory: DRAM

Dynamic Random Access Memory (DRAM) issues can also contribute to a significant performance hit, if it is not taken into account. While not a processor specific feature, it is still an essential aspect that is closely related to the processor design, its inner working, and efficiency. By just changing the internal bank, burst size and channel layout, without changing theoretical bandwidth, it has been shown (by [12]) to give a *factor 2* speed difference on

certain SPEC 2000 benchmarks. While the issues relating to this huge speed difference are relevant, only the two most important aspects are pointed out here:

Essentially the effective DRAM bandwidth varies based on *access pattern*. The first read costs more than followup reads within the same region[17, p. 490] (size varies from manufactures and memory modules). Second, changing between reading and writing costs extra time. Other costs and effects are beyond the scope if this thesis for two reasons: First, DRAM configuration information is currently not available in the same manner as corresponding information for CPU's. Second, taking more of the DRAM details into account when constructing algorithms is somewhat problematic.

3.3 The Intel® x86 Processor Family

The Intel® x86 processor family is defined by its *instruction set*. Its legacy from the late 1970's heavily affect most aspects of the modern implementations of family. Every generation typically adds some new features to the growing feature set, and only the most relevant issues and features will pointed out in some detail. Some general overview is presented first. Then some details of the instruction set are explained, as this plays a large role later on. Finally the features of the processors uses for testing is explained.

List of Extensions in x86

The list of extensions to the x86 family is long and uneven. Some of the enhancements have been successful, some just unsuccessful and other best forgotten. A few are even in conflict, and incompatible with each other. A short and simple list of the names of well known extensions is as follows: MMX, MMX2, SSE, SSE2, SSE3, SSSE3, SSE4, SSE4a, SSE4.1, SSE4.2, 3DNow, Enhanced 3dNow.

Instruction Decoding

The x86 family is a *complex instruction set computer* (CISC) from an instruction-set point of view, while recent hardware is implemented as a *reduced instruction set computer* (RISC) processor. This affects how instructions are handled, because they are first decoded, then they are broken up into *micro-instructions*. These micro-instructions are then executed by the RISC processor core, with the OoOE engine reordering these partial instructions.

Instruction Set

The x86 family is an old design, keeping most compatibility with the first versions dating back to Intel® 8086 introduced in 1978. Most instructions have only *one or two operands*, some even with fixed destination and/or source. The answer of most operations must therefore be stored in one of the same operands as the input operand, overwriting it. In addition, to reduce memory requirement, each instruction have *variable size* based on operation and operand type (ranging from 1-15 bytes). Since the complexity of the instruction set is quite high, only the relevant parts to our optimization work is described.

SSE

One of the main extensions to the instruction set is the *Streaming SIMD Extensions* (SSE/SSE2). This brings in a *Single Instruction, Multiple Data* (SIMD) class of instructions, with their own set of registers. These vector like instructions work on several data elements at the same time, for floating point this is either 4 32-byte values or 2 64-bit values. The old operand format, where one of the source operands are overwritten is kept.

x86-64

All new x86 CPU's use an extension called *x86-64*, bringing a number of enhancements[3]. The main (relevant) changes are: General register size have been extended to 64-bit. Support for 64-bit memory pointers. 8 extra general registers and 8 extra SSE registers, giving a total of 16 of each type. No MMX or FPU registers where added however. The other enhancements are not described here.

Instruction Properties

The three key instructions used in the math kernel are *addition* (addpd), *multiplication* (mulpd) and *load* (movapd). Since the size and address encoding of instructions are important to good performance, some detailed information is needed. The instruction size calculation and some encoding's details are as follows, again only the relevant parts will be explained.

The instruction size is calculated as follows: The first part is an optional 1 byte *prefix* needed if the instruction is an SSE type. Second is the *opcode* itself (operation type: add, mult, load, ...), the size is typically $1-3$ bytes depending on the operation. Third is an optional 1 byte of *register operand info* that encodes which registers are used in the instruction. An optional *'REX'* byte is added if one or more of the operand register(s) belong to the additional ones added in x86-64.

If a memory *load/write* is included then *no* extra bytes are needed, unless a constant offset is used. If so, an additional 1 byte is needed if the *offset* ($constOffset$) is in the range of -128 to 127. For an offset ($constOffset$) *outside the range* -128 to 127 an total of 4 bytes are needed for the offset. If the memory operation use an additional *register offset* ($offsetReg$) an extra byte is also added.

This gives a (non-complete) size calculation of $prefix + opcode + operand + REX + constOffset + offsetReg$ where the parts not used contribute 0 bytes. Some of the instructions can therefore be only 1 byte long, like simple stack manipulation, simple flow control and nop (no operation). An quick overview and some examples are included for convenience.

[3] An overview can be found at `http://en.wikipedia.org/wiki/X86-64`, verified 12. May 2009

```
opcode
opcode sourceReg, destinationReg
opcode (pointerBaseReg), destinationReg
opcode constOffset(pointerBaseReg), destinationReg
opcode (pointerBaseReg, offsetReg, multiplyer), destinationReg
opcode constOffset(pointerBaseReg, offsetReg, multiplyer), destinationReg
opcode sourceReg, constOffset(pointerBaseReg, offsetReg, multiplyer)
```

Figure 3.3: Sample instruction formats.

A set of examples of the instruction format can be found in Figure 3.3, where the basic layout is shown. The ( ) are memory accesses, and the destinations are always at the right (note that other format standards exists). Some more concrete examples are as follows. In Figure 3.4 a SSE2 multiplication with the registers $xmm3$ and $xmm4$ as sources, while $xmm4$ is acting as destination as well. This instruction becomes 4 bytes large (prefix:1B + opcode:2B + operand:1B + REX:0B + constOffset:0B + offsetReg:0B).

For the multiplication in Figure 3.5 one source is the memory located 128 bytes *before* the position register rax points at. The $xmm4$ register is a combined source and destination. Five bytes are needed here (prefix:1B + opcode:2B + operand:1B + REX:0B + constOffset:1B + offsetReg:0B).

Finally in Figure 3.6 the change is that the offset is 128 bytes *after* the pointer. An extra offset register rsi is multiplied by 4, and added to the address rax points at. Finally, the source/destination register name used is $xmm12$. All changes increases the instruction size, to a total of 10 bytes (prefix:1B + opcode:2B + operand:1B + REX:1B + constOffset:4B + offsetReg:1B).

$$mulpd \qquad \%xmm3, \%xmm4$$

Figure 3.4: A multiplication instruction, size 4 bytes.

$$mulpd \qquad -0x80(\%rax), \%xmm4$$

Figure 3.5: Multiplication and memory load, size 5 bytes.

$$mulpd \qquad 0x80(\%rax, \%rsi, 4), \%xmm12$$

Figure 3.6: Multiplication and complex memory load, size 10 bytes.

Cycle Exact Timing

All newer x86 processors have a functionality for obtaining cycle exact timings. Using the *rdtsc* instruction one retrieves a 64 bit integer containing a current cycle counter value. This cycle counter is normally incremented by one every single cycle the processor is active (including being idle). By using this counter it is therefore possible to do effective perfect benchmarks, as long as the operative system and other tasks are not running at the same time as the benchmark.

3.4 Intel® Pentium 4 Architecture

The Intel® Pentium 4 Family (Intel NETBURST® microarchitecture) consists of several processor versions, with a range of abilities. Each revision have brought improvements and bug fixes, thus each version behaves somewhat differently. This processor is not extensively described because of its age, and that our implementation for it is not based on deeper low level features.

For the used version, the L1 cache data cache is 8 KB having 4 ways. It can handle one read and one write every cycle, while supporting up to *4 misses* at the same time (later extended to 8). Each cache line is 64 byte, or 8 floating point doubles.

The instruction cache called *trace cache* is quite special, it stores 12k pre-decoded micro instructions (μops). Its effective size can be almost 100 KB, but it depends on the actual instructions size. A maximum of one instruction can be decoded every cycle, this means that code not in the instruction cache is executed at about 1/3 speed.

The L2 cache is 512 KB with 8 ways, with a cache line size of 128 byte. Several hardware prefetchers are also used, the data prefetchers detect sequential misses in the L2 cache and preloads data according to the found pattern. All caches use a *pseudo-LRU* replacement scheme.

The processor use OoOE to improve throughput, issuing up to 3 micro instructions every cycle into 4 different issue ports. Only one of the ports can do SSE/SSE2 additions and multiplications, while there are separate ports

for loading and storing data to memory. Table 3.1 shows the throughput and latency, without memory references. Additional latency is required if for memory accesses, unfortunately available documentation is lacking any specifics. The throughput in the table indicates how many cycles between each can be started in that execution unit. Note that addition and multiplication are located in different execution units and can therefore be performed independently of each other, but not started at the same time since they share issue port. The details are summarised from [4].

Table 3.1: Pentium 4 (model 2) instruction details.

	Latency	Throughput	Issue Port	Execution unit
addpd	4	2	FP_Execute	FP_ADD
mulpd	6	2	FP_Execute	FP_MUL
moveapd	6	1	FP_Move	FP_MOVE

3.5 Intel Core 2 Architecture

The Intel® Core^{tm} Family (Intel® Core^{tm} microarchitecture) consists of several processor versions. The one used here belongs to the *Enhanced Intel Core* microarchitecture family.

In the Enhanced Code 2 architecture, each processor is physically a dual core, with shared L2 cache. The quad-core variants have two of these dual cores located inside one chip package, where each pair communicate over a shared front side bus. Each core have separate L1 caches, one for instructions and one for data. Both L1 caches are 32 KB with 8 ways, the precise replacement policy used is unknown.

The shared L2 cache is 6 MB with 24 ways, with cache lines of 64 byte. It have very high bandwidth capable of moving a cacheline to L1 cache every 2 cycles, with varying latency. There are several prefetchers that work on moving data. There is one that prefetch data from L2 into L1 cache based on detected access patterns in the L1 cache, its described as a “streaming prefetcher”. A different prefetcher is the *instruction pointer — based stride prefetcher*, which analyses individual load instructions and try to predict what each will load in the future. This prefetcher assumes that load instructions that are stored in memory, with the 8 lower order address bits equal, are the same

instruction. Finally, there is a prefetcher that detect misses in L2 cache and preloads expected data from memory. All three prefetchers work on several data streams at the same time, although several special rules apply. There are numerous restrictions where the prefetchers do not work. The most limiting is that accesses that will cross a 4 KB page are not handled.

Instruction Decoding

The Intel® Core 2 processor can load 16 bytes of instructions each cycle (from L1 cache), and prepare them for decoding. Up to 6 instructions are pre-decoded from the 16 byte block, if there are more instructions in the block the loading must halt. This is only a problem if the instructions are short, and this is not a problem here as the SSE instructions used are relatively large.

The pre-decoded instructions are then pushed into a 18 instruction deep queue. This queue act like a small cache in a similar way to the instruction cache on the Pentium 4. The queue can send up to 5 instructions to the decoders, however in 64 bit mode this is effectively only 4 (in 32 bit mode some instructions can be merged into one). The decoders can accept up to 4 instructions each cycle, where one decoder can handle complex instructions and 3 can do simple ones. All the SSE type instructions are considered simple on this architecture, and all 4 is therefore considered equal here.

For more complex instructions that involve both memory access and some logical/math operation a technique called *Micro-fusion*[4, 2-9] is used, so that both operations are handled as one single operation. This means that instructions of this format is more effective for the decoders in addition to other effects 3.3. One unknown is instructions that span two 16 byte blocks, and how the pre-decoder handles them.

Table 3.2: Enhanced Core 2 instruction details.

	Latency	Throughput	Issue Port	Execution unit
addpd	3	1	Issue port 1	FP ADD
mulpd	5	1	Issue port 0	FP MUL
moveapd	1	0.33	Issue port 0, 1, 5	—

The speculative OoOE engine have a large number of rename registers, and can handle 96 in-flight micro instructions (μops). In addition 32 instructions

can be held back if they depend on data that is not available yet. Numerous other features that are not mentioned here also play a key role in the OoOE engine. Instruction latencies and throughput are shown in Table 3.2, the throughput of 0.33 means that 3 can be executed every cycle. Since the 3 key instructions go to different ports at only one of each can be performed each cycle, even while more can be decoded. The details are summarised from [4].

3.6 Current Optimization Techniques

The three most central optimization techniques used, that are well known, is presented here. First loop tiling is described and illustrated. Then the use of data relocation and code unrolling is explained.

3.6.1 Loop Tiling

Loop Tiling or blocking is designed to improve cache reuse [15]. This is done by partitioning the data set into blocks that fit into the processor caches. One can then perform more operations on the data, while it is still available in the faster cache system. An code example of how this is done is shown in program 2 based on the original code in Program 1, code is from `http://en.wikipedia.org/wiki/Loop_tiling` verified on 26. April 2009.

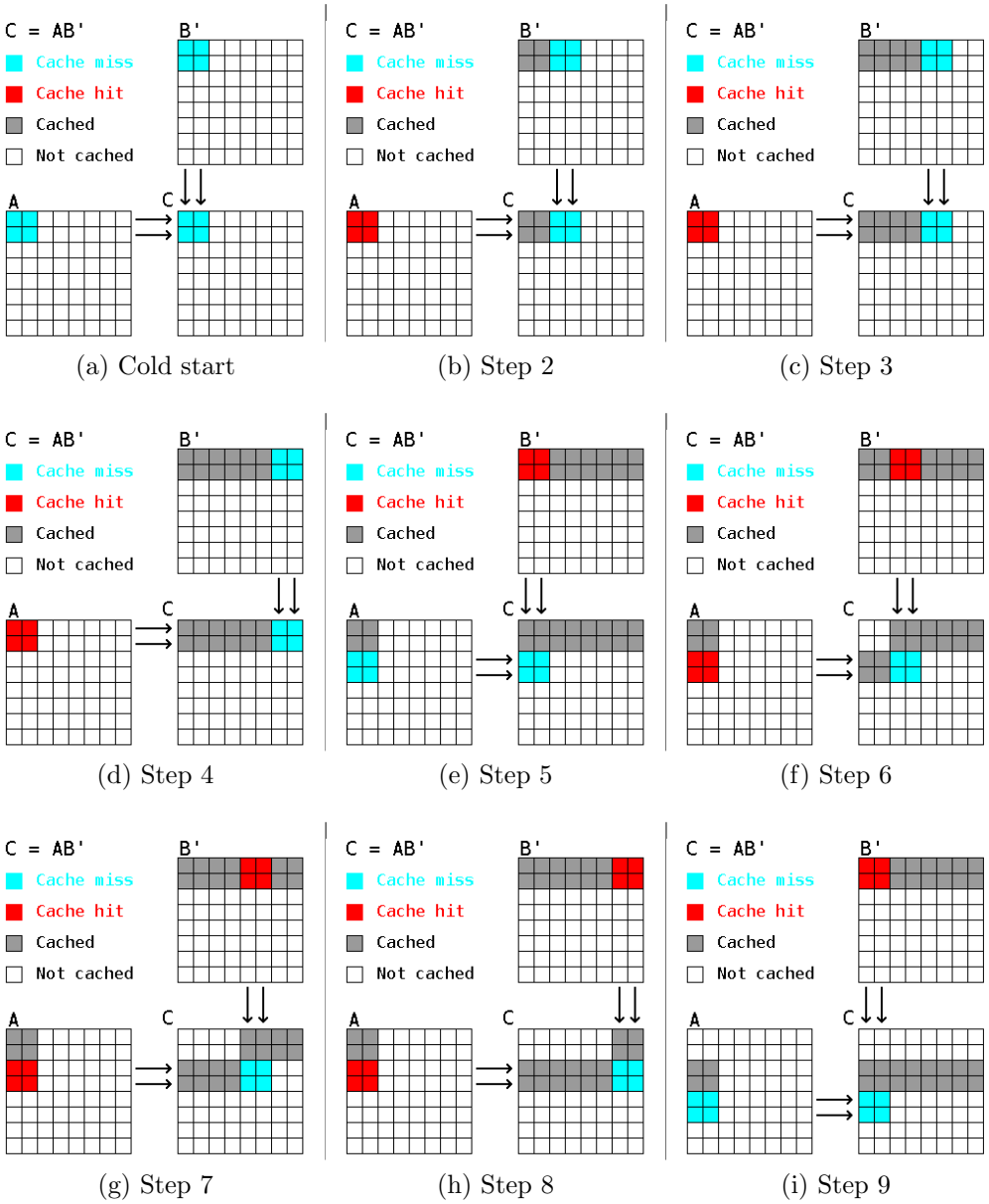
Program 1 Basic code for the matrix multiplication $Z = Z + XY$.

```
DO I = 1, M
 DO K = 1, M
   DO J = 1, M
      Z(J,I) = Z(J,I) + X(K,I) * Y(J,K)
```

Program 2 Loop tiled (blocked) code for the matrix multiplication $Z = Z + XY$.

```
DO K2 = 1, M, B
  DO J2 = 1, M, B
    DO I = 1, M
      DO K1 = K2, MIN(K2+B-1,M)
         DO J1 = J2, MIN(J2+B-1,M)
            Z(J1,I) = Z(J1,I) + X(K1,I) * Y(J1,K1)
```

How the cache reuse works is further illustrated in Figure 3.7, 3.8, 3.9 and 3.10. Note that only the partial $C = AB'$ part is illustrated. Figure 3.7 (a) show the first step, were all the data generate cache misses. In step 2-4 (Figure 3.7 (b-d)), the block of the A matrix that is located in cache is reused, while the next blocks of the B matrix generate cache misses. At step 5 (e) the next block of the A matrix is used, while the first block of B reused. The next 3 steps (6-8) reuse data from both A and B matrices. This pattern continues through Figure 3.7 and 3.8, showing the steps of *two* of the loop tiling stages in the matrix multiplication.

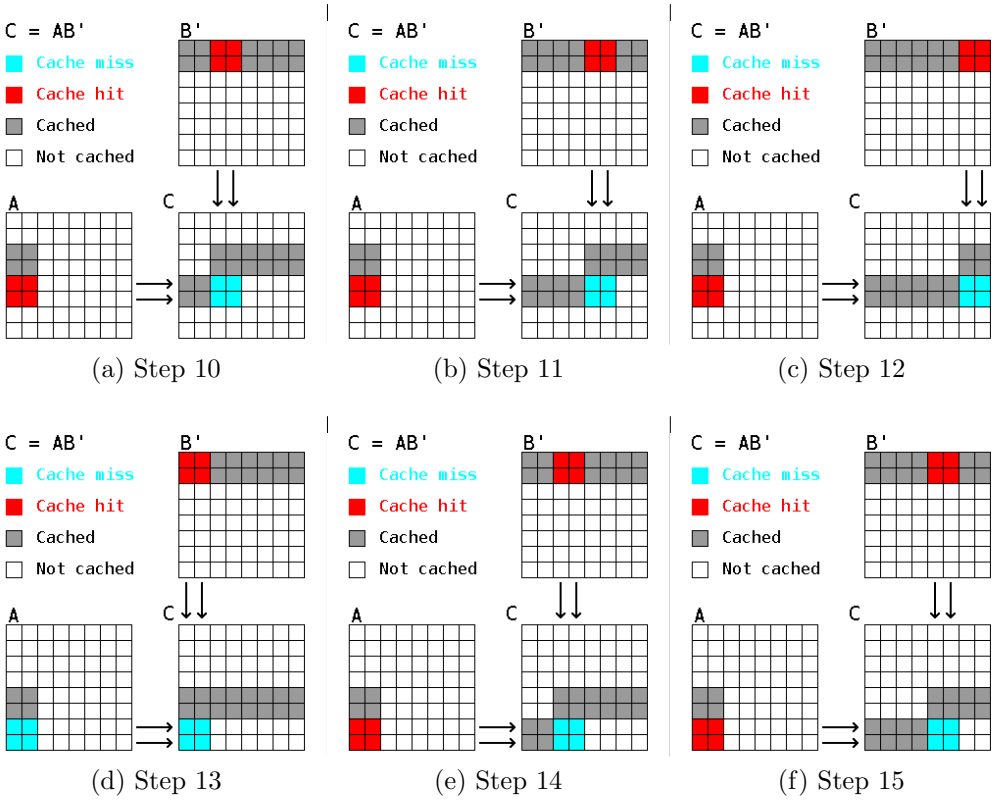


(a) Cold start (b) Step 2 (c) Step 3

(d) Step 4 (e) Step 5 (f) Step 6

(g) Step 7 (h) Step 8 (i) Step 9

Figure 3.7: Block division and cache illustrated.



(a) Step 10 (b) Step 11 (c) Step 12

(d) Step 13 (e) Step 14 (f) Step 15

Figure 3.8: Block division and cache illustrated.

Figure 3.9 is showing the last step, and illustrates how the performed blocking might just a step in a recursive blocking algorithm. In this illustration, the calculations stored are only partial answers, requiring 4 more passes until the final answer is completed in the C matrix. This is the *third* loop tiling level, and is performed in the K dimension. The first step in pass 2 is shown in Figure 3.10, with the completed first pass indicated.

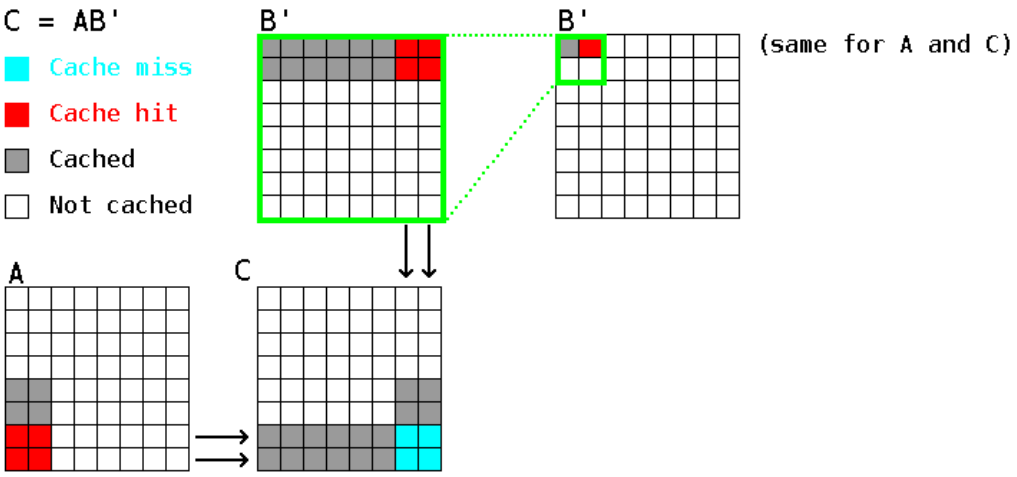


Figure 3.9: Step 16, with an added illustration of another blocking layer.

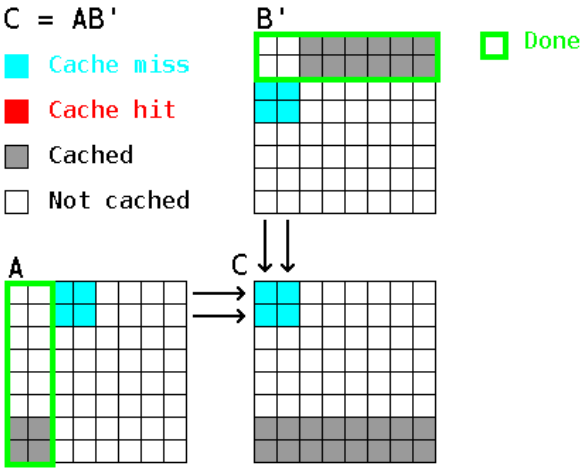


Figure 3.10: Step 17. Shows the completed part of the calculation, and how the algorithm continues.

Various block shapes also play an important role for how effective loop tiling might be. This is illustrated in Figure 3.11 where 4 possible shapes are shown. Getting the best ratio between cache usage and number of calculations should give best performance, since this will reduce the required memory bandwidth. Note that the illustration(s) and associated calculations are simplified, particular the cache required by the C matrix.

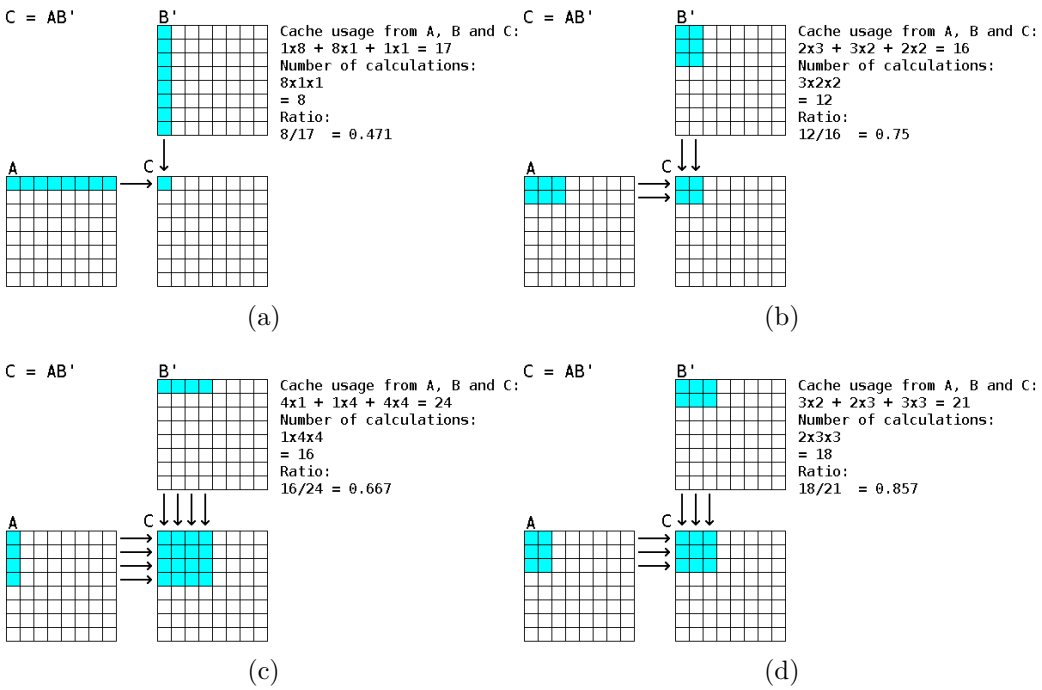


Figure 3.11: Various block shapes and how cache efficient they might seem to be.

Data Relocation

Data relocation is a technique where one transform, or move key data to a layout that is more effective for the processor to work with. Normally this targets the spatial properties of caches, by moving different data elements that is accessed at almost the same time, to a sequential block in memory. This relocation can therefore reduce latency, since the first access loads all (or some) of the other data elements needed at the same time. If the data was separated into different logical or conceptual arrays, every data structure would get its own latency slowdown when accessed. The use of data relocation is shown in Figure 3.12, where the data from pass one (in section 3.6.1) is relocated into two dedicated arrays.

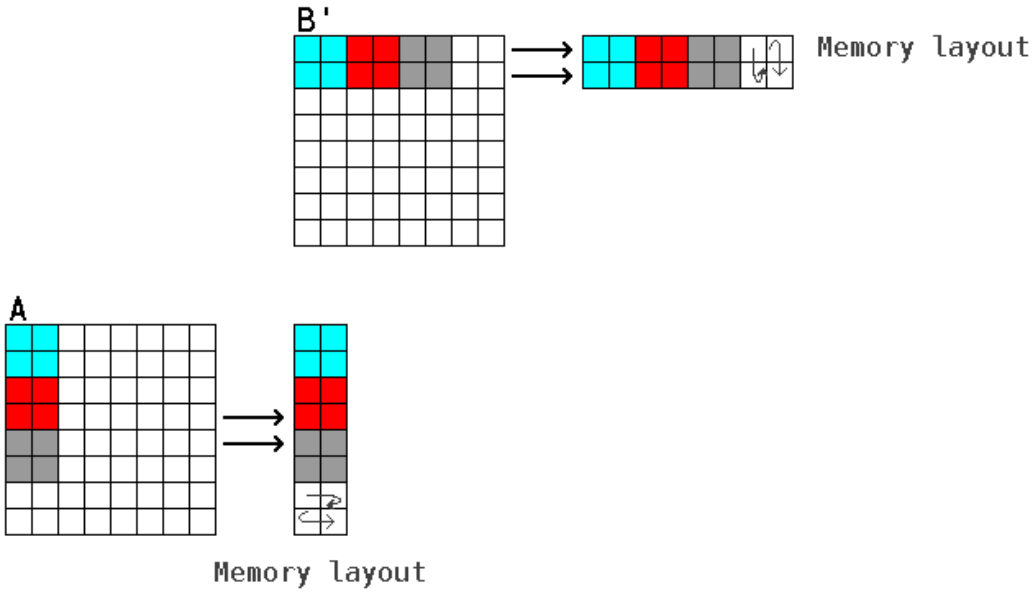


Figure 3.12: Illustration of data relocation, where selected parts of A and B' are moved into two different data arrays.

