



TECHNISCHE
UNIVERSITÄT
WIEN

VIENNA
UNIVERSITY OF
TECHNOLOGY

Dissertation

Automatic SIMD Vectorization

ausgeführt zum Zwecke der Erlangung des akademischen Grades
eines Doktors der technischen Wissenschaften
unter der Leitung von

Ao. Univ.-Prof. Dipl.-Ing. Dr. techn. Christoph W. Überhuber
E101 – Institut für Analysis und Scientific Computing

eingereicht an der Technischen Universität Wien
Fakultät für Informatik

von

Dipl.-Ing. Jürgen Lorenz

Matrikelnummer 9525135

Zenogasse 3/4

1120 Wien

Wien, am 16. März 2004

Kurzfassung

Automatisch generierter numerischer Quellcode kann von handelsüblichen Compilern nicht optimal verarbeitet werden. Algorithmische Strukturen werden nur unzureichend erkannt und dementsprechend schlecht ausgenutzt. Es werden daher spezialisierte Compilertechniken benötigt um zu einem zufriedenstellend effizienten Zielcode zu gelangen.

Die immer weiter fortschreitende Zunahme der Bedeutung von Multimedia-Anwendungen hat zur Entwicklung von SIMD-Befehlssatzerweiterungen auf allen gängigen Mikroprozessoren geführt. Die konventionellen Vektorisierungsmethoden der handelsüblichen Compiler sind jedoch nicht in der Lage, Parallelismus in einem längeren numerischen Straight-Line-Code zu erkennen.

In dieser Dissertation werden Techniken, die speziell die Übersetzung numerischer Routinen zum Ziel haben, vorgestellt. Die *automatische 2-fach-SIMD-Vektorisierung* entdeckt den in einer Routine vorhandenen SIMD-Parallelismus und garantiert einen zufriedenstellenden Grad der Nutzung von SIMD-Befehlen. Die *Gucklock-Optimierung* ersetzt bestimmte Kombinationen von Instruktionen durch effizientere. Für den letzten Teil der Übersetzung werden *synthese-spezifische Techniken* eingesetzt, die deutlich zur Steigerung der Qualität des übersetzten Programmcodes – über das Maß handelsüblicher Übersetzer hinaus – beitragen.

Die in dieser Dissertation präsentierten Konzepte stellen die Grundlage des *Special Purpose Compilers MAP* [70, 71, 72, 73, 75] dar, der speziell für die effiziente Vektorisierung großer numerischer Straight-Line-Codes entwickelt wurde. MAP unterstützt die führenden automatischen Performance-Tuning-Softwareprodukte: FFTW¹, SPIRAL² und ATLAS³. Damit werden die wichtigsten Algorithmen der numerischen linearen Algebra und der digitalen Signalverarbeitung abgedeckt.

Zu den praktischen Ergebnissen dieser Dissertation zählen: (i) die schnellsten derzeit verfügbaren FFT-Codes für AMD Athlon basierte Systeme; (ii) automatisch erzeugte FFT-Codes für den Intel Pentium 4, die vergleichbare Leistung wie die schnellsten handoptimierten Codes liefern und (iii) die schnellsten derzeit verfügbaren FFT-Codes für den IBM PowerPC 440 FP2 Prozessor, der im derzeit von IBM entwickelten Supercomputer BlueGene/L Verwendung findet.

Die experimentellen Resultate wurden durch Anwendung des MAP *Special Purpose Compilers* auf von FFTW (Frigo and Johnson [35]), SPIRAL (Moura et al. [88]) und ATLAS (Whaley et al. [115]) erzeugten Code erzielt.

¹FFTW ist die Abkürzung von "Fastest Fourier Transform in the West" (schnellste Fourier-Transformation des Westens).

²SPIRAL ist die Abkürzung von "Signal Processing Algorithms Implementation Research for Adaptive Libraries" (Implementierungsforschung an Signalverarbeitungsalgorithmen für adaptive Software).

³ATLAS ist die Abkürzung von "Automatically Tuned Linear Algebra Software" (automatisch angepasste Software für Probleme der Linearen Algebra).

Summary

General purpose compilers produce suboptimal object code when applied to automatically generated numerical source code. Moreover, general purpose compilers have natural limits in deducing and utilizing information about the structure of the implemented algorithms. Specialized compilation techniques, as introduced in this thesis, are needed to realize such structural transformations.

Increasing focus on multimedia applications has resulted in the addition of short vector SIMD extensions to most existing general-purpose microprocessors. This added functionality comes primarily with the addition of short vector SIMD instructions. Unfortunately, access to these instructions is limited to proprietary language extensions, in-line assembly, and library calls. Generally, it has been assumed that vector compilers provide the most promising means of exploiting multimedia instructions. But although vectorization technology is well understood, it is inherently complex and fragile. Conventional vectorizers are incapable of locating SIMD style parallelism within a basic block without introducing unacceptably large overhead. Without the adoption of SIMD extensions, 50 % to 75 % of a machine's possible performance is wasted, even in conjunction with state-of-the-art performance tuning software. Automatic exploitation of SIMD instructions therefore requires new compiler technology.

This thesis presents techniques tailor-made for the compilation of numerical straight line code: (i) *Automatic 2-way SIMD vectorization* extracts SIMD parallelism out of numerical straight line code in a way that guarantees a satisfactory level of SIMD utilization. (ii) *Peephole optimization* rewrites the SIMD vectorized code in a domain specific manner to further improve efficiency. (iii) *Backend specific techniques* are used for compiling the vectorized and optimized code, yielding object code whose performance is significantly better than the performance of code produced by general purpose compilers.

The new compiler techniques presented in this thesis form the basis of the "MAP Vectorizer and Backend" [70, 71, 72, 73, 75], a special purpose compiler, which was designed especially for the efficient vectorization and backend optimization of large basic blocks of numerical straight line code. MAP is currently applicable to the state-of-the-art performance tuning systems FFTW⁴, SPIRAL⁵ and ATLAS⁶. Thus, MAP is covering a broad range of highly important algorithms in the fields of digital signal processing and numerical linear algebra.

The experimental results were obtained using the MAP vectorizer and backend in connection with FFTW [35], SPIRAL [88] and ATLAS [115].

⁴FFTW is the abbreviation of "Fastest Fourier Transform in the West".

⁵SPIRAL is the abbreviation of "Signal Processing Algorithms Implementation Research for Adaptive Libraries".

⁶ATLAS is the abbreviation of "Automatically Tuned Linear Algebra Software".

Contents

Kurzfassung / Summary	2
Introduction	8
1 Hardware vs. Algorithms	14
1.1 Current Hardware Trends	14
1.2 Performance Implications	16
1.3 Automatic Performance Tuning	21
2 Standard Hardware	27
2.1 Processors	27
2.2 Advanced Architectural Features	31
2.3 The Memory Hierarchy	33
3 Short Vector Hardware	36
3.1 Short Vector Extensions	37
3.2 Intel's Streaming SIMD Extensions	42
3.3 AMD's 3DNow!	47
3.4 The IBM BlueGene/L Supercomputer	48
3.5 Vector Computers vs. Short Vector SIMD	48
4 Fast Algorithms for Linear Transforms	52
4.1 Discrete Linear Transforms	52
5 Matrix Multiplication	58
5.1 The Problem Setting	58
6 A Portable SIMD API	61
6.1 Definition of the Portable SIMD API	62
7 Automatic Vectorization of Straight-Line Code	67
7.1 Vectorization of Straight Line Code	68
7.2 The Vectorization Approach	68
7.3 Benefits of The MAP Vectorizer	69
7.4 Virtual Machine Models	70
7.5 The Vectorization Engine	75
7.6 Pairing Rules	83

8	Rewriting and Optimization	93
8.1	Optimization Goals	94
8.2	Peephole Optimization	94
8.3	Transformation Rules	96
8.4	The Scheduler	101
9	Backend Techniques for Straight-Line Code	107
9.1	Optimization Techniques Used in MAP	107
9.2	Backend Optimization Goals	108
9.3	One Time Optimizations	109
9.4	Feedback Driven Optimizations	114
10	Experimental Results	118
10.1	Experimental Layout	118
10.2	BlueGene/L Experiments	119
10.3	Experiments on IA-32 Architectures	121
	Conclusion and Outlook	129
A	The Kronecker Product Formalism	130
A.1	Notation	131
A.2	Extended Subvector Operations	134
A.3	Kronecker Products	134
A.4	Stride Permutations	138
A.5	Twiddle Factors and Diagonal Matrices	142
B	Compiler Techniques	144
B.1	Register Allocation and Memory Accesses	144
B.2	Instruction Scheduling	145
C	Performance Assessment	147
C.1	Short Vector Performance Measures	149
C.2	Empirical Performance Assessment	149
D	Short Vector Instruction Set	157
D.1	The Intel Streaming SIMD Extensions 2	157
E	The Portable SIMD API	161
E.1	Intel Streaming SIMD Extensions 2	161
E.2	BG/L Double FPU SIMD Extensions	163
F	SPIRAL Example Code	167
F.1	Scalar C Code	167
F.2	Short Vector Code	170

G FFTW Example Code	175
G.1 Scalar C Code	175
G.2 Short Vector Code	176
H ATLAS Example Code	179
H.1 Scalar C Code	179
H.2 Short Vector Code	181
Table of Abbreviations	186
Bibliography	187
Curriculum Vitae	197

Acknowledgements

First of all, I would like to express my sincere gratitude to my Ph.D. advisor Christoph Ueberhuber, who gave me the opportunity of working in the ASCOT team operating in the interesting research field of scientific computing. I especially want to express my gratitude for leading my first steps in the area of high performance computing and for giving me the possibility to work at the forefront of scientific research. He supported my efforts with great dedication, not only during the work on my thesis, but also throughout the time of my diploma thesis. Finally I would like to thank him for spending enjoyable evenings talking about interesting this and that apart from work.

Very, very special thanks go to my colleague and friend Peter Wurzinger, with whom I have spent countless rewarding programming afternoons in nerd friendly atmosphere.

Franz Franchetti deserves great appreciation for his support and guidance throughout my work. His ideas and comments were invaluable. Without his effort of regulating my weird thoughts to their true semantically meaning, the writing style of this document and especially this work would not be as they are.

I truly want to thank the outstanding Stefan Kral, the most discriminating hacker I have ever met, for valuable discussions and for “gently” forcing me into avantgarde programming. Without his help and private advice, this work could not have been done at all.

I would like to thank all the people from the AURORA Project 5 and the Institute for Analysis and Scientific Computing at the Vienna University of Technology for having made the work on my thesis a rewarding and enjoyable experience.

In addition, I would like to acknowledge the financial support of the Austrian Science Fund FWF.

I want to thank all my friends, too many to list all of them, for their friendship and support.

Above all, I want to thank my family—my parents, grandparents and my brother—for always supporting me and for giving me guidance. I want to thank my parents for giving me the opportunity to study and for their support in all the years. I especially want to thank my mother for always believing in me.

Finally, I am grateful to Johann Sebastian Bach and John Coltrane for their invaluable music.

JÜRGEN LORENZ

Introduction

A few years ago major vendors of *general purpose* microprocessors have started to include short vector single instruction, multiple data (SIMD) extensions into their instruction set architecture (ISA) with the main objective of improving the performance of multi-media applications. The fundamental idea behind SIMD architectures is that a single instruction operates concurrently on several data elements using several functional units, traditionally located on different processors.

Examples of SIMD extensions supporting both integer and floating-point operations include Intel's streaming SIMD extensions (SSE, SSE 2, and SSE 3), AMD's 3DNow! as well as its successors "enhanced 3DNow!" and "3DNow! professional", Motorola's AltiVec, and last but not least IBM's Double FPU floating-point unit for BlueGene/L supercomputers.

Drawbacks of General Purpose Compilers. Although the newly introduced SIMD instruction set extensions have the potential for outstanding speed-up, a significant obstacle to their application in signal processing and scientific computing has been the lack of compilers producing code comparable to carefully hand-optimized code, in terms of overall floating-point performance. Not fully exploiting the available short vector SIMD extensions wastes 50 % to 75 % of a processor's capabilities.

Automatic Performance Tuning. Extensive experience shows that in numerical linear algebra and in digital signal processing, high performance is achieved either by extremely costly hand optimization or by automatic performance tuning—a new software paradigm in which optimized code for numerical computation is generated automatically [35, 88, 115].

All high performance numerical vendor-supplied libraries (Intel MKL, IBM ESSL, Apple vDSP, ...) are hand-coded and/or hand-tuned to utilize SIMD extensions to a satisfactory degree. They have to be adapted to each new hardware generation by hand, thus they are not performance portable.

Apart from automatic performance tuning software, there is currently no other type of portable high-performance numerical software. However, at the moment, automatic performance tuning software does not include proper support for SIMD extensions. For example, FFTW 2—the de-facto standard for portable high performance FFT computation—does not include support for short vector SIMD extensions. This shortcoming demonstrates the need of short vector SIMD support in conjunction with numerical performance tuning software.

New Compiler Technology. This thesis presents new techniques for the compilation of automatically generated code coming out of automatic performance tuning systems like FFTW, SPIRAL, and ATLAS. The techniques introduced in

this thesis include automatic 2-way SIMD vectorization of numerical straight line code, domain-specific peephole optimization, and compiler backend optimization (see [71], [76] and [26]).

The newly introduced concepts are realized in the *MAP compiler*, which is capable of generating SIMD vectorized code for digital signal processing algorithms like FFTs, DCTs and DSTs, as well as algorithms in numerical linear algebra like matrix multiplication.

MAP achieves the same level of performance as hand-tuned vendor libraries while providing portability (see Chapter 10). Combined with leading-edge self-tuning numerical software—FFTW, SPIRAL, and ATLAS—it produces

- (i) the fastest FFTs running on x86 architecture machines (Intel and AMD processors) for both real and complex FFTs and for arbitrary vector sizes;
- (ii) the only FFT routines supporting IBM's Power PC 440 FP2 double FPU used in BlueGene/L machines (see [25] and Section 3.4);
- (iii) the only automatically tuned vectorized implementations of DSP transforms (including discrete sine and cosine transforms, Walsh-Hadamard transforms, and multidimensional transforms);
- (iv) the only fully automatically vectorized ATLAS kernels.

This thesis points out the following major issues: (i) SIMD vectorization cannot be achieved easily. Nonstandard compiler techniques are required to obtain automatic performance tuning systems featuring satisfactory performance for processors with SIMD extensions. (ii) Performance portability across platforms and processor generations is not a straightforward matter, especially in the case of short vector SIMD extensions. Even the members of a family of binary compatible processors featuring the same short vector SIMD extension are different and adaptation is required to utilize them satisfactorily. (iii) Most standard vectorizing compilers are not able to deliver competitive performance due to the structural complexity of discrete linear transforms algorithms. (iv) Conventional vector computer libraries optimized for dealing with long vector lengths do not achieve satisfactory performance on short vector SIMD extensions.