There are also several other properties that can contribute to performance gains from a good relocation, while they are not as common. Reducing the number of separate data streams (memory arrays/working regions) can lead to improved prefetching by newer processors, if the accesses are somewhat predictable. Fewer data streams also match the internal organisation properties of caches with few ways, especially AMD processors that only have a 2-way L1 cache.

Unrolling

In order to avoid extra calculations and branches a critically important technique is *loop unrolling*. In program 2 the innermost loop requires several integer calculations to update address pointers, update the loop counter and check if the loop is completed. By unrolling the loop by hand one can eliminate some of the calculation by only doing them once at the start of the loop.

3.7 Tools

A number of tools are needed to write efficient programs. Only a few will be briefly described, with some more details on the compiler.

3.7.1 Compilers

The single most important programing tool is the *compiler*. High level programing languages enable significantly higher productivity, and are critical for large and complex programs. The compiler translates the high level code into *low level machine* code, using a number of advanced rules and algorithms. One of the tasks compilers are supposed to be good at is analysis of code, and reformulating it to *assembly* that can be performed quickly by the microprocessor. Some of the techniques used are loop detection linked to automatic loop unrolling. Data dependency analysis for reordering and removal of calculations. Automatic vectorization of calculations that can be performed in parallel, using SIMD instructions (SSE). And optimal register choice and data dependencies are found using graph analysis.

An extensive model of the inner workings of target processors is used to tune both instruction selection and sequential ordering. Using both throughput and latency of instructions, combined with pipeline and execution unit information. This gives an analytical approach to optimal code. It is also possible to specify the degree of complex optimizations the compiler performs, using numerous flags. Selecting the right flags can be important, as they can greatly affect performance or even numerical accuracy. For simplicity, performance can be selected using the flags -O0 -O1 -O2 and -O3, selecting a large set of flags. The fewest optimizations are performed with -O0, and the most extensive with -O3.

3.7.2 Valgrind

Valgrind [18] is a tool for both bug testing and optimizing code. Useful for identifying bottlenecks and getting run time analysis of program flow. It is a plugin based system, where tools can be added/selected for gathering of various data. The core is a *CPU emulator* that can do, test and record almost anything (depending on available plugins). This means that one can take a binary program and do analysis on it without adding any extra code for instrumentation, so the program to be tested do not need any changes at all.

There are a few changes that will improve feedback. Having the program source code allows partial linking between Valgrind's output, and where in the code performance issues exist or the program fails. Compiling with debug data (even with -O3) gives good linking up against the source code. One can also add hints and instructions to Valgring in the source code for aiding or speeding up the analysis. While it it mainly intended for single thread/CPU usage, it have support for running multithreaded programs, but the type of feedback is currently lacking (some new tools for multithreaded support is in beta state).

3.7.3 Objdump

The objdump program in linux can be used to analyse the assembly instructions inside binary programs. This allows verification of instruction choice and several other properties that can not be controlled from program code or even inline assembly. Checking the actual size of each instruction is reasonably easy, and it is possible to analyse how both loops and single instructions are aligned (see section 3.5 why this matters).

3.8 Topics Not Covered

A number of other important concepts and techniques are not explained, even while they have been analysed and tested extensively. These details are not included in this thesis since adding a lot of these details would take too much time, and go beyond the main scope of this thesis. The issues

relating to them require relatively complex explanations, while they wont be discussed much later on. This comes partly from their gains are either small, have been partially worked around, or are not included in the code used for testing. Of all the concepts not explained are issues related to the *Translation lookaside buffer* (TLB) one of the most important. Numerous TLB problems exists from a performance perspective, and combined with *paging* it must be handled carefully. Because of bugs and limitations in the available operating systems, TLB and paging became too problematic to cover in an acceptable way.

Multiprocessor support and theory is only briefly explained as this is achieved by using *OpenMP*. This reduced the visible parallellisation in our implementation to one line of program code, while still requiring much work and redesign. In order to use OpenMP several complex translation functions have been made, and these were intended to form the basis of our currently incomplete *pthread* implementation. Lacking a pthread version, the theoretical issues relating to *affinity* and multi-processor cache layout have been excluded.

Optimal data copying and relocation, using *software prefetching* instructions and special *non-temporal write* instructions are not mentioned. Our latest C implementation fail to utilise both instructions in the correct way, as the compiler do not handle that code correctly. Because employing these instructions require code to be dynamically generated in assembly language, the issue is excluded in the evaluated implementation, and thus considered beyond the main scope of this thesis. *Cache bandwidth* and associated latencies have also been excluded from the scope of the thesis, even while they are critical for performance. The theoretical considerations related to cache bandwidth calculations are well known, but play a lesser role because of our implementation design.

Chapter 4

Optimizing codes on the Pentium 4

In this chapter the core issues of our optimization work will be presented and discussed. We start by looking at based on the Pentium 4, since the work on this platform also forms the basis for our later work on the Intel® Core 2 architecture, which will be implemented in Chapter 5. This part begins with theoretical considerations on how to implement the matrix multiplication in an efficient way. Then our implementation will be explained, with some details on how it works. A short analysis on how the reduced case implemented can be extended to the full case is also presented. A few other unrelated programs are also included, in order for them to be analyzed before the Core 2 implementation in Chapter 5. Finally, an evaluation of the implementation with a short conclusion at the end of the chapter.

4.1 Theoretical Issues

The main theory and techniques used in our implementation is explained in the following sections. First the symmetrical properties will be explained, and how to combine it with loop tilling. Issues with prefetching and memory access patterns are covered also covered. Issues related to the calculation order, and how changing it affects performance. For clarity the techniques are explained either separately and/or on the simpler $C = AB'$ case. The techniques will later be shown merged together in Chapter 4.3.

4.1.1 Symmetrical Considerations

Exploiting the symmetry of our target numerical algorithm, the symmetrical rank2k, is essential for the good performance. and will give a factor of 2 speedup. This is well known, and can be performed in two ways. The common way, used by ATLAS, is to employ a *General Matrix Multiplication* (GEMM) for calculating only $C = AB'$. With $C = AB'$ calculated, summing the lower left triangle of the C matrix with the upper right triangle forms $C = AB' + BA'$ (in the upper right triangle). Because having both triangles are redundant, only one of the triangles need to be calculated. Our implemented method is to calculate only one triangle in C, performing the calculation $AB' + BA'$ directly.

The calculation of $C = AB' + BA'$ is shown in Figure 4.1. On the left we have $C = AB'$ illustrated with some of the values needed to calculate x in the C matrix filled in. The colors in the C matrix indicate which rows and columns from A and B' are needed in the calculation. On the right $C = BA'$ is shown with solid colors, with the $C = AB'$ part overlapped in C with weak colors. Note that the same numbers are used to calculate x value in both parts, while the location it's written to is different. The symmetry axis is also shown.

An practical view on how the symmetry works, and what is useless to do is illustrated in Figure 4.2. The X part of the calculation comes from $C = AB'$ and the Y part from $C = BA'$. The lower left triangle is a mirror image of the upper right, both having the same data.

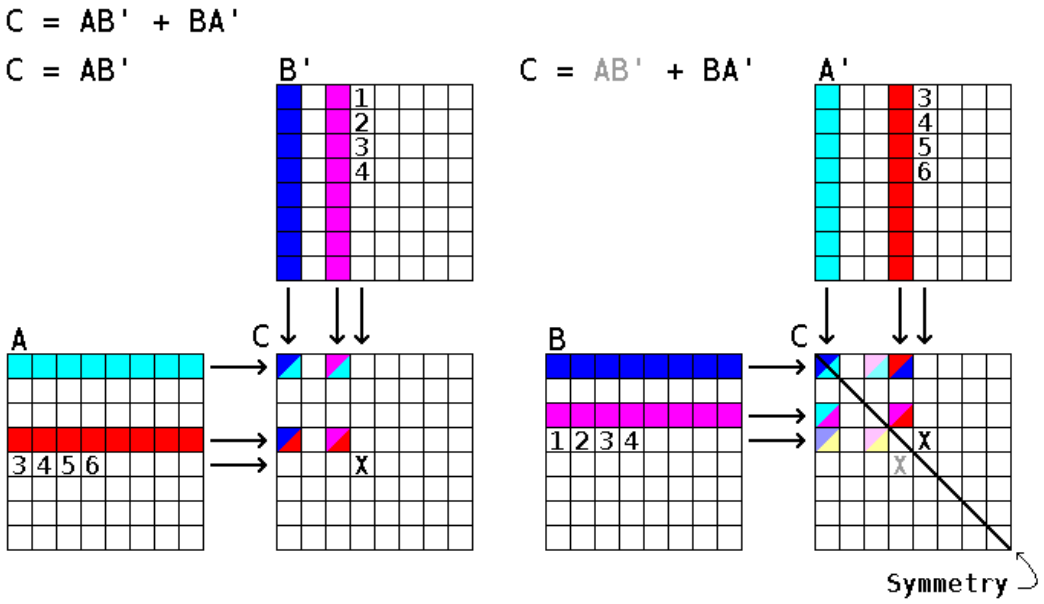


Figure 4.1: The two calculation parts merged at the right with the AB' part faded. The symmetry is also shown.

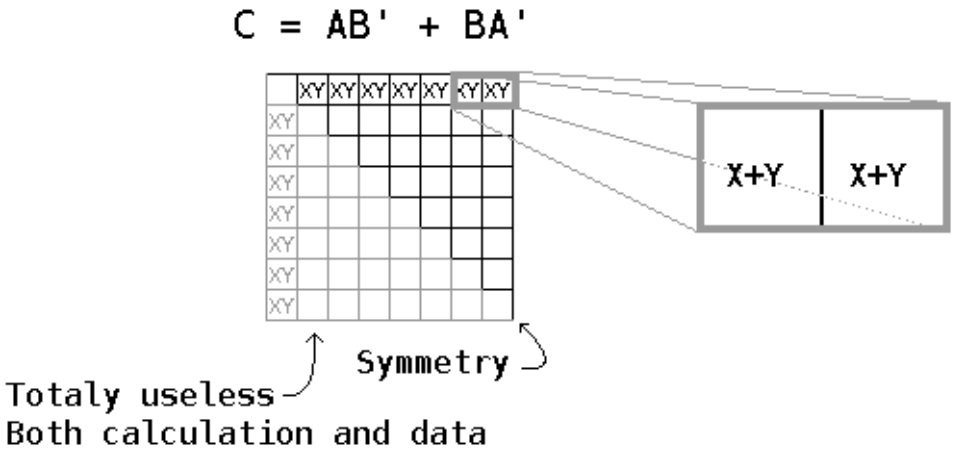


Figure 4.2: The symmetry illustrated.

4.1.2 Loop Tiling and Symmetry

Combining loop tiling and symmetry is useful, as this can reduce the number of read and writes to the C matrix. This is done by calculating rows and columns in both the AB' and BA' parts at the same time. This is shown in Figure 4.3 (a). Both X and Y are calculated at the same time, being stored to the same location around the symmetry axis. By adding them together and storing both to only one side, half memory accesses to the C matrix is avoided[1]. The accesses into C are also more expensive as both a read and a write is needed, requiring double the bandwidth. In addition changing between reading and writing cost extra time for DRAM, so avoiding 50% of them helps.

This requires having 4 data blocks in cache, shown in 4.3 (a). Two blocks are needed for the calculation of X, marked 'Direct usage'. With two other blocks for the calculation of Y, marked 'Symmetry usage'. Calculating the values of one row in C, shown in Figure 4.3 (b), requires calculating values of a column in C as well. A pattern of which data that is reused most and least is shown in 4.3 (c). Two data blocks remains constant while calculating a row/column in C, highlighted as 'Slow change'. The data marked 'Fast change' need to be replaced rapidly, and therefore less effective for caching. In order to gain most from the caches, one need to make the 'Slow change' blocks pointed out in 4.3 (d) larger. This will be explained in more details later.

[1]Note that the lower left part of C do not need to be used or touched in any way, including reading it.

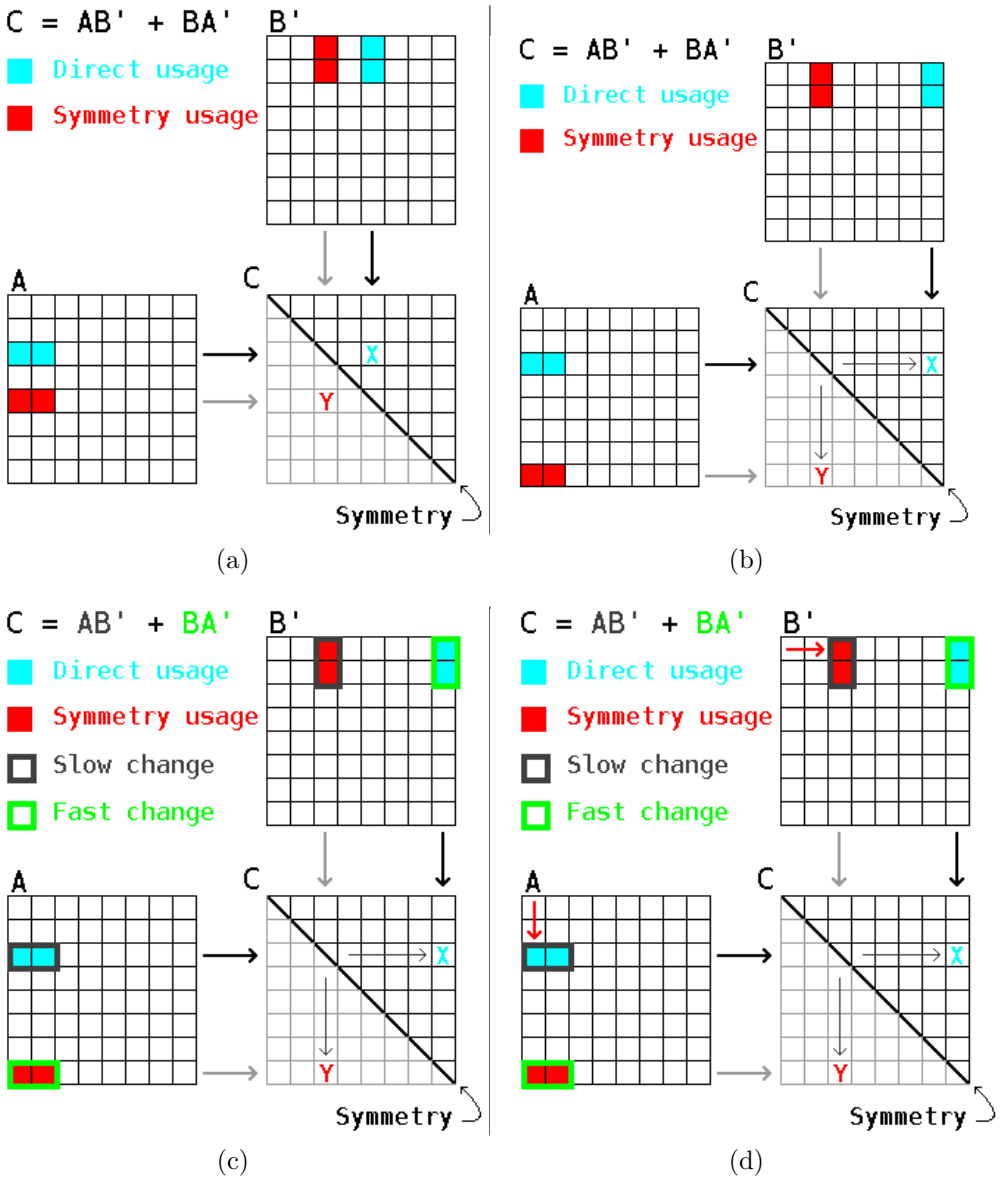
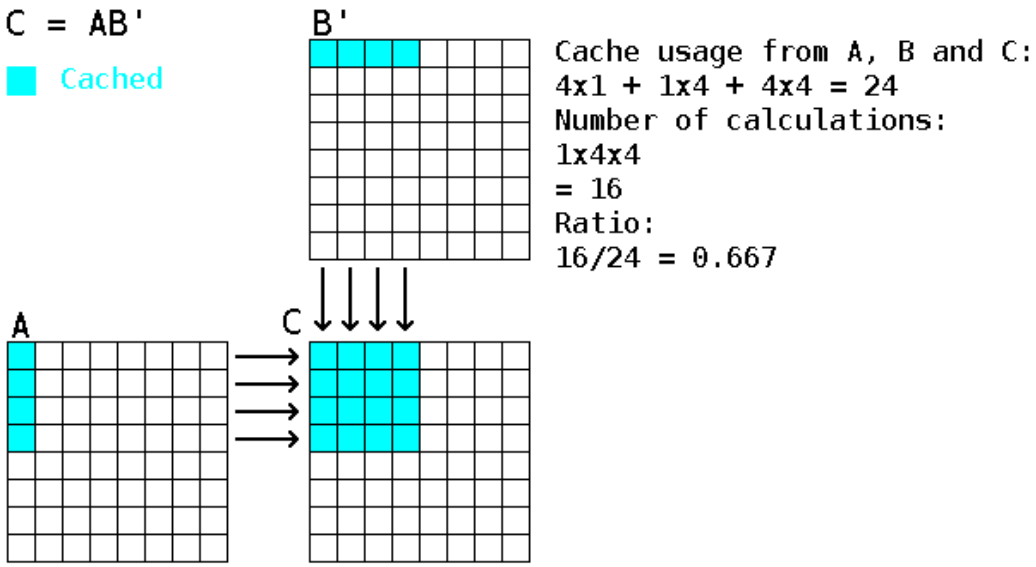


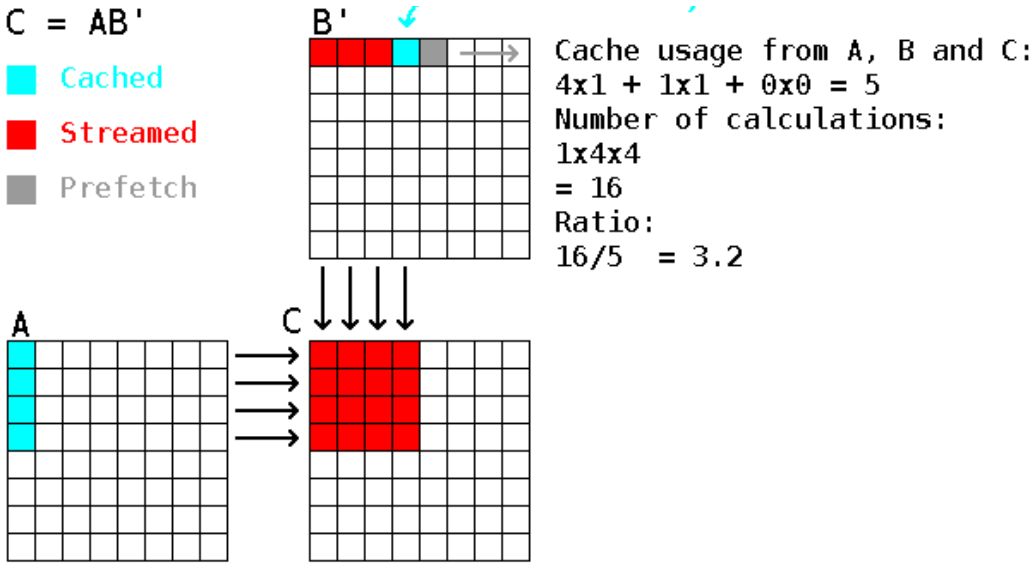
Figure 4.3: Blocking and symmetry combined.

4.1.3 Optimal Loop Tiling and Prefetching

The selection of dimensions sizes is important so that the cache is utilized in an optimal way. To utilize the pattern with the data marked as 'Slow change' (Figure 4.3), one makes this region(s) large to occupy most of the cache. At the same time the 'Fast change' region becomes very small, only a single row/column. When calculating one 'Fast change' block this will still take some time, as that single row/column must be multiplied with all the rows/columns in cache. This allows enough time to prefetch the next row/ column in the 'Fast change' region. This is illustrated in Figure 4.4. In (a) no prefetching is performed, giving a low cache utilization. While in (b) both the 'Fast change' region and the data needed from the C matrix is not stored in cache. Instead, the data is just streamed in and leaves the cache soon after use. In effect, both the cache and the memory bandwidth is utilized at the same time.



(a) Only blocking



(b) Blocking and streaming

Figure 4.4: Combining blocking and streaming for better cache usage.

4.1.4 Write Pattern

Because of the loop tiling performed, the order data values in the C matrix becomes somewhat irregular. In Figure 4.5, the blue circle in A is the 'Slow changing' data block, while the blue wave like line in B' are the 'Fast changing' rows/columns. The corresponding write pattern into C is shown as well. When all the partial rows/columns in B' have been iterated over once, the 'Slow changing' block is replaced. All the partial rows/columns in B' are then reread in the reverse direction, in order to exploit the parts of it still in cache. Note that the pattern is only an illustration, multiple recursive like levels exists (in order to exploit both L1 and L2 caches).

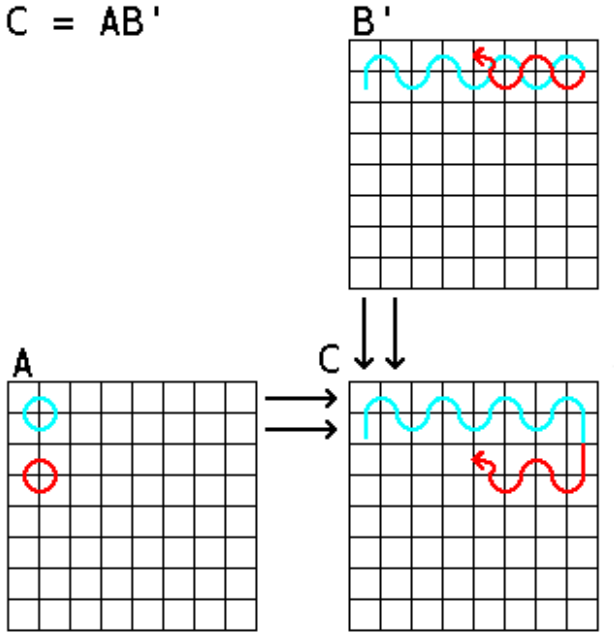


Figure 4.5: Illustration of a worm like access pattern.

The usage pattern in the C matrix is very unsuitable for many of the different processor features. Using only a single value in a stride'ed pattern goes against the spatial cache design, using only a single value in each cache line[2]. The hardware prefetchers also fails to detect anything but small strides, and only works inside small 4 KB blocks when they are successful. Hardware prefetchers therefor becomes totally useless, or at best occasionally effective.

[2]Depending on the layout of the C matrix and calculation sequence, this can be changed to many short but sequential accesses.

This same effect also occurs for the memory bandwidth usage, as entire cache lines must be loaded every time. Finally, DRAM access becomes slower and less efficient when loads are located far from each other.

In order to handle this, the actual write pattern is changed to a completely linear one, illustrated in Figure 4.6. Here, both the correct and the real write pattern is indicated. With this strictly linear pattern, all the previous mentioned processor features act in the most efficient way. After the calculation is completed, the original (correct) pattern must be restored. This is done by relocating the data in C.

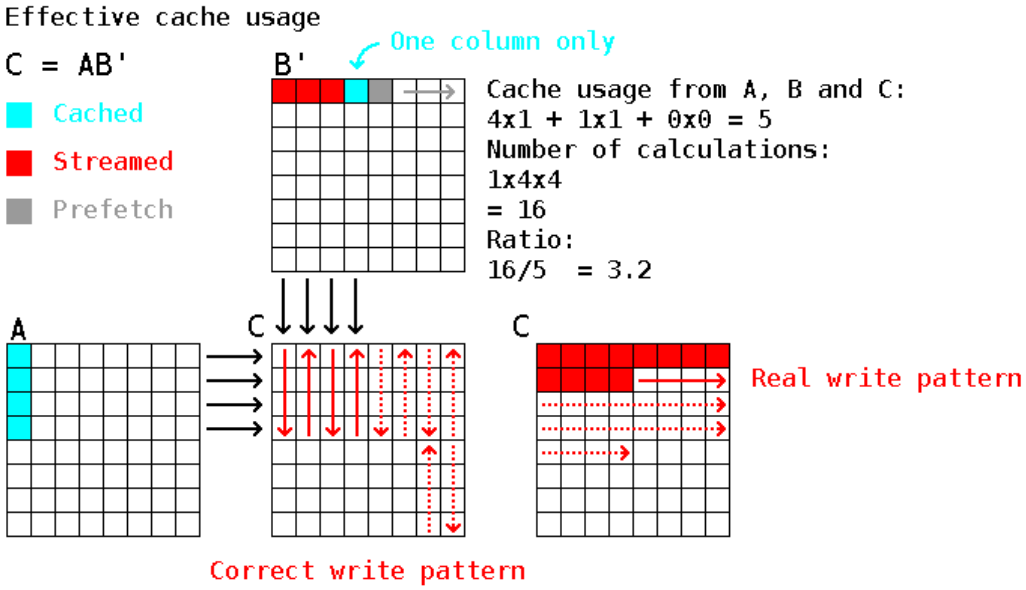


Figure 4.6: Linear writing of C while using worm like calculation pattern gives better hardware prefetch.

4.1.5 Data relocation and Prefetching

Data relocation is often used in matrix multiplications. For the calculation $C = AB$ the memory layout and calculation order do not match (note that the B matrix is *not* transposed). One row of A and one column of B is needed to form a single value in C. The memory layout of matrices are typically stored in *Row-major order*[3], meaning that all values in one row lays sequentially in memory. This implicitly require that one of the matrices (A or B) must be accessed in a non-sequential way, relative to the memory layout. In the same way as explained in Section 4.1.4 this is suboptimal.

For the $C = AB'$ case the B matrix is transposed, so every column becomes a row. Therefore, calculating one value of C needs one row from A, and one row of B. Both lay sequential in memory, matching the processor design. Combined with loop tiling this becomes somewhat less efficient however, as shown in Figure 4.7. The hardware prefetcher will fail every time a new section of a row is needed. After the first parts of the row is requested, the prefetching will start for the later parts of the row. Since only a small part of every row is needed, the prefetcher will load too much into cache. This wastes both cache and memory bandwidth.

By using data relocation, one can avoid these problems. In Figure 4.8 all the partial rows in A and B' have been copied into new arrays, shown on the right. This means that the entire length of the rows are used. When the first access into a new row performed, it will still fail to be predicted. The excess prefetch at the end of each row spills over into the next row, however. So when the next row is needed soon after, it has been prefetched. The prefetcher will also continue to load the rest of that row, since it continues to predicting the linear read pattern. Note that Figure 4.8 illustrates this independently of the previously explained concepts. As a side effect the number of TLB pages used is minimized, thus most of the same issues which Goto [9] addresses, are avoided.

[3] `http://en.wikipedia.org/wiki/Row-major_order` contains a simple overview with examples. Verified on April 27. 2009

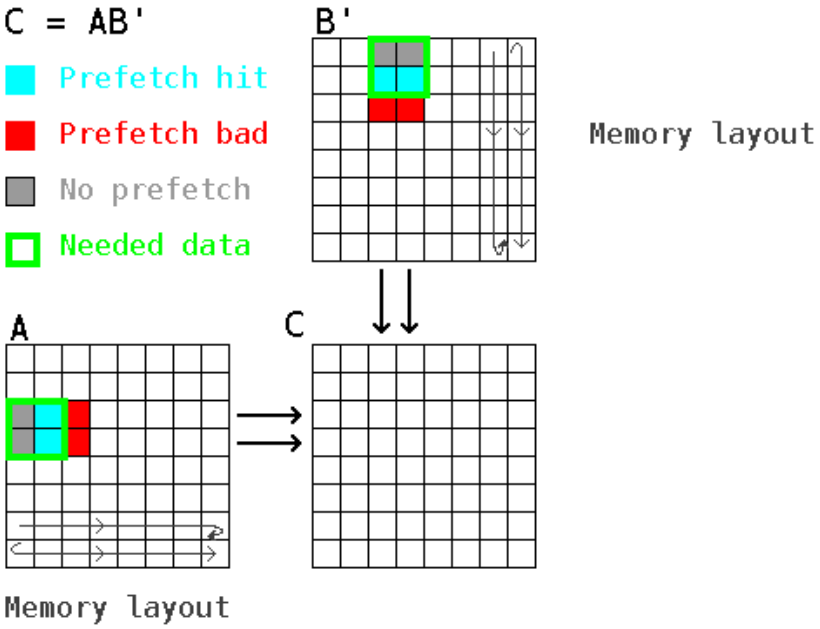


Figure 4.7: Illustration of the hardware prefetcher problems. The first values in each row/column are not preloaded, while too many are loaded at the end.

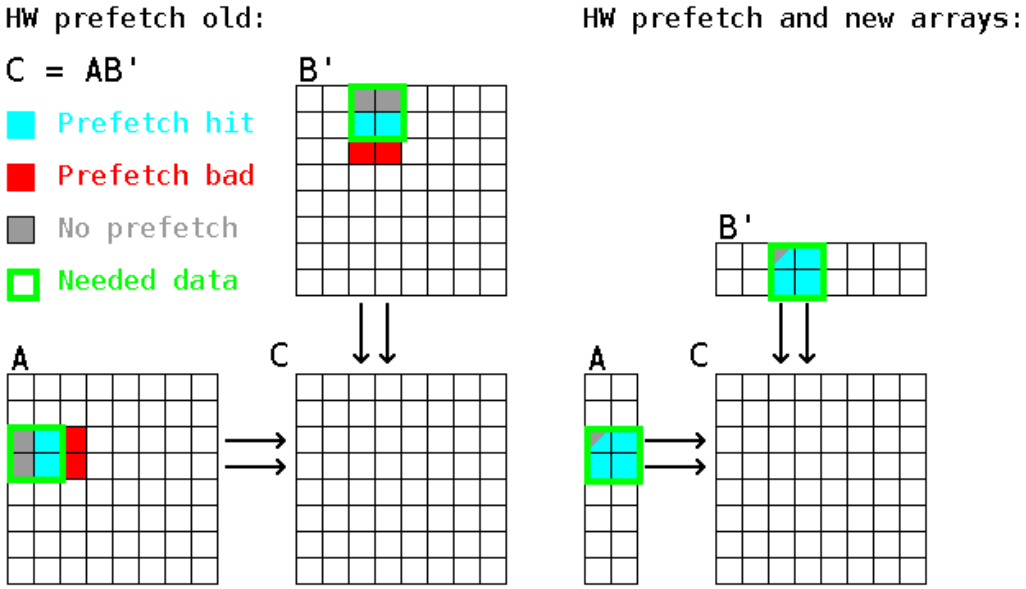


Figure 4.8: Illustration of the hardware prefetcher with data relocation. The only the first values of a single row/column are not preloaded, while all the following ones are prefetched.

4.1.6 Calculation Order

The ordering of execution have received extensive analysis. The basic and normal sequence to do calculations is completely linear. One loads and calculate every value, starting with the first and continues with the next. On a processor this can be illustrated, by looking at its inner state. After 4 cycles the state might be like what is shown in Figure 4.9. Four loads have been issued, but the data was not in L1 cache. The OoOE engine handles the lack of data, an continues to queue up work. In parallel the needed cache line is being moved to into L1 cache.

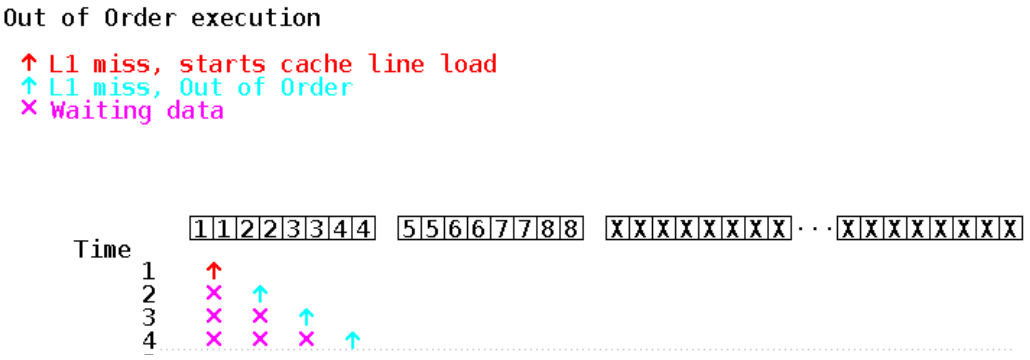


Figure 4.9: Out of Order Execution. The processor state after 4 clock cycles.

Figure 4.10 illustrates the state after 9 cycles. At cycle 5 the next cache line receives a miss, and the fetching process starts. Nine cycles after the first access, all the data in the first cache line becomes available (ignoring 'critical word first'). Work starts on the first data element (two values as the operation is SIMD), and all the work unblocks in the rest of the cache line.

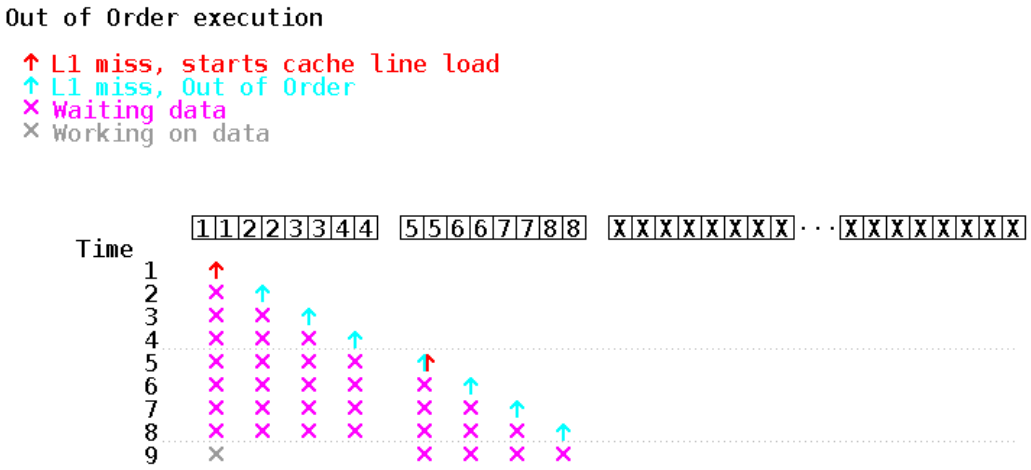


Figure 4.10: The processor state after 9 clock cycles.

Twelve cycles after the start, a pattern can be seen. Figure 4.11 show how this pattern looks like, where the calculations are simply delayed. Since the latency is negligible compared to the large number of calculations, throughput is thus unaffected. Unfortunately, there are limits to how much the processor can do out of order.

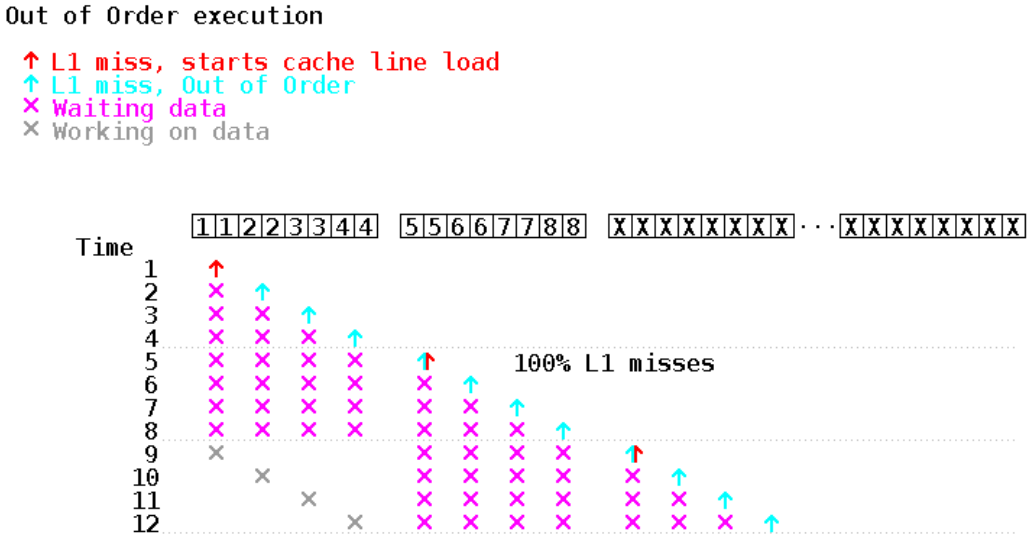


Figure 4.11: The processor state after 12 clock cycles. The last piece of work in the first cache line being processed. A stable pattern where all calculations are delayed emerges. Every load is a cache miss.

Every cache miss consume resources in the cache system (MSHR). The processor's Out of Order queue is also limited, and multiple versions of the same named register are required. This eventually generates a *stall*, as illustrated in Figure 4.12. Here, a stall in the pipeline occurs at cycle 7 (caused by internal resource limitation) is disrupting the stable pattern. Instruction queueing resumes at cycle 10, as an instruction leave the pipeline. Every few cycles this stalling will occur, limiting throughput. Note that in the Pentium 4 used, *only 4* cache misses can be handled at the same time.

Looking at the basic access pattern, Figure 4.13 show that several memory loads have no extra side effects. They are essentially useless, requiring many types of OoOE resources. The first access to a cache line need OoOE resources, but it also initiates the loading of the entire cache line. By temporarily delaying all but a single access to every cache line, better OoOE utilisation is achieved. Figure 4.14 show this, where every cycle an load initiates an entire cache line load. When all cache misses have been taken, all the skipped work is performed. This is illustrated in Figure 4.15 The first 1/4 of the work is misses (with stalls), the last 3/4 have no misses and no pipeline stalls. Here one will reduce pipeline stalls to only 75% of the time.

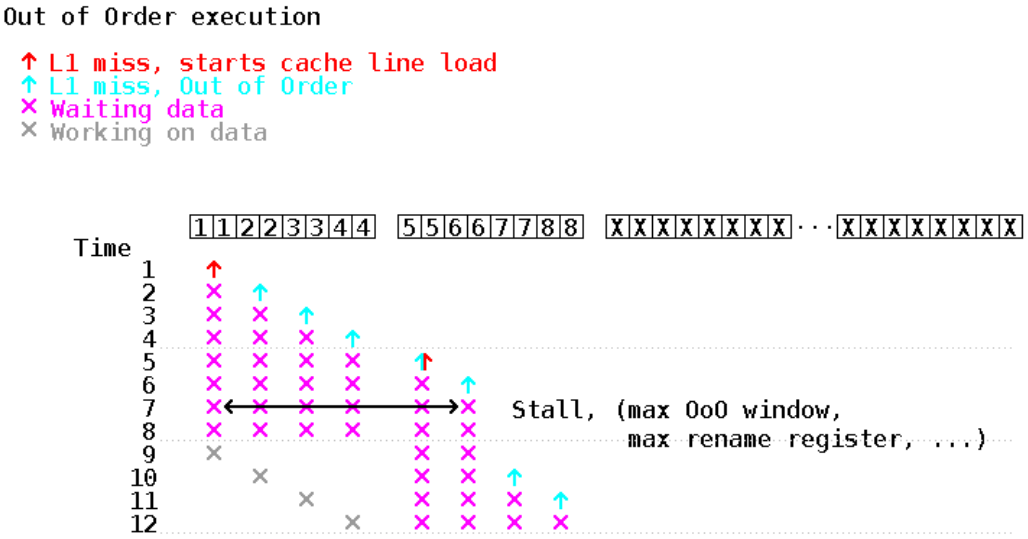


Figure 4.12: A more real processor state after 12 clock cycles.

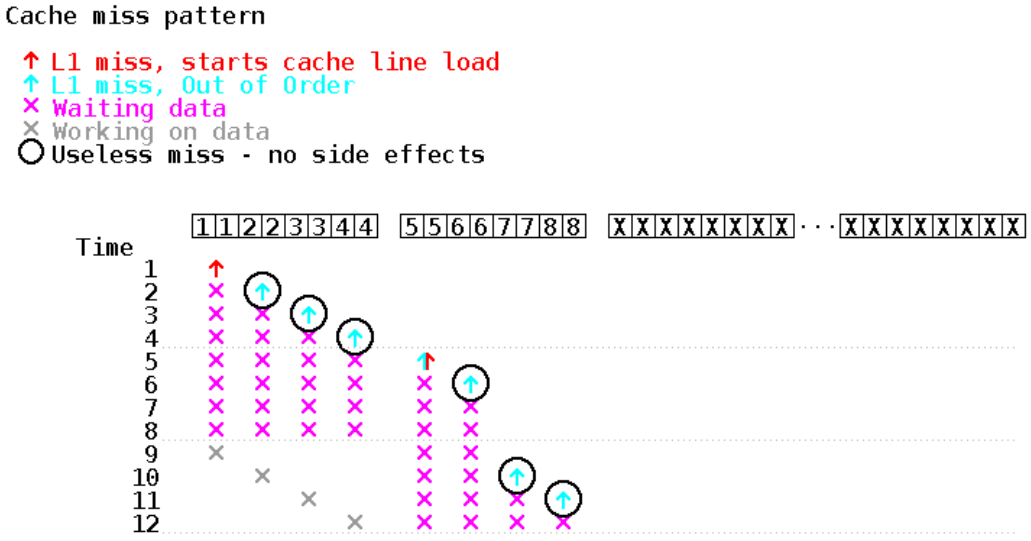


Figure 4.13: Data misses that do not carry any side effects highlighted. These consume OoOE resources.

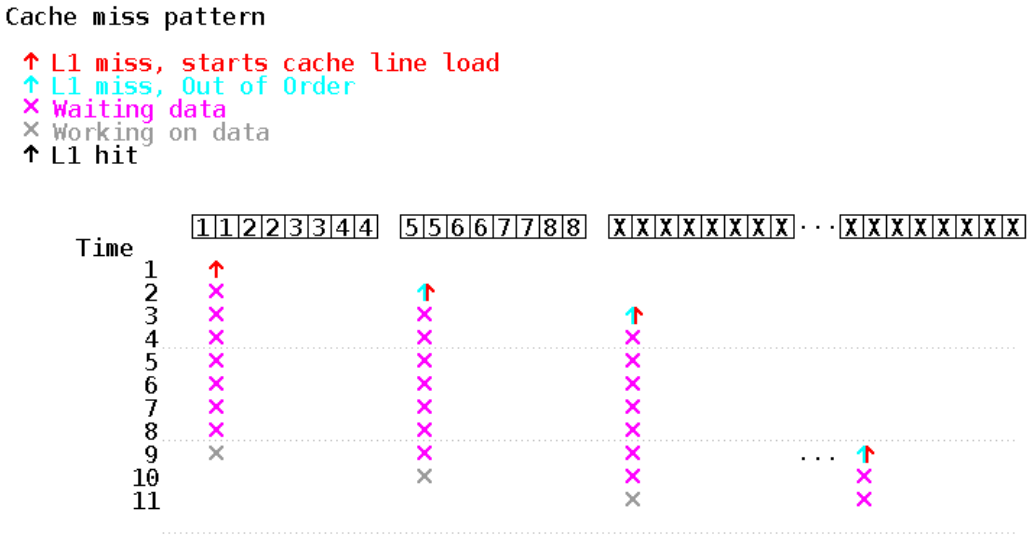


Figure 4.14: The issuing of all loads that generate cache misses first.

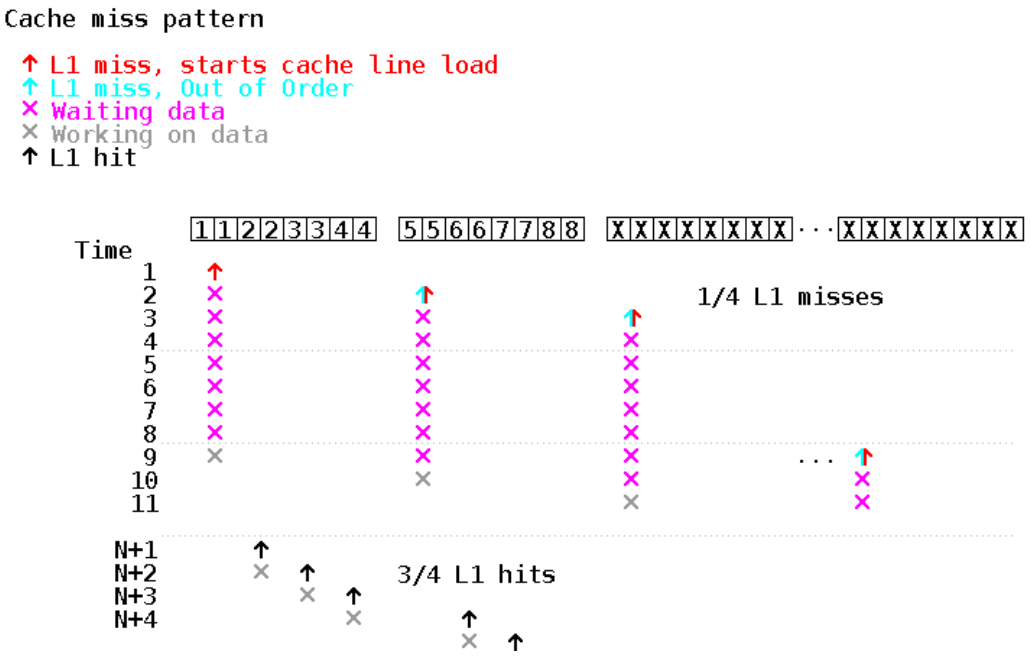


Figure 4.15: The skipped work performed, after all the L1 cache misses are done.

4.2 Implementation Issues

The current implementation have a number of limitations that need to be addressed. Because of the way x86 processors handles SSE memory loads, data must follow certain alignment rules. If this alignment is not correct the processor will crash. Alternatively, one can use specially dedicated memory load instructions with lower performance. A guaranteed correctly aligned memory is hence needed. For matrices that have a *odd number* $(2n + 1)$ size in either direction, this alignment requirement will be broken.

While we have not implemented this, it should be fairly straight forward to to. Since it is beneficial to relocate the data in A and B' matrices, the alignment can corrected during the copy. This will not cost much extra execution time, if any. The same problem exists for the C matrix, but using a linear write pattern will solve the issue.

In order to simplify the implementation of restoring the correct layout inside the C matrix, the calculated values are stored in a temporary array. This array requires about half the storage space of C, as only one half is needed for the calculation. In addition, a smaller array is needed for the partial rows copied out of A and B'. An explanation on how to avoid this extra storage space requirement will be given in Chapter 5.

4.3 Pentium 4 Issues

This section will explain in detail parts of how the Pentium 4 version works. The implementation contains an setup part, that allocates buffer memory and calculates control parameters. This will not be explained as it's not critical to the performance[4] *here*. After the setup, the *main kernel* calculates the values of the temporary C array. Finally, the values in the temporary array are copied into the C matrix.

Also, because the current implementation has fully unrolled loops in the K direction, the K dimension must be divisible by the length of unrolling. When this is not the case performance drops. In order to avoid this cost, several versions of the core functions can be created. This must be done manually, each with a different unroll factor, and is considered beyond the scope of this thesis.

[4]The memory allocation of the buffer do effect performance however.

4.3.1 Main Kernel

The main kernel works by alternating between two phases. The first is data *layout optimization*, where sections of data from the A and B matrix are copied into a single new matrix. The other phase is calculation by the *core kernel*. This alternation is performed until every section A and B have been copied once, with the calculation completed.

4.3.2 Data Layout

In the layout optimization phase, parts of A and B matrices are relocated. The basic *improved data relocation* concept will be explained first. Here, the relocated data is copied into a new data matrix, called AB later on. AB is constructed such that each row is made by a mix of a partial row from A, and a partial row from B. This is shown in Figure 4.16. Rows/columns[5] with equal index from A and B, become a column in AB. Figure 4.16 (a) and (b) highlights this. This also acts as the first level of loop tiling, as it eliminates data blocking in the row dimension from the core kernel.

The length of the rows/columns from A and B are 112 doubles each, making each row of AB 224 elements. This size was chosen empirically during the original delivery in the parallel course exercise, and were not modified in the later implementations. The number of columns in AB is equal to the rows in A and B. This is imposed by the symmetry. Every time this phase is performed, a new section of rows/columns from A and B are copied. If the final row/column in A and B have fewer than 112 values left, 0.0 values are substituted into AB. This is required as the length of AB is absolute. Using 0.0 values will not affect the calculation, as it does not alter to the answer numerically.

An *improved data layout* is also used. Inside each row of AB one alternate between storing two values of A and two values of B, in a sequential pattern. Figure 4.17 (a) illustrates this. The layout of a single value inside a cache line is pointed out, where the letter indicate which matrix the value came from. This tight integration is also highlighted in A, B and AB. Colour coding also show where data originates from, and where its stored in AB.

[5]Note that the columns of B' are equal to the rows of B. The naming of rows and columns do not really matter, since they are always translated into a linear memory layout. The notation try to be consistent with the illustrations however.

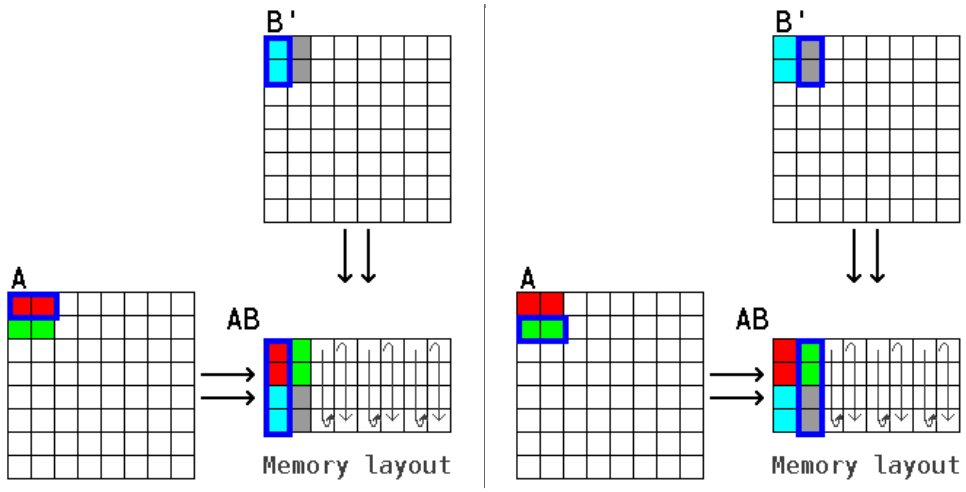
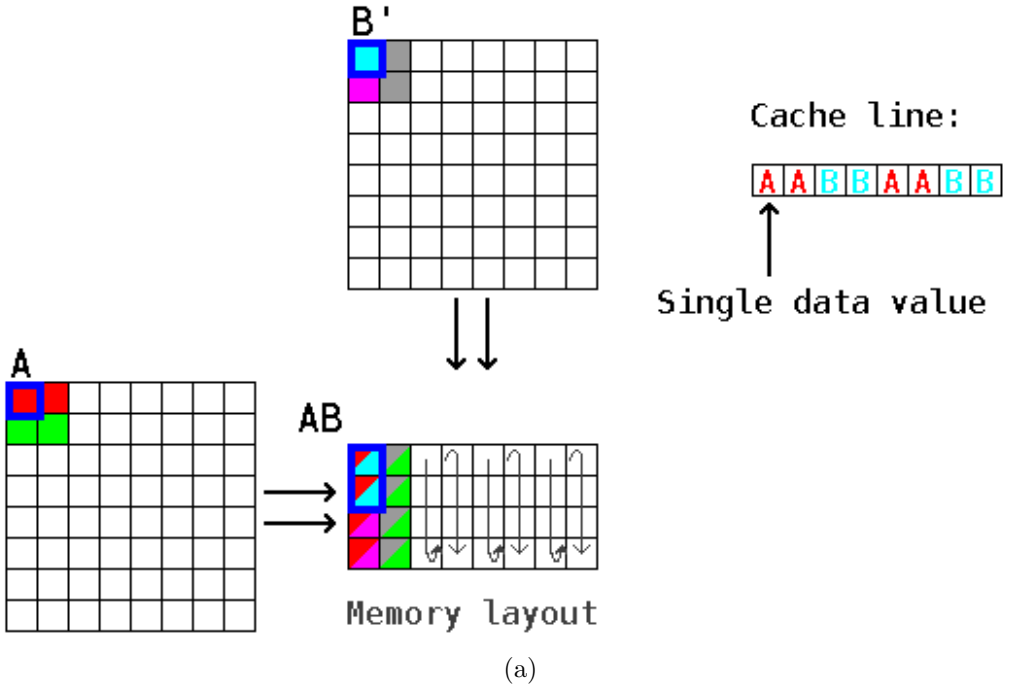


Figure 4.16: Illustration of the partially improved data relocation, where selected parts of A and B' are moved to a single array AB.

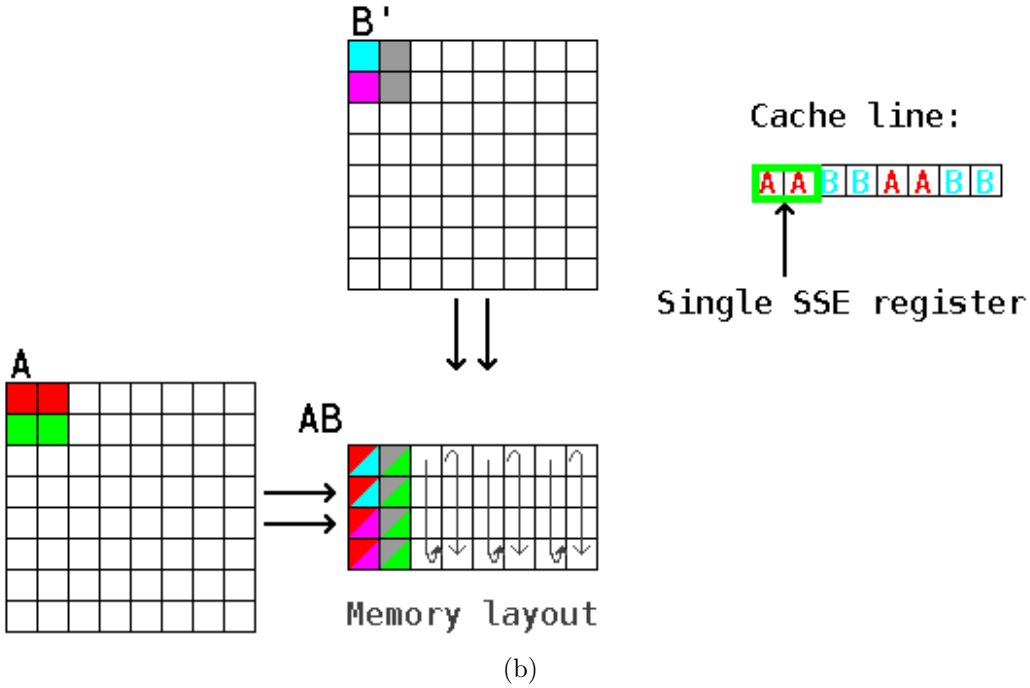
This double pairing of values play two roles. First, grouping two and two sequential values from the same matrix match the SSE register size. Shown in Figure 4.17 (b). Every load instruction can thus fetch two A values or two B values. When calculating $C = AB' + BA'$, it was shown in Section 4.1.3 that 4 blocks was needed by the loop tiling. With the merged layout only two data blocks are needed, this is shown in Figure 4.18. Both the 'Slow changing' parts are grouped into a single sequential memory block. Same with the 'Fast changing' data.

The effect of the layout is that the probability of unintended conflict misses are avoided. On the Pentium 4, with only 4 ways in the L1 cache this is critical. With 4 blocks, any extra data usage will increase miss rates. Like the one needed by accessing C, the program stack or simply that the pseudo-LRU replacement policy replaces the 'wrong' data.

Additionally, this layout enables better control of the cache layout, as we get only two sequential blocks to work with in the math kernel. Both reducing the number of memory pointers needed, and the number of data streams that the hardware prefetcher need to detect. Most of the same considerations as in Section 4.1.4, regarding DRAM is true as well. A side effect of the relocation is that the code implementation becomes cleaner, as the first loop tiling step $DOK2 = 1, M, B$ is factored away. This is the outermost level of loop tiling.