Related Work

As the implementation of short vector SIMD extensions in general purpose processors is a relatively new accomplishment, only few mathematical libraries and vectorizing compilers providing SIMD support are available.

Numerical Libraries with SIMD Support

A radix-4 FFT implementation for the NEC V80R DSP processor featuring a 4-way integer SIMD extension has been presented in 1999 [68]. Apple Computers Inc. included the vDSP library with Altivec support into their operating system

MAC OS X [11, 17]. Intel's⁷ math kernel library (MKL) and performance primitives (IPP) provide support for SSE, SSE 2, SSE 3, and the Itanium processor family [62]. An SSE split-radix FFT implementation has been published in an Intel application note [49]. SIMD-FFT [101] is a radix-2 FFT implementation for SSE and AltiVec.

ATLAS dgemm kernels featuring (partially hand-coded) SIMD support are available for several problem sizes and machines [1]. LIBSIMD [92] is an ongoing effort to develop a portable library utilizing short vector SIMD extensions.

Symbolic FFT Vectorization

The Kronecker product formalism (see Appendix A) was introduced in the early 1990s as a tool for developing FFT algorithms for parallel and vector machines [65]. The SCIPOrt library [78] is a portable Fortran implementation of the proprietary Cray SCILIB library targeted at traditional vector computers. A SPIRAL based approach (i.e., SPIRAL-SIMD) to portably vectorize discrete linear transforms utilizing structural knowledge is presented in [24, 27, 28, 29, 31]. The method utilizes structural knowledge that is available on the *algorithm level* only. The vectorization algorithm translates DSP algorithms into vectorized DSP algorithms suitable for short vector SIMD architectures by means of algebraic algorithm transformations using the Kronecker product formalism. A vectorized algorithm can be mapped to any of the short vector architectures currently available. The vectorization algorithm works for arbitrary lengths of the data vectors and is parameterized by the short vector SIMD architecture's vector length ν . The methods were incorporated into FFTW and SPIRAL.

Compiler Vectorization and Backends

Vectorizing compilers have to be divided into compilers targeted at (i) "classic" loop vectorization, and (ii) basic block vectorization. Krall and Lelait [77] give a good overview of loop vectorization techniques, as well as the implementation of vectorization by loop unrolling.

Loop Vectorization. Established vectorization techniques mainly focus on finding loop-constructs which can be vectorized. If vectorization is possible, compiler known functions, i.e., intrinsics, are inserted into the code through language extensions by the compiler. However, there is a strong need for complex techniques for analyzing the source program. Additionally, the inserted instructions have a great impact on the code generation steps following the vectorization process.

Intel's C++ compiler [50] and Codeplay's VECTOR C compiler [15] are able to vectorize loop code for both integer and floating-point short vector extensions.

⁷<http://developer.intel.com>

An upgrade to the SUIF⁸ compiler that vectorizes loop code for MMX is described in [104]. The compiler identifies parallel sections of the code using scalar and array dependence analysis. The enhanced compiler applies C source to source translation with inline assembly instructions for the vectorizable sections of the code. Another loop vectorization extension has been added to SUIF addressing Berkeley's Torrent MIPS-II compliant processor with vector enhancements as coprocessor (DeVries [18]).

A code generator for energy aware compilation on DSP processors using loop level vectorization for code improvement w.r.t. execution time and therefore energy consumption has been introduced in Lorenz et al. [81].

Basic Block Vectorization. Due to the complex analysis required for loop vectorization, proposed to perform the vectorization on basic blocks resulting from increased parallelism by applying loop unrolling. After that, scalar instructions whose semantics allow to be executed as SIMD instructions are *packed* into groups. Vectorization of basic blocks by loop unrolling has the advantage that the analysis is less complex compared to loop vectorization. However, for basic block vectorizers addressing a wide range of different codes, i.e., codes where no domain specific knowledge about code inherent parallelism is available, there is a high risk of increasing code size for loops which cannot be vectorized.

A vectorizing compiler exploiting *superword level parallelism* (i.e., SIMD style parallelism) has been introduced in Larsen and Amarasinghe [79]. It identifies blocks of scalar operations offering enough parallelism for vectorization and joins them with scalar code blocks not suited for vectorization via expensive data reorganization operations. IBM's XL C compiler for BlueGene/L [3] utilizes this vectorization technique.

A vectorization approach that performs a data-flow graph based *code selection* technique for media processors with support for SIMD instructions has been introduced in Leupers and Bashford [80]. This advanced approach exploits SIMD instructions in code selection for plain ANSI C code.

General purpose compilers are not capable of detecting potential parallelism in their standard, fast code selection techniques.

As this technique for basic block code selection is highly related to the vectorization techniques introduced in this thesis, a short comparison of the two approaches follows.

The main difference to the automatic vectorization approach introduced in this thesis is that (i) only consecutive load and store operations are allowed, (ii) instruction parings of different operation type (e.g., add/sub) are not allowed, and (iii) two scalar instructions are to be covered by *one* SIMD instruction.

The graph based code selection technique for processors with SIMD instruc-

⁸SUIF is the abbreviation of "Stanford University Intermediate Format", a public domain compiler that takes either Fortran or C as input language and automatically translates sequential scientific programs into parallel code for scalable parallel machines.

tions deals with suboptimal cases by allowing an incomplete graph coverage by SIMD instructions. On the other hand, the approach introduced in this thesis always demands for a complete coverage but possibly with suboptimal SIMD utilization (cf. the vectorization levels in Section 7.5).

These differences stem from the different target architectures. While the code selection technique is targeted at embedded DSP processors, the vectorization technique of this thesis addresses general purpose SIMD processors. It is generally cheap to interleave scalar instructions with SIMD instructions for DSP processors, which is not the case for processors such as AMD's K7 or Intel's Pentium 4.

Basic Block Backend. Standard compilers are intended for handwritten code normally having only short basic blocks. However, code generated by *automatic performance tuning systems* often contains very large basic blocks of loop-unrolled code.

A compiler backend for MIPS processors targeting numerical straight-line code is presented in Guo et al. [44]. The simple *farthest first* algorithm [105] has been assessed as spilling strategy for register allocation used on large basic blocks. The test codes, the experiments were carried out with, have been generated by ATLAS and SPIRAL, two state of the art automatic performance tuning systems automatically generating completely or partly unrolled code. Experiments show that the farthest first algorithm is superior when used as spilling strategy for large blocks of straight-line code.

Synopsis

This thesis introduces and discusses the MAP vectorizer and backend in detail. It consists of four main parts.

Part I: Foundations

Chapter 1 discusses the reasons why it is hard to achieve high performance implementations of numerical algorithms on current computer architectures. The three major automatic performance tuning systems for discrete linear transforms—ATLAS, FFTW and SPIRAL—are discussed.

Chapter 2 describes current hardware trends and advanced hardware features. The main focus lies on CPUs and memory hierarchies.

Chapter 3 discusses current short vector SIMD extensions and available programming interfaces.

Chapter 4 deals with fast algorithms for discrete linear transforms, i.e., matrix-vector products with specially structured matrices. The discrete Fourier transform is discussed in detail. Classical iterative and modern recursive algorithms are summarized. The mathematical approach of SPIRAL and FFTW is presented.

Chapter 5 covers the matrix-matrix multiplication in scientific computation. Differences between a straightforward implementation and the highly tuned BLAS library ATLAS are pointed out.

Chapter 6 introduces a *portable SIMD API* as a prerequisite for the implementation of the short vector algorithms presented in subsequent chapters.

Part II: The MAP Special Purpose Compiler

Chapter 7 describes a new method for the automatic 2-way SIMD vectorization of numerical straight line code. The new technique guarantees a satisfactory degree of SIMD utilization and has been successfully used to vectorize (i) complex FFT kernels of arbitrary length, (ii) real-to-halfcomplex FFT kernels of even lengths, (iii) various DSP transform kernels, and (iv) kernels used in numerical linear algebra.

Chapter 8 deals with domain-specific peephole optimization techniques used in the code generator of MAP.

Chapter 9 describes domain-specific backend optimization techniques implemented in the x86 compiler backend of MAP.

Part III: Experimental Results

Chapter 10 presents a number of experimental results. The MAP vectorizer and backend have been assessed in combination with ATLAS, FFTW and SPIRAL on a variety of target processors, including Intel's Pentium 4, AMD's Athlon, and IBM's PowerPC 440 FP2. Experimental evidence for the superior performance achievable by using the newly introduced methods is given.

Part IV: Appendices

Appendix A summarizes the mathematical framework required to express the results presented in this thesis. The Kronecker product formalism and its connection to programs for discrete linear transforms is discussed. The translation of complex arithmetic into real arithmetic within this framework is described.

Appendix B contains elementary information about the compiler techniques of register allocation and instruction scheduling.

Appendix C discusses the performance assessment of scientific software.

Appendix D summarizes the relevant parts of short vector SIMD instruction sets.

Appendix E shows the implementation of the portable SIMD API required for the numerical experiments of this thesis.

Appendix F displays example code obtained using the newly developed short vector SIMD extension for SPIRAL.

Appendix G contains codelet examples taken from the short vector SIMD version of FFTW.

Chapter 1

Hardware vs. Algorithms

The fast evolving microprocessor technology, following Moore's law, has turned standard, single processor off-the-shelf desktop computers into powerful computing devices with peak performances of, at present, several gigaflop/s. Thus, scientific problems that a decade ago required powerful parallel supercomputers, are now solvable on a PC. On a smaller scale, many applications can now be performed under more stringent performance constraints, e. g., in real time.

Unfortunately, there are several problems inherent to this development on the hardware side that make the development of top performance software an increasingly difficult task feasible only for expert programmers.

(i) Due to the memory-processor bottleneck the performance of applications depends more on the pattern, e. g., locality of data access rather than on the mere number of arithmetic operations.

(ii) Complex architectures make a performance prediction of algorithms a difficult, if not impossible task.

(iii) Most of the modern microprocessors introduce special instructions like FMA (fused multiply-add), or short vector SIMD instructions (like SSE on Pentium processors). These instructions provide superior potential speed-up but are difficult to utilize.

(iv) High-performance code, hand-tuned to a given platform, becomes obsolete as the next generation (in cycles of typically about two years) of microprocessors comes onto the market.

As a consequence, the development of top performance software, portable across architectures and time, has become one of the key challenges associated with Moore's law. As a result there has been a number of efforts recently, collectively referred to as *automatic performance tuning*, to automate the process of implementation and optimization for given computing platforms. Important examples include FFTW by Frigo and Johnson [34], ATLAS by Whaley et al. [115], and SPIRAL by Püschel et al. [98].

1.1 Current Hardware Trends

The gap between processor performance, memory bandwidth and network link bandwidth is constantly widening. Processor power grows by approximately 60 % per year while memory bandwidth is growing by a relatively modest 6 % per year. Although the overall sum of the available network bandwidth is doubling every

year, the sustained bandwidth *per link* is only growing by less than 6% per year. Thus, it is getting more and more complicated to build algorithms that are able to utilize modern (serial or parallel) computer systems to a satisfactory degree.

Only the use of sophisticated techniques both in hardware architecture and software development allows to overcome these difficulties. Algorithms which were optimized for a specific architecture several years ago, fail to perform well on current and emerging architectures. Due to the fast product cycles in hardware development and the complexity of today's execution environments, it is of utmost importance to provide users with easy-to-use self-adapting numerical software.

The development of algorithms for modern high-performance computers is getting more and more complicated due to the following facts. (i) The performance gap between CPUs, memories, and networks is widening. (ii) Hardware tricks partially hide this performance gap. (iii) Performance modelling of programs running on current and future hardware is getting more and more difficult. (iv) Non-standard processor extensions complicate the development of programs with satisfactory performance characteristics.

In the remainder of this section, these difficulties are outlined in detail.

Computing Cores

Computing power increases at an undiminished rate according to Moore's law. This permanent performance increase is primarily due to the fact that more and more non-standard computing units are incorporated into microprocessors. For instance, the introduction of *fused multiply add* (FMA) operations doubled the floating-point peak performance. The introduction of *short vector SIMD extensions* (e.g., Intel's SSE or Motorola's AltiVec) enabled the increase of the peak performance by another factor of 2 or 4.

Using standard algorithms and general purpose compiler technology, it is not possible to utilize these recently introduced hardware extensions to a satisfactory degree. Special algorithms have to be developed for high-performance numerical software to achieve an efficient utilization of modern processors.

Memory Subsystems

Memory access is getting more and more expensive relatively to computation speed. Caching techniques try to hide latency and the lack of satisfactory memory bandwidth but require locality in the algorithm's memory access patterns. Deep memory hierarchies, cache associativity and size, transaction lookaside buffers (TLBs), and automatic prefetching introduce another level of complexity. The parameters of these facilities even vary within a given computer architecture leading to an intrinsic problem for algorithm developers who try to optimize floating-point performance for a set of architectures.

Symmetrical multiprocessing introduces the problem of cache sharing as well as cache coherency and the limited memory bandwidth becomes an even more limiting factor. Non-uniform memory access on some architectures hides the complexity of distributed memory at the cost of higher latencies for some memory blocks.

1.1.1 Performance Modelling

For modern computer architectures, modelling of system characteristics and performance characterization of numerical algorithms is extremely complicated. The number of floating-point operations is no longer an adequate measure for predicting the required run time.

The following features of current hardware prevent the accurate modelling and invalidate current performance measures for a modern processor: (i) Pipelining and multiple functional units, (ii) super-scalar processors and VLIW processors, (iii) fused multiply-add (FMA) instructions, (iv) short-vector SIMD extensions, (v) branch prediction, (vi) virtual registers, (vii) multi-level set-associative caches as well as shared caches, and (viii) transaction lookaside buffers (TLBs).

As modelling of algorithms with respect to their actual run time is not possible to a satisfactory degree, the only reasonable performance assessment is an empirical run time study carried out for given problems.

Chapter 2 explains performance relevant processor features and the respective techniques in detail. Appendix C explains the methodology of run time measurement and performance assessment.

1.2 Performance Implications

This section exemplary shows the drawback of the standard approach of optimizing software to a given platform and shows that the asymptotic complexity and even the actual number of operations is no adequate performance measure.

The standard approach to obtain an optimized implementation for a given algorithm is summarized as follows.

- The algorithm is adapted to the hardware characteristics by hand, focussing, e. g., on the memory hierarchy and/or processor features.
- The adapted algorithm is coded using a high-level language to achieve portability and make the programming manageable.
- Key portions of the code may be coded by hand in assembly language to improve performance.

The complexity of current hardware and the pace of development make it impossible to produce optimized implementations which are available at or shortly after

FFT Program	Vector Length N			
	2^5	2^{10}	2^{15}	2^{20}
NAG/c60fcf	11.6	6.0	3.3	2.6
IMSL/dfftcf	2.0	1.7	2.7	3.9
Numerical Recipes/four1	2.6	2.1	2.2	3.9
FFTPACK/cfftf	1.4	1.0	2.1	4.0
Green's FFT	1.6	1.1	1.0	—
FFTW 2.1.3	1.0	1.1	1.1	1.0

Table 1.1: Slow-down factors of various FFT routines relative to the run time of the best performing routine (with factor 1.0). Performance data were obtained on one processor of an SGI Power Challenge XL (Auer et al. [12]).

a processor's release date. This section shows the run time differences resulting from the intrinsic problems.

1.2.1 Run Time vs. Complexity

For all Cooley-Tukey FFT algorithms the asymptotic complexity is $O(N \log N)$ with N being the length of the vector to be transformed. Even the constant is nearly the same for all algorithms.

Performance of Scalar Code

However, Table 1.1 shows that the run times of the corresponding programs vary tremendously. It is a summary of experiments described in Auer et al. [12] where the performance of many FFT routines was measured on various computer systems.

For instance, on one processor of an SGI Power Challenge XL, for a transform length $N = 2^5$ the function `c60fcf` of the NAG library is 11.6 *times* slower than the fastest implementation, FFTW.

For $N = 2^{10}$ `cfftf` of FFTPACK is the fastest program and `c60fcf` is six times slower while FFTW is a moderate 10% slower.

For $N = 2^{20}$ FFTW is again the fastest program. `c60fcf` is 2.6 times slower and `cfftf` of FFTPACK is four times slower than FFTW.

This assessment study shows that (i) the performance behavior of FFT programs depends strongly on the problem size, and (ii) architecture adaptive FFTs are within 10% of the best performance for all problem sizes.

Performance of Short Vector SIMD Code

The arguments in favor of architecture adaptive software become even more striking by extending this study to machines featuring short vector SIMD extensions.

Figure 1.1 shows the performance of various one dimensional complex FFT routines (SIMD and scalar) on one CPU of a prototype of BlueGene/L's PowerPC 440 FP2 running at 500 MHz featuring a *double FPU*. The *double FPU* provides two-way vector extensions (for single-precision and double-precision, respectively) resulting in a theoretical speed-up of two.

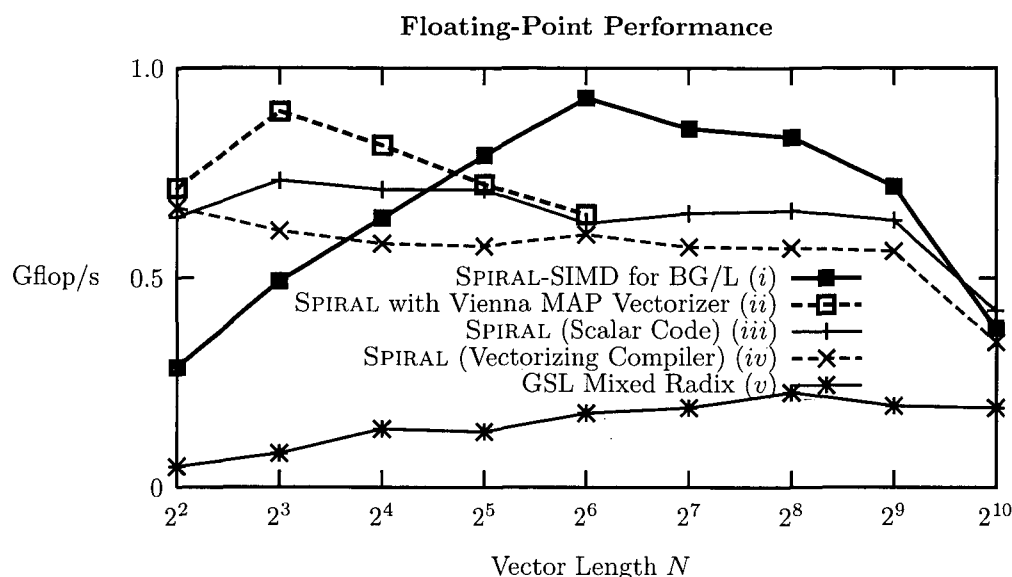


Figure 1.1: FFT Floating-point performance and speed-up of the vectorization techniques applied by the MAP vectorizer for BG/L and SPiRAL-SiMD for BG/L (formal vectorization) compared to the best scalar code and the best vectorized code (utilizing the VisualAge XL C for BG/L vectorizing compiler) found by SPiRAL. Performance is displayed in *pseudo Gflop/s* ($5N \log N / \text{runtime}$ with N being the vector length; Frigo and Johnson [35]).

The following scalar and vectorization techniques were tested:

The MAP Vectorizer is one of the main subjects of this thesis and its vectorization techniques will be explained in detail in Chapter 7. MAP provides vectorization of single static assignment straight-line code for two-way short vector extensions as a source-to-source transformation. The source codes to be vectorized may contain array accesses, index computation, and arithmetic operations. Typically, such source codes consist of thousands of arithmetic operations. In addition to SIMD vectorization, MAP provides support for FMA instructions.

MAP was adapted to support BlueGene/L's double FPU and connected to the SPiRAL system to provide BlueGene/L specific vectorization of SPiRAL generated code. SPiRAL's SPL compiler was used to translate formulas generated by the formula generator into fully unrolled implementations leading to large straight-line codes. These codes were subsequently vectorized by MAP and thus transformed into a C program where all arithmetic operations and memory access operations are expressed using XL C's intrinsic

functions. Finally the codes were compiled utilizing the XL C compiler with vectorization turned off. A detailed assessment of the MAP Vectorizer can be found in Chapter 10.

SPIRAL-SIMD. A SIMD vectorizing version of SPIRAL's SPL compiler that is portable across different SIMD architectures was adapted to vectorize codes for FFTs [28] targeting BlueGene/L's double FPU. Certain mathematical constructs used in SPIRAL's formula representation of a DSP algorithm are mapped to vectorized code by this compiler. These constructs occur in virtually every DSP algorithm. Furthermore, in several important cases, including the FFT, the formulas generated by SPIRAL are built exclusively from these vectorizable constructs, and thus can be completely vectorized using the adapted SIMD vectorizing SPL compiler.

In addition, a formally derived "short vector FFT variant" is utilized to vectorize FFT codes [30]. This variant guarantees, that the generated FFT codes fit to the need of generic short vector SIMD extensions. The short vector FFT variant was adapted to support BlueGene's double FPU.

Utilizing both methods of formal vectorization, FFT formulas generated by SPIRAL's formula generator are translated into vector code utilizing BlueGene/L's double FPU efficiently.

SPIRAL. For scalar implementations, SPIRAL (besides FFTW) provides the currently fastest publicly available FFT programs. It delivers codes which are as fast as FFT programs specifically optimized for a given architecture. Thus, SPIRAL may serve as the baseline in Figure 1.1.

IBM XL C Compiler extracts instruction level parallelism [79]. It identifies blocks of scalar operations offering enough parallelism for vectorization and joins them with scalar code blocks not suited for vectorization via expensive data reorganization operations. Therefore, it has no domain specific knowledge about the codes to vectorize. The IBM XL C compiler supports FMA extraction.

GNU GSL Mixed Radix. The GNU scientific library GSL features sophisticated algorithms (without SIMD support) and is even much slower than the best scalar adaptive SPIRAL code. It is slower than the best SIMD vectorized codes by a factor of five and more.

Experimental Results. The presented scalar and vectorization techniques were evaluated on an early BlueGene/L prototype. Performance data of 1D FFTs with vector lengths $N = 2^2, 2^3, \dots, 2^{10}$ were obtained on a single PowerPC 440 FP2 running at 500 MHz.

In particular the following FFT implementations were tested: (i) The best vectorized code found by SPIRAL utilizing formal vectorization, (ii) the best vectorized code found by SPIRAL utilizing the MAP vectorizer, (iii) the best scalar FFT implementation found by SPIRAL (XLC's vectorizer and FMA extraction turned off), (iv) the best vectorized FFT implementation found by SPIRAL using the XLC compiler's vectorizer and FMA extraction turned on, and (v) the mixed-radix FFT implementation provided by the GNU scientific library (GSL). Fig. 1.1 displays the respective performance data.

The best scalar codes found by SPIRAL—referenced by (iii) in Fig. 1.1—serve as baseline for the assessment of the various vectorization techniques. These codes are very fast scalar implementations featuring no FMA instructions.

Formal vectorization—referenced by (i) in Fig. 1.1—provides up to 40% speed-up w.r.t. the best scalar codes generated by SPIRAL for problem sizes $N \geq 64$. Thus formal vectorization provides significant speed-up for larger problem sizes.

The MAP vectorizer—referenced by (ii) in Fig. 1.1—is restricted to problem sizes that can fully be unrolled fitting into instruction cache and the resulting code can be handled well by the XLC compiler's register allocator. For problem sizes $N \leq 32$, the MAP vectorizer provides the same level of performance as formal vectorization for larger problem sizes.

The third-party GNU GSL FFT library—referenced by (v) in Fig. 1.1—reaches about 30% of the performance of the best scalar SPIRAL generated code thus performing badly.

XLC's vectorization and FMA extraction—referenced by (iv) in Fig. 1.1—produces code 15% slower than scalar XLC without FMA extraction. Thus, the vectorization techniques to vectorize straight-line code currently used within the XLC compiler cannot handle SPIRAL generated FFT codes well.

Conclusion. The numerical experiments carried out on the BlueGene/L prototype summarized in Figure 1.1 show that automatic performance tuning in combination with the two newly developed vectorization approaches is able to speed up FFT code considerably, while vectorization by IBM's XLC compiler does not speed up the automatically generated scalar codes at all. The two vectorization approaches, the MAP vectorizer and SPIRAL-SIMD, are able to provide high-performance FFT kernels for BlueGene/L by fully utilizing the new double FPU.

These experiments provide evidence that modern (vectorizing) compilers are not able to generate fast machine code in conjunction with portable libraries. This gives an impression of how much performance can be gained by using automatic performance tuning and utilizing special processor features as short vector SIMD extensions.

The next section explains, why this performance gain is possible and gives an

short overview over current automatic performance tuning systems.

1.3 Automatic Performance Tuning

Automatic performance tuning is a step beyond standard compiler optimization. It is required to overcome the problem that today's compilers on current machines cannot produce high performance code any more as outlined in the previous section.

Automatic performance tuning is a problem specific approach and thus is able to achieve much more than general purpose compilers are capable of.

Current automatic empirical optimization systems (AEOS) focus on (i) CPU level optimizations (loop unrolling, source code scheduling, FMA utilization), and (ii) memory hierarchy utilization (loop tiling, cache blocking). Actual code runtime is used to steer the automatic optimization process.

For instance, ATLAS' search for the correct loop tiling for carrying out a matrix-matrix product is a loop transformation a compiler could in principle do (and some compilers try to), if the compiler would have an accurate machine model to deduce the correct tiling. But compilers do not reach ATLAS' performance. The same phenomenon occurs with the source code scheduling done by SPIRAL and FFTW for straight line code, which should be done satisfactorily by the target compiler.

1.3.1 Compiler Optimization

Modern compilers make extensive use of optimization techniques to improve the program's performance. The application of a particular optimization technique largely depends on a static program analysis based on simplified machine models. Optimization techniques include high level loop transformations, such as loop unrolling and tiling. These techniques have been extensively studied for over 30 years and have produced, in many cases, good results. However, the machine models used are inherently inaccurate, and transformations are not independent in their effect on performance making the compiler's task of deciding the best sequence of transformations difficult (Aho et al. [2]).

Typically, compilers use heuristics that are based on averaging observed behavior for a small set of benchmarks. Furthermore, while the processor and memory hierarchy is typically modelled by static analysis, this does not account for the behavior of the entire system. For instance, the register allocation policy and strategy for introducing spill code in the backend of the compiler may have a significant impact on performance. Thus static analysis can improve program performance but is limited by compile-time decidability.

1.3.2 The Program Generator Approach

A method of source code adaptation at compile-time is code generation. In code generation, a *code generator* (i.e., a program that produces other programs) is used. The code generator takes as parameters the various source code adaptations to be made, e.g., instruction cache size, choice of combined or separate multiply and add instructions, length of floating-point and fetch pipelines, and so on. Depending on the parameters, the code generator produces source code having the desired characteristics.

Example 1.1 (Parameters for Code Generators) In `genfft`, the codelet generator of FFTW, the generation of FMA specific code can be activated using the `-magic-enable-fma` switch. Calling `genfft` using

```
genfft 4 -notwiddle -magic-enable-fma
```

results in the generation of a no-twiddle codelet of size four which is optimized for FMA architectures.

1.3.3 Compile-Time Adaptive Algorithms Using Feedback-Information

Not all important architectural variables can be handled by *parameterized* compile-time adaptation since varying them actually requires changing the underlying source code. This brings in the need for the second method of software adaptation, compile-time adaptation by *feedback directed* code generation, which involves actually generating different implementations of the same operation and selecting the best performing one.

There are at least two different ways to proceed:

- (i) The simplest approach is to get the programmer to supply various hand-tuned implementations, and then to choose a suitable one.
- (ii) The second method is based on automatic code generation. In this approach, parameterized code generators are used. Performance optimization with respect to a particular hardware platform is achieved by searching, i.e., varying the generator's parameters, benchmarking the resulting routines, and selecting the fastest implementation. This approach is also known as *automated empirical optimization of software* (AEOS) (Whaley et al. [115]).

In the remainder of this section the existing approaches are introduced briefly.

PHiPAC

Portable high-performance ANSI C (PHiPAC) was the first system which implemented the “generate and search” methodology (Bilmes et al. [14]). Its code generator produces matrix multiply implementations with various loop unrolling

depths, varying register and L1- and L2-cache tile sizes, different software pipelining strategies, and enables other options. The output of the generator is C code, both to make the system portable and to allow the compiler to perform the final register allocation and instruction scheduling. The search phase benchmarks code produced by the generator under various options to select the best performing implementation.

ATLAS

The *automatically tuned linear algebra software* (ATLAS¹) project [102, 115, 114, 113, 112] is an ongoing research effort (at the University of Tennessee, Knoxville) focusing on empirical techniques in order to produce software having portable performance. Initially, the goal of the ATLAS project was to provide a portably efficient implementation of the BLAS. Now ATLAS provides at least some level of support for all of the BLAS, and first tentative extensions beyond this level have been taken.

While originally the ATLAS project's principle objective was to develop an efficient library, today the field of investigation has been extended. Within a couple of years new methodologies to develop self-adapting programs have become established, the AEOS approach has been established which forms a new sector in software evolution. ATLAS' adaptation approaches are typical AEOS methods; even the concept of "AEOS" was coined by ATLAS' developers (Whaley et al. [115]). In this manner, the second main goal of the ATLAS project is the general investigation in program adaptation using AEOS methodology.

ATLAS uses automatic code generators in order to provide different implementations of a given operation, and involves sophisticated search scripts and robust timing mechanisms in order to find the best way of performing this operation on a given architecture.