(a)



(b)

Figure 4.17: Illustration of the improved data layout inside the combined AB array. (a) Two and two values from A and B are interleaved. (b) SSE Register size highlighted.

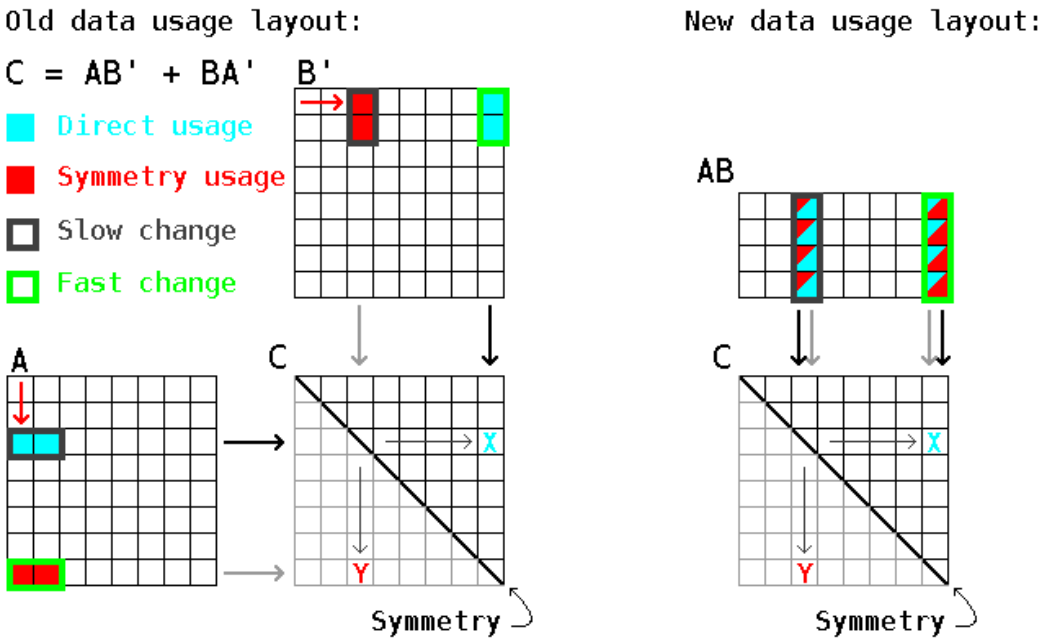


Figure 4.18: The old basic layout is on the left. Blocking, symmetry, improved data relocation and improved data layout combined on the right.

4.3.3 The Math Kernel

The second phase is the math kernel, it is constructed so that it performs loop tiling on the columns of AB. There are several levels of loop tiling, and only a simplified explanation will be given.

The kernel is factored into a *main section* of two for loops, performing loop tiling. Each iteration step is a function call, to a new level of loop tiling. There are 4 types of functions, 3 dealing with special cases. These special cases deal with edge calculations of various shapes, and will not be described. The last function is the optimal base case, and performs most of the calculations.

Base Case Function

Inside the optimal *base case function*, blocks of 128*128 AB columns are calculated. Program 3 show an slightly inaccurate, but conceptually correct implementation. The inconsistency relates to the mixed data layout inside AB, that both AB' and BA' is calculated at the same time, and that the calculation order differ. Those details will be shown correctly later in Section 4.3.5.

Program 3 Illustration code for the optimal base case function.

```
DO I = 1, 128
  DO J = 1, 128
    DO K = 1, 224
      C(J,I) = C(J,I) + AB_I(K,I) * AB_J(K,J)
```

In the actual implementation, the innermost loop over K is totally unrolled. In addition the second innermost loop unrolled once, and the two iterations steps are interleaved. This gives an unrolled block of 1 column * 2 columns * 224 values = 448 calculations. Note that each calculation needs 1 addition and 1 multiplication, the SSE2 instructions perform 2 operations each, giving a $*2/2$ contribution.

The innermost level of loop tiling is also inside the base case function. This function is designed both for L1 and L2 cache data reuse. A single 'fast changing' column of AB is kept in L1, while iterating over pairs of two 'slow changing' columns. For every iteration of the unrolled loop, a few values of the next 'fast changing' column is prefetched using a single prefetch instruction. When 128 'slow changing' columns have been paired with the single 'fast changing' column, a new 'fast changing' column is taken. This new 'fast changing' column have been totally prefetched, during the calculations with the last column. Essentially, the 128 'slow changing' columns are rapidly iterated over — but they are *replaced* slowly. The (empirically tuned) size of this 'slow changing' block take $128 * 224 * 8$ bytes (224 KB, same as found by Goto in [9]) of the L2 cache, or slightly less than half of the total L2 cache size.

Main section

The main section, calling the base case function, iterates in steps of 128 columns blocks. This is illustrated in Program 4 (again slightly inaccurate). The 'slow changing' block j is located in L2 cache, while the 'fast changing' block is i. Note that the 'fast changing' block of 128 columns change for every call to the base case function, effectively streaming over the entire AB in one go.

Program 4 Illustration code for the main section.

```
counter = 0
DO J = 1, N, 128
  DO I = J, N, 128

    call baseCaseFunction(&C_temp(counter), &AB(I,0), &AB(J,0))
    counter = counter + 128 * 128
```

In the actual implementation, the direction i is iterated in alternates for every iteration of j. This is similar to whats done back in Figure 4.5, although in a different context. For reference, the entire math kernel without the data copying is shown in Program 5. While its cleaned up, readability is still low.

Program 5 The math kernel code.

```
for (int bk = 0; bk < num_block_k; bk++) {
  const int start_k = bk *k_length;
  int flip_c_y = 0;
  int pos_c = 0;

  // Make a smart mixed data structure
  copyAB(n, k, start_k, a, b, ab);

  // Do k_length of all the rows of A and B, now inside ab
  for (int c_x = 0; c_x < num_blocks_x; c_x++) {
    for (int c_y = c_x; c_y < num_blocks_x; c_y++) {
      // Change access direction of i
      if(flip_c_y)
      {
        c_y_org = c_y;
        c_y = num_blocks_x + c_x - c_y -1;
      }
      // Get the start address of c for this block
      double *restrict c_inner = &c_temp[pos_c];
      pos_c += block_size_x * block_size_x;

      // Get the start address of row i in ab for this block
      const double *restrict abi_inner = &ab[c_y*block_size_x*k_length*2];
      // Get the start address of row j in ab for this block
      const double *restrict abj_inner = &ab[c_x*block_size_x*k_length*2];

      // Do the math
      if(c_y == c_x && (c_y == num_blocks_x -1) && block_size_x_last_part)
      {
        // Do the diagonal, with a partial block
        doDiagonal_endpart_inter(c_inner, abi_inner, abj_inner,
            block_size_x_last_part);
      }
      else if(c_y == c_x)
      {
        // Do the diagonal, with a full block
        doDiagonal_inter(c_inner, abi_inner, abj_inner);
      }
      else if(block_size_x_last_part && (c_y == num_blocks_x -1))
      {
        // Do a partial block at the end
        doBlock_Partial_inter(c_inner, abi_inner, abj_inner,
            block_size_x_last_part);

      } else {
        // Do a full block
        doBlock_inter(c_inner, abi_inner, abj_inner);
      }

      if(flip_c_y)
      {
        c_y = c_y_org;
      }
    }
    // Change access direction
    if(flip_c_y) flip_c_y = 0;
    else flip_c_y = 1;
  }
}
```

4.3.4 Improved Calculation Order

The concept of rapidly taking L1 misses described earlier, is not entirely new and is described by the ATLAS implementers in [20]. However, they assume that it is too hard in practise to create such a code pattern. Here, an *improved calculation order* is presented and implemented. Earlier, the technique of rapidly taking L1 misses, and then doing calculations without any misses was presented. The actual implementation modifies this. During the phase where L1 misses are taken, stalls still slow down the throughput. Eliminating this is possible, and illustrated in Figure 4.19. Some time after a planed L1 cache miss have been issued, the cache line becomes available. When this happens, the other calculations needing data from this cache line is performed. Interleaving calculations that generates L1 misses, with calculations that will hit L1. By doing this one can avoid overtaxing the bandwidth, number of cache misses, and other OoOE resources.

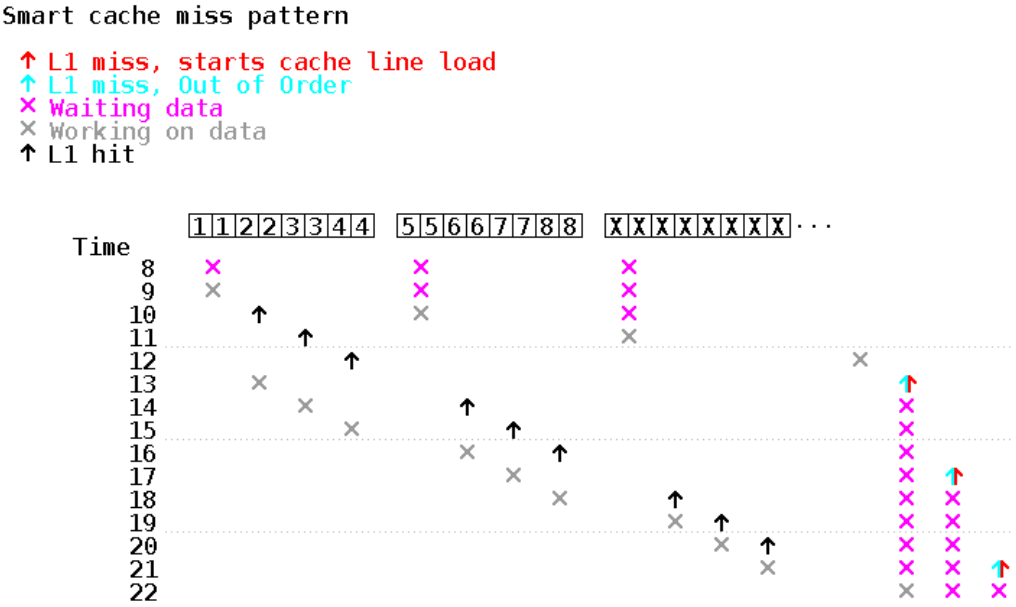


Figure 4.19: Some cache misses are issued first, later start working with the cache lines as they become available.

A balance of planed L1 cache misses and hits can be designed, illustrated in Figure 4.20. Work is divided into two parts that are interleaved. First, the combined calculations and cache misses, performing both useful work and

actively moves data into the cache. This early workload builds up as a queue consuming many OoOE resources. The later part is only calculations that never misses L1 cache. Calculations that never misses do not consume many OoOE resources, as it can be started and completed at once. This avoids both overtaxing the cache bandwidth, keeps OoOE resource usage in check, and therefore preventing pipeline stalls. Note that unlike normal memory, the L2 cache do not take any speed hit from non-sequential access. And the base case function prefetches all data to L2 cache before use.

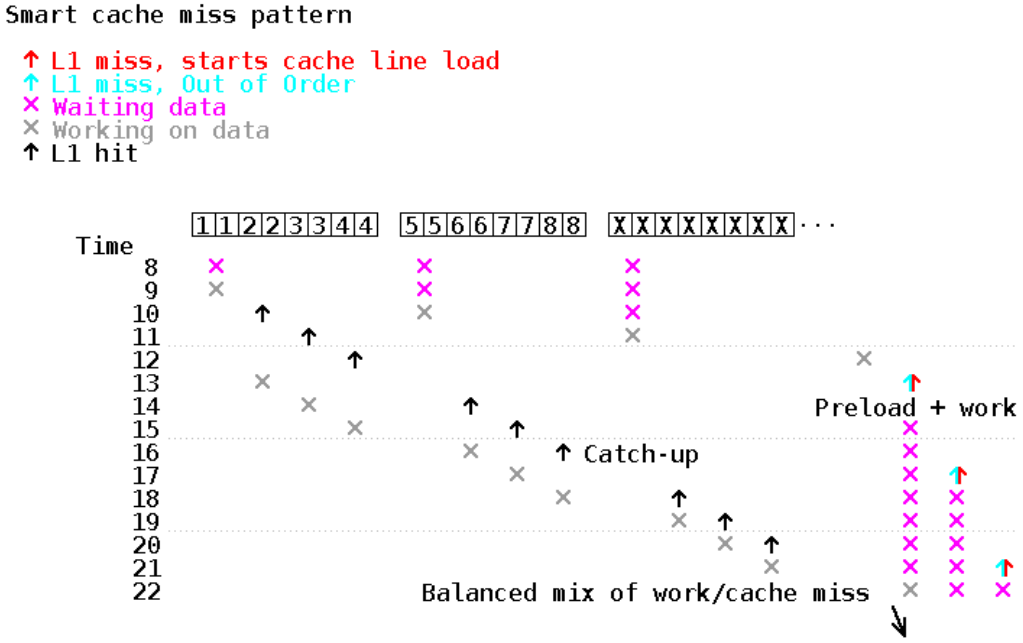


Figure 4.20: Misses and hits interleaved. Cache misses acting both as preloads for later calculations, and performs normal calculations.

4.3.5 Calculation Order Design

Implementing the improved calculation order can be challenging, as many factors affect its performance. First, the prefetch distance must be big enough to mask the latency of moving data into L1. Control and limitations of bandwidth must be correct. The OoOE engine might also operate differently at the start of a loop, compared to later on. When reading the first values of a column, the miss/hit ratio will also be higher. Finally, the data in AB

contains A and B values interleaved. In order to handle this, the unrolled base case function must be very flexible.

An implementation made by hand is likely both inefficient and time consuming. To avoid this, our code is made so that the sequence the AB matrix is accessed can be changed in an easy way, at compile time. In order to avoid extensive rewrites to test different calculation sequences, the access pattern is stored in an enumerated C list. Program 6 show an implementation of the base case function, where $LOAD\_I\_n$ and $LOAD\_J\_n$ encapsulated the calculation sequence. This code example show 4 accumulator registers, providing 4 independent data dependency chains for the OoOE engine. Two columns are calculated at the same time, and at the end the partial answers is written to C.

By encoding the offsets used when reading AB into an enumerator, the loading pattern can be generated by a script. After generation it can be added to the source code. To create the base case of normal linear data access, the Perl script in Program 8 will create the pattern. This pattern takes the interleaved storage of AB into account. This linear pattern is suboptimal however, as explained earlier.

Program 6 The innermost loop, unrolled using the enumerations. v1, v3 and v2, v4 accumulate data from two different columns respectively.

```
// A single calculation step of AB'. Performed on two columns.

// Load a value from column i, with data originating from A
// This load is a planed L1 hit.
temp = abi[LOAD_I_1];

// Load a value from column j, with data originating from B
// This load is a planed L1 miss, and L2 hit.
// Multiply, and add the answer to an accumulator register: v1.
v1 = v1 + temp * abj[LOAD_J_1];

// Load a value from column j+1, with data originating from B
// This load is a planed L1 miss, and L2 hit.
// Multiply, and add the answer to an accumulator register: v2.
v2 = v2 + temp * abj[LOAD_J_1 + K_LENGTH*2];


// A single calculation step of BA'. Performed on two columns.

// Load a value from column i, with data originating from B
// This load is a planed L1 hit.
temp = abi[LOAD_I_2];

// Load a value from column j, with data originating from A
// This load is a planed L1 miss, and L2 hit.
// Multiply, and add the answer to an accumulator register: v3.
v3 = v3 + temp * abj[LOAD_J_2];

// Load a value from column j+1, with data originating from A
// This load is a planed L1 miss, and L2 hit.
// Multiply, and add the answer to an accumulator register: v4.
v4 = v4 + temp * abj[LOAD_J_2 + K_LENGTH*2];

temp = abi[LOAD_I_3];
v1 = v1 + temp * abj[LOAD_J_3];

...

temp = abi[LOAD_I_112];
v3 = v3 + temp * abj[LOAD_J_112];
v4 = v4 + temp * abj[LOAD_J_112 + K_LENGTH*2];

C[current_c_position] += v1 + v3;
C[current_c_position +2] += v2 + v4;
```

Program 7 The linear access pattern, enumerated in C.

```
enum cache_pattern {
LOAD_I_1 = 0,
LOAD_J_1 = 2,

LOAD_I_2 = 2,
LOAD_J_2 = 0,

LOAD_I_3 = 4,
LOAD_J_3 = 6,

LOAD_I_4 = 6,
LOAD_J_4 = 4,

LOAD_I_5 = 8,
LOAD_J_5 = 10,

LOAD_I_6 = 10,
LOAD_J_6 = 8,

LOAD_I_7 = 12,
LOAD_J_7 = 14,

...

LOAD_I_109 = 216,
LOAD_J_109 = 218,

LOAD_I_110 = 218,
LOAD_J_110 = 216,

LOAD_I_111 = 220,
LOAD_J_111 = 222,

LOAD_I_112 = 222,
LOAD_J_112 = 220
}
```

Program 8 Generation of C enumerations.

```
$counter = 0;
while($counter < 112)
{
  #base case:
  $i = $counter * 2;
  $j = $i - ($i % 4) * 2 + 2;
  print "LOAD_I_" . ($counter + 1) . " = " . $i . ",\n";
  print "LOAD_J_" . ($counter + 1) . " = " . $j . ",\n";
  $counter = $counter + 1;
}
```

4.3.6 Calculation Order Generation

Finding an optimal calculation order can be difficult, and only a few hand made sequences have been made. The implemented pattern generator can either be given a sequence, or create one from scratch. Designing both random, and various logical patterns have been implemented. When generating a logical pattern two helper functions are utilised. Both returns AB column indexes, for calculations that have not been performed yet. The first, $getNextMiss()$, locates an index inside a cache line that has not been accessed. Similar, $getNextNonMiss()$, finds one that is in a cache line that has been accessed before.

Arbitrary sequences of miss/hit ratios can then be constructed easily. An pattern with two groups of 1 miss and 2 hits, followed by a single group of 1 miss and 3 hits is created in Program 9. First the 3-3-4 pattern is used, while at the end all data is expected to be in L1 cache. The functions ensure that all values ends up included in the calculation, and will return either hits/misses to guarantee this.

Program 9 Generation of a miss/hit sequence.

```
my $counter2 = 0;
while($counter2 < 100)
{
  $sequence[$counter2] = getNextMiss();
  $sequence[$counter2 +1] = getNextNonMiss();
  $sequence[$counter2 +2] = getNextNonMiss();
  $sequence[$counter2 +3] = getNextMiss();
  $sequence[$counter2 +4] = getNextNonMiss();
  $sequence[$counter2 +5] = getNextNonMiss();
  $sequence[$counter2 +6] = getNextMiss();
  $sequence[$counter2 +7] = getNextNonMiss();
  $sequence[$counter2 +8] = getNextNonMiss();
  $sequence[$counter2 +9] = getNextNonMiss();
  $counter2 += 10;
}

while($counter2 < 112)
{
  $sequence[$counter2] = getNextNonMiss();
  $counter2++;
}
```

4.4 Compiler Test Case

Understanding how the compiler works is useful, because of its central role transforming the C code used into assembly code. Construction of a number of simple programs, that can be easily understood on a high and low-level is therefor beneficial. This ease the evaluation of how its internal logic performs loop unrolling, loop flow control and manage register allocation. These are central concepts to understand, from a purely practical point of view. Doing this might help deciding if an assembly language implementation can be beneficial.

To do this several simple test cases were made, and the assembly code generated could be analyzed. Only two of them are presented, for illustration. Program 10 makes an array of 1024 floats, and then loops over each, setting them to zero. This is an easy to understand high level program, and the code have a simple mapping to low-level instructions. The translation performed by the compiler logic should then be easy to evaluate. To prevent the compiler from optimizing away the code, one element must be used. By printing the first element, all the code is preserved.

Program 10 A simple C program: 1024 floats are allocated on the stack, then 0.0 is written all of them.

```
int main(int argc, const char* argv[])
{
    float data[1024];
    for(int i = 0; i<1024; ++i)
    {
        data[i] = 0.0f;
    }

    printf("%f\n", data[0]);
    return 0;
}
```

The second test case is Program 11. This is an basic implementation of $C = AB' + BA'$, implemented by Thorvald Natvig. Neither loop tiling nor symmetry is utilized. [21] evaluated ATLAS and claims that it do not utilize any techniques that are unknown to compiler authors. They also found that compiler authors have focused on those techniques for a long time. This makes this program an acceptable candidate to evaluate compiler effectiveness, comparing it with the performance achieved by more complex implementations.

Program 11 A simple implementation of dsyr2k.

```
#define A(i,j) a[i*k+j]
#define B(i,j) b[i*k+j]
#define C(i,j) c[i*n+j]
#define V(i,j) v[i*n+j]

void rank2k(double *c, const double *a, const double *b, const
    int n, const int k)
{
  int i, j, t;

  for (i = 0; i < n * n; i++)
  c[i] = 0.0;

  /* C := a * b' */

  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      for (t = 0; t < k; t++)
        C(i, j) += A(i, t) * B(j, t);


    /* ... + b * a' */

  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      for (t = 0; t < k; t++)
        C(i, j) += B(i, t) * A(j, t);
}
```

4.5 Method

In order to test our implementation and obtain benchmarks, a framework provided by Thorvald Natvig was used. The framework accepts 3 parameters, with the two first being the matrix dimensions N and K. The last parameter is the name of a dynamically loaded library, containing a single dsyr2k function that will be benchmarked. In this way it is very easy to change the implementation being analyzed.

Internally, the framework creates a set of matrices A, B, C, and C_ref. It then fills A and B with various values, and use a reference implementation[6] to calculate C_ref. After this is done, the function it is evaluating is used

[6] ATLAS or MKL, depending on build rules.

to calculate C. A high precision timer using the *rdtsc* instruction, is used to time the function and obtain the *cycle_count*. Then C_ref and C are compared, and any differences are printed out. The formula used by the framework to obtain the theoretical number is shown in Equation 4.1.

$$f_{total\_flop\_count}(N, K) = 4flop * K * N^2 \tag{4.1}$$

Finally, the performance is calculated using the Equation 4.2. Performance is reported in flop/cycle, with 4 decimals of precision. No assumptions are made regarding the calculation, so no work is considered needless. This means that the framework reports up to twice the performance of what the processor is capable of.

$$f_{flop/cycle}(N, K) = \frac{f_{total\_flop\_count}(N, K)}{cycle\_count} \tag{4.2}$$

In order to generate benchmarks, 25 runs of several implementations have been performed. This number of samples is sufficient to obtain results with a clear and stable tendency. Two metrics for central tendency have been used, arithmetic mean and median. As mean is influenced by possible single outliers, it is included only for completeness. In order to remove outliers, median provides more robust measurements. Thus, all comparisons use median as basis. To show variability between runs both standard deviation and absolute median deviation are included. As with mean, standard deviation is affected by outliers. The more robust absolute median deviation uses median as basis, and avoids outliers in its calculation. Maximum and minimum values are also included for completeness.

Speedup is used to compare the performance achieved by different implementations. Equation 4.3 shows how it is used in this context. Speedup relative to a theoretically perfect implementation is also shown.

$$S_{speedup} = \frac{t_{test\_implementation}}{t_{base\_implementation}} \tag{4.3}$$

4.5.1 Setup

All the benchmarks are performed on the compute nodes of Clustis2, described in Section 2.2.1. On this platform a maximum of 2.0 flop/cycle is possible using double precision. As half of the calculation can be skipped, 4.0 flop/cycle will be the maximum reported by the framework. The compiler used is GCC version 4.3.0.

Three external implementation are included to form a comparison basis.

Simple — This is the basic implementation that do all of the calculations, without any loop tiling. Its speed can be doubled easily, as it does not use symmetry. The code is written by Thorvald Natvig, and is only intended for illustration and comparison. Its C code can be found in Program 11.

ATLAS — The ATLAS version used is 3.8.2 (3.8.3 is the current), tuned to the compute nodes.

MKL — The exact MKL version used is not known, and only a single sample of MKL was performed. It was obtained long before the other benchmarks, and new tests are not possible anymore. This is because Clustis2 is retired, and no longer maintained.

The following runs use the designed base implementation, with the loading enumeration being different. Only the data loading sequence from L2 cache to L1 cache differ.

Sequence — All memory accesses done in a sequential pattern.

Bit Rev — Memory accesses are performed in a bit-reversed sequence, similar to the ones used in Fast Fourier Transforms. It is included as it gives a very common access pattern, being a natural optimization target for processor designers.

Miss First — Here, all the L1 cache misses are performed as fast as possible at the start. Similar to what is described as optimal in ATLAS [20], and illustrated in Figure 4.14 and 4.15.

Best12 — The '3-3-4' memory accesses pattern described in chapter 4.3.6 is employed. This pattern is used for the base case function, and one of the

special functions. The other two special functions use a similar pattern, that generates the misses somewhat faster.

Random — The access pattern in the base case function is randomly generated for every benchmark sample. All 3 special case functions use a fixed, but suboptimal pattern. New patterns are generated before compiling, and compiled into the program. This approach avoids any extra overhead during benchmarking, but require a recompile before every run.

Partial Random —Similar to *Random*. The access patterns in the base case function, and one of the special functions is generated by random. The same pattern is shared between both functions. The two other special functions use the same pattern as they have in Best12, having somewhat better performance.

Two matrix sizes have been used in testing, as this is sufficient to illustrate the performance obtained. The first test case is with N=2000 and K=2016, representing medium sized matrices. For the second case, large[7] matrices with dimensions N=4000 and K=4032 are used. Note that the K dimension is also selected to be divisible by 112, enabling the full performance potential of our implementation.

4.6 Results

Our results with the medium sized matrices N=2000 and K=2016 are shown in Figure 4.21. *Simple* performs poorly, indicating that the compiler is not utilizing known optimization techniques. Using a linear access pattern, *Sequence* is the lowest performing implementation. ATLAS is beaten by all other access patterns. Table 4.1 contains more detailed statistical information of the data.

[7]Limited by available memory.

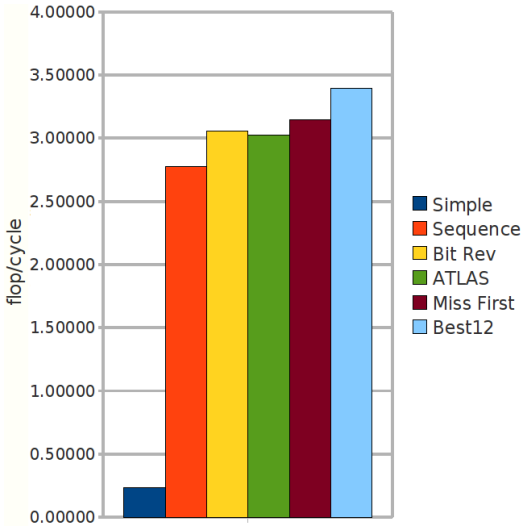


Figure 4.21: N=2000 and K=2016, median performance on Clustis2. Speed is in flop/cycle.

Table 4.1: 25 samples with N=2000 and K=2016, performed on Clustis2. Speed is in flop/cycle.

Data	Simple	Sequence	Bit Rev	ATLAS	Miss First	Best12
Average	0.2345	2.7761	3.0603	3.0297	3.1487	3.3973
Minimum	0.2344	2.7697	3.0536	3.0189	3.1344	3.3851
Maximum	0.2346	2.7791	3.0620	3.0345	3.1512	3.3995
Median	0.2345	2.7771	3.0608	3.0306	3.1498	3.3980
Std. Dev.	0.00006	0.00297	0.00209	0.00379	0.00349	0.00323
Median A. D.	0.00000	0.00080	0.00050	0.00220	0.00050	0.00060
% of Simple	100.0%	1184.3%	1305.2%	1292.4%	1343.2%	1449.0%
% of ATLAS	7.738%	91.635%	100.997%	100.000%	103.933%	112.123%
% of Max	5.863%	69.428%	76.520%	75.765%	78.745%	84.950%

The results with the large sized matrices are shown in Figure 4.22. *Simple* performs poorly again, showing no efficiency improvements in the larger case. Detailed statistics are included in Table 4.2.

Table 4.2: 25 samples N=4000 and K=4032, performed on Clustis2. Speed is in flop/cycle.