The remainder of this chapter introduces the two leading projects dealing with architecture adaptive implementations of discrete linear transforms, SPIRAL and FFTW. One result of this thesis is the usage of the output codes of these systems with the newly developed MAP vectorizer and backend.

FFTW

FFTW² (*fastest Fourier transform in the west*) was the first effort to automatically generate FFT code using a special purpose compiler and use to the actual run time as optimization criterion (Frigo [33], Frigo and Johnson [34, 35, 36, 37]). Typically, FFTW performs faster than publicly available FFT codes and faster to equal with hand optimized vendor-supplied libraries across different machines. It provides comparable performance to SPIRAL). Several extensions to FFTW exist,

¹available from <http://math-atlas.sourceforge.net/>

²available from <http://www.fftw.org/>

including the AC FFTW package and the UHFFT library. Currently, FFTW is the most popular portable high performance FFT library that is publicly available.

FFTW provides a recursive implementation of the Cooley-Tukey FFT algorithm. The actual computation is done by automatically generated routines called *codelets* which restrict the computation to specially structured algorithms called right expanded trees (see Section 4.1 and Haentjens [46]). The recursion stops when the remaining right subproblem is solved using a codelet. For a given problem size there are many different ways of solving the problem with potentially very different run times. FFTW uses dynamic programming with the actual run time of problems as cost function to find a fast implementation for a given DFT_N on a given machine. FFTW consists of the following fundamental parts. Details about FFTW can be found in Frigo and Johnson [35].

The Planner. At run time but as a one time operation during the initialization phase, the *planner* uses dynamic programming to find a good decomposition of the problem size into a tree of computations according to the Cooley-Tukey recursion called *plan*.

The Executor. When solving a problem, the *executor* interprets the *plan* as generated by the planner and calls the appropriate codelets with the respective parameters as required by the plan. This leads to data access patterns which respect memory access locality.

The Codelets. The actual computation of the FFT subproblems is done within the *codelets*. These small routines come in two flavors, (i) *twiddle codelets* which are used for the left subproblems and additionally handle the twiddle matrix, and (ii) *no-twiddle codelets* which are used in the leaf of the recursion and which additionally handle the stride permutations. Within a larger variety of FFT algorithms is used, including the Cooley-Tukey algorithm, the split-radix algorithm, the prime factor algorithm, and the Rader algorithm (Van Loan [109]).

The Codelet Generator *genfft*. At install time, all codelets are generated by a special purpose compiler called the *codelet generator genfft*. As an alternative the preponderated codelet library can be downloaded as well. In the standard distribution, codelets of sizes up to 64 (not restricted to powers of two) are included. But if special transform sizes are required, the required codelets can be generated.

SPIRAL

SPIRAL³ (*signal processing algorithms implementation research for adaptive libraries*) is a generator for high performance code for discrete linear transforms like the DFT, the discrete cosine transforms (DCTs), and many others by Püschel et al. [99], Moura et al. [87, 88, 89]. SPIRAL uses a mathematical approach that

³available from <http://www.ece.cmu.edu/~spiral/>

translates the implementation problem of discrete linear transforms into a search in the space of structurally different algorithms and their possible implementations to generate code that is adapted to the given computing platform. SPIRAL's approach is to represent the many different algorithms for a transform as formulas in a concise mathematical language. These formula are automatically generated and automatically translated into code, thus enabling an automated search. Chapter 4 summarizes the discrete linear transforms and Appendix A summarizes the mathematical framework.

More specifically, the SPIRAL approach is based on the following observations.

- For every discrete linear transform transform there exists a *very large* number of different *fast* algorithms. These algorithms differ in dataflow but are essentially equal in the number of arithmetic operations.
- A fast algorithm for a discrete linear transform can be represented as a *formula* in a concise mathematical notation using a small number of mathematical constructs and primitives (see Appendix A).
- It is possible to *automatically generate* the alternative formulas, i. e., algorithms, for a given discrete linear transform.
- A formula representing a fast discrete linear transform algorithm can be mapped *automatically* into a program in a high-level language like C or Fortran.

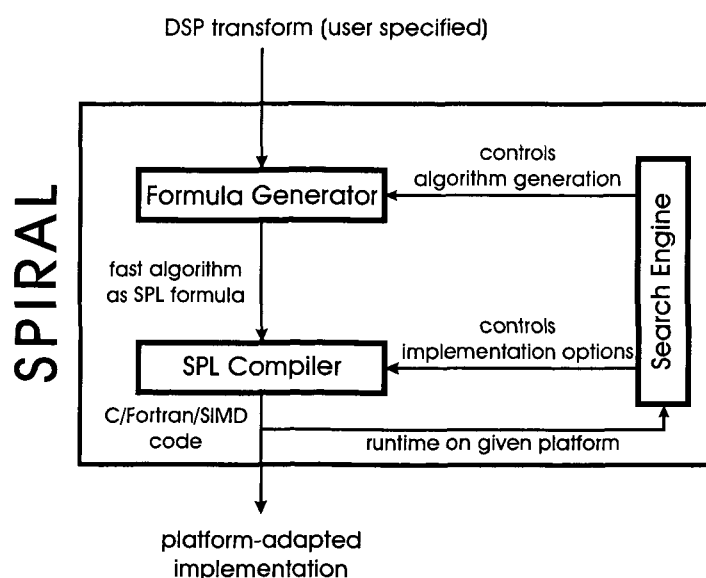


Figure 1.2: SPIRAL's architecture.

The architecture of SPIRAL is shown in Figure 1.2. The user specifies a transform to be implemented, e. g., a DFT of size 1024. The *formula generator* expands the transform into one (or several) formulas, i.e., algorithms, represented in the SPIRAL proprietary language SPL (signal processing language). The *formula translator* (also called SPL compiler) translates the formula into a C or Fortran program. The run time of the generated program is fed back into a *search engine* that controls the generation of the next formula and possible implementation choices, such as the degree of loop unrolling. Iteration of this process yields a platform-adapted implementation. Search methods in SPIRAL include dynamic programming and evolutionary algorithms. By including the mathematics in the system, SPIRAL can optimize, akin to a human expert programmer, on the implementation level *and* the algorithmic level to find the best match to the given platform. Further information on SPIRAL can be found in Püschel et al. [97], Singer and Veloso [103], Xiong et al. [116].

GAP and AREP. SPIRAL's formula generator uses AREP, a package by Egner and Püschel [22] implemented in the language GAP [107] which is a computer algebra system for computational group theory. The goal of AREP was to create a package for computing with group representations up to equality, not up to equivalence, hence, AREP provides the data types and the infrastructure to do efficient symbolic computation with representations and structured matrices which arise from the decomposition of representations.

Algorithms represented as formulas are written in mathematical terms of matrices and vectors which are specified and composed symbolically in the AREP notation. Various standard matrices and matrix types are supported such as many algebraic operations, like DFT and diagonal matrices, and the Kronecker product formalism.

One result of the work presented in this thesis will be the MAP vectorizer which allows for the automatic vectorization of computational kernels generated by self-tuning numerical software and the MAP backend, a special purpose kernel backend to translate vectorized code into high-performance assembly code. The MAP vectorizer and backend can be connected to FFTW, SPIRAL, and ATLAS, thus addressing many different numerical computations ranging from FFTs and many other DSP transforms to BLAS kernels.

Chapter 2

Standard Hardware

This chapter gives an overview over standard features of single processor systems relevant for the computation of discrete linear transforms, i. e., microprocessors and the associated memory subsystem. Sections 2.1 and 2.2 discuss the features on the processor level while Section 2.3 focusses on the memory hierarchy.

Further details can be found, for example, in Gansterer and Ueberhuber [40] or Hlavacs and Ueberhuber [47].

2.1 Processors

Due to packaging with increased density and architectural concepts like RISC, the peak performance of processors has been increased by about 60 percent each year (see Figure 2.1). This annual growth rate is likely to continue for at least another

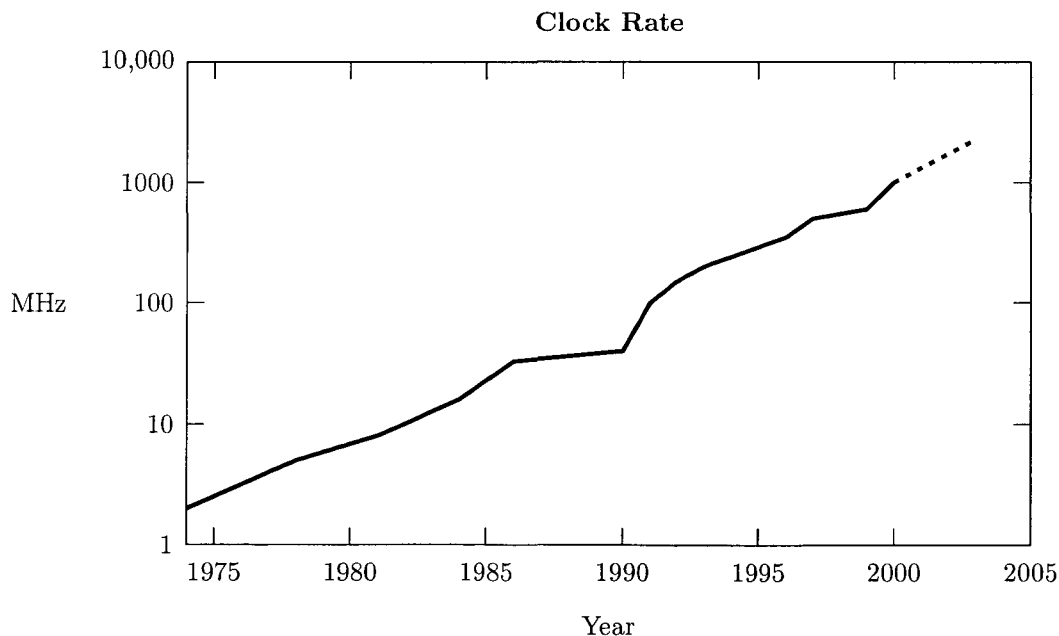


Figure 2.1: Clock rate trend for off-the-shelf CPUs.

decade. Then physical limitations like Heisenberg's principle of uncertainty will impede package density to grow.

2.1.1 CISC Processors

The term CISC is an acronym for *complex instruction set computer*, whereas RISC is an acronym for *reduced instruction set computer*. Until the 1980s, practically all processors were of CISC type. Today, the CISC concept is quite out-of-date, though most existing processors are designed to understand CISC instruction sets (like Intel x86 compatibles). Sixth and seventh generation x86 compatible processors (Intel's Pentium II, III, and 4, and AMD's Athlon line) internally use advanced RISC techniques to achieve performance gains. They fetch x86 CISC style instructions from memory in order and translate them to an internal instruction format of one or more simpler fixed length RISC style instructions. Multiple RISC execution units are utilized to feed them into their multiple pipelines and other functional units.

Microcode

When designing new processors, it is not always possible to implement instructions by means of transistors and resistors only. Instructions that are executed directly by electrical components are called *hardwired*. Complex instructions, however, often require too much effort to be hardwired. Instead, they are emulated by simpler instructions inside the processor. The "program" of hardwired instructions emulating complex instructions is called *microcode*. Microcode makes it possible to emulate instruction sets of different architectures just by adding or changing ROMs containing microcode information.

Compatibility with older computers also forces vendors to supply the same set of instructions for decades, making modern processors to deal with old fashioned, complex instructions instead of creating new, streamlined instruction sets to fit onto RISC architectures.

Example 2.1 (Microcode) Intel's Pentium 4, and AMD's Athlon XP dynamically translate complex CISC instructions into one or more equivalent RISC instructions. Each CISC instruction thus is represented by a microprogram containing optimized RISC instructions.

2.1.2 RISC Processors

Two major developments paved the road to RISC processors.

High Level Languages. Due to portability and for faster and affordable software development high level languages are used instead of native assembly. Thus, optimizing compilers are needed that can create executables having an efficiency comparable to programs written in assembly language. Compilers prefer small and simple instructions which can be moved around more easily than complex instructions with more dependencies.

Performance. Highest performance has to be delivered at any cost. This goal is achieved by either increasing the packaging density, by increasing the clock rate or by reducing the cycles per instruction (CPI) count. The latter is impossible when using ordinary CISC designs. On RISC machines, instructions are supposed to take only one cycle, yet several stages were needed for their execution. The answer is a special kind of parallelism: *pipelining*. Due to the availability of cheap memory it is possible to design fixed length instruction sets, the most important precondition for smooth pipelining. Also, cheaper SRAMs are available as caches, thus providing enough memory-processor bandwidth for feeding data to faster CPUs.

RISC processors are characterized by the following features: (i) pipelining, (ii) uniform instruction length, (iii) simple addressing modes, (iv) load/store architecture, and (v) more registers.

Additionally, modern RISC implementations use special techniques to improve the instruction throughput and to avoid pipeline stalls (see Section 2.2): (i) low grain functional parallelism, (ii) register bypassing, and (iii) optimized branches. As current processors understanding the x86 CISC instruction set feature internal RISC cores, these advanced technologies are used in x86 compatible processors as well.

In the remainder of this section the features are discussed in more detail.

2.1.3 Pipelines

Pipelines consist of several stages which carry out a small part of the whole operation. The more complex the function is that a pipeline stage has to perform, the more time it needs and the slower the clock has to tick in order to guarantee the completion of one operation each cycle. Thus, designers face a trade-off between the complexity of pipeline stages and the smallest possible clock cycle. As pipelines can be made arbitrarily long, one can break complex stages into two or more separated simple ones that operate faster. Resulting pipelines can consist of ten or more stages, enabling higher clock rates. Longer pipelines, however, need more cycles to be refilled after a pipeline hazard or a context switch. Smaller clock cycles, however, reduce this additional overhead significantly.

Processors containing pipelines of ten or more stages are called *superpipelined*.

Example 2.2 (Superpipeline) The Intel Pentium 4 processor core contains pipelines of up to 20 stages. As each stage needs only simple circuitry, processors containing this core are able to run at more than 3 GHz.

2.1.4 VLIW Processors

When trying to issue more than one instruction per clock cycle, processors have to contain several pipelines that can operate independently. In *very long instruction*

word (VLIW) processors, instruction words consist of several different operations without interdependence. At run time, these basic instructions can be brought to different units where they are executed in parallel.

The task of scheduling independent instructions to different functional units is done by the compiler at compile time. Typically, such compilers try to find a good mixture of both integer and floating-point instructions to form up long instruction words.

Example 2.3 (VLIW Processor) The *digital signal processors* (DSPs) VelociTI from TI, Trimedia-1000 from Philips and FR500 from Fujitsu are able to execute long instruction words.

The 64 bit Itanium processor family (IPF, formerly called IA-64) developed in a cooperation between Intel and HP, follows the VLIW paradigm. This architecture is also called EPIC (*explicit parallel instruction computing*).

It is very difficult to build compilers capable of finding independent integer, memory, and floating-point operations for each instruction. If no floating-point operations are needed or if there are much more integer operations than floating-point operations, for example, much of this kind of parallelism is wasted. Only programs consisting of about the same number of integer and floating-point operations can exploit VLIW processors efficiently.

2.1.5 Superscalar Processors

Like long instruction word processors, superscalar processors contain several independent functional units and pipelines. Superscalar processors, however, are capable of scheduling independent instructions to different units dynamically at run time. Therefore, such processors must be able to detect instruction dependencies. They have to ensure that dependent instructions are executed in the same order they appeared in the instruction stream.

Modern superscalar RISC processors belong to the most complex processors ever built. To feed their multiple pipelines, several instructions must be fetched from memory at once, thus making fast and large caches inevitable. Also, sophisticated compilers are needed to provide a well balanced mix of independent integer and floating-point instructions to ensure that all pipelines are kept busy during execution. Because of the complexity of superscalar processors their clock cycles cannot be shortened to the same extent than in simpler processors.

Example 2.4 (Superscalar Processor) The IBM POWER 4 is capable of issuing up to 8 instructions per cycle, with a sustained completion rate of five instructions. As its stages are very complex, it runs at only 1.30 GHz.

The AMD Athlon XP 2100+ processor running at 1733 MHz features a floating-point adder and a floating-point multiplier both capable of issuing one two-way vector operation per cycle.

2.2 Advanced Architectural Features

Superscalar processors dynamically schedule instructions to multiple pipelines and other functional units. As performance is the top-most goal, all pipelines and execution units must be kept busy in order to achieve maximum performance. Dependencies among instructions can hinder the pipeline from running smoothly, advanced features have to be designed to detect and resolve dependencies. Other design features have to make sure that the instructions are executed in the same order they entered the processor.

2.2.1 Functional Parallelism

As was said before, superscalar RISC processors contain several execution units and pipelines that can execute instructions in parallel. To keep all units busy as long as possible, it must be assured that there are always instructions to execute, waiting in buffers. Such buffers are called *reservation stations* or *queues*. Every execution unit can have a reservation station of its own or get the instructions from one global queue. Also, for each instruction leaving such a station, another one should be brought from memory. Thus, memory and caches have to deliver several instructions each cycle.

Depending on the depths of the used pipelines, some operations might take longer than others. It is therefore possible that instruction $i + 1$ is finished, while instruction i is still being processed in a pipeline. Also, an integer pipeline may get idle, while the floating-point unit is still busy. Thus, if instruction $i + 1$ is an integer operation, while instruction i is a floating-point operation, $i + 1$ might be put into the integer pipeline, before i can enter the floating-point unit. This is called *out-of-order* execution. The instruction stream leaving the execution units will often differ from the original instruction stream. Thus, earlier instructions must wait in a *reorder buffer* for all prior instructions to finish, before their results are written back.

2.2.2 Registers

Registers obviously introduce some kind of bottleneck, if too many values have to be stored in registers within a short piece of code. The number of existing registers depends on the designs of the underlying architecture. The set of registers known to compilers and programs is called the *logical* register file. To guarantee software compatibility with predecessors, the number of logical registers cannot be increased within the same processor family. Programs being compiled for new processors having more registers could not run on older versions with a smaller number of registers. However, it is possible to increase the number of *physical* registers existing within the processor and to use them to store intermediate values.

2.2.3 Fused Multiply-Add Instructions

In current microprocessors equipped with fused multiply-add (FMA) instructions, the floating-point hardware is designed to accept up to three operands for executing FMA operations, while other floating-point instructions requiring fewer operands may utilize the same hardware by forcing constants into the unused operands. In general, FPUs with FMA instructions use a multiply unit to compute $a \times b$, followed by an adder to compute $a \times b + c$.

FMA operations have been implemented in the floating-point units, e.g., of the HP PA-8700+, IBM POWER4, Intel IA-64 and Motorola POWERPC microprocessors. In the Motorola POWERPC, FMA instructions have been implemented by chaining the multiplier output into the adder input requiring rounding between them. On the contrary, processors like the IBM POWER4 implement the FMA instruction by integrating the multiplier and the adder into one multiply-add FPU. Therefore in the POWER4 processor, the FMA operation has the same latency (two cycles) as an individual multiplication or addition operation. The FMA instruction has one other interesting property: It is performed with one round-off error. In other words, in $a = b \times c + d$, $b \times c$ is first computed to quadruple (128 bit) precision, if b and c are double (64 bit) precision, then d is added, and the sum rounded to a . This use of very high precision is used by IBM's RS6000 to implement division, which still takes about 19 times longer than either multiply or add. The FMA instruction may be used to simulate higher precision cheaply.

2.2.4 Short Vector SIMD Extensions

A recent trend in general purpose microprocessors is to include short vector SIMD extensions. Although initially developed for the acceleration of multi-media applications, these extensions have the potential to speed up digital signal processing kernels, especially discrete linear transforms. The range of general purpose processors featuring short vector SIMD extensions starts with the Motorola MPC G4 (featuring the AltiVec extension), [85] used in embedded computing and by Apple. It continues with Intel processors featuring SSE and SSE 2 (Pentium III and 4, Itanium and Itanium 2) [50] and AMD processors featuring different 3DNow! versions (Athlon and successors) [4]. These processors are used in desktop machines and commodity clusters. But short vector SIMD extensions are even included into the next generation of supercomputers like the IBM BG/L machine currently in development.

All these processors feature two-way or four-way floating-point short vector SIMD extensions. These extensions operate on a vector of ν floating-point numbers in parallel (where ν denotes the extension's vector length) and feature constrained memory access: only naturally aligned vectors of ν floating-point numbers can be loaded and stored efficiently. These extensions offer a high potential

speed-up (factors of up to two or four) but are difficult to use: (i) vectorizing compilers cannot generate satisfactorily code for problems with more advanced structure (as discrete linear transforms are), (ii) direct use is beyond standard programming, and (iii) programs are not portable across the different extensions.

The efficient utilization of short vector SIMD extensions for discrete linear transforms in a performance portable way is the core of this thesis. Details about short vector SIMD extensions are given in Chapter 3.

2.3 The Memory Hierarchy

Processor technology has improved dramatically over the last years. Empirical observation shows that processor performance annually increases by 60 %. RISC design goals will dominate microprocessor development in the future, allowing pipelining and the out-of-order execution of up to 6 instructions per clock cycle. Exponential performance improvements are here to stay for at least 10 years. Unfortunately, other important computer components like main memory chips could not hold pace and introduce severe bottlenecks hindering modern processors to fully exploit their power.

Memory chips have only developed slowly. Though fast *static* RAM (SRAM) chips are available, they are much more expensive than their slower *dynamic* RAM (DRAM) counterparts. One reason for the slow increase in DRAM speed is the fact that during the last decade, the main focus in memory chip design was primarily to increase the number of transistors per chip and therefore the number of bits that can be stored on one single chip.

When cutting the size of transistors in halve, the number of transistors per chip is quadrupled. In the past few years, this raising was observed within a period of three years, thus increasing the capacity of memory chips at an annual rate of 60 % which corresponds exactly to the growth rate of processor performance. Yet, due to the increasing address space, address decoding is becoming more complicated and finally will nullify any speed-up achieved with smaller transistors. Thus, memory latency can be reduced only at a rate of 6 percent per year. The divergence of processor performance and DRAM development currently doubles every six years.

In modern computers memory is divided into several stages, yielding a *memory hierarchy*. The higher a particular memory level is placed within this hierarchy, the faster and more expensive (and thus smaller) it is. Figure 2.2 shows a typical memory hierarchy

The fastest parts belong to the processor itself. The *register file* contains several processor registers that are used for arithmetic tasks. The next stages are the primary or L1-cache (built into the processor) and the secondary or L2-cache (on extra chips near the processor). Primary caches are usually fast but small. They directly access the secondary cache which is usually larger but slower.

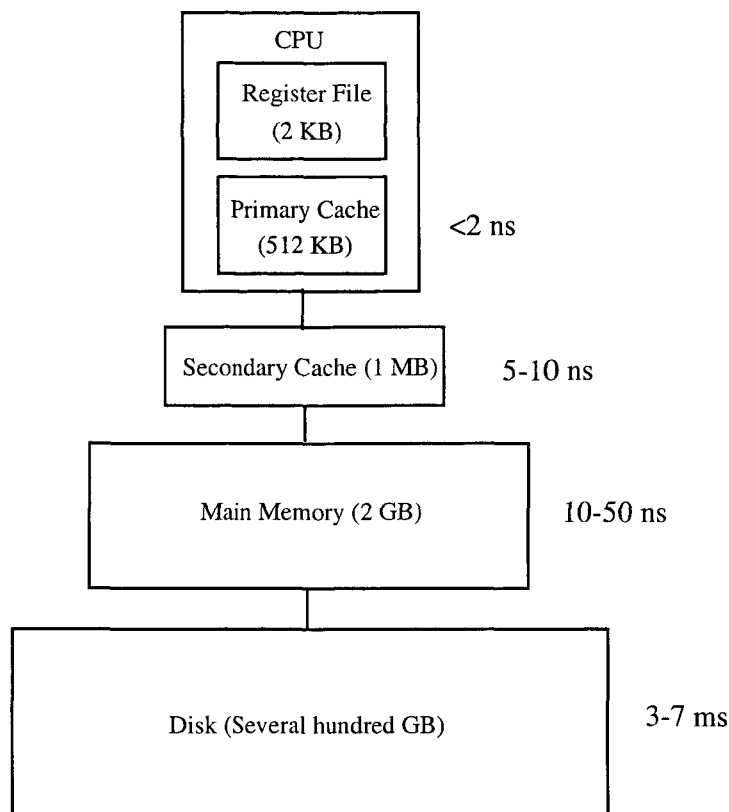


Figure 2.2: The memory hierarchy of a modern computer.

The secondary cache accesses main memory which—on architectures with virtual memory—exchanges data with disk storage. Some microprocessors of the seventh generation hold both L1- and L2-cache on chip, and have an L3-cache near the processor.

Example 2.5 (L2-Cache on Chip) Intel's Itanium processors hold both L1- and L2-cache on chip. The 2 MB large L3-cache is put into the processor cartridge near the processor.

2.3.1 Cache Memory

To prevent the processor from waiting most of the time for data from main memory, *caches* were introduced. A *cache* is a fast, but small memory chip placed between the processor and main memory. Typical cache sizes vary between 128 KB and 8 MB.

Data can be moved to and from the processor within a few clock cycles. If the processor needs data that is not currently in the cache, the main memory has to send it, thus decreasing the processor's performance. The question arises whether caches can effectively reduce main memory traffic. Two principles of locality that have been observed by most computer programs support the usage of caches:

Temporal Locality: If a program has accessed a certain data element in memory, it is likely to access this element again within a short period of time.

Spatial Locality: If a program has accessed a data element, it is likely to access other elements located closely to the first one.

Program monitoring has shown that 90 % of a program's work is done by only 10 % of the code. Thus data and instructions can effectively be buffered within small, fast caches, as they are likely to be accessed again and again. Modern RISC processors would not work without effective caches, as main memory could not deliver data to them in time. Therefore, RISC processors have built-in caches, so-called *primary* or *on-chip caches*. Many RISC processors also provide the possibility to connect to them extra cache chips forming *secondary caches*. The current generation of processors even contains this secondary cache on chip and some processors are connected to an external *tertiary cache*. Typical caches show latencies of only a few clock cycles. State-of-the-art superscalar RISC processors like IBM's POWER4 architecture have caches that can deliver several different data items per clock cycle. Moreover, the on-chip cache is often split into *data cache* and *instruction cache*, yielding the so-called *Harvard architecture*.

Cache Misses

If an instruction cache miss is detected, the whole processor pipeline has to wait until the requested instruction is supplied. This is called a *stall*. Modern processors can handle more than one *outstanding load*, i. e., they can continue execution of other instructions while waiting for some data items brought in from cache or memory. But cache misses are very expensive events and can cost several tens of cycles.

Chapter 3

Short Vector Hardware

Major vendors of general purpose microprocessors have included single instruction, multiple data (SIMD) extensions to their instruction set architectures (ISA) to improve the performance of multi-media applications by exploiting the subword level parallelism available in most multi-media kernels.

All current SIMD extensions are based on the packing of large registers with smaller datatypes (usually of 8, 16, 32, or 64 bits). Once packed into the larger register, operations are performed in parallel on the separate data items within the vector register. Although initially the new data types did not include floating-point numbers, more recently, new instructions have been added to deal with floating-point SIMD parallelism. For example, Motorola's AltiVec and Intel's streaming SIMD extensions (SSE) operate on four single-precision floating-point numbers in parallel. IBM's double FPU extension and Intel's SSE 2 and SSE 3 can operate on two double-precision numbers in parallel.

IBM's double FPU extension which is part of the BlueGene initiative and is implemented in BG/L processor prototypes is still classified and will therefore be excluded from a detailed discussion. However, this particular SIMD extension is a major target for the technology presented in this thesis.

By introducing double-precision short vector SIMD extensions this technology entered scientific computing. Conventional scalar codes become obsolete on machines featuring these extensions as such codes utilize only a fraction of the potential performance. However, SIMD extensions have strong implications on algorithm development as their efficient utilization is not straightforward.

The most important restriction of all SIMD extensions is the fact that only *naturally aligned vectors* can be accessed efficiently. Although, loading subvectors or accessing unaligned vectors is supported by some extensions, these operations are more costly than aligned vector access. On some SIMD extensions these operations feature prohibitive performance characteristics. This negative effect has been the major driving force behind the work presented in this thesis.

The intra-vector parallelism of SIMD extensions is contrary to the inter-vector parallelism of processors in vector supercomputers like those of Cray Research, Inc., Fujitsu or NEC. Vector sizes in such machines range to hundreds of elements. For example, Cray SV1 vector registers contain 64 elements, and Cray T90 vector registers hold 128 elements. The most recent members of this type of vector machines are the NEC SX-6 and the Earth Simulator.

3.1 Short Vector Extensions

The various short vector SIMD extensions have many similarities, with some notable differences. The basic similarity is that all these instructions are operating in parallel on lower precision data packed into higher precision words. The operations are performed on multiple data elements by single instructions. Accordingly, this approach is often referred to as *short vector* SIMD parallel processing. This technique also differs from the parallelism achieved through multiple pipelined parallel execution units in superscalar RISC processors in that the programmer explicitly specifies parallel operations using special instructions.

Two classes of processors supporting SIMD extensions can be distinguished: (i) Processors supporting only integer SIMD instructions, and (ii) processors supporting both integer and floating-point SIMD instructions.

The *vector length* of a short vector SIMD architecture is denoted by ν .