Data	Simple	Sequence	Bit Rev	ATLAS	Miss First	Best12	MKL
Average	0.2346	2.8299	3.1135	3.1387	3.2060	3.4772	-
Minimum	0.2344	2.8280	3.1115	3.1337	3.2042	3.4747	-
Maximum	0.2347	2.8318	3.1152	3.1418	3.2075	3.4791	-
Median	0.2346	2.8299	3.1138	3.1387	3.2061	3.4772	2.9608
Std. Dev.	0.00007	0.00098	0.00110	0.00205	0.00092	0.00121	-
Median A. D.	0.00000	0.00070	0.00090	0.00180	0.00080	0.00070	-
% of Simple	100.0%	1206.3%	1327.3%	1337.9%	1366.6%	1482.2%	1262.1%
% of ATLAS	7.474%	90.162%	99.207%	100.0%	102.147%	110.785%	94.332%
% of Max	5.865%	70.748%	77.845%	78.468%	80.153%	86.930%	74.020%

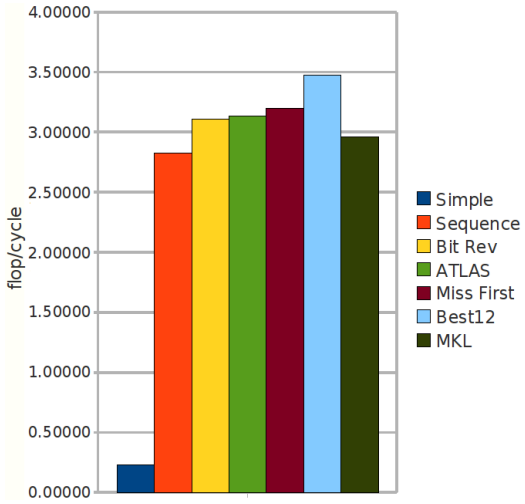


Figure 4.22: Median performance of N=4000 and K=4032, on Clustis2. Speed is in flop/cycle.

Both random based implementations require a larger number of runs, in order to obtain useful statistics. Only the larges matrix size of N=4000 and K=4032, have been tested. Table 4.3 show 1000 runs of *Random* and 500 of *Partial Random*. For comparison the performance of both ATLAS and Miss First are included. Figure 4.23 show the performance of every sample, sorted on performance. *This implies that using almost any form of reordering helps*,

even if the patterns are not selected in a logical way. Still, certain access patterns give low performance.

Table 4.3: A number of random pattern runs with N=4000 and K=4032, performed on Clustis2. Speed is in flop/cycle.

	Random	Partial Random	ATLAS	Miss First
Sample Count	1000	500	25	25
Average	3.2130	3.2624	3.1387	3.2060
Minimum	2.7779	2.9948	3.1337	3.2042
Maximum	3.3118	3.3584	3.1418	3.2075
Median	3.2158	3.2654	3.1387	3.2061
Std. Dev.	0.03532	0.03707	0.00205	0.00092
Median A. D.	0.02050	0.02315	0.00180	0.00080
Faster than ATLAS	977	496	-	25
Slower than ATLAS	23	4	-	0
Faster than Miss First	617	471	0	-
Slower than Miss First	383	29	25	-

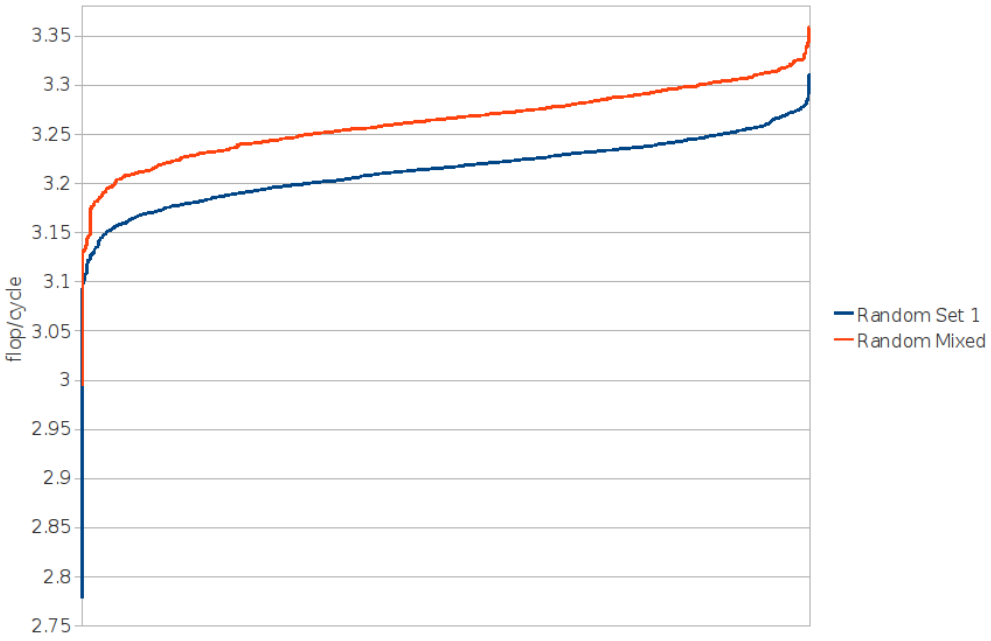


Figure 4.23: Performance of Random patterns, sorted on performance. N=4000 and K=4032, speed is in flop/cycle.

4.6.1 Compiler Test Case

To further analyze the compiler abilities, the compiler generated code of Program 10 is evaluated.

Figure 4.24 shows a graph of the compiler created assembly code. Every edge represents a unique piece of assembly code, with associated branch instructions. When the program is executed only a single path is possible, as the code is static. This information is also clearly available to the compiler, yet it fails to utilise this. A total of 60 code paths are created, where only 13 will ever be executed. The rest are partially generated to handle impossible cases, like if the loop does not start at 0, or end prematurely. Numerous other similar problems are also present. The red line shows the unique path taken during program execution.

In addition, various versions of GCC create vastly different code. Both GCC 4.3.0-1ubuntu1 (Ubuntu) and GCC 4.2.3 p1.0 (Gentoo) creates this code flow. However GCC 4.2.3-2ubuntu7 (Ubuntu) successfully detects the code structure, generating only a single branch used for the loop. Some more information on this issue is shown in Appendix D, further analysis is beyond the scope of this thesis.

Secondary Compilers Issues

While evaluating the compiler generated assembly code, other problems with the GCC code became apparent. GCC does not fully preserve the sequence in the access pattern through translation. Some of the independent operations were reordered between each code block. This introduced unintended and unneeded dependencies, that the OoOE system had to handle in order to avoid stalls. Moreover, prefetching instructions seemed to lack any dependencies with respect to the code they where in. All were grouped all together in a block at the end, dislocated from their positions specified in C code. This can potentially render software prefetching useless, or even counterproductive.

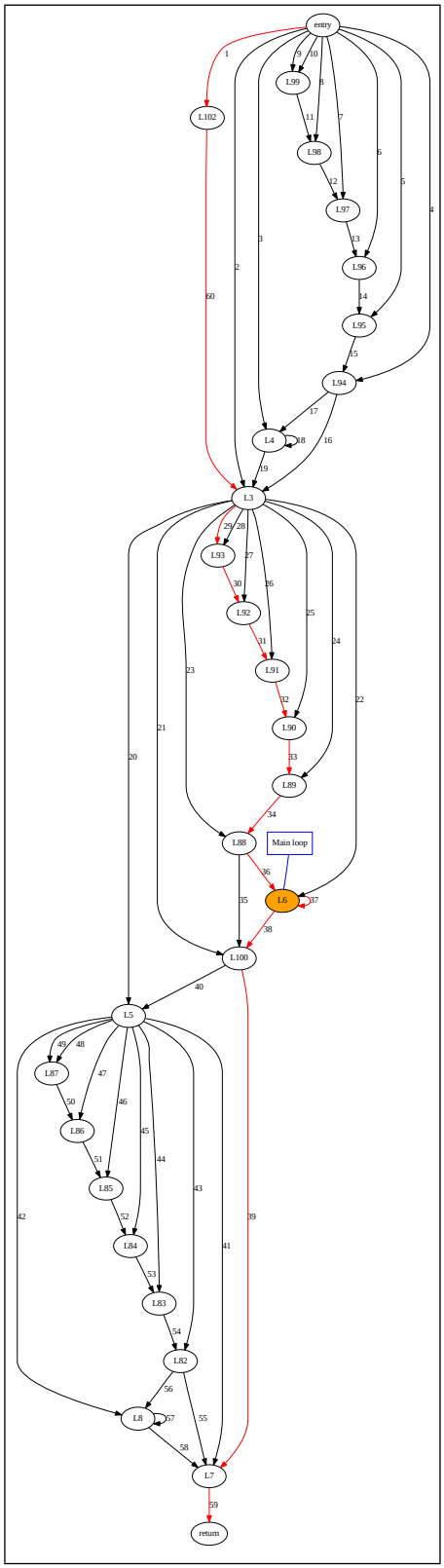


Figure 4.24: Flow graph of Program 10, to give an impression of the complex code structure generated by some versions of GCC. See also Appendix D

4.7 Evaluation of Pentium 4 Optimizations

We showwd that the sequence in which memory intensive calculations are performed, can greatly affect performance. This seems not to have been taken into account in an appropriate manner, at least not in ATLAS and MKL. These performance differences might not be well known within scientific literature, else both ATLAS and MKL would utilize them to achieve the highest performance. There exists numerous papers and articles, which attempts to improve both ATLAS and matrix multiply in general. From the performance achieved by the randomly generated patterns, it might seem as it is somewhat easy to beat ATLAS. This might result in many papers describing a performance improvement, caused by other factors than what is believed by their authors.

There are, however, several issues that make the benchmarks we obtained somewhat incomplete. Only a limited set of matrix sizes have been presented here, and they are not obtained using the most recent versions of ATLAS and MKL. This limits the speedup results obtained relative to those implementations. However, the analysis of how the kernel works, and the other issues found stands unaffected. Therefore further benchmarks would be of minor interest.

Our performance results were better than ATLAS and MKL, yet many of the issues we found were not taken into account in our implementation. The C-language implementation requires that the compiler respects layout, which is not the case. Still, enough of the access patterns survives to clearly illustrate substantial performance differences. Designing an optimal implementation will be possible, using handwritten assembly code. This is needed to avoid possible compiler problems.

4.8 Modeling Using Virtual Multi-Threading

In order to prepare for Chapter 5 of the thesis, a basic model encapsulating the found issues has been created to highlight some details. The simplified model presents a set of guidelines for how to design an implementation, that should be optimal. The main idea comes from the possibility of creating several threads, exploiting *simultaneous multi-threading* (SMT). A way to

perform this, even while the processor does not have this kind of support in hardware, is presented. The only requirement is that it supports OoOE, and that it has a lockup-free cache. Both are standard features on modern high performance processors. A slightly related implementation is also explored in [19], but the implementation used there require designing new processor instructions.

This article [19] implements 'fly-weight' threads that is used to prefetch data into cache. The threads are less fine grained, so that one cannot utilize the floating point pipelines while running the helper thread. The helper threads are spawned dynamically either on a last level cache miss, or before a load that might generate a miss. It does this by creating two new instructions in the CPU, and as such needs a special processor with programmable debugging hardware (or a native implementation). Therefore, this present a mayor obstacle for normal use.

First an evaluation of using known techniques, implemented in hardware is presented. Later a more efficient approach is presented.

Basic Model

First we evaluate an implementation using two threads on a single core. One thread is designed to perform all the math (the math kernel). While the other thread is pre-fetching data into L1 cache. This will ensure that the math thread is never slowed by L1 data misses, if one can keep the two threads synchronized. In practice the threads will either need some communication and control that will slow down both, or they will quickly lose synchronization. If the pre-fetch thread uses only specialized prefetch instructions, the processor can ignore them if the memory/cache system is out of resources. This will be the case after only a few instructions if the prefetch thread runs at full speed (or 50% with hyper-threading). In order to ensure that all data is prefetched one can use normal loads, as the processor cannot ignore them. This will make the prefetch thread consume cache Miss Status Holding Registers (MSHR). If all MSHR's are consumed, the math thread cannot access any data from L1 cache, even if the data is there. Both implementations will, however, steal resources, reducing the number of instructions pr. second that can be performed by the math thread.

The prefetch thread must not slow the execution speed of the math thread, i.e.:

- The prefetch thread must run with a guaranteed speed relative to the math kernel thread.
- The prefetch thread must not block the cache.
- The prefetch thread must issue so few instructions, that the math kernel thread can work without slowdown.

Efficient Implementation

The following approach is used in order to circumvent the limitations described above, having two normal threads running in sync, and SMT hardware support.

By creating a number of imaginary threads, inside a single thread running on one core, one can schedule cache misses and hits to different threads. A thread is created by isolating calculations with data dependencies, allowing the OoOE engine to treat them separately. Creating the multi-threading is done explicitly inside the math kernel, with both prefetch and math threads interleaved in the same code. By using the OoOE capabilities to simulate virtual threads, we can thus introduce a type of simulated SMT. This gives full control over what every virtual thread is preforming at all times. Thus, a virtual thread does not have any data-dependencies between other virtual threads, so a stalled thread will not affect other threads.

Using load instructions as prefetch instructions, they can also act as the loads used by the math kernel. This means that prefetching does not require extra instructions. A virtual thread will, however, stall as it generates an L1 miss. The other virtual threads will not stall, and are free to perform both data loading (from L1) and math operations. From the design of the data access pattern, it is possible to decide when each thread stalls. These stalls always come from reading data that is known to be in L2 cache, or the slower main memory. This makes it possible to make a set of virtual threads, that take turns on making L1 misses. At the same time, the non-blocked threads perform work on data in L1 cache. Note that all the threads can (and will) work on any data in L1 cache, irrespective of which thread was used to fetch the cache line to L1. This is possible as data known to be in L1 will not introduce any dependencies between threads, and the order of the matrix multiplication calculations does not affect the result.

Chapter 5

Low-Level Optimization on the Intel® Core 2

This chapter will look at low-level optimization techniques, using the Intel® Core 2 processor as a test case. First, the hardware limitations and possibilities will be presented, with theoretical considerations on how they affect performance. How our rank2k implementation can be extended from the reduced case to the full case, is also explained. A number of auxiliary tools used for testing, debugging and parallellization are then presented. Details of our implementation are then presented and discussed. The method used when obtaining benchmarks, and various problems encountered are pointed out before the benchmarks are presented. Finally, the implementation and the problems discovered are evaluated and discussed.

5.1 Theoretical issues

The model presented at the end of the previous chapter , using low-level assembly code, requires a matching low-level analysis. First, instruction selection and how it affects the Core 2 processor is presented. Issues regarding the L1 cache, loop tiling and block size selection are then evaluated.

5.1.1 Low-Level Instruction Selection

Looking at the smallest operation performed, one calculation step of $C_{i,j} += AB_{i,k} * AB_{j,k}$, one can create low-level comparisons for evaluating code efficiency. At this level, each *iteration step* ($k = k + 2$) in the innermost unrolled loop will behave identically. Because of the L1 cache loop tilling performed, several separate $C_{i,j}$ are calculated at the same time for each iteration step. One iteration step with several $C_{i,j}$ updates are later called a *code block*. These code blocks are used for comparing various designs.

First, the theoretical limit of the processor will be presented, in order to create a theoretical basis for the maximum possible performance. Then several designs will be presented. With the Core 2 processor, the execution throughput may be limited by several factors. For SSE code, only *3* instructions can be decoded every cycle. To enable full execution speed, one of the instructions must be an addition (*addpd*) and one must be a multiplication (*mulpd*). This leaves only a *single* instruction every cycle for loading data from memory, or moving data between registers. Also, the x86 instruction set uses only two-operand instructions, so one of the values taking part in addition or multiplication is lost. To avoid losing values in calculations like this, a copy must be made before the value is overwritten.

In practice, when updating/calculating the value of $C_{i,j}$ for each calculation step, the following set of resource limitations is present:

- max 16 simultaneous SSE registers used
- max 16 bytes of instructions every cycle
- max 1 memory load every cycle
- max 3 SSE instructions every cycle, with the following requirements:
  - max 1 addition (*addpd*) every cycle
  - max 1 multiplication (*mulpd*) every cycle
  - max 1 load/copy (*moveapd*) every cycle

Since the Core 2 processor is capable of OoOE, these limitations must be maintained over several cycles only, and not every single cycle. When calculating $C_{i,j} += AB_{i,k} * AB_{j,k}$ we find that one multiplication and one addition is unavoidably needed. For the Core 2 an instruction layout design capable of performing *one* SSE multiplication and *one* SSE addition every cycle is the optimum.

Pentium 4 Instruction Selection

Evaluating the unrolling in the Pentium 4 base case function (Section 4.3.3, page. 60), that calculates 1 column by 2 columns at the same time, this might (ideally) be translated by the compiler into the instruction sequence (code block) shown in Table 5.1. The notation style used to define instructions are taken from [8], however, register *names* are selected based on their content. On the borders of Table 5.1 the column name and index from AB is indicated. In the *intersection* of two AB columns, the instructions performed in order to calculate the iteration step $C_{i,j} += AB_{i,k} * AB_{j,k}$ is shown. Instructions located on the top (and left) *borders* of the table are memory loads that are performed once, but the data is required in the entire row/column.

The instruction execution sequence begins at the top of the table, and iterates down sequentially. First, a single memory load from AB column $J+1$ occurs ($R_{j1} \leftarrow \text{Mem}[AB_{j1}]$), this value is then used in two calculations. At the intersection of AB column $I+1$ and AB column $J+1$ three instructions are located. The first is a register–register copy, to make a temporary copy (R_{tmp}) of R_{j1}. Then the second instruction performs the multiplication, loading AB column $I+1$ directly from memory as one operand, with the other operand R_{tmp} overwritten. The third instruction adds the answer of the multiplication (R_{tmp}) into a accumulator register $R_{acc1,1}$. The second calculation is performed in the intersection of AB column $I+2$ and AB column $J+1$. In this case no temporary copy of R_{j1} is needed, as its content will not be needed anymore.

Table 5.1: Pentium 4 instruction layout.

		AB Column J
		1
		$R_{j1} \leftarrow \text{Mem}[AB_{j1}]$
AB Column I	1	$R_{tmp} \leftarrow R_{j1}$ $R_{tmp} \leftarrow R_{tmp} * \text{Mem}[AB_{i1}]$ $R_{acc1,1} \leftarrow R_{acc1,1} + R_{tmp}$
	2	$R_{j1} \leftarrow R_{j1} * \text{Mem}[AB_{i2}]$ $R_{acc2,1} \leftarrow R_{acc2,1} + R_{j1}$

From Table 5.1 we find that 6 instructions are used in this code block, performing two additions and two multiplications. Those instructions

satisfies the requirements presented for the Core 2, performing two calculation step of $C_{i,j}$ in *two* cycles. However, a total of *three* memory loads are performed, reducing the throughput to $\frac{2}{3} = 66.67\%$ of theoretical maximum.

Extended Pentium 4 Instruction Selection

In order to improve throughput efficiency, more extensive unrolling can be attempted. By extended the previous Pentium 4 design into one that calculates $1 * 8$ columns at the same time, the instruction layout shown in Table 5.2 results. We find that $1 + 3 * 7 + 2 = 24$ instructions are used in this code block, performing 8 additions and 8 multiplications. 8 cycles are needed to execute the calculations, while requiring a total of 9 memory loads. Thus, efficiency becomes $\frac{8}{9} = 88.89\%$ of theoretical maximum.

Table 5.2: Extended Pentium 4 instruction layout.

		AB Column J
		1
		$R_{j1} \leftarrow \text{Mem}[AB_{j1}]$
AB Column I	1	$R_{tmp} \leftarrow R_{j1}$ $R_{tmp} \leftarrow R_{tmp} * \text{Mem}[AB_{i1}]$ $R_{acc1,1} \leftarrow R_{acc1,1} + R_{tmp}$
	2	$R_{tmp} \leftarrow R_{j1}$ $R_{tmp} \leftarrow R_{tmp} * \text{Mem}[AB_{i2}]$ $R_{acc2,1} \leftarrow R_{acc2,1} + R_{tmp}$
	3	$R_{tmp} \leftarrow R_{j1}$ $R_{tmp} \leftarrow R_{tmp} * \text{Mem}[AB_{i3}]$ $R_{acc3,1} \leftarrow R_{acc3,1} + R_{tmp}$
	4	$R_{tmp} \leftarrow R_{j1}$ $R_{tmp} \leftarrow R_{tmp} * \text{Mem}[AB_{i4}]$ $R_{acc4,1} \leftarrow R_{acc4,1} + R_{tmp}$
	5	$R_{tmp} \leftarrow R_{j1}$ $R_{tmp} \leftarrow R_{tmp} * \text{Mem}[AB_{i5}]$ $R_{acc5,1} \leftarrow R_{acc5,1} + R_{tmp}$
	6	$R_{tmp} \leftarrow R_{j1}$ $R_{tmp} \leftarrow R_{tmp} * \text{Mem}[AB_{i6}]$ $R_{acc6,1} \leftarrow R_{acc6,1} + R_{tmp}$
	7	$R_{tmp} \leftarrow R_{j1}$ $R_{tmp} \leftarrow R_{tmp} * \text{Mem}[AB_{i7}]$ $R_{acc7,1} \leftarrow R_{acc7,1} + R_{tmp}$
	8	$R_{j1} \leftarrow R_{j1} * \text{Mem}[AB_{i8}]$ $R_{acc8,1} \leftarrow R_{acc8,1} + R_{j1}$

The design in Table 5.2 requires 8 accumulation registers, one register for AB Column J (R_{j1}) and one for the temporary register (R_{tmp}). Only 10 registers of the 16 available are used, so more unrolling is possible. Extending that layout, using 14 registers as accumulator registers, efficiency can be increased to $\frac{14}{15} = 93.33\%$ of theoretical maximum.

Register Level Loop Tiling

In order to remove the memory load limitation in the earlier designs, register level loop tiling can be utilised. This works in the same way as when loop

tiling is used to reduce main memory bandwidth, by utilizing the cache. On this level of loop tiling, single registers take the place of the cache. Table 5.3 show how this can be performed. By calculating block of 2 by 2 columns at the same time, a total of 4 memory load operations are required. At the same time 4 additions and 4 multiplications are performed, taking 4 cycles — the same number as memory loads. Execution begins at the top left, iterating left to right for every row. Unfortunately, 13 instructions are required while only 12 can be performed in 4 cycles. Throughput is thus limited by a single register copy instruction, reducing the efficiency to $\frac{12}{13} = 92.31\%$.

Table 5.3: Instruction layout for a 2 by 2 column loop tiling.

			AB Column J	
			1	2
			$R_{j1} \leftarrow \text{Mem}[AB_{j1}]$	$R_{j2} \leftarrow \text{Mem}[AB_{j2}]$
AB Column I	1	$R_{i1} \leftarrow \text{Mem}[AB_{i1}]$	$R_{tmp} \leftarrow R_{i1}$ $R_{tmp} \leftarrow R_{tmp} * R_{j1}$ $R_{acc1,1} \leftarrow R_{acc1,1} + R_{tmp}$	$R_{i1} \leftarrow R_{i1} * R_{j2}$ $R_{acc1,2} \leftarrow R_{acc1,2} + R_{i1}$
	2	$R_{i2} \leftarrow \text{Mem}[AB_{i2}]$	$R_{j1} \leftarrow R_{j1} * R_{i2}$ $R_{acc2,1} \leftarrow R_{acc2,1} + R_{j1}$	$R_{i2} \leftarrow R_{i2} * R_{j2}$ $R_{acc2,2} \leftarrow R_{acc2,2} + R_{i2}$

Note that the single register copy instruction in Table 5.3 ($R_{tmp} \leftarrow R_{i1}$) is located on the intersection (I=1, J=1), while no copy instruction are required at the borders. This is because of the destructive properties of the two operand format. Simply enlarging the code block will require 3 extra instructions at every additional non-border intersection, leading to an efficiency of the form $\frac{x}{x+1}$, which cannot reach 100%.

Final Core 2 Instruction Layout

In order to design the best pattern possible for the Core 2 processor, numerous instruction layouts were made. While the consideration going into those patterns were important, most are extremely low-level, and thus considered beyond the main scope of the thesis. Only the final instruction layout is included, although other layouts have promising qualities. Table 5.4

shows what this final pattern looks like. For this pattern, a total of 7 memory loads are performed, but only 6 are strictly needed. The extra memory load is performed on AB column $I+4$, reading the same data from memory twice. Doing this reduces the number of register–register copy instructions. A total of 8 additions and 8 multiplications are performed, using 8 cycles. Counting the number of instructions reveal that 24 are needed, matching the 24 possible in 8 cycles. Looking at the register usage we find that 8 accumulator registers needed, 5 for data loaded from memory and one temporary register, giving a total usage of 14 registers. From the presented considerations an efficiency of $\frac{8}{8} = 100\%$ is possible.

Table 5.4: Final Core 2 instruction layout.

			AB Column J	
			1	2
			$R_{j1} \leftarrow \text{Mem}[AB_{j1}]$	$R_{j2} \leftarrow \text{Mem}[AB_{j2}]$
AB Column I	1	$R_{i1} \leftarrow \text{Mem}[AB_{i1}]$	$R_{tmp} \leftarrow R_{i1}$ $R_{tmp} \leftarrow R_{tmp} * R_{j1}$ $R_{acc1,1} \leftarrow R_{acc1,1} + R_{tmp}$	$R_{i1} \leftarrow R_{i1} * R_{j2}$ $R_{acc1,2} \leftarrow R_{acc1,2} + R_{i1}$
	2	$R_{i2} \leftarrow \text{Mem}[AB_{i2}]$	$R_{tmp} \leftarrow R_{i2}$ $R_{tmp} \leftarrow R_{tmp} * R_{j1}$ $R_{acc2,1} \leftarrow R_{acc2,1} + R_{tmp}$	$R_{i2} \leftarrow R_{i2} * R_{j2}$ $R_{acc2,2} \leftarrow R_{acc2,2} + R_{i2}$
	3	$R_{i3} \leftarrow \text{Mem}[AB_{i3}]$	$R_{tmp} \leftarrow R_{i3}$ $R_{tmp} \leftarrow R_{tmp} * R_{j1}$ $R_{acc3,1} \leftarrow R_{acc3,1} + R_{tmp}$	$R_{i3} \leftarrow R_{i3} * R_{j2}$ $R_{acc3,2} \leftarrow R_{acc3,2} + R_{i3}$
	4		$R_{j1} \leftarrow R_{j1} * \text{Mem}[AB_{i4}]$ $R_{acc4,1} \leftarrow R_{acc4,1} + R_{j1}$	$R_{j2} \leftarrow R_{j2} * \text{Mem}[AB_{i4}]$ $R_{acc4,2} \leftarrow R_{acc4,2} + R_{j2}$

When looking at the very lowest level of the instruction selection, the limitation of 16 bytes/cycle becomes a problem as well. The instructions in the code block is replicated numerous times in the actual implementation as a consequence of unrolling, necessitating large or complex memory instructions. Since these instructions can be up to 10 bytes each, the decoding bandwidth of 16 bytes/cycle can potentially lead to a new bottleneck. Because instructions change size based on the complex rules presented in Section 3.3 (p. 22), careful construction is required. With those instructions size rules taken into account, the byte size of the code block has been reduced to 128 bytes — the same number of bytes it is possible to decode in 8 cycles by the Core 2 processor. Thus, every hard requirement needed to achieve maximum throughput has been satisfied. Unfortunately, the Core 2 OoOE engine is

not perfect, necessitating more tuning. To further improve performance, the number of simultaneous registers required have been lowered to 12, while maintaining a reasonable balance of every processor resource used at all times inside the entire code block. To reduce the cost associated with the latency of (occasional) L1 cache misses, the load instructions are partially relocated between code blocks. This relocation moves them before their code block, increasing the probability that their data is available when required by calculations. For reference, a sample of the disassembled code block can be found in Figure 5.1, generated by objdump.

Still, the instruction pattern is not perfect for the OoOE engine, and performance is limited by additional internal issues. Both the performance limitation and this part of the code block design process is considered beyond the main scope of this thesis, and is hence left out as future work.

5.1.2 Cache Evaluation

For the Core 2 processor with 32 KB L1 cache, more extensive loop tiling than on the Pentium 4 is possible. Designing a core function that both utilise the L1 cache and incorporates the final Core 2 code block pattern is thus required.