3.1.1 Integer SIMD Extensions

MAX-1. With the PA-7100LC, Hewlett-Packard introduced a small set of multi-media acceleration extensions, MAX-1, which performed parallel subword arithmetic. Though the design goal was to support all forms of multi-media applications, the single application that best illustrated its performance was real-time MPEG-1, which was achieved with C codes using macros to directly invoke MAX-1 instructions.

VIS. Next, Sun introduced VIS, a large set of multi-media extensions for Ultra-Sparc processors. In addition to parallel arithmetic instructions, VIS provides novel instructions specifically designed to achieve memory latency reductions for algorithms that manipulate visual data. In addition, it includes a special-purpose instruction that computes the sum of absolute differences of eight pairs of pixels, similar to that found in media coprocessors such as Philips' Trimedia.

MAX-2. Then, Hewlett-Packard introduced MAX-2 with its 64 bit PA-RISC 2.0 microprocessors. MAX-2 added a few new instructions to MAX-1 for subword data alignment and rearrangement to further support subword parallelism.

MMX. Intel's MMX technology is a set of multi-media extensions for the x86 family of processors. It lies between MAX-2 and VIS in terms of both the number and complexity of new instructions. MMX integrates a useful set of multi-media instructions into the somewhat constrained register structure of the x86 architecture. MMX shares some characteristics of both MAX-2 and VIS, and also includes new instructions like parallel 16 bit multiply-accumulate instruction.

VIS, MAX-2, and MMX all have the same basic goal. They provide high-performance multi-media processing on general-purpose microprocessors. All

three of them support a full set of subword parallel instructions on 16 bit subwords. Four subwords per 64 bit register word are dealt with in parallel. Differences exist in the type and amount of support they provide driven by the needs of the target markets. For example, support is provided for 8 bit subwords when target markets include lower end multi-media applications (like games) whereas high quality multi-media applications (like medical imaging) require the processing of larger subwords.

3.1.2 Floating-Point SIMD Extensions

Floating-point computation is the heart of each numerical algorithm. Thus, speeding up floating-point computation is essential to overall performance.

AltiVec. Motorola's AltiVec SIMD architecture extends the recent MPC74xx G4 generation of the Motorola POWER PC microprocessor line—starting with the MPC7400—through the addition of a 128 bit vector execution unit. This short vector SIMD unit operates concurrently with the existing integer and floating-point units. This new execution unit provides for highly parallel operations, allowing for the simultaneous execution of four arithmetic operations in a single clock cycle for single-precision floating-point data.

Technical details are given in the Motorola AltiVec manuals [85, 86].

SSE. In the Pentium III streaming SIMD Extension (SSE) Intel added 70 new instructions to the IA-32 architecture.

The SSE instructions of the Pentium III processor introduced new general purpose floating-point instructions, which operate on a new set of eight 128 bit SSE registers. In addition to the new floating-point instructions, SSE technology also provides new instructions to control cacheability of all data types. SSE includes the ability to stream data into the processor while minimizing pollution of the caches and the ability to prefetch data before it is actually used. Both 64 bit integer and packed floating-point data can be streamed to memory.

Technical details are given in Intel's architecture manuals [55, 56, 57] and the C++ compiler manual [50]. The features relevant for this thesis are summarized in Appendix D.1.

SSE 2. Intel's Pentium 4 processor is the first member of a new family of processors that are the successors to the Intel P6 family of processors, which include the Intel Pentium Pro, Pentium II, and Pentium III processors. New SIMD instructions (SSE 2) were introduced in the Pentium 4 processor architecture and included floating-point SIMD instructions, integer SIMD instructions, as well as conversion of packed data between XMM registers and MMX registers.

The added floating-point SIMD instructions allow computations to be performed on packed double-precision floating-point values (two double-precision values per 128 bit XMM register). Both the single-precision and double-precision

floating-point formats and the instructions that operate on them are fully compatible with the IEEE Standard 754 for binary floating-point arithmetic.

Technical details are given in Intel's architecture manuals [55, 56, 57] and the C++ compiler manual [50]. The features relevant for this thesis are summarized in Appendix D.1.

SSE 3. The newly developed 90nm version of Intel's Pentium 4 (formerly code name Prescott) includes the SSE 3 technology with improved performance over SSE and SSE 2. SSE 3 incorporates new instructions that help to significantly improve (i) complex arithmetic evaluation, (ii) the evaluation of quantities like dot products, (iii) the conversion to integer for applications using x87 floating point code, and (iv) 128-bit unaligned memory accesses.

Technical details are given in Intel's software developer's guide [63] and the technology journal [64].

IPF. Support for Intel's SSE is maintained and extended in Intel's and HP's new generation of Itanium processor family (IPF) processors when run in the 32 bit legacy mode. Native 64 bit instructions exist which split the double-precision registers in a pair of single-precision registers with support of two-way SIMD operations. In the software layer provided by Intel's compilers these new instructions are emulated by SSE instructions.

Technical details are given in Intel's architecture manuals [59, 60, 61] and the C++ compiler manual [50].

3DNow! Since AMD requires Intel x86 compatibility for business reasons, they implemented the MMX extensions in their processors too. However, AMD specific instructions were added, known as "3DNow!".

AMD's Athlon has instructions, similar to Intel's SSE instructions, designed for purposes such as digital signal processing. One important difference between the Athlon extensions (Enhanced 3DNow!) and those on the Pentium III are that no extra registers have been added in the Athlon design. The AMD Athlon XP features the new 3DNow! professional extension which is compatible to both Enhanced 3DNow! and SSE. AMD's new 64 bit architecture x86-64 and the first processor of this new line called Hammer supports a superset of all current x86 SIMD extensions including SSE 2.

Technical details can be found in the AMD 3DNow! manual [4] and in the x86-64 manuals [8, 9].

Double FPU. Within the PowerPC 440 FP2 the standard FPU is replicated leading to a double FPU that is capable to operate well on complex numbers. Up to four floating-point operations (one two-way vector fused multiply-add operation) can be issued every cycle. This double FPU has many similarities to industry-standard two-way short vector SIMD extensions like AMD's 3DNow! or Intel's SSE 2 and SSE 3. In particular, data to be processed by the double FPU has to be naturally aligned on 16-byte boundaries in memory.

However, the PowerPC 440 FP2 features some characteristics that are different from standard short vector SIMD implementations:

(i) Non-standard fused multiply-add (FMA) operations required for complex multiplications, (ii) computationally expensive data reorganization within 2-way registers, and (iii) cheap intermix of scalar and vector operations.

A more detailed description of the PowerPC 440 FP2 processor and the double FPU extension can be found in Section 3.4.

Overview. Table 3.1 gives an overview over the SIMD floating-point capabilities found in current microprocessors. In the context of this thesis, especially Intel's SSE 2 and AMD's 3DNow! extensions as well as IBM's double FPU are of importance. Thus, only 2-way vectorization is of concern.

Vendor	Name	<i>n</i> -way	Prec.	Processor	Compiler
Intel	SSE	4-way	single	Pentium III Pentium 4	MS Visual C++ Intel C++ Compiler GNU C Compiler 3.0
Intel	SSE 2	2-way	double	Pentium 4 (Willamette)	MS Visual C++ Intel C++ Compiler GNU C Compiler 3.0
Intel	SSE 3	2-way	double	Pentium 4 (Prescott)	MS Visual C++ Intel C++ Compiler 8.0 GNU C Compiler 3.3.3
Intel	IPF	2-way	single	Itanium Itanium 2	Intel C++ Compiler
AMD	3DNow!	2-way	single	K6, K6-II	MS Visual C++ GNU C Compiler 3.0
AMD	Enhanced 3DNow!	2-way	single	Athlon (K7)	MS Visual C++ GNU C Compiler 3.0
AMD	3DNow! Professional	4-way	single	Athlon XP Athlon MP	MS Visual C++ Intel C++ Compiler GNU C Compiler 3.0
Motorola	AltiVec	4-way	single	MPC74xx G4	GNU C Compiler 3.0 Apple C Compiler 2.96
IBM	Double FPU	2-way	double	PowerPC 440 FP2	IBM XL Century

Table 3.1: Short vector SIMD extensions providing floating-point arithmetic found in general purpose microprocessors. In the context of this thesis, especially Intel's SSE 2 and AMD's 3DNow! extensions as well as IBM's double FPU are of importance.

3.1.3 Data Streaming

One of the key features needed in fast multi-media applications is the efficient streaming of data into and out of the processor. Multi-media programs such as video decompression codes stress the data memory system in ways that the multilevel cache hierarchies of many general-purpose processors cannot handle efficiently. These programs are data intensive with working sets bigger than many first-level caches. Streaming memory systems and compiler optimizations aimed at reducing memory latency (for example, via prefetching) have the potential to improve these applications' performance. Current research in data and computational transforms for parallel machines may provide for further gains in this area.

3.1.4 Software Support

Currently, application developers have three common methods for accessing multi-media hardware within a general-purpose micro processor: (i) They can invoke vendor-supplied libraries that utilize the new instructions, (ii) rewrite key portions of the application in assembly language using the multi-media instructions, or (iii) code in a high-level language and use vendor-supplied macros that make available the extended functionality through a simple function-call like interface.

System Libraries. The simplest approach to improving application performance is to rewrite the system libraries to employ the multi-media hardware. The clear advantage of this approach is that existing applications can immediately take advantage of the new hardware without recompilation. However, the restriction of multi-media hardware to the system libraries also limits potential performance benefits. An application's performance will not improve unless it invokes the appropriate system libraries, and the overheads inherent in the general interfaces associated with system functions will limit application performance improvements. Even so, this is the easiest approach for a system vendor, and vendors have announced or plan to provide such enhanced libraries.

Assembly Language. At the other end of the programming spectrum, an application developer can benefit from multi-media hardware by rewriting key portions of an application in assembly language. Though this approach gives a developer great flexibility, it is generally tedious and error prone. In addition, it does not guarantee a performance improvement (over code produced by an optimizing compiler), given the complexity of today's microarchitectures.

Programming Language Abstractions. Recognizing the tedious and difficult nature of assembly coding, most hardware vendors which have introduced multi-media extensions have developed programming-language abstractions. These give an application developer access to the newly introduced hardware without having

to actually write assembly language code. Typically, this approach results in a function-call-like abstraction that represents one-to-one mapping between a function call and a multi-media instruction.

There are several benefits of this approach. First, the compiler (not the developer) performs machine-specific optimizations such as register allocation and instruction scheduling. Second, this method integrates multi-media operations directly into the surrounding high-level code without an expensive procedure call to a separate assembly language routine. Third, it provides a high degree of portability by isolating from the specifics of the underlying hardware implementation. If the multi-media primitives do not exist in hardware on the particular target machine, the compiler can replace the multi-media macro by a set of equivalent operations.

The most common language extension supplying such primitives is to provide within the C programming language function-call like intrinsic (or built-in) functions and new data types to mirror the instructions and vector registers. For most SIMD extensions, at least one compiler featuring these language extensions exists. Examples include C compilers for HP's MAX-2, Intel's MMX, SSE, and SSE 2, Motorola's AltiVec, and Sun's VIS architecture as well as the GNU C compiler which supports a broad range of short vector SIMD extensions.

Each intrinsic directly translates to a single multi-media instruction, and the compiler allocates registers and schedules instructions. This approach would be even more attractive to application developers if the industry agreed upon a common set of macros, rather than having a different set from each vendor. For the AltiVec architecture, Motorola has defined such an interface. Under Windows both the Intel C++ compiler and Microsoft's Visual Studio compiler use the same macros to access SSE, SSE 2, and SSE 3. The Intel C++ compiler for Linux uses these macros as well. These two C extensions provide defacto standards on the respective architectures.

Vectorizing Compilers. While macros may be an acceptably efficient solution for invoking multi-media instructions within a high-level language, subword parallelism could be further exploited with automatic compilation from high-level languages to these instructions. Some vectorizing compilers for short vector SIMD extensions exist, including the Intel C++ compiler, the PGI Fortran compiler, the Vector C compiler and the IBM's XL C compiler for BlueGene/L. The latter compiler will be assessed together with the vectorization methods introduced in this thesis in Section 10.2.

3.2 Intel's Streaming SIMD Extensions

The Pentium III processor was Intel's first processor featuring the streaming SIMD extensions (SSE). SSE instructions are Intel's floating-point and integer SIMD extensions to the P6 core. They also support the integer SIMD operations

(MMX) introduced by its predecessor, the Pentium II processor.

Appendix D.1 lists all SSE 2 instructions relevant in the context of this thesis.

SSE offers general purpose floating-point instructions that operate on a set of eight 128 bit SIMD floating-point registers. Each register is considered to be a vector of four single-precision floating-point numbers. The SSE registers are not aliased onto the floating-point registers as are the MMX registers. This feature enables the programmer to develop algorithms that can utilize both SSE and MMX instructions without penalty. SSE also provides new instructions to control cacheability of MMX technology and IA-32 data types. These instructions include the ability to load data from memory and store data to memory without polluting the caches, and the ability to prefetch data before it is actually used. These features are called *data streaming*. SSE provides the following extensions to the IA-32 programming environment: (i) one new 128 bit packed floating-point data type, (ii) 8 new 128 bit registers, and (iii) 70 new instructions.

The 128 bit Packed Floating Data Type

The new data type is a vector of single-precision floating-point numbers, capable of holding exactly four single-precision floating-point numbers.

The new SIMD floating-point unit (FPU) can be used as a replacement for the standard non-SIMD FPU. Unlike the MMX extensions, the new floating-point SIMD unit can be used in parallel with the standard FPU.

The New Registers

The 8 new registers are each capable of holding exactly one 128 bit SSE data type. Unlike the standard Intel FPU, the SSE FPU registers are not viewed as register stack, but rather are directly accessible by the names **XMM0** through **XMM7**.

Unlike the general purpose registers, the new registers operate only on data, and can not be used to address memory (which is sensible since memory locations are 32 bit addressable). The SSE control status register **MXCSR** provides the usual information such as rounding modes, exception handling, for a vector as a whole, but not for individual elements of a vector. Thus, if a floating-point exception is raised after performing some operation, one may be aware of the exception, but cannot tell where in the vector the exception applies to.

Through the introduction of new registers the Pentium III processor has operating system visible state and thus requires operating system support. The integer SIMD (MMX) registers are aliased to the standard FPU's registers, and thus do not require operating system support. Operating system support is needed if on a context switch the contents of the new registers are to be stored and loaded properly.

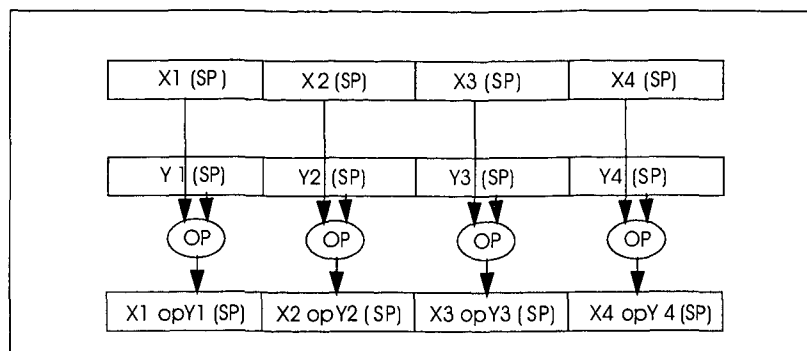


Figure 3.1: Packed SSE operations.