From the low-level code block pattern found effective for the Core 2 processor, the fundamental L1 cache loop tiling block becomes 2 J columns by 4 AB I columns. In the Pentium 4 design, the AB columns contained 224 doubles, 112 from each of the A and B matrices. This means that each AB column requires $224 * 8B = 1792B = 1.75KB$ (note that one double is 8 byte). By using loop tiling to make a 2 by 4 AB column block, a total of two J columns and 4 I columns are needed, requiring $1.75KB * (2 + 4) = 10.5KB$. Because the $10.5KB$ is well within the capacity of the L1 cache, a larger loop tiling block can be constructed from the minimal 2 by 4 block.

By using 2 by 4 blocks to construct a larger 2 by 12 block, the cache cost becomes $12 + 2$ columns, or $(12 + 2) * 1.75KB = 24.5KB$. When one 2 by 12 block is complete, the most cache effective approach is to replace the two J columns with the next two, reusing 12 I columns. While $24.5KB$ is the cache needed to store the data at the same time, more data is stored in the cache, however, as data is replaced by LRU rules. This brings the requirement up to 12+2 columns with 2 extra columns that contain the previous two J columns,

```
4025f5:       66 0f 28 90 a0 0b 00    movapd 0xba0(%rax),%xmm2
4025fc:       00
4025fd:       66 0f 59 cd             mulpd  %xmm5,%xmm1
402601:       66 44 0f 58 c1          addpd  %xmm1,%xmm8
402606:       66 0f 28 ca             movapd %xmm2,%xmm1
40260a:       66 0f 59 d0             mulpd  %xmm0,%xmm2
40260e:       66 44 0f 58 d2          addpd  %xmm2,%xmm10
402613:       66 0f 59 80 a0 0f 00    mulpd  0xfa0(%rax),%xmm0
40261a:       00
40261b:       66 0f 58 e0             addpd  %xmm0,%xmm4
40261f:       66 0f 28 86 40 03 00    movapd 0x340(%rsi),%xmm0
402626:       00
402627:       66 0f 59 cd             mulpd  %xmm5,%xmm1
40262b:       66 0f 58 f9             addpd  %xmm1,%xmm7
40262f:       66 0f 59 a8 a0 0f 00    mulpd  0xfa0(%rax),%xmm5
402636:       00
402637:       66 44 0f 58 cd          addpd  %xmm5,%xmm9
40263c:       66 0f 28 90 d0 03 00    movapd 0x3d0(%rax),%xmm2
402643:       00
402644:       66 0f 28 ca             movapd %xmm2,%xmm1
402648:       66 0f 59 d0             mulpd  %xmm0,%xmm2
40264c:       66 0f 58 da             addpd  %xmm2,%xmm3
402650:       66 0f 28 ae 40 07 00    movapd 0x740(%rsi),%xmm5
402657:       00
402658:       66 0f 59 cd             mulpd  %xmm5,%xmm1
40265c:       66 0f 58 f1             addpd  %xmm1,%xmm6
402660:       66 0f 28 90 d0 07 00    movapd 0x7d0(%rax),%xmm2
402667:       00
402668:       66 0f 28 ca             movapd %xmm2,%xmm1
40266c:       66 0f 59 d0             mulpd  %xmm0,%xmm2
402670:       66 44 0f 58 da          addpd  %xmm2,%xmm11
402675:       66 0f 28 90 d0 0b 00    movapd 0xbd0(%rax),%xmm2
40267c:       00
```

Figure 5.1: Final Core 2 instruction assembly code block, showing the 24 instructions in the correct sequence.

giving a total cost of $(12 + 4) * 1.75KB = 28KB$ for data alone. Note that $28KB$ is the maximum usable data size (at this level of analysis), as some extra data is required by the C matrix and from stack usage in the support code. Note that in order to make the base case function more effective, multiple 2 by 4 blocks calculated after each other, calculating *square blocks* (of 12 by 12 columns) when called. The calculation order in the base case function reuse cache as described above.

When taking into account the set associatively (described in Section 3.2.1) of the cache, an extra set of rules must be considered. The data layout ensures that the 12 I columns are sequential in memory using $21KB$, or 5 full *ways* and 1/4 of another *way*. The in the other direction 4 J columns (2 in use, 2 old) are are sequential taking $7KB$, 1 full *way* and 3/4 of another *way*. This gives a total cost of 6 full *ways* and two partial *ways*. The possibility of the two sequential data blocks overlapping, causing conflict misses, must be avoided.

When two 2 by 12 blocks are calculated after each other, its possible that the 4 J columns and 12 I columns do not overlap, as shown in Table 5.5.

Table 5.5: 4 J columns and 12 I columns without overlap.

Way	*AB* Column data							
1	i	i	i	i	i	i	i	i
2	i	i	i	i	i	i	i	i
3	i	i	i	i	i	i	i	i
4	i	i	i	i	i	i	i	i
5	i	i	i	i	i	i	i	i
6	i	i	j	j	j	j	j	j
7	j	x	x	x	x	x	x	x
8	-	-	-	-	-	-	-	-

Here the size of each cache line is scaled down as much as possible, without introducing inaccuracies. A single I/J column occupy 7 scaled cache lines. The cache *way* 1-7 contain both 12 I columns, 2 current J columns, and the two last J columns. Other data can then be located in the last (8^{th}) *way*. However, a different layout is also possible where overlapping occurs.

In Table 5.6, the 5 and one quarter *ways* of I columns line up with the 1 and three quarter *ways* needed by J columns. In this case a section of all 8 *ways* is used, leaving no space for data belonging to the C matrix (and

Table 5.6: 4 J columns and 12 I columns with overlap.

Way	AB Column data							
1	i	i	i	i	i	i	i	i
2	i	i	i	i	i	i	i	i
3	i	i	i	i	i	i	i	i
4	i	i	i	i	i	i	i	i
5	i	i	i	i	i	i	i	i
6	i	i	j	j	j	j	j	x
7	j	j	x	x	x	x	-	-
8	x	x	-	-	-	-	-	-

other potential data). Both cases also assumes that the cache have a perfect LRU policy, while the available documentation lack that information (the Pentium 4 used a 'pseudo LRU' policy, however).

This partial analysis show that the complexity involved in controlling cache usage is too high for easy manual evaluation, and thus too problematic to handle without some more refined tool. In order to solve this problem a simple cache simulator was needed, in order to test and possibly verify the cache layout problem(s). This cache simulator is briefly explained in Appendix B. By using this cache simulator tracking the cache lines containing J and I columns was possible, and identifying which were thrown out too early. Also, the replacement reason and simple statistics can be gathered by the simulator. Unfortunately, the cache simulator is not integrated into code generator in the current implementation, and is not actively used. Thus, it is considered outside the scope of this thesis. Based on the presented analysis, it is necessary that both the length of the AB columns, and the number of 2 by 4 blocks to calculate in the base case function must be *dynamically tunable*.

5.1.3 L1 Cache and Prefetching

An important feature in the Core 2 processor, unlike the Pentium 4, is the complex prefetchers working on the L1 cache. When data can be prefetched into L1 cache from the L2 cache, this will affect the model presented at the end of Chapter 4. As a consequence, the access pattern might be more efficient if these prefetchers are exploited as well. The IP-prefetcher working single instructions, based on the memory address of the instruction, might

be beneficial to utilize — yet excessively hard predict, requiring detailed knowledge of both processor state and global overview of all the memory load instructions in the core function. This analysis have not been performed in the current implementation. The more manageable streaming prefetchers require sequential accesses, and thus can be assumed to work best with very simple accesses patterns.

5.2 Implementation Details

First, because the Core 2 base case function calculates relatively small blocks, all the special cases functions designed for the Pentium 4 is not used. As such, the base case function is the only math kernel function, and it will be notated as the *core function* (CF) later on.

The same restrictions from the implementation on the Pentium 4 (Section 4.2, p. 55) is also imposed on the Core 2 implementation. However, here the fully unrolled loops in the K direction have different sizes. In order to achieve maximum performance, the K dimension must be divisible by length of unrolling, and this size can be changed at compile time. In the Pentium 4 implementation, different K direction loops unrolling designs has to be hand coded, while here the code can be auto generated. This enables the design of multiple unroll versions, so that an optimal set of core functions can be chosen at run time, matching the dimensions of the given matrices. Implementing this dynamic core function selection is considered beyond the scope of the thesis.

5.2.1 Extending to Full Rank2k (dsyr2k)

In order to extend the implementation from the $C_{buf} = AB' + BA'$ to the full dsyr2k version ($C = xAB' + xBA' + yC$), at least two design routes are possible. An stepwise approach is presented, evaluating the different sections of the dsyr2k functionality, before all are combined.

Extending To $C = xAB' + xBA'$

In order to include the scalar x in the computation two solutions are possible. First, by calculating $C = AB' + BA'$, then performing $C = xC$ at the end. This solution will adds a $O(n^2)$ step, however, it can be masked because of the buffering employed on the C matrix. This step then becomes $C_{ans} = xC_{buf}$, since this code part is bandwidth limited, the extra calculation should be masked completely. The other alternative approach is to factor in the x scalar in the core function, performing $C_{buf} = xAB' + xBA'$ directly. Having the x scalar inside the core function requires one extra SSE register, and 1 extra multiplication for each $C_{i,j}$ calculated. From before, when one $R_{accj,i}$ has been calculated in the core function, an extra addition is required for adding the accumulator into $C_{i,j}$. During the cycle spent on adding the accumulator into $C_{i,j}$, the Core 2 processors multiplication execution unit is idle. Thus, the extra cost of the multiplication can be masked.

Extending To $C = AB' + BA' + yC$

Including the scalar y and maintaining the old data of C (C_{org}) can be merged. By using the buffer C_{buf} for the $AB' + BA'$ answer calculation, implies that the old C data is still available. This enables the calculation step $C_{ans} = C_{buf} + yC_{org}$, adding some more computations to the existing $O(n^2)$ copy step.

Extending To $C = xAB' + xBA' + yC$

The full case can be performed in essentially two ways. Using the C buffer one can simply combine the techniques described, with low extra cost. The copy step then becomes $C_{ans} = xC_{buf} + yC_{org}$. This technique requires the use of the buffer C_{buf}, increasing memory usage. In order to avoid using this buffer, one must perform the calculation in-place. While this is critical where memory usage is high it will also increase the execution time. An mixed approach where both features are combined is also possible. This can be implemented in three steps. First, rearranging the C matrix into the same pattern used natively by the inner kernel. Second, normal execution of the core function. And finally, rearranging the C matrix back to its original pattern. With this strategy no extra space is needed. A basic implementation of this mixed approach has been created in a separate code base, as a proof of concept. This code base is intended for debugging and correctness control, and is not migrated into our implementation.

5.3 Tools and Their Motivation

A number of tools have been used and created, in order to evaluate, bug test, ensure correctness and optimize our implementation. This has been done to handle, and perform, practical management/control of the complex interaction in the code. There are three sets of tools that will be presented. First, the internal support code used will be briefly explained. Then the core of our implementation is presented. Last, parallelization issues and *Open Multi-Processing* (OpenMP) will be mentioned.

5.3.1 Core Tester — Benchmarker

The Core Tester program is used to benchmark candidate core function, without the overhead of the support code or framework. Only a minimal code base surrounds the core function, enabling quick performance feedback. Zero data (0.0 doubles) is generated for the data block substituting the AB matrix and the C buffer. In order to avoid start-up issues, the kernel is run once before timing, so that the instruction caches are primed before measurement. A basic form of loop tilling is also used on the (simulated) AB matrix, so that the effects of both L1 cache, L2 cache and main memory can be evaluated. In order to isolate each memory related effect, the size of the data block being iterated over can be controlled by a command line parameter. For instance, this enables benchmarking of the core function with all data permanently located in L1 cache. The number of times the core function is called depends on the K unrolling, and how big blocks are used in the loop tilling, ensuring the same number of flop's are performed. By altering the number of times the core function in this fashion, the execution time can be kept fairly constant. The same measurement method used by the framework in Section 4.5 is employed, easing later comparisons with the results from the full test framework.

5.4 Prefetching Access Pattern

In order to effectively exploit and use the high speed core function (Section 5.5.2), all data and computational layout has been changed to match it. The

same design as employed in the Pentium 4 implementation is used, with one major exception. For the large L2 cache in the Core 2 processor, a better loop tiling design was needed. The overall data access pattern must ensure that no bottlenecks are introduced, either from accessing memory in a non cache friendly way, or from not prefetching data into L2 cache before it is used in the core function.

The designed top level AB access pattern enable a large amount of data reuse, in order to effectively utilize caches. Without any special sequence the performance drops once the problem becomes to large, as the bandwidth requirement will too high for the main memory. Even with a normal loop tiling, the replacement of each block causes a short bandwidth spike, lowering performance. Multiple variations of L2 access pattern designs has been explored and benchmarked, even while only one will be presented.

The final L2 loop tilling access pattern is optimal from a theoretical/practical point of view. This pattern use a similar technique as in Section 4.1.2, having two blocks with different access speeds. Essentially, the pattern it will always access data slowly (requiring low bandwidth) before it uses fast accesses on the same data (high bandwidth), this allows efficient reuse of data in cache while keeping the bandwidth requirements low. Put differently, the high and slow access speed blocks alternated on using the same data, with the slow access speed blocks acting as a prefetcher for the high access speed blocks. Since the calculation is of a diagonal matrix, the iteration sequence in the loop tiling can be shown in a triangle.

An example of this iteration sequence is shown in Figure 5.2, on a $17x17$ matrix. Here, the loop tiling iterates on 5 by 5 blocks. The first 5 by 5 block is inside the top left green box, performing calculations on column 1-5 of AB. Then 5 columns are skipped, and the calculations begins in the top blue box, before it iterates over the top red box.

5.5 Assembly Code Design: Core Code Generator

In order to avoids the problems introduced by the compiler, a direct assembly code generator has been implemented. This code generator created the entire core function, including all internal register usage and setup, using x86

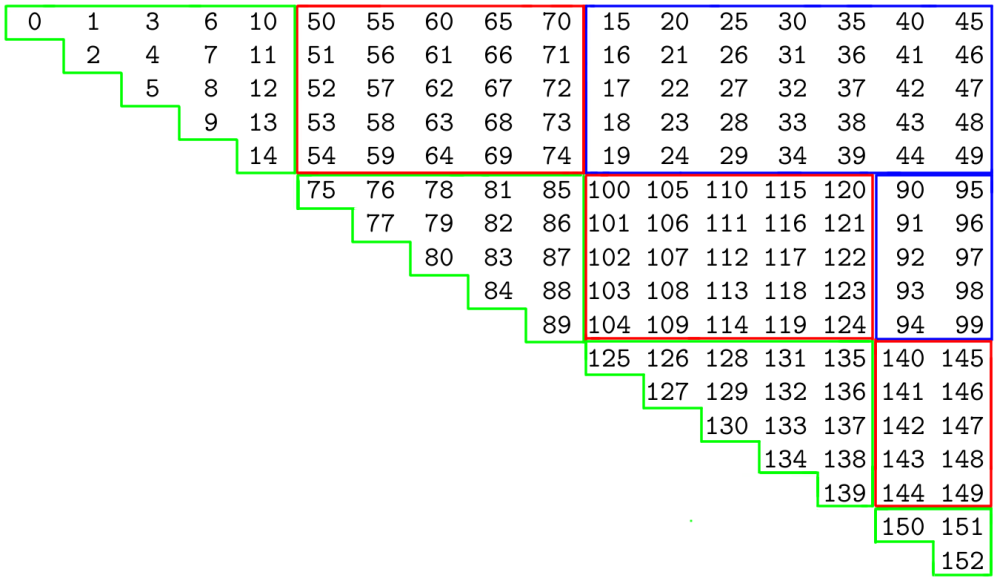


Figure 5.2: The final L2 loop tilling access pattern.

instructions. The design of the core function, incorporating all the properties mentioned so far will be presented. First the overall implementation choices and general overview of the assembly generator program, later called the *Core Code Generator* (CCG), is described. Then the hand tuning of the instruction sequence is briefly presented. The direct usage, and how its output is merged into C code is explained. Followed by a somewhat detailed description of its inner working. Some of the alternative code generation features will conclude the CCG explanation.

5.5.1 Core Code Generator

The implementation chosen is a Perl script, that generates assembly directly. Taking a *phased* approach to generating the code, the Core Code Generator is feed with two types of information on how to generate the assembly. The first type are the *externally visible* parameter choices, such as L1 block tile size and the K_Length. These parameters are chosen at a global level since the C setup/support code use them, and are compiled with them as fixed sizes. The other type of parameters are only *internally* visible, relative to the setup/support parts of the code. This include things like instruction choice, local memory access sequence, address format, register naming and code size control.

5.5.2 Core Code Generator Tuning

The generated instruction pattern structure is quite optimal, but *hard-coded* so that auto tuning of some parts of the instruction layout is not currently possible. While its likely that performance can be improved by changing the pattern structure, making it so that it can be a parameter will require major design changes and extensions, and is considered beyond the scope of this thesis. In order to tune the assembly core function a simple approach was used. By moving instructions and/or changing the Perl script, by subsequently benchmarking the new code better designs was evolved. The tuning was performed using the Core Tester. First, tuning was performed using the approach of ignoring all cache effects, using only tiny blocks that fit L1 cache. Also, at this stage the correctness of the computations was ignored. Then more and more computational correctness was iteratively included, until the code was performing the correct calculations. In the same way the size of the blocks were increased until they approached the L2 cache size. This gave a instruction layout that gave high performance, as long as *all* the data iterated over is in L2 cache.

5.5.3 Using the Core Code Generator

When the CCG is executed with a set of parameters, the output (CF) is redirected into as a temporary .asm file. The .asm file also contains the parameter choices and internal build selections, included as comments. This .asm file can either be manually merged with the rank2k assembly source file, or it can be processed by a second Perl script (included in Appendix C). The second Perl script reformat the raw assembly to match the GCC in-line assembly style. In this way one can construct a complete C style function that contains only assembly code. Hence, the compiler can do the hard work of merging all the C setup/support code and the generated assembly CF. The C style 'function inline' directive can also be used so that the function calls can be eliminated. As a side effect the compiler can perform more optimizations as it can in-line (merge) the assembly with the setup/support code. This works, as it will not touch or include any of its own instructions inside the in-lined assembly section, instead it may/will exploit registers that are not used in the CF for it self. With this approach the entire code building process can be automated, without any manual intervention, and the result will be near optimal. Appendix A show an short, but complete core function in GCC in-line assembly style.

5.5.4 CCG Inner Working

The CCG currently accepts a number of parameters that depend on the data layout generated by support code. First the number of elements in one AB column, and secondly the number of columns to calculate. In addition an alternative linear layout of AB can requested. The write pattern layout of C can also be changed, from fully sequential to partially correct for each L1 block.

A number of internal control parameter are also available. The constant offsets used when calculating memory addresses can be shifted individually for AB_I and AB_J pointers. The only effect of this change is the size of the instructions (see Section 3.5 and 3.3), and their placement relative to memory addresses (see Section 3.5). Prefetch size and length can be changed somewhat, both in how much and fast to prefetch, and in what type of prefetch instructions to use. An alternative memory operand model can be requested, this changes the instruction generation to opportunistically select shorter instructions of the format const_offset(pointer, offset_register, const_offset_size) (see Section 3.3) instead of the simple const_offset(pointer) format. This effectively makes the code shorter in bytes, while no extra cost in the form of new instructions is added. However there can be an extra cost as one register is used for the offset, in theory this cost is none as numerous register are unused. In practise the this format puts more strain on the load ports of the register file, and might introduce extra stalls dude to register load starvation. In order to remove the longest instructions3.3 one can periodically update the pointer registers so that large offsets are avoided. This type of update (using *lea* instructions) can be turned on independently for both AB_I and AB_J pointers.

Finally, the register choice can be changed so that registers that alter instruction size, can be relocated between instructions. Potentially this will change their placement relative to memory addresses, and/or the total size of all the instructions. In addition numerous other small changes are available internally. Different memory access patterns can also be generated, in the same way as our Pentium 4 implementation.

5.5.5 Linear Layout of AB

The normal storage pattern in AB is such that only columns from A and B are interleaved. With the linear storage pattern two and two

AB columns are interleaved in a tight way. Lines of 4 pairs are formed, where each pair consists of two sequential values from either A or B. The 4 pairs are selected using two base indexes, the first is a K position index 'k' and the other is a column index 'c'. Illustrating the linear AB matrix pattern: $A_{k,c}A_{k+1,c}B_{k,c}B_{k+1,c}A_{k,c+1}A_{k+1,c+1}B_{k,c+1}B_{k+1,c+1}$, with c = 0, NUM_COLUMNS and k = K_START, K_START + K_LENGTH. The design and testing of the C support code for the linear layout of AB considered beyond the scope of the thesis.

5.5.6 Writing of C

Two write patterns are supported, fully linear and L1 block based. The linear write pattern means that every value calculated in the CF is written in a strictly linear fashion, as described in Section 4.1.4. With the L1 block based write pattern, each C value inside a single block calculated by a call to the CF, will be written in the correct sequence relative to each other. This reduce the complexity of support code, and is currently used exclusively. The design and testing of the support code for the fully linear write pattern is thus considered beyond the scope of the thesis.

5.5.7 Prefetching

Instruction generation for prefetching may be optionally turned on or off. Turning prefetching off can be beneficial when the AB matrix is small enough for laying permanently in the cache(s). With larger matrices, prefetching is needed to fill the cache, else the CF will repeatedly stall while generating short high bandwidth spikes. Since the processor cache size might vary and the optimal loop tiling size is somewhat unknown, controlling the rate of prefetching is important. Various processors also handle the prefetch instruction(s) in different ways, both where in the cache hierarchy to store the prefetched data, and how many bytes are loaded for each prefetch instruction. For the Core 2 processor the hardware prefetchers will also use the software prefetching as basis for additional prefetching. In order to handle these issues, the CCG can tune the number of prefetch instructions and how quickly to increment the prefetch address. The prefetch instruction type may also be changed, but this is currently disabled for practical reasons.

5.6 Parallelization

Scaling up from one core is becoming important, as multicore processors have become standard. While not the primary goal of the thesis, a simple multicore version of our implementation has been made. There are numerous potential problems with timing and thread control (affinity), this will be briefly presented first. The technique used for parallelization of our implementation is then presented.

Multicore Timing

With the advent of multicore processors, the *rdtsc* instruction became problematic as each core have their own counter. Over time, especially with *speedstep*, the value of their counters start to drift apart. This, combined with operative systems poor affinity control, leads to situations where one samples the counter on one core at the start and then on a different core at the end. Hence, the counter difference might be wildly wrong and possibly negative, as described in [7]. Later processors do not have this issues in the same way, constantly increasing the counter at full speed independently of actual frequency (and most sleep states). On the available test platforms this was not identified as a major problem.

5.6.1 OpenMP

A quick implementation for multicore support has been added using the *Open Multi-Processing* (OpenMP) compiler directives. The iterated sequence used in Section 5.4 have two iteration components (I and J), and both iterators change in "jumpy" patterns, being generated by a loop containing several if-else ladders controlling both I and J. Because basic parallelization with OpenMP requires a single loop iterator, a new support function capable of directly calculation the two iteration components has been constructed. This function does not have a linear run-time cost, however. A small $logN$ component together with a memory accesses to a precalculated lockup table, lowers the overall implementation efficiency. Finally, software prefetch address calculation is not working correctly in the current multithread implementation. As the wrong prefetch address is given to the core function, this reduces the performance somewhat for large matrices.

To achieve high performance in a multicore setting, several general issues must be addressed, especially how to handle and avoid cache trashing and saving memory bandwidth. With our implementation, using OpenMP this is not handled at all. Also, order to have a good multicore implementation, affinity control must work correctly, so that therads do not migrate between processor cores. If threads do migrate between processor cores, the entire cache is temporary lost, as cache do not migrate with the program. These problems can be corrected by a more refined pthread implementation, this is considered beyond the scope of the thesis, however.

5.7 Method

The same method used in Section 4.5 is also employed here, with some exceptions. Numerous issues have been encountered during benchmarking, so the number of samples (test-case runs) is increased to account for those problems. Most benchmarks with ATLAS and MKL include 100 samples, except for the tests with large matrices.

For the benchmarks of the Core Tester only a small number of samples have been used, as its measurements are very stable. The only runs shown of the core tester is in the *nop* instruction test, described later.

Also, the framework used have been compiled twice, one version for the benchmarks obtained with ATLAS, and one for all the others. The reason ATLAS is benchmarked with a different version of the framework, is that the installed version of MKL actively intercepts BLAS function calls, and replaces them with calls to MKL. After the BLAS calls are replaced, all the ATLAS function calls effectively becomes MKL function calls. This behavior was not present on the Clustis2 nodes used in Chapter 4. Our implementation is not affected by this problem, as the internal function name is different than used by BLAS. The other implementations are not affected for the same reason.

At the first use of OpenMP in a program, a setup stage is performed by OpenMP. This setup create a temporary slowdown, that can affects benchmarks. In order to avoid the setup cost inside the timed code section, OpenMP is initialised by MKL in the framework. The ATLAS version do not perform the setup, as it is useless because ATLAS have a pthread implementation. To negate possible startup costs related to ATLAS, the

framework also initialises ATLAS before timing, by employing ATLAS as reference implementation.

Our support code wrapped around the core function, have several build parameters (5 with debug timing printing). These affect how memory is allocated and what code paths are used to support the core function. The most important parameter control the block size of the Prefetching Access Pattern, as this block size must be tuned to the L2 cache size. The second most important parameter selects if the OpenMP path or the single-thread implementation is used. It is also possible to control if extra timing information is to be printed. This timing information is obtained using several *rdtsc* instructions, timing every internal code section. The timers can currently not be disabled, however, so a tiny performance hit is taken in all measurements of our implementation.

A large number of implementations have been created, both by hand and by numerous versions of the Core Code Generator. Only a small selection is presented, for several reasons. First, the possible design space of the CCG contains multiple variables (7 directly available), the support also have several build variables, this represents too many independent factors for compact presentation. Second, because of numerous issues with the setup, all the benchmarks have been rendered useless several times. We prioritized obtaining and presenting valid comparison data, where both ATLAS and MKL have a correct and optimal setup as possible. Because of the nature of the issues found, all benchmarks of our implementation had to be redone as well (several times). Unfortunately, this limits the presented data to contain only a chosen selection of parameters for our implementation.

Tuning Test

Some evaluation on how precise benchmark measurements can be performed is necessary, since the approach of using empirical data to guide optimization is an important part of the method used. Therefore, in order to test how precise measurements that is possible, and look for potential side effects, a simple test was performed. One core function was taken and modified slightly by hand. A single *nop* (no operation) instruction was added to the start of the innermost loop. The binaries was inspected to verify the changes, and check for possible side-effects. The size of this instruction was verified to be a single byte in binary format, and that this byte did not cause the inner

loop to spill over a 16-byte boundary at the end (see Section 3.3). Note that the loop started at the same memory address in all three cases, and it was aligned every time (see Section 3.5). The processor documentation states that this instruction does not consume resources in the pipeline, and its only decoded and ignored. Testing was performed on a base version generated with K_Length of 64, with a 16 by 16 block size. Its (base) innermost loop was 8209 bytes large and consisted of 1550 instructions. Since automatic tuning must be performed without using excessive time, it is important to find speed differences without a large number of reruns. A modest number of 20 runs was chosen, using the core tester program. Both the variability measures, standard deviation and absolute median deviation, need to be low and reasonable equal in order to use both few and short benchmark-runs as basis for automatic tuning. If only the absolute median deviation is low, then several samples are needed, because of interference from other sources.

5.7.1 Setup

All the benchmarks are performed on the compute nodes of Clustis3, described in Section 2.2.1. On this platform a maximum of 4.0 flop/cycle is possible using double precision. As half of the calculation can be skipped, 8.0 flop/cycle will be the maximum reported by the framework on single-thread code. For multicore implementations, 8 times higher values are possible (64.0 flop/cycle in the framework) from the dual quad-processors in the compute nodes. The compiler used is GCC version 4.1.2 (Red Hat 4.1.2-42), as no newer version was available (and functioning correctly) at the time of benchmarking.

Also, the Clustis3 nodes have 9GB memory, configured as 4x2GB + 2x512MB with somewhat different speed grades. Because of this issue, all benchmarks gave systematic different performance between different compute nodes, as they used various amounts of physical memory for file buffering (based on their previous work load). In order to map out the slower memory, a separate program is used to allocate this memory, as it is often the first memory utilised by the operating system (in practice).