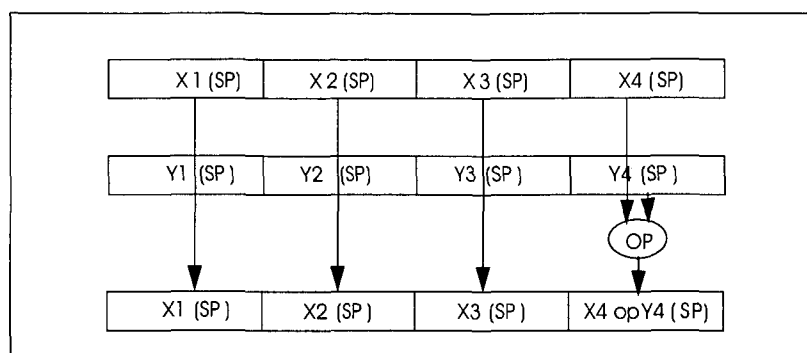


Figure 3.2: Scalar SSE operations.

3.2.1 The SSE Instructions

The 70 SSE instructions are mostly SIMD floating-point related, however, some of them extend the integer SIMD extension MMX, and others relate to cache control. There are: (i) data movement instructions, (ii) arithmetic instructions, (iii) comparison instructions, (iv) conversion instructions, (v) logical instructions, (vi) shuffle instructions, (vii) state management instructions, (viii) cacheability control instructions, and (ix) additional MMX SIMD integer instructions. These instructions operate on the MMX registers, and not on the SSE registers.

The SSE instructions operate on either all (see Figure 3.1) or the least significant (see Figure 3.2) pairs of packed data operands in parallel. In general, the address of a memory operand has to be aligned on a 16 byte boundary for all instructions.

The data movement instructions include pack/unpack instructions and data shuffle instructions that enable to “mix” the indices in the vector operations. The instruction `SHUFPS` (shuffle packed, single-precision, floating-point) is able to shuffle any of the packed four single-precision, floating-point numbers from one source operand to the lower two destination fields; the upper two destination fields are generated from a shuffle of any of the four floating-point numbers from the second source operand (Figure 3.3). By using the same register for both

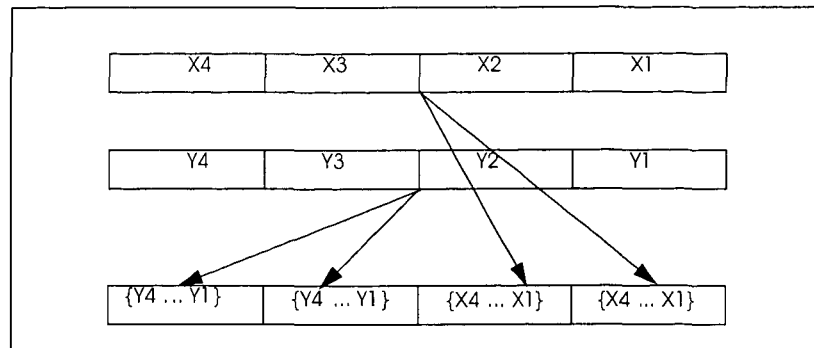


Figure 3.3: Packed shuffle SSE operations.

sources, SHUFPS can return any combination of the four floating-point numbers from this register.

When stored in memory, the floating-point numbers will occupy consecutive memory addresses. Instructions exist which allow data to be loaded to and from memory, in 128 bit, 64 bit, or 32 bit blocks, that is: (i) instructions for moving all 4 elements to and from memory, (ii) instructions for moving the upper two elements to and from memory, (iii) instructions for moving the lower two elements to and from memory, and (iv) instructions for moving the lowest element to and from memory.

Some important remarks about the SSE instruction set have to be made.

- The SSE instruction set offers no means for moving data between the standard FPU registers and the new SSE registers, as well as no provision for moving data between the general purpose registers and the new registers (without converting types).
- Memory access instructions, as well as instructions which use a memory address as an operand like the arithmetic instruction MULPS (which can use a memory address or a register as one of its operands) distinguish between 16 byte aligned data and data not aligned on a 16 byte boundary. Instructions exist for moving aligned and unaligned data, however, instructions which move unaligned data suffer a performance penalty of 9 to 12 extra clock cycles. Instructions which are able to use a memory location for an operand (such as MULPS) assume 16 byte alignment of data. If unaligned data is accessed when aligned data is expected, a general protection error is raised.
- The Pentium III SIMD FPU is a true 32 bit floating-point unit. It does all computations using 32 bit floating-point numbers. The standard FPU on the Intel IA-32 architecture defaults all internal computations to 80 bits (IEEE 754 extended), and truncates the result if less than 80 bits is

needed. Thus, noticeable differences can be observed when comparing single-precision output from the two units.

Documentation. The SSE instruction set is described in the IA-32 manuals [55, 56]. Further information on programming Intel's SSE can be found in the related application notes [49, 52] and the IA-32 optimization manual [53]. Further information on data alignment issues is given in [51].

3.2.2 The SSE 2 Instructions

The streaming SIMD extensions 2 (SSE 2) add 144 instructions to the IA-32 architecture and allow the Pentium 4 to process double-precision data using short vector SIMD instructions. In addition, extra long integers are supported.

SSE 2 is based on the infrastructural changes already introduced with SSE. In particular, the SSE registers are used for SSE 2, and all instructions appear as two-way versions of the respective four-way SSE instructions. Thus, most restrictions of the SSE instructions are mirrored by the SSE 2 instructions. Most important, the memory access restriction is the same as in SSE: Only naturally (16 byte) aligned vectors can be accessed efficiently. Even the same shuffle SSE operations are implemented as two-way SSE 2 versions.

The SSE 2 arithmetic offers full IEEE 754 double-precision arithmetic and thus can be used in science and engineering applications. SSE 2 is designed to replace the standard FPU. This can be achieved by utilizing scalar SSE 2 arithmetic operating on the lower word of the two-way vector. The main impact is that floating-point code not utilizing the SSE 2 extension becomes obsolete and again the complexity of high-performance programs is raised.

SSE 2 introduces the same data alignment issues as SSE. Efficient memory access requires 16-byte aligned vector memory access.

Documentation. The SSE 2 instruction set is described in the IA-32 manuals [55, 56]. Further information on programming Intel's SSE 2 can be found in the IA-32 optimization manuals [53, 58]. Further information on data alignment issues is given in [51].

3.2.3 The SSE 3 Instructions

The streaming SIMD extensions 3 (SSE 3) add 11 instructions to the SSE 2 instruction set. Thereby, they don't so much add new functionality as much as they improve the efficiency of the previous SSE and SSE 2 instructions.

Among the new instructions the most important are the following.

Overall, five instructions have been added to significantly accelerate complex arithmetics. In contrast to SSE 2 it is possible to perform a mix of addition and subtraction operations, hence removing the need for changing the sign of some

operands and therefore saving instructions. These new instructions are useful for evaluating complex products on packed single and double-precision data.

While most SIMD instructions operate in parallel, four SSE 3 instructions allow intraoperand addition and subtraction both on single and double precision operands, meaning that contiguous data elements from the same operand are used to produce a result data element.

One instruction has been added to ease the converting of SIMD packed to x87 floating point data.

One instruction now allows 128-bit unaligned data load, designed to avoid cache line splits and thus increasing memory transfer performance.

Three instructions combine loads with duplication of data elements saving the need for a shuffle instruction on the loaded data.

Documentation. The newly added SSE 3 instructions are described in [63].

3.3 AMD's 3DNow!

AMD is Intel's competitor in the field of x86 compatible processors. In response to Intel's MMX technology, AMD released the 3DNow! technology line which is MMX compatible and additionally features two-way floating-point SIMD operation. In the first step, 21 instructions were included defining the original 3DNow! extension. The original 3DNow! was released with the AMD K6-II processor. Up to two 3DNow! instructions could be executed per clock cycle, including one two-way addition and one two-way multiplication leading to a peak performance of four floating-point operations per cycle.

With the introduction of the AMD Athlon processor, AMD has taken 3DNow! technology to the next level of performance and functionality. The AMD Athlon processor features an enhanced version of 3DNow! that adds 24 instructions to the existing 21 original 3DNow! instructions. These 24 additional instructions include: (i) 12 standard SIMD instructions, (ii) 7 streaming memory access instructions, and (iii) 5 special DSP instructions.

AMD's Athlon XP and Athlon MP processor line introduces SSE compatibility by the introduction of 3DNow! professional. Thus, Athlon XP and Athlon MP processors are both enhanced 3DNow! and SSE compatible.

The 3DNow! extension shares the FPU registers and features a very fast switching between MMX and the FPU. Thus, no additional operating system support is required. Information about the AMD 3DNow! extension family can be found in the related manuals [5, 4, 7, 6]

With AMD's next generation processor (codename Hammer [8, 9]), a new 64 bit architecture called x86-64 will be introduced. This architecture features *128 bit media instructions* and *64 bit media programming*. The new 64 bit instruction set will support a superset of all IA-32 SIMD extensions, thus supporting MMX, all 3DNow! versions, SSE, and SSE 2.

3.4 The IBM BlueGene/L Supercomputer

IBM's BlueGene/L (BG/L) planned to be in operation in 2005 will be an order of magnitude faster than the Earth Simulator, currently being the number one on the TOP 500 list. BlueGene/L is developed to run large scale simulations on 64k processors in parallel making new classes of problems solvable. Its custom double floating-point unit provides support for complex arithmetic. However, as a non-standard feature, it is difficult to utilize this FPU efficiently when not using complex arithmetic explicitly.

Efficient computation of fast Fourier transforms (FFTs) is required in many applications planned to be run on BlueGene/L. In most of these applications, very fast one-dimensional FFT routines for small problem sizes (up to 2048 data points) running on a single processor are required as major building blocks.

The BlueGene/L machine [3] will be built from 65,536 PowerPC 440 FP2 processors connected by a 3D torus network leading to 360 Tflop/s peak performance. The Earth Simulator, currently leading the TOP 500 list, provides 40 Tflop/s peak performance. A small prototype of the BlueGene/L machine was built recently. In contrast to BlueGene/L's 700 MHz target frequency, the current prototype runs at only 500 MHz.

BlueGene/L's Floating-Point Unit. To boost BlueGene/L's floating-point performance, a custom "double FPU" is applied: The standard FPU is replicated within the PowerPC 440 FP2 leading to a double FPU that is capable to operate well on complex numbers: Up to four floating-point operations (one 2-way vector fused multiply-add operation) can be issued every cycle. This double FPU has many similarities to industry-standard 2-way short vector SIMD extensions like AMD's 3DNow! or Intel's SSE 2. In particular, data to be processed by the double FPU has to be naturally aligned on 16-byte boundaries in memory.

However, the PowerPC 440 FP2 features some characteristics that are different from standard short vector SIMD implementations: (i) Non-standard fused multiply-add (FMA) operations required for complex multiplications, (ii) computationally expensive data reorganization within 2-way registers, and (iii) cheap intermix of scalar and vector operations.

Without tailor-made adaptation of established short vector SIMD vectorization techniques to the specific features of BlueGene/L's double FPU no high-performance short vector code can be obtained.

3.5 Vector Computers vs. Short Vector SIMD

Vector computers are supercomputers used for large scientific and engineering problems, as many numerical algorithms allow those parts which consume the majority of computation time to be expressed as vector operations. This holds especially for almost all linear algebra algorithms [42, 20]. It is therefore a straight-

forward strategy to improve the performance of processors used for numerical data processing by providing an instruction set tailor-made for vector operations as well as suitable hardware.

This idea materialized in vector architectures comprising specific *vector instructions*, which allow for componentwise addition, multiplication and/or division of vectors as well as the multiplication of the vector components by a scalar. Moreover, there are specific load and store instructions enabling the processor to fetch all components of a vector from the main memory or to move them there.

The hardware counterparts of vector instructions are the matching *vector registers* and *vector units*. Vector registers are memory elements which can contain vectors of a given maximum length. Vector units performing vector operations, as mentioned above, usually require the operands to be stored in vector registers.

These systems are specialized machines not comparable to general purpose processors featuring short vector SIMD extensions. The most obvious difference on the vector extension level is the larger machine vector length, the support for smaller vectors and non-unit memory access. In vector computers actually multiple processing elements are processing vector data, while in short vector SIMD extensions only a very short fixed vector length is supported.

Example 3.1 (Vector Computers) The Cray T90 multiprocessor uses Cray Research Inc. custom silicon CPUs with a clock speed of 440 MHz, and each processor has a peak performance of 1.7 Gflop/s. Each has 8 vector registers with 128 words (vector elements) of eight bytes (64 bits) each.

Current vector computers provided by NEC range from desktide systems (the NEC SX-6i featuring one CPU and a peak performance of 8 Gflop/s) up to the currently most powerful computer in the world: the *Earth Simulator* featuring 5120 vector CPUs running at 500 MHz leading to a peak performance of 41 Tflop/s.

The high performance of floating-point operations in vector units is mainly due to the concurrent execution of operations (as in a very deep pipeline).

There are further advantages of vector processors as compared with other processors capable of executing overlaid floating-point operations.

- As vector components are usually stored contiguously in memory, the access pattern to the data storage is known to be linear. Vector processors exploit this fact using a very fast vector data fetch from a massively interleaved main memory space.
- There are no memory delays for a vector operand which fits completely into a vector register.
- There are no delays due to branch conditions as they might occur if the vector operation were implemented in a loop.

In addition, vector processors may utilize the super-scalar principle by executing several vector operations per time unit [21].

Parallel Vector Computers

Most of the vector supercomputer manufacturers produce multiprocessor systems based on their vector processors. Since a single node is so expensive and so finely tuned to memory bandwidth and other architectural parameters, the multiprocessor configurations have only a few vector processing nodes.

Example 3.2 (Parallel Vector Computers) A NEC SX-5 multi node configuration can include up to 32 SX-5 single node systems for the SX-6A configuration.

However, the latest vector processors fit onto single chips. For instance, NEC SX-6 nodes can be combined to form much larger systems in multiframe configuration (up to 1024 CPUs are combined) or even the earth simulator with its 5120 CPUs.

3.5.1 Vectorizing Compilers

Vectorizing compilers were developed for the vector computers described above. Using vectorizing compilers to produce *short vector SIMD* code for discrete linear transforms in the context of adaptive algorithms is not straightforward. As the vectorizing compiler technology originates from completely different machines and in the short vector SIMD extensions other and new restrictions are found, the capabilities of these compilers are limited. Especially automatic performance tuning poses additional challenges to vectorizing compilers as the codes are generated automatically and intelligent search is used which conflicts with some compiler optimization. Thus compiler vectorization and automatic performance tuning cannot be combined easily. The two leading adaptive software systems for discrete linear transforms cannot directly use compiler vectorization in their code generation and adaptation process.