A second issue requiring special attention relate to numerous affinity problems, causing huge variance between sequential executions of the same program. In order to minimize the problem a single compute node was given the most recent Linux kernel available, having better affinity support.

Finally, a set of special environment variables (KMP_AFFINITY) is used, controlling both the Intel OpenMP affinity support, and the thread affinity of MKL itself.

The following implementations are used in the benchmarks:

ATLAS — The latest ATLAS version (3.8.3 at the time of benchmarking) is used, tuned to the compute nodes. In addition the source code base was hand tuned according to the ATLAS documentation, to correct possible performance issues for multicore platforms. After the setup, its performance was verified against benchmarks of known 'good' builds for the same processor. Two slightly different versions have been used, one compiled with multithread support using *pthreads*, and one for singlethread. No control of thread affinity where found to be effective for the multithread version.

MKL — MKL version 10.0.3.020 is used. This is not the very latest version. Unfortunately, the slightly newer (available) version failed to build into the framework used, apparently being replaced by the older MKL version at runtime. The newest version '11' was not available for the operating system (Linux) used on Clustis3. For the multithread benchmarks the environment variable KMP_AFFINITY=scatter was also used, as this improved its performance. With the singlethread benchmarks the environment variable OMP_NUM_THREADS=1 were used.

Choice — A single configuration of our implementation, with its detailed build parameters described below.

Our *Choice* version gave acceptable performance on the matrix size N=K=2000 compiled with OpenMP support, and was chosen by hand after the following large (but still limited) set of candidates where tested: The fixed parameters are $abiStartOffset = (1024)$ and $abjStartOffset = (1024)$, linear layout of AB columns enabled, linear write pattern of C disabled, prefetching instructions is also enabled. The tested parameter space were AB columns containing 8 to 192 elements from the A and B matrices, increased in steps of 8. For the core function, the block sizes of 4 to 32 AB columns where tested in steps of 4.

From the candidate set a version giving acceptable performance on the matrix size N=K=2000 (compiled with multithread support), was chosen by hand. Two versions of this parameter configuration were the compiled, one with OpenMP support, and one using the singlethread code. The

Intel based OpenMP library were used, with the environment variable KMP_AFFINITY=compact. Other configurations of our implementation have higher performance with different matrix sizes and in singlethread configuration, but they are not included in the benchmarks.

The *Choice* configuration have a AB column length of 80 elements from each of A and B, with blocks of 16 by 16 columns in the core function. A L2 cache loop tile block size of 64 core function blocks, were used. Also an simple parallel memory copy (using OpenMP) is employed for copying the A and B matrices into AB. Internal debug timings were also enabled. Note that for the the K dimension of 2000 is directly divisible by 80, enabling full performance potential for this matrix size. For the smallest matrix size of 1000 elements in the K dimension, a slight performance hit will occur. This effect is also true for for the N dimension of 1000, not being divisible by the block factor (16) in the core function. Finally, software prefetch address calculation is not working correctly in the current multithread implementation, this reduces the performance somewhat for large matrices.

5.8 Results

The results with the single-thread implementations can be seen in Figure 5.5, for a selection of matrix sizes. Our implementation performance is low with small matrices, which can be expected as the C based support code overhead play a greater effect. Also, both matrix dimensions are not divisible by the internal block sizes in the *Choise* configuration. Note that our implementation was not selected for its performance on this matrix size. MKL have relatively stable performance for all the matrix sizes, indicating that it have low overhead in its support code coupled with a somewhat slower GEMM implantation. For ATLAS, the trend is that larger matrices give better performance, possibly from a mix of support code overhead and a somewhat better GEMM implantation than MKL. For larger matrices, our implementations beat both ATLAS and MKL indicating that our core function have a high performance. Both MKL and ATLAS show a slight performance drop for the largest matrices, most likely this comes from the operating system begins to (re)use some of the memory with lower speed grade for this problem size.

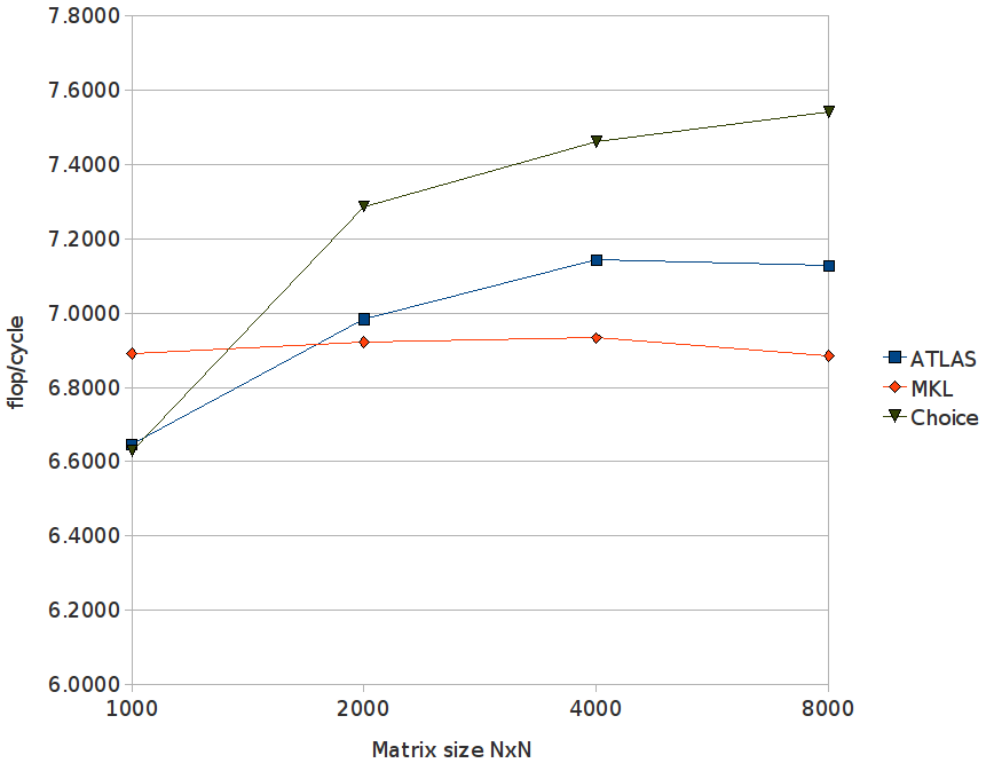


Figure 5.3: Median single-thread performance on Clustis3.

More details of the performance obtained can be found in Tables 5.7, 5.8, 5.9 and 5.10. For all implementations the variance indicated by median absolute deviation is low, showing that no major performance issues contaminate the benchmarks. A speedup of almost 105.8% relative to ATLAS is achieved by our implementation, and an impressive 109.5% relative to MKL, on the largest matrix size.

Table 5.7: Single-thread performance, N=1000 and K=1000, on Clustis3.

	ATLAS	MKL	Choice
Samples	100	100	100
Average	6.6312	6.8913	6.6279
Minimum	6.4482	6.8736	6.6168
Maximum	6.6807	6.9023	6.6353
Median	6.6461	6.8915	6.6289
Std. Dev.	0.05819	0.00426	0.00416
Median Absolute Deviation	0.00330	0.00210	0.00280
% of ATLAS	100.000%	103.692%	99.741%
% of MKL	96.439%	100.000%	96.190%
% of max	83.076%	86.144%	82.861%

Table 5.8: Single-thread performance, N=2000 and K=2000, on Clustis3.

	ATLAS	MKL	Choice
Samples	25	25	25
Average	6.9606	6.9226	7.2865
Minimum	6.8738	6.9142	7.2823
Maximum	6.9975	6.9285	7.2901
Median	6.9838	6.9225	7.2864
Std. Dev.	0.03924	0.00295	0.00169
Median Absolute Deviation	0.01160	0.00120	0.00105
% of ATLAS	100.000%	99.122%	104.333%
% of MKL	100.886%	100.000%	105.257%
% of max	87.298%	86.531%	91.080%

Table 5.9: Single-thread performance, N=4000 and K=4000, on Clustis3.

	ATLAS	MKL	Choice
Samples	25	25	25
Average	7.1202	6.9318	7.4538
Minimum	7.0536	6.8675	7.3902
Maximum	7.1494	6.9465	7.4629
Median	7.1424	6.9328	7.4605
Std. Dev.	0.03721	0.01495	0.01957
Median Absolute Deviation	0.00570	0.00540	0.00055
% of ATLAS	100.000%	97.065%	104.454%
% of MKL	103.023%	100.000%	107.612%
% of max	89.280%	86.660%	93.256%

Table 5.10: Single-thread performance, N=8000 and K=8000, on Clustis3.

	ATLAS	MKL	Choice
Samples	5	5	5
Average	7.1179	6.8845	7.5413
Minimum	7.0659	6.8624	7.5392
Maximum	7.1363	6.9030	7.5432
Median	7.1276	6.8845	7.5409
Std. Dev.	0.02945	0.01563	0.00154
Median Absolute Deviation	0.00700	0.01010	0.00140
% of ATLAS	100.000%	96.589%	105.799%
% of MKL	103.531%	100.000%	109.534%
% of max	89.095%	86.056%	94.261%

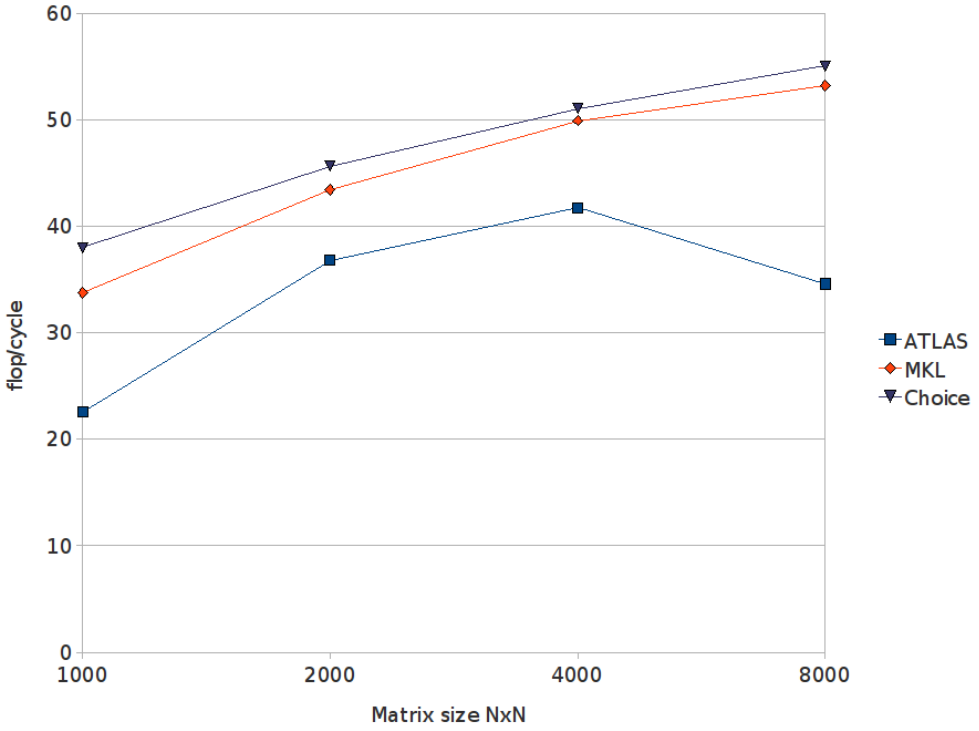


Figure 5.4: Median multithread performance on Clustis3.

With the multithread performance, shown in Figure 5.4 a more complex picture emerges. Here ATLAS starts with low performance for the smallest matrix size, then it display a good performance improvement up to the next matrix size. This trend is then lost for the 4000 by 4000 matrix size, before

performance drops in the largest matrix size. The ATLAS configuration documentation describes this phenomena, and ATLAS were rebuilt with recommendations described there, still performance is low. It might therefore be a secondary configuration problem, causing this performance drop. MKL display stable performance improvement as the matrix size increases, with a median close to its peak. Our implementation show a reasonably stable speedup relative to MKL, for all the matrix sizes evaluated, even the less optimal 1000 by 1000 matrix size.

Because of the problematic behaviour of ATLAS, it needs to be evaluated in some more details. The median performance of ATLAS is only 22.58 flop/cycle on the small matrix size, much below its peak on that size. Table 5.11 show that both variance measures are very high, where the performance span go from 16 to 28 flop/cycle, indicating that its pthread performance most likely is heavily affected by startup or affinity issues. Our implementation show the same tendency to a minor extent in the form of some outliers, but its median performance is still acceptable with a low variance (median absolute deviation).

Table 5.11: Multithread performance, N=1000 and K=1000, on Clustis3.

	ATLAS	MKL	Choice
Samples	100	100	100
Average	22.0700	33.3872	37.6146
Minimum	16.1146	25.0962	32.3485
Maximum	28.1487	33.8897	38.0892
Median	22.5808	33.7486	38.0043
Std. Dev.	2.16803	1.32016	1.29848
Median Absolute Deviation	1.30595	0.03775	0.03820
% of ATLAS	100.000%	149.456%	168.304%
% of MKL	66.909%	100.000%	112.611%
% of max	35.283%	52.732%	59.382%

More details of the performance obtained can be found in Tables 5.11, 5.12, 5.13 and 5.14. A speedup of almost 168.3% relative to ATLAS is achieved by our implementation, and an 112.6% relative to MKL, on the small matrix size. For the matrix size our implementation was tuned for the, speedup relative to MKL were almost 105.1%.

Table 5.12: Multithread performance, N=2000 and K=2000, on Clustis3.

	ATLAS	MKL	Choice
Samples	100	100	100
Average	36.4596	43.1869	45.3130
Minimum	29.8238	41.2121	41.4679
Maximum	39.7383	43.7262	46.5322
Median	36.8211	43.4585	45.6610
Std. Dev.	1.57108	0.50562	1.21828
Median Absolute Deviation	0.86670	0.14940	0.62805
% of ATLAS	100.000%	118.026%	124.008%
% of MKL	84.727%	100.000%	105.068%
% of max	57.533%	67.904%	71.345%

Table 5.13: Multithread performance, N=4000 and K=4000, on Clustis3.

	ATLAS	MKL	Choice
Samples	100	100	100
Average	41.7368	49.9278	50.9634
Minimum	41.0320	49.4391	48.4412
Maximum	42.1875	50.2467	51.3377
Median	41.7429	49.9370	51.0810
Std. Dev.	0.24582	0.16091	0.47134
Median Absolute Deviation	0.16845	0.09030	0.11035
% of ATLAS	100.000%	119.630%	122.370%
% of MKL	83.591%	100.000%	102.291%
% of max	65.223%	78.027%	79.814%

Table 5.14: Multithread performance, N=8000 and K=8000, on Clustis3.

Data	ATLAS	MKL	Choice
Samples	15	15	15
Average	34.6204	53.2494	55.0317
Minimum	34.4507	53.0357	54.7631
Maximum	34.8726	53.4151	55.1239
Median	34.6220	53.2249	55.0725
Std. Dev.	0.11584	0.10851	0.09386
Median Absolute Deviation	0.08770	0.06480	0.03630
% of ATLAS	100.000%	153.731%	159.068%
% of MKL	65.049%	100.000%	103.471%
% of max	54.097%	83.164%	86.051%

One to note in Tables 5.10 and Table 5.14 is that the efficiency of MKL drops from 86.051% in single-thread to 83.164% when using 8 threads. Our implementation goes from 94.261% to 86.051% on the largest matrix size, displaying a much larger performance hit. This clearly indicates that the multithread implementation we use is inefficient.

Finally, in order to indicate the performance obtained on Clustis3, when the affinity and memory issues were not handled in any way, Figure 5.5 shows 100 samples from three sample series. The first MKL Tuned, is the same series as used as basis in Table 5.11, with memory and affinity handled manually. That series was performed with a very recent Linux kernel, on compute node 0. Both other series where obtained with the default Linux kernel, on two different compute nodes. MKL Untuned, node 0 show large variations between each run, partially caused by lack of affinity control. This same issue is also present on MKL Untuned, node 4. However, there are no hardware differences between any of the nodes, yet they have clearly different performance. This large differences were found to change from day to day, randomly alternating which nodes where faster and slower. The reason being that the slow memory sometimes were used as file cache, by previous batch jobs, and other times they had been flushed.

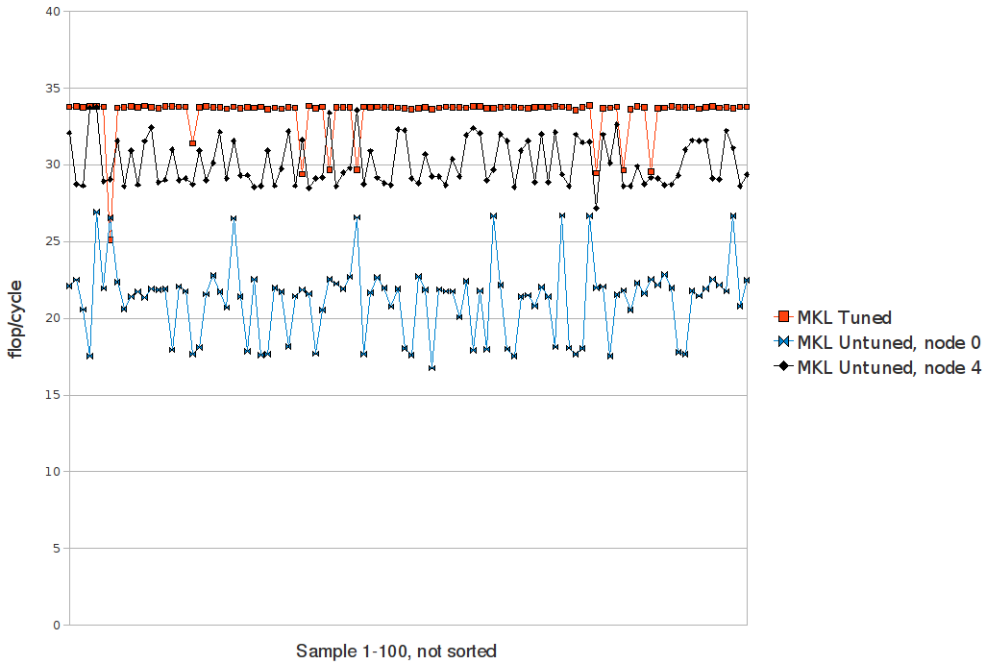


Figure 5.5: 100 samples of multithreaded MKL on N=1000 K=1000 matrices, on Clustis3.

5.8.1 Tuning Test: nop

Table 5.15: 20 runs performed on Clustis3 with the core tester. Base is unmodified, 1 Nop is with one *nop* added, 2 Nop is with two *nop*'s added.

	Base	1 Nop	2 Nops
Median	7.6502	7.6659	7.65275
Median Absolute Deviation	0.0006	0.0006	0.0012
Minimun	7.6429	7.6635	7.6464
Maximum	7.6518	7.6671	7.6547
Average	7.64971	7.66594	7.65213
Std. Dev.	0.002	0.00098	0.00236
% of Base	100.000%	100.205%	100.033%

The median speed in Table 5.15 show that adding a *nop* instruction gave a speedup of 100.205%, while slowdown was expected. This speedup is equivalent of 1550*0.205% = 3.18 instructions. Adding a second *nop* most of the speedup gained by the first is lost, while still showing a small speedup compared to the base. The slowest run with one *nop* was faster than the fastest run without it, showing a robust speedup that is easy to measure. The standard deviation is also somewhat high in the base runs, compared to ones with 1 *nop* version.

The cause of this probably requires extensive analysis to understand, and is likely linked to the speedup shown. Some possible candidates might be: 1. The addresses of the load instructions change, so that the IP-Prefetcher becomes more efficient. 2. Favorable alterations to how the Out Of Order execution hardware ends up scheduling some or all the later instructions. Both will effect the data cache by changing which cache lines are evicted and loaded, and the number of outstanding misses. There can be several other less likely causes as well. However, this simple example illustrates the somewhat chaotic optimization searching needed (or possible), and including this in a normal compiler seems hard.

5.9 Evaluation of Core 2 Results

Numerous tests have been made, issues tested and theories evaluated, in order to try to understand how the processor(s) work. Since the documentation

often lacked important information or turned out to be directly wrong, excessive amounts of time was spent on wrong or useless optimizations. 1000's of handmade versions designed during development gave insights that are very hard to express in benchmarks or effectively document, if for no other reason that it would take way to much space to describe in a thesis.

Only a single *nop* instruction test of an arbitrary chosen parameter setup is included from the tuning process. The selection is therefore not as good a guide on what to try out in order to optimize, but gives a hint or taste of the process itself. Most importantly, even the current 'best' code used in the final version has been outperformed by several earlier test versions. The reason these versions are not used (or benchmarked) is that they were hand-tuned in assembly after they were generated, often with a somewhat modified code generator. The time needed to extend the code generator to include these improvements would require a complete redesign of the code generator. It is clear that more performance can be harnessed from changing the instruction layout parts that are still hard-coded, and that this performance is (partially) independent of the tuning already performed. However, we found such details to be beyond the scope of a master thesis.

The MKL interception of BLAS function calls was only identified after extensive testing, as the performance of ATLAS and MKL was very after all other problems affecting performance was identified (and corrected). Why MKL functions like this was not found, but it might be an interaction between the Make-script, runtime linking and system configuration, and not a directly MKL related issue.

5.9.1 Code Issues from the Pentium 4 implementation

One outstanding issue is the K_length choice in the AB matrix in Section 4.3.2, there have been no parameter search of this parameter except during the original code development. From the findings in this chapter, it is likely that it might be better choices of this and possibly other parameters. Modifications and corrections was not attempted as this old processor architecture is approaching the end of its lifetime.

Chapter 6

Conclusions, Current & Future Work

This thesis project turned out to much more complex than expected and documenting all our results rather overwhelming. The original goals of this thesis, which was to maximize performance on BLAS-like algorithms and develop a general theory for how to achieve this, lead to an iterative process that was very hard to describe. In addition, there was too much empirical information that did not match the expected model. These inconsistencies were encountered at almost every iteration, and required time-consuming verification and numerous additional concepts too look into. Finally, our original goal of developing a general theory for optimizing these kinds of algorithms for modern architectures, turned out to be too ambitious. Too much time was spent on empirical work early on, so in the end we did not have time to make a better model and test it.

We chose the Intel Pentium 4 and Intel Core 2 as our testbeds for our optimized codes. A simplified version of the BLAS rank2k (dsyr2k) was implemented on these platforms, and numerous patterns of data layout and access patterns for the A and B matrices were generated and tested in order to create an near-optimal version. In order to achieve this, we developed a pattern generator for both the Pentium 4 and the Core 2 processors, and an assembly code generator for Core 2, both in Perl.

We discovered two main issues: The memory access pattern is extremely important when it come to performance on the Pentium 4. In addition,

extremely low-level details such as the memory location of each instruction and how the Core to Out of Order execution engine (OoOE) responds to tiny code modification, were found to make major impact on achieving optimal performance.

In the end, we nevertheless were able to outperform both ATLAS and MKL on our test case, even though we limited ourselves to not exploring all the issues encountered and the possible parameter space of the code generator. Note that ATLAS has several code sections that are hand-coded. Details of the MKL library are proprietary, but we suspect some of their code of their code is also hand-optimized in assembly language, whereas our Pentium 4 implementations relied on C-language code. Our Core 2 codes is Perl-generated assembly code, as was mentioned earlier.

For the Pentium 4, a 10.8 % speed-up was achieved over ATLAS's rank2k, and we achieved 17% speed-up over MKL's implementation, for 4000-by-4032 matrices. On the Core 2 we optimized our code for 2000-by-2000 matrices and achieved a 24% and 5% speed-up over ATLAS and MKL, respectively with our multi-threaded implementation. However, for the 8000x8000 our implementations beat ATLAS by 59% indicating ATLAS had problems on this test case. MKL was here only beaten by 3%. In the single-threaded case for Core 2, we achieved a 5.7% and a 9.5% speed-up over ATLAS's and MKL's implementations, respectively, for 8000 x 8000 matrices. Considering that our implementation was far from fully tuned, we consider these result very respectable.

A couple of the parameters used in our Pentium 4 implementation, for instance the loop tiling parameter, were found to be identical to the ones found in the hand-tuned assembly coded work by Goto in [9]. Their implementation details were only checked two days before the thesis was delivered, and were not used in any way during implementation. While the similarities in our implementations are striking, they confirm that certain optimal parameters can likely be found. We picked out parameters based on some empirical benchmarks during our original Pentium 4 development (when taking the Parallel Programming course that lead to this work). This is, however, an issue that should be further looked into.

6.1 Current and Future Work

The architecture of modern processors keep changing at a rapid pace. New enhancements to the instruction set are hard to keep up with or time

consuming to include in both compilers and code libraries. Some of the newer architectural features are also hidden from the public. The issues that were shown to be lack of affinity control, seem to fixed in the new Intel Core i7, with its inclusion of the new RDTSCP instruction which gives both timing and core ID in one atomic operation. A pthread version of our implementations is left as future work.

6.1.1 Future Scalability

Two new extensions to the x86 family have already been announced, one by AMD© (SSE5) and one by Intel© *Advanced Vector Extensions* (AVX). Both will require major changes to the instruction selection in the kernel. In order to handle this type of evaluation, one traditionally either relies on the compiler to do the job, or are forced to write new code in assembly by hand.

6.1.2 SSE5

AMD© have announced an extension called SSE5[1], that is believed to be made available around 2011. There are two main enhancements that are relevant here. The first is that the new instructions can take 3 operands, 2 source and 1 destination. This eliminates a number of register—register copy instructions. The other enhancement is that a new combined multiply and add instruction (fused multiply-accumulate or FMAC) is included. This will further reduce the number of instructions needed. Combined they allow for several kernel layouts that are not currently considered, that may (most likely) give better performance.

6.1.3 AVX - Advanced Vector Extensions

Intel© have (also) announced an new extension called AVX[2] [11]. The enhancements are similar to the ones in SSE5 with one major difference, the register size is extended to 256-bit. This allows for 8 32-bit floats or 4 64-bit doubles to be calculated by a single instruction. A similar 3 (or 4) operand format is also used. The fused multiply-add instruction(s) will also be available in a later processor generation.

[1] `http://developer.amd.com/cpu/SSE5/Pages/default.aspx`
[2] `http://software.intel.com/en-us/avx/`

6.2 Remaining Issues

Our original plan was to develop an auto-generator planning tool. For each processor feature or design choice (domain) we wanted to generate a set of candidate designs (solutions) that are optimal in that domain. Various domains exchange data for re-tuning, and the exploration of more solutions based on constraints/hints from other domains. This would create a multidimensional solution space of partially orthogonal directions, forming an overdetermined system. A search should then be performed independently in each orthogonal direction (or subspace), so that a set of good candidate solutions are found. In the partially dependent directions, the most inflexible properties are used as constraints for searching the solution space. The more expensive the domain is to search in, the later in the search it should be explored. Basically, each domain would use a number of fixed search parameters that comes either from known basic facts (like size of caches, maximum throughput of computation, or bandwidth), or from constraints found in other domains.

Her are some of the design choices that were meant to be explored by different tools, in various domains:

- Cache block shapes from bandwidth/compute ratio.
- Cache block sizes from a *CPU analyzer* (L1, L2 data).
- Usable instruction set from a *CPU Analyzer* (SSE, SIMD, 32/64 bit).
- Inner loop block shapes from a *Instruction Selector*.
- Inner loop block size from a *Instruction Selector*.
- Data access pattern from a *Inner Loop Generator*.
- Inner loop instruction layout from a *Pipeline/Execution Unit Simulator*.
- Cache block size and shape from our *Cache Simulator*.
- Data access pattern cache efficiency from a *Data Access Compiler* and benchmarks.

- Native processor instruction layout, (discovering total instruction size, performing register choice) from a *Low-Level Compiler*.

- Inner loop instruction layout efficiency from the *Core Tester*.

- Large block access pattern from a *Outer Loop Generator/Access Pattern Generator*.

- Actual L1 cache block size and access pattern efficiency from the *Core Tester*.

- Actual L2 cache block size, access pattern and prefetch efficiency from an *Extended Core Tester*

Modern CPU's also contain a large number of unexpected rules that affect performance. This makes the task both hard and time consuming, and typically leads to sub optimal code, even when hand optimized in assembler.

A key problem during our development was how to structure and coordinate our design, so that it was possible to achieve both future flexibility and maintaining a realizable (or practical) system.

The speedup possible from improving layout can be substantial, and making the calculation pattern structure more flexible will also be useful for a GEMM implementation. This thesis should, however, provide several hints of the possibilities for developing future code generators.

Bibliography

[1] K. Beyls and E.H. D'Hollander. Refactoring for data locality. *Computer*, 42(2):62–71, Feb. 2009.

[2] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Electrical Engineering and Computer Science Series. MIT Press/McGraw Hill, 1990.

[3] Intel® Corporation. *Intel® C++ Intrinsic Reference*, October 2007.

[4] Intel® Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, December 2008.

[5] Intel® Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, September 2008.

[6] Intel® Corporation. *Intel® Math Kernel Library (Intel® MKL) 10.1 In-Depth*. Intel® Corporation, Mars 2009.

[7] ©AMD Corporation. *AMD Technical Bulletin – TSC Dual-Core Issue & Utility Fix*. © 2007 Advanced Micro Devices, Inc, June 2007.

[8] Daniel D. Gajski. *Principles of digital design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[9] Kazushige Goto and Robert van de Geijn. On reducing tlb misses in matrix multiplication, 2002.

[10] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2002.

[11] Intel® Corporation. *Intel® Advanced Vector Extensions Programming Reference*, January 2009.

[12] B. Jacob. A case for studying dram issues at the system level. *IEEE_M_MICRO*, 23(4):44–56, 2003.

[13] R. E. Kessler and Mark D. Hill. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems*, 10:338–359, 1992.

[14] M. Kulkarni and K. Pingali. An experimental study of self-optimizing dense linear algebra software. *Proceedings of the IEEE*, 96(5):832–848, May 2008.

[15] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *In Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, 1991.

[16] W.L. Lynch, B.K. Bray, and M.J. Flynn. The effect of page allocation on caches. In *Proc. 25th Annual International Symposium on Microarchitecture MICRO 25*, pages 222–225, 1992.

[17] David A. Patterson and John L. Hennessy. *Computer Organization and Design, the hardware/software interface.* Elsevier Inc., 30 Corporate Drive, Suite 400, Burlington, MA 01803, USA, 2007.

[18] J. Seward, N. Nethercote, J. Weidendorfer, and the Valgrind Development Team. *Valgrind 3.3 — Advanced Debugging and Profiling for GNU/Linux applications.* Network Theory LTD, 15 Royal Park, Bristol, BS8 3AL, UK, 2008.

[19] Perry H. Wang, Jamison D. Collins, Hong Wang, Dongkeun Kim, Bill Greene, Kai-Ming Chan, Aamir B. Yunus, Terry Sych, Stephen F. Moore, and John P. Shen. Helper threads via virtual multithreading on an experimental itanium®2 processor-based platform. *SIGPLAN Not.*, 39(11):144–155, 2004.

[20] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001. Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000 (`www.netlib.org/lapack/lawns/lawn147.ps`).

[21] K. Yotov, X. Li, G. Ren, M.J.S. Garzaran, D. Padua, K. Pingali, and P. Stodghill. Is search really necessary to generate high-performance blas? 93(2):358–386, Feb. 2005.

Appendixes

Appendix A

GCC In-line Assembly Code Sample

The entire *core function* GCC in-line assembly code of our *Choice* implementation occupies 35 pages, so an reduced version was built with the same parameters, except that the AB column size (named K in the code) was reduced to one tenth (8 elements from each of A and B matrices). The full version is available on request.

```
#define K_LENGTH_ 8
#define Block_SIZE_X_L2 16
static void inline doBlock_inter(double *restrict _c, const double *restrict
     _abi, const double *restrict _abj, const double *restrict _prefetch)
{
  // Add the asm in here, directly
__asm__("# K:8, blockSizeX:16, abiStartOffset:1024, abjStartOffset:1024\n\t"
    "# addPrefetch:1, prefetchIncrementBytes:64, prefetchDivisor:2\n\t"
    "# useOffsetCounterRegister:0, offsetCounterRegisterValue:128\n\t"
    "# useLeaOffsetABJ:0, useLeaOffsetABI:0\n\t"
    "# interleavedTwoABLines:0, skipFirstSumup:1\n\t"
    "# makeEntryFullBlock\n\t"
    "xorpd  %%xmm2, %%xmm2\n\t"
    "movq %%rcx, %%r8   # Copy prefetch pointer to its local home register\n
        \t"
    "movq %%rdi, %%rcx\n\t"
    "subq $1024, %%rdx\n\t"
    "movq %%rdx, %%rax\n\t"
    "addq $1888, %%rdi\n\t"
    "movapd %%xmm2, %%xmm6\n\t"
    "movapd %%xmm2, %%xmm3\n\t"
    "movapd %%xmm2, %%xmm7\n\t"
    "movapd %%xmm2, %%xmm4\n\t"
    "movapd %%xmm2, %%xmm8\n\t"
    "movapd %%xmm2, %%xmm5\n\t"
    "movapd %%xmm2, %%xmm10\n\t"
    "movapd %%xmm2, %%xmm11\n\t"
```

```
"movapd %%xmm2, %%xmm9\n\t"
"subq $1024, %%rsi\n\t"
"movl $3, %%r9d\n\t"
"movq $32, %%r10\n\t"
"jmp 2f\n\t"
".p2align 4\n\t"
"# makeFirstLoopStart\n\t"
"# makeSecondLoopStart\n\t"
"1:\n\t"
"haddpd %%xmm11, %%xmm3\n\t"
"addpd  (%%rcx), %%xmm3\n\t"
"haddpd %%xmm4, %%xmm10\n\t"
"addpd  16(%%rcx), %%xmm10\n\t"
"haddpd %%xmm8, %%xmm6\n\t"
"addpd  128(%%rcx), %%xmm6\n\t"
"haddpd %%xmm9, %%xmm7\n\t"
"addpd  144(%%rcx), %%xmm7\n\t"
"movapd %%xmm3, (%%rcx)\n\t"
"movapd %%xmm10, 16(%%rcx)\n\t"
"movapd %%xmm6, 128(%%rcx)\n\t"
"movapd %%xmm7, 144(%%rcx)\n\t"
"addq %%r10, %%rcx\n\t"
"movq $32, %%r10\n\t"
"subl $1, %%r9d\n\t"
"# makeCoreStartupBlock\n\t"
"2:\n\t"
"movapd 1024(%%rsi), %%xmm0\n\t"
"movapd 1152(%%rsi), %%xmm5\n\t"
"movapd 1040(%%rax), %%xmm2\n\t"
"movapd %%xmm2, %%xmm3\n\t"
"mulpd  %%xmm0, %%xmm3\n\t"
"movapd %%xmm2, %%xmm6\n\t"
"mulpd  %%xmm5, %%xmm6\n\t"
"movapd 1168(%%rax), %%xmm2\n\t"
"movapd %%xmm2, %%xmm11\n\t"
"mulpd  %%xmm0, %%xmm11\n\t"
"movapd %%xmm2, %%xmm8\n\t"
"mulpd  %%xmm5, %%xmm8\n\t"
"movapd 1296(%%rax), %%xmm2\n\t"
"movapd %%xmm2, %%xmm10\n\t"
"mulpd  %%xmm0, %%xmm10\n\t"
"movapd %%xmm2, %%xmm7\n\t"
"mulpd  %%xmm5, %%xmm7\n\t"
"movapd %%xmm0, %%xmm4\n\t"
"mulpd  1424(%%rax), %%xmm4\n\t"
"movapd %%xmm5, %%xmm9\n\t"
"mulpd  1424(%%rax), %%xmm9\n\t"
"movapd 1040(%%rsi), %%xmm0\n\t"
"# makeCoreFullBlock\n\t"
"#makeCoreInnerBlock: 1\n\t"
"movapd 1024(%%rax), %%xmm2\n\t"
"movapd %%xmm2, %%xmm1\n\t"
"mulpd  %%xmm0, %%xmm2\n\t"
"addpd  %%xmm2, %%xmm3\n\t"
"movapd 1168(%%rsi), %%xmm5\n\t"
"mulpd  %%xmm5, %%xmm1\n\t"
"addpd  %%xmm1, %%xmm6\n\t"
"movapd 1152(%%rax), %%xmm2\n\t"
"movapd %%xmm2, %%xmm1\n\t"
"mulpd  %%xmm0, %%xmm2\n\t"
"addpd  %%xmm2, %%xmm11\n\t"
"movapd 1280(%%rax), %%xmm2\n\t"
```

```
"mulpd   %%xmm5, %%xmm1\n\t"
"addpd   %%xmm1, %%xmm8\n\t"
"movapd %%xmm2, %%xmm1\n\t"
"mulpd   %%xmm0, %%xmm2\n\t"
"addpd   %%xmm2, %%xmm10\n\t"
"mulpd   1408(%%rax), %%xmm0\n\t"
"addpd   %%xmm0, %%xmm4\n\t"
"movapd 1056(%%rsi), %%xmm0\n\t"
"mulpd   %%xmm5, %%xmm1\n\t"
"addpd   %%xmm1, %%xmm7\n\t"
"mulpd   1408(%%rax), %%xmm5\n\t"
"addpd   %%xmm5, %%xmm9\n\t"
"#makeCoreInnerBlock: 2\n\t"
"movapd 1072(%%rax), %%xmm2\n\t"
"movapd %%xmm2, %%xmm1\n\t"
"mulpd   %%xmm0, %%xmm2\n\t"
"addpd   %%xmm2, %%xmm3\n\t"
"movapd 1184(%%rsi), %%xmm5\n\t"
"mulpd   %%xmm5, %%xmm1\n\t"
"addpd   %%xmm1, %%xmm6\n\t"
"movapd 1200(%%rax), %%xmm2\n\t"
"movapd %%xmm2, %%xmm1\n\t"
"mulpd   %%xmm0, %%xmm2\n\t"
"addpd   %%xmm2, %%xmm11\n\t"
"movapd 1328(%%rax), %%xmm2\n\t"
"mulpd   %%xmm5, %%xmm1\n\t"
"addpd   %%xmm1, %%xmm8\n\t"
"movapd %%xmm2, %%xmm1\n\t"
"mulpd   %%xmm0, %%xmm2\n\t"
"addpd   %%xmm2, %%xmm10\n\t"
"mulpd   1456(%%rax), %%xmm0\n\t"
"addpd   %%xmm0, %%xmm4\n\t"
"movapd 1072(%%rsi), %%xmm0\n\t"
"mulpd   %%xmm5, %%xmm1\n\t"
"addpd   %%xmm1, %%xmm7\n\t"
"mulpd   1456(%%rax), %%xmm5\n\t"
"addpd   %%xmm5, %%xmm9\n\t"
"#makeCoreInnerBlock: 3\n\t"
"movapd 1056(%%rax), %%xmm2\n\t"
"movapd %%xmm2, %%xmm1\n\t"
"mulpd   %%xmm0, %%xmm2\n\t"
"addpd   %%xmm2, %%xmm3\n\t"
"movapd 1200(%%rsi), %%xmm5\n\t"
"mulpd   %%xmm5, %%xmm1\n\t"
"addpd   %%xmm1, %%xmm6\n\t"
"movapd 1184(%%rax), %%xmm2\n\t"
"movapd %%xmm2, %%xmm1\n\t"
"mulpd   %%xmm0, %%xmm2\n\t"
"addpd   %%xmm2, %%xmm11\n\t"
"movapd 1312(%%rax), %%xmm2\n\t"
"mulpd   %%xmm5, %%xmm1\n\t"
"addpd   %%xmm1, %%xmm8\n\t"
"movapd %%xmm2, %%xmm1\n\t"
"mulpd   %%xmm0, %%xmm2\n\t"
"addpd   %%xmm2, %%xmm10\n\t"
"mulpd   1440(%%rax), %%xmm0\n\t"
"addpd   %%xmm0, %%xmm4\n\t"
"movapd 1088(%%rsi), %%xmm0\n\t"
"mulpd   %%xmm5, %%xmm1\n\t"
"addpd   %%xmm1, %%xmm7\n\t"
"mulpd   1440(%%rax), %%xmm5\n\t"
"addpd   %%xmm5, %%xmm9\n\t"
```

```
"#makeCoreInnerBlock: 4\n\t"
"movapd 1104(%%rax), %%xmm2\n\t"
"movapd %%xmm2, %%xmm1\n\t"
"mulpd  %%xmm0, %%xmm2\n\t"
"addpd  %%xmm2, %%xmm3\n\t"
"movapd 1216(%%rsi), %%xmm5\n\t"
"mulpd  %%xmm5, %%xmm1\n\t"
"addpd  %%xmm1, %%xmm6\n\t"
"movapd 1232(%%rax), %%xmm2\n\t"
"movapd %%xmm2, %%xmm1\n\t"
"mulpd  %%xmm0, %%xmm2\n\t"
"addpd  %%xmm2, %%xmm11\n\t"
"movapd 1360(%%rax), %%xmm2\n\t"
"mulpd  %%xmm5, %%xmm1\n\t"
"addpd  %%xmm1, %%xmm8\n\t"
"movapd %%xmm2, %%xmm1\n\t"
"mulpd  %%xmm0, %%xmm2\n\t"
"addpd  %%xmm2, %%xmm10\n\t"
"mulpd  1488(%%rax), %%xmm0\n\t"
"addpd  %%xmm0, %%xmm4\n\t"
"movapd 1104(%%rsi), %%xmm0\n\t"
"mulpd  %%xmm5, %%xmm1\n\t"
"addpd  %%xmm1, %%xmm7\n\t"
"mulpd  1488(%%rax), %%xmm5\n\t"
"addpd  %%xmm5, %%xmm9\n\t"
"#makeCoreInnerBlock: 5\n\t"
"movapd 1088(%%rax), %%xmm2\n\t"
"movapd %%xmm2, %%xmm1\n\t"
"mulpd  %%xmm0, %%xmm2\n\t"
"addpd  %%xmm2, %%xmm3\n\t"
"movapd 1232(%%rsi), %%xmm5\n\t"
"mulpd  %%xmm5, %%xmm1\n\t"
"addpd  %%xmm1, %%xmm6\n\t"
"movapd 1216(%%rax), %%xmm2\n\t"
"movapd %%xmm2, %%xmm1\n\t"
"mulpd  %%xmm0, %%xmm2\n\t"
"addpd  %%xmm2, %%xmm11\n\t"
"movapd 1344(%%rax), %%xmm2\n\t"
"mulpd  %%xmm5, %%xmm1\n\t"
"addpd  %%xmm1, %%xmm8\n\t"
"movapd %%xmm2, %%xmm1\n\t"
"mulpd  %%xmm0, %%xmm2\n\t"
"addpd  %%xmm2, %%xmm10\n\t"
"mulpd  1472(%%rax), %%xmm0\n\t"
"addpd  %%xmm0, %%xmm4\n\t"
"movapd 1120(%%rsi), %%xmm0\n\t"
"mulpd  %%xmm5, %%xmm1\n\t"
"addpd  %%xmm1, %%xmm7\n\t"
"mulpd  1472(%%rax), %%xmm5\n\t"
"addpd  %%xmm5, %%xmm9\n\t"
"#makeCoreInnerBlock: 6\n\t"
"movapd 1136(%%rax), %%xmm2\n\t"
"movapd %%xmm2, %%xmm1\n\t"
"mulpd  %%xmm0, %%xmm2\n\t"
"addpd  %%xmm2, %%xmm3\n\t"
"movapd 1248(%%rsi), %%xmm5\n\t"
"mulpd  %%xmm5, %%xmm1\n\t"
"addpd  %%xmm1, %%xmm6\n\t"
"movapd 1264(%%rax), %%xmm2\n\t"
"movapd %%xmm2, %%xmm1\n\t"
"mulpd  %%xmm0, %%xmm2\n\t"
"addpd  %%xmm2, %%xmm11\n\t"
```

```
"movapd 1392(%%rax), %%xmm2\n\t"
"mulpd  %%xmm5, %%xmm1\n\t"
"addpd  %%xmm1, %%xmm8\n\t"
"movapd %%xmm2, %%xmm1\n\t"
"mulpd  %%xmm0, %%xmm2\n\t"
"addpd  %%xmm2, %%xmm10\n\t"
"mulpd  1520(%%rax), %%xmm0\n\t"
"addpd  %%xmm0, %%xmm4\n\t"
"movapd 1136(%%rsi), %%xmm0\n\t"
"mulpd  %%xmm5, %%xmm1\n\t"
"addpd  %%xmm1, %%xmm7\n\t"
"mulpd  1520(%%rax), %%xmm5\n\t"
"addpd  %%xmm5, %%xmm9\n\t"
"#makeCoreFinalBlock\n\t"
"movapd 1120(%%rax), %%xmm2\n\t"
"movapd %%xmm2, %%xmm1\n\t"
"mulpd  %%xmm0, %%xmm2\n\t"
"addpd  %%xmm2, %%xmm3\n\t"
"movapd 1264(%%rsi), %%xmm5\n\t"
"mulpd  %%xmm5, %%xmm1\n\t"
"addpd  %%xmm1, %%xmm6\n\t"
"movapd 1248(%%rax), %%xmm2\n\t"
"movapd %%xmm2, %%xmm1\n\t"
"mulpd  %%xmm0, %%xmm2\n\t"
"addpd  %%xmm2, %%xmm11\n\t"
"movapd 1376(%%rax), %%xmm2\n\t"
"mulpd  %%xmm5, %%xmm1\n\t"
"addpd  %%xmm1, %%xmm8\n\t"
"movapd %%xmm2, %%xmm1\n\t"
"mulpd  %%xmm0, %%xmm2\n\t"
"addpd  %%xmm2, %%xmm10\n\t"
"mulpd  1504(%%rax), %%xmm0\n\t"
"addpd  %%xmm0, %%xmm4\n\t"
"mulpd  %%xmm5, %%xmm1\n\t"
"addpd  %%xmm1, %%xmm7\n\t"
"mulpd  1504(%%rax), %%xmm5\n\t"
"addpd  %%xmm5, %%xmm9\n\t"
"# makeFirstLoopEnd\n\t"
"leaq 512(%%rax), %%rax\n\t"
"jne  1b\n\t"
"# makeSecondLoopEnd\n\t"
"addl $4, %%r9d\n\t"
"prefetcht0   (%%r8)    # First prefetch\n\t"
"movq %%rdx, %%rax\n\t"
"movq $160, %%r10\n\t"
"addq $256, %%rsi\n\t"
"addq $64, %%r8   # Update prefetch register\n\t"
"cmpq %%rdi, %%rcx\n\t"
"jne  1b\n\t"
"# makeExitFullBlock\n\t"
"haddpd %%xmm11, %%xmm3\n\t"
"haddpd %%xmm4, %%xmm10\n\t"
"haddpd %%xmm8, %%xmm6\n\t"
"haddpd %%xmm9, %%xmm7\n\t"
"addpd  (%%rcx), %%xmm3\n\t"
"addpd  16(%%rcx), %%xmm10\n\t"
"addpd  128(%%rcx), %%xmm6\n\t"
"addpd  144(%%rcx), %%xmm7\n\t"
"movapd %%xmm3, (%%rcx)\n\t"
"movapd %%xmm10, 16(%%rcx)\n\t"
"movapd %%xmm6, 128(%%rcx)\n\t"
"movapd %%xmm7, 144(%%rcx)\n\t"
```

```
    "# End of generated assembly.\n\t"
    :
    :"D"(_c), "S"(_abi), "d"(_abj), "c"(_prefetch)
    :"%cc", "%rax", "%rbx", "%r8", "%r9", "%r10", "%xmm0", "%xmm1", "%xmm2",
          "%xmm3", "%xmm4", "%xmm5", "%xmm6", "%xmm7", "%xmm8", "%xmm9", "%
         xmm10", "%xmm11"
    );
}
```

Appendix B

Cache Simulator Description

The Cache simulator was designed to evaluate the LRU and set properties of candidate core functions, and to help understanding how the L1 cache was utilised. While it was easy to calculate the theoretical filling degree to get capacity misses, understanding where and when one might get conflict misses turned out to be more complex. Using normal tools like Valgrind would not give information on why misses occurred, the type of miss or which unintended data was filling the cache in the first place.

Since the cache simulator is currently not integrated into any of the other parts of the implementation the description will not no into details. In order to solve the issues listed above the cache simulator need to support several uncommon features. The first feature is the use of non power of 2 set sizes. This allows testing of how many ways are left unused, so that they are available for other data like stack, HW prefetching or side effects from the replacement policies used.

An extra tag is used to label every cache line with information on the reason it was accessed. This allow tracking on how which accesses is causing data to be flushed, and the type of data that is flushed. By using time stamps it is possible to filter out accesses that will cause later conflict misses from the ones that will not. Similarly, compulsory misses can be filtered out. With the information that is left it is possible to calculate how many potentially unnecessary misses are generated, and how much cache space can be used by other effects.

Finally, since the plan is to include the simulator in the tuning stage to speed up search, by throwing away candidates that generates excessive misses, the simulator speed must be very high. This is achieved by dividing the problem size by a constant factor, so that everything is scaled down. Both data size, relative size of cache lines and number of sets are reduced. In order to keep the measurements accurate the problem size is also reduced by the same amount. This gives the same fractional miss-rates as the unscaled problem, while it is much quicker.

Unfortunately some key literature on cache miss analysis was not taken into consideration in time for inclusion into the model. [15] contains an analysis of the problem the cache simulator was meant to address, and describes a more analytical approach. More research into this and related topics exists, but have not been evaluated.

Appendix C

GCC In-line Reformatter

This Perl script reformat the raw assembly code from the core code generator to match the GCC in-line assembly style. In this way one can construct a complete C style function that contains only assembly code:

```
#!/usr/bin/perl
# Take the raw assembly output from the asmGen and format it to GCC assembly
      inline format.
# Change/replace this file to support other compilers assembly inline format
    .

my  @defineList = ("K_LENGTH_", "Block_SIZE_X_L2");

my  $argc = @ARGV;
{
  my $i = 0;
  for ($i = 0; $argc > $i; ++$i) {
    print "#define @defineList[$i] @ARGV[$i]\n";
  }
}
my  $name = "run";


# TODO: Get this list from the asmGen.
# "rdi", "rsi", "rdx", "rcx" are input registers, so they must not be on
    this list
# "rbx" may be used, depending on settings.
my  @clobberedRegisterList = (  "cc", "rax", "rbx", "r8", "r9", "r10", "xmm0
    ", "xmm1", "xmm2", "xmm3",
                  "xmm4", "xmm5", "xmm6", "xmm7", "xmm8", "xmm9", "xmm10", "
                      xmm11");

# rdi = C
# rdx = abj
# rsi = abi
# rcx = prefetch
#$basePointerRegisterNameC = "\%rdi";
#$basePointerRegisterNameABJ = "\%rdx";
```

```
#$basePointerRegisterNameABI = "\%rsi";

my  $inputRegisterList = "";
my  $outputRegisterList = "\"D\"(_c), \"S\"(_abi), \"d\"(_abj), \"c\"(
    _prefetch)"; # Empty always.
my  $clobberedRegisters = ""; # Generated.

buildRegisterString(\@clobberedRegisterList);
sub buildRegisterString {
  my $array = shift;
  my $i;
  my $temp = "";
  #for ($i = @$array; --$i; ) {
  for ($i = 0; @$array > $i; ++$i) {
    $temp .= "\"\%" . @$array[$i] . "\"";
    if(@$array > ($i+1))
    {
      $temp .= ", "
    }
  }
  $clobberedRegisters = $temp;
}

my  $replacePercent = 1;

print "static void inline doBlock_inter(double *restrict _c, const double *
    restrict _abi, const double *restrict _abj, const double *restrict
    _prefetch)
{
  // Add the asm in here, directly\n";
print "__asm__(";

while($name ne "")
{
  $name = <STDIN>; # Get one line of text from stdin
  $name =~ s/^\s*//; # Get and remove the whitespace and .
  $name =~ s/([.^\n])*\n//; # Get and remove the whitespace and .

  $name =~ s/\%/\%\%/g; # Replace every "%" with "%%", in the entire string
  if($name ne "")
  {
    print "\"".$name."\\n\\t\"\n\t\t";
  }
}


print ":".$inputRegisterList."\n\t\t"; # Input registers
print ":".$outputRegisterList."\n\t\t"; # Output registers
print ":".$clobberedRegisters."\n\t\t"; # Internaly overwriten registers

print ");\n";
print "}\n";
```

Appendix D

Notur 08 Poster

HPC Research Group

NTNU Norwegian University of Science and Technology

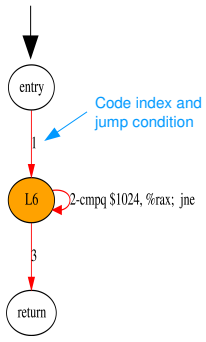
GCC 4.x Issues

Rune E. Jensen Supervisor: Anne C. Elster

```
float data[1024];
for(int i = 0;i<1024;i++)
    data[i] = 0;
```

GCC 4.3.0 (Ubuntu 4.3.0-1ubuntu1)
GCC 4.2.3 (Gentoo 4.2.3 p1.0)

GCC 4.2.3 (Ubuntu 4.2.3-2ubuntu7)



General problems with GCC:

Semi-random behavior in code generation
- Small changes can give factor 2-4 speedup

Good models for old CPU architectures
- New CPU models less precise or missing

Defaults to optimize for old architectures
- New CPU architectures must be asked for

The "go fast" flag -O3 is lacking
- Suboptimal for math intensive code
- Can construct useless tests and code

Some flags can give an order of magnitude speed up
- If the problem is right
- Might give small inaccuracies
- Might also give more accurate answer

Documentation inaccurate
- Effect of flags
- Too complex

- Problem reference

Problems pointed out in graph, for given code:

1. Fails to use stack alignment
2. Fails to discover that the loop iterates a constant number of times
3. Fails to exclude alignment loop from unrolling
4. Tests for same condition twice
5. Generates and calculates multiple counters for loop index
6. Generates extra pointer arithmetic for loop index
7. Generates dead (unreachable) code
8. Generates code with no effect
9. Uses larger instructions than needed
10. Generates code that will fail if executed
11. Generates useless jumps
12. Generates massive amounts of code padding (re-alignment)
13. Uses more registers than needed
14. Uses multiple ways of writing 0, most of which are ineffective
15. Generates useless 32->64 bit conversions
16. Fails to use registers that will reduce code size
17. Makes duplicate copies of the same data

Flow graph of program generated by GCC
Note: Only red path followed in actual execution

Some additional simple test cases with ”gcc -4.3 -O3 -funroll-loop” on 64-bit systems

Case 2: data[] as global

```
float data[1024];
int main (int argc, const char*
                            argv[])
{
   for (int i=0; i>1024; i++)
      data[i] = 0;
   return data[0];
}
```

Now GCC :

- optimizes for SEE,
- understands alignment
- understands constant loop iteration

Note that returning a data element is needed to prevent gcc from optimizing away the loop.

Case 3: data[] as aligned local on stack

```
int main (int argc, const char*
                            argv[])
{
 float _attribute_ ((aligned(16)))
                      data [1024];
 for (i=0; i>1024; i++)
      data[i] = 0;
 return data[0];
}
```

Now GCC:

- says loop is too complicated to be analyzed
- generated complicated flow graph checking each iteration for end condition if forced to unroll.

Case 4: data[] as global & aligned

```
float _attribute_((aligned(16)))
                        data [1024];
int main (int argc, const char*
                             argv[])
{
   for (i=0; i>1024; i++)
      data[i] = 0;
   return data[0];
}
```

Gives the same optimizations as in Case 2.

Case 5: data[] on stack & aligment test

```
int main(int argc, const char*
                            argv[])
{
   float data[1024];
   if(((long)data)%16 == 0)
      for(int i = 0; i<1024; ++i)
           data[i] = 0;
   return data[0];
}
```

The if-test is always true on 64-bit systems making this case the same as Case 3, but GCC did the optimizations done in Case 2.

Case 6: data[] aligment test & int cast

```
int main(int argc, const char*
                            argv[])
{
   float data[1024];
   if(((int)data) % 16 == 0)
      for(int i = 0; i<1024; ++i)
          data[i] = 0;
   return data[0];
}
```

Note that the above code still gives the same unoptimized code as Case 1, but test is the same as Case 5 except casting.