FFTW. Due to the recursive structure of FFTW and the fact that memory access patterns are not known in advance, vectorizing compilers cannot prove alignment and unit stride properties required for vectorization. Thus FFTW cannot be vectorized automatically using compiler vectorization.

SPIRAL. The structure of code generated by SPIRAL implies that such code cannot be vectorized directly by using vectorizing compilers without some hints and changes in the generated code. A further difficulty is introduced by optimizations carried out by SPIRAL. Vectorizing compilers only vectorize rather large loops, as in the general case the additional cost for prologue and epilogue has to be amortized by the vectorized loop. Vectorizing compilers require hints about which loop to vectorize and to prove loop carried data dependencies. It is required to guarantee the proper alignment. The requirement of a large number of loop iterations conflicts with the optimal code structure, as in discrete linear transforms a small number (sometimes as small as the extension's vector length) turns out to be most efficient. In addition, straight line codes cannot be vectorized.

3.5.2 Vector Computer Libraries

Traditional vector processors have typically vector lengths of 64 and more elements. They are able to load vectors at non-unit stride but feature a rather high startup cost for vector operations [65]. Codes developed for such machines do not match the requirements of modern short vector SIMD extensions. Highly efficient implementations for DFT computation that are portable across different conventional vector computers are not available. For instance, high-performance implementations for Cray machines were optimized using assembly language [66]. An example for such an library is Cray's proprietary SCILIB which is also available as the Fortran version SCIPORT which can be obtained via NETLIB [78].

Chapter 4

Fast Algorithms for Linear Transforms

This chapter defines discrete linear transforms and fast algorithms for such transforms, following the methodology introduced by the SPIRAL team. The approach is based on Kronecker product factorizations of transform matrices and on recursive factorization rules.

4.1 Discrete Linear Transforms

This section defines discrete linear transforms as a foundation for the specific discussion of fast Fourier transform algorithms in the next section. In this thesis, the main focus is on the discrete Fourier transform and its fast algorithms based on the Cooley-Tukey recursion.

Discrete linear transforms are represented by real or complex valued matrices and their application means to calculate a matrix-vector product. Thus, they express a base change in the vector space of sampled data.

Definition 4.1 (Real Discrete Linear Transforms) Let $M \in \mathbb{R}^{m \times n}$, $x \in \mathbb{R}^n$, and $y \in \mathbb{R}^m$. The real linear transform M of x is obtained by the matrix-vector multiplication

$$y = Mx.$$

Examples include the Walsh-Hadamard transform and the sine and cosine transforms.

Definition 4.2 (Complex Discrete Linear Transforms) Let $M \in \mathbb{C}^{m \times n}$, $x \in \mathbb{C}^n$, and $y \in \mathbb{C}^m$. The complex linear transform M of x is again given by the matrix-vector multiplication

$$y = Mx.$$

A particularly important example and the main focus in this thesis is the discrete Fourier transform (DFT), which, for size N , is given by the following definition.

Definition 4.3 (Discrete Fourier Transform Matrices) The matrix DFT_N is defined for any $N \in \mathbb{N}$ with $i = \sqrt{-1}$ by

$$\text{DFT}_N = (e^{2\pi i k \ell / N} \mid k, \ell = 0, 1, \dots, N-1).$$

The values $\omega_N^{k\ell} = e^{2\pi i k \ell / N}$ are called *twiddle factors*.

Example 4.1 (DFT Matrices) The first five DFT matrices are

$$\begin{aligned} \text{DFT}_1 &= (1), \quad \text{DFT}_2 = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \quad \text{DFT}_3 = \begin{pmatrix} 1 & 1 & 1 \\ 1 & e^{-2\pi i/3} & e^{-4\pi i/3} \\ 1 & e^{-4\pi i/3} & e^{-2\pi i/3} \end{pmatrix} \\ \text{DFT}_4 &= \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix}, \quad \text{DFT}_5 = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & e^{-2\pi i/5} & e^{-4\pi i/5} & e^{-6\pi i/5} & e^{-8\pi i/5} \\ 1 & e^{-4\pi i/5} & e^{-8\pi i/5} & e^{-2\pi i/5} & e^{-6\pi i/5} \\ 1 & e^{-6\pi i/5} & e^{-2\pi i/5} & e^{-8\pi i/5} & e^{-4\pi i/5} \\ 1 & e^{-8\pi i/5} & e^{-6\pi i/5} & e^{-4\pi i/5} & e^{-2\pi i/5} \end{pmatrix}. \end{aligned}$$

DFT_4 is the largest DFT matrix having only *trivial* twiddle-factors, i.e., $1, i, -1, -i$.

Definition 4.4 (The Discrete Fourier Transform) The discrete Fourier transform $y \in \mathbb{C}^N$ of a data vector $x \in \mathbb{C}^N$ is given by the matrix-vector product

$$y = \text{DFT}_N x.$$

Fast Algorithms

An important property of discrete linear transforms is the existence of fast algorithms. Typically, these algorithms reduce the complexity from $O(N^2)$ arithmetic operations, as required by direct evaluation via matrix-vector multiplication, to $O(N \log N)$ operations. This complexity reduction guarantees their very efficient applicability for large N .

Mathematically, any fast algorithm can be viewed as a factorization of the transform matrix into a product of sparse matrices. It is a specific property of discrete linear transforms that these factorizations are highly structured and can be written in a very concise way using the formalisms of Kronecker (tensor) products [109], in combination with permutation and twiddle factor matrices.

The permutation operator L_n^{mn} sorts the components of x according to their index modulo n . Thus, components with indices equal to $0 \bmod n$ come first, followed by the components with indices equal to $1 \bmod n$, and so on.

Definition 4.5 (Stride Permutation) For a vector $x \in \mathbb{C}^{mn}$ with

$$x = \sum_{k=0}^{mn-1} x_k e_k^{mn} \quad \text{with} \quad e_k^{mn} = e_i^n \otimes e_j^m, \quad \text{and} \quad x_k \in \mathbb{C},$$

the stride permutation L_n^{mn} is defined by its action on the tensor basis of \mathbb{C}^{mn} .

$$L_n^{mn}(e_i^n \otimes e_j^m) = e_j^m \otimes e_i^n.$$

An important class of matrices arising in FFT factorizations are diagonal matrices whose diagonal elements are roots of unity. Such matrices are called twiddle factor matrices.

Definition 4.6 (Twiddle Factor Matrices) Let $\omega_N = e^{2\pi i/N}$ denote the N th root of unity. The twiddle factor matrix, denoted by T_m^{mn} , is a diagonal matrix defined by

$$T_m^{mn}(e_i^m \otimes e_j^n) = \omega_{mn}^{ij}(e_i^m \otimes e_j^n), \quad i = 0, 1, \dots, m-1, \quad j = 0, 1, \dots, n-1,$$

$$T_m^{mn} = \bigoplus_{i=0}^{m-1} \bigoplus_{j=0}^{n-1} \omega_{mn}^{ij} = \bigoplus_{i=0}^{m-1} \Omega_{n,i}(\omega_{mn}),$$

where $\Omega_{n,k}(\alpha) = \text{diag}(1, \alpha, \dots, \alpha^{n-1})^k$.

Example 4.2 (DFT₄) Consider a factorization, i. e., a fast algorithm, for DFT₄. Using the mathematical notation from [109] it follows that

$$\begin{aligned} \text{DFT}_4 &= \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & i \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (4.1) \\ &= (\text{DFT}_2 \otimes I_2) \cdot T_2^4 \cdot (I_2 \otimes \text{DFT}_2) \cdot L_2^4. \end{aligned}$$

T_2^4 denotes a twiddle matrix, i. e., $T_2^4 = \text{diag}(1, 1, 1, i)$. L_2^4 denotes a stride permutation which swaps the two middle elements x_1 and x_2 in a four-dimensional vector, i. e.,

$$\begin{pmatrix} x_0 \\ x_2 \\ x_1 \\ x_3 \end{pmatrix} = L_2^4 \begin{pmatrix} x_0 \\ x_1 \\ x_3 \\ x_2 \end{pmatrix}.$$

Automatic Derivation of Fast Algorithms

In [23] a method has been introduced that *automatically* derives fast algorithms for a given transform and size. This method is based on algebraic symmetries of the transformation matrices utilized by the software package AREP [22], a library for the computer algebra system GAP [43] used in SPIRAL. AREP is able to factorize transform matrices and to find fast algorithms automatically. In [96] an algebraic derivation of fast sine and cosine transform algorithms is described.

Recursive Rules

One key element in factorizing a discrete linear transform matrix into sparse factor matrices is the application of *breakdown rules* or simply *rules*.

A rule is a sparse factorization of the transform matrix and breaks down the computation of the given transform into transforms of smaller size. These smaller transforms, which can be of a different type, can be further expanded using the same or other rules. Thus rules can be applied recursively to reduce a large linear transform to a number of smaller discrete linear transforms.

In breakdown rules, the transform sizes have to satisfy certain conditions which are implicitly given by the rule. Here the transform sizes are functions of some parameters which are denoted by lowercase letters. For instance, a breakdown rule for DFT_N , whose size N has to be a product of at least two factors (say m and n), is given by an equation for DFT_{mn} . In such a rule, m and n are subsequently used as parameters in the right-hand side of the rule.

Examples of discrete linear transforms featuring rules include the Walsh-Hadamard transform (WHT), the discrete cosine transform (DCT) used, for instance, in the JPEG standard [100], as well as the fast Fourier transform [16].

In the following examples P_n , P'_n , and P''_n denote permutation matrices, S_n denotes a bidiagonal and D_n a diagonal matrix [110].

Example 4.3 (Walsh-Hadamard Transforms) The WHT_N for $N = 2^k$ is given by

$$\text{WHT}_N = \overbrace{\text{DFT}_2 \otimes \dots \otimes \text{DFT}_2}^{k \text{ times}}.$$

A particular example of a rule for this transform is

$$\text{WHT}_{2^k} = \prod_{i=1}^k (\text{I}_{2^{k_1+\dots+k_{i-1}}} \otimes \text{WHT}_{2^{k_i}} \otimes \text{I}_{2^{k_{i+1}+\dots+k_t}}), \quad k = k_1 + \dots + k_t. \quad (4.2)$$

Example 4.4 (Discrete Cosine Transforms) The DCT_N for arbitrary N is given by

$$\text{DCT}_N = (\cos((\ell + 1/2)k\pi/N) \mid k, \ell = 0, 1, \dots, N-1).$$

A corresponding rule is

$$\text{DCT}_{2n} = P_{2n} (\text{DCT}_n \oplus S_{2n} \text{DCT}_n D_{2n}) P'_{2n} (\text{I}_n \otimes \text{DFT}_2) P''_{2n}.$$

Example 4.5 (Discrete Fourier Transforms) A rule for the DFT_N matrix is given by

$$\text{DFT}_{mn} = (\text{DFT}_m \otimes \text{I}_n) T_n^{mn} (\text{I}_m \otimes \text{DFT}_n) L_m^{mn}. \quad (4.3)$$

(4.3) is the Cooley-Tukey FFT written in the Kronecker product notation [65].

Transforms of higher dimension are also captured in this framework and naturally possess rules. For example, if M is an $N \times N$ transform, then the corresponding two-dimensional transform is given by $M \otimes M$. Using the respective property of the tensor product, the rule

$$M \otimes M = (M \otimes \text{I}_N) (\text{I}_N \otimes M) \quad (4.4)$$

is obtained.

The set of rules used in SPIRAL is constantly growing. A set of important rules can be found in [97].

Formulas and Base Cases

Eventually a mathematical *formula* is obtained when all transforms are expanded into base cases.

Example 4.6 (Fully Expanded Formula for WHT_8) According to rule (4.2), WHT_8 can fully be expanded into

$$(\text{DFT}_2 \otimes \text{I}_4)(\text{I}_2 \otimes \text{DFT}_2 \otimes \text{I}_2)(\text{I}_4 \otimes \text{DFT}_2)$$

with DFT_2 being the base case.

Trees and Recursion

The recursive decomposition of a discrete linear transform into smaller ones using recursion rules can be expressed by trees. FFTW calls these trees *plans* while SPIRAL calls these trees *rule trees*. In these trees the essence of the recursion—the type and sizes of the child transforms—is specified.

As an example, rule trees for a recursion rule that breaks down a transform of size N into two smaller transforms is discussed.

Figure 4.1 shows a tree of a discrete linear transform of size $N = mn$ that is decomposed into smaller transforms of the same type of sizes m and n . When specific rules are used, the nodes have to carry the rule name.

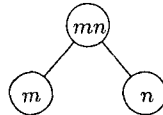


Figure 4.1: Tree representation of a discrete linear transform of size $N = mn$ with one recursion step applied.

The node marked with mn is the *parent node* of the *child nodes* lying directly below, which indicate here transforms of sizes m and n .

Analogously, Figure 4.2 shows a tree of a discrete linear transform of size $N = kmn$ where in a first step the transform is decomposed into discrete linear transforms of size k and mn . In a second step the transforms of size mn are further decomposed into transforms of size m and n .

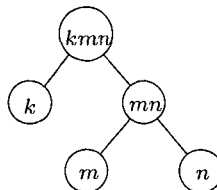


Figure 4.2: Right-expanded tree, two recursive steps.

In general, the splitting rules are *not* commutative with respect to m and n . Thus, the trees are generally *not* symmetric. *Left-* and *right-child* nodes have to be distinguished which is done simply by left and right branches.

In every tree there exists a *root node*, i.e., the upmost node which has no “parents”. There are lowest nodes without “children” which are the *leave nodes*.

The upmost recursive decomposition in a tree, the one of the root node, is called the *top level decomposition*. If its two branches are equivalent the tree is called *balanced*, if they are nearly equivalent it is said to be “somewhat” balanced. But there also exist trees that are not balanced at all. They may be even extremely unsymmetrical. A tree with just leafs as left children is formed strictly to the right. Such a tree is called *right-expanded*, its contrary *left-expanded*.

The Search Space

By selecting different breakdown rules, a given discrete linear transform expands to a large number of formulas that correspond to different fast algorithms. For example, for $N = 2^k$, there are $k - 1$ ways to apply rule (4.3) to DFT_N . A similar degree of freedom recursively applies to the smaller DFTs obtained, which leads to $O(5^k/k^{3/2})$ different formulas for DFT_{2^k} . In the case of the DFT, allowing breakdown rules other than (4.3) further extends the formula space.

The problem of finding an efficient formula for a given transform translates into a search problem in the space of formulas for that specific transform. The size of the search space depends on the rules and transforms actually used.

The conventional approach for solving the search problem is to make an educated guess (with some machine characteristics as hints) which formula might lead to an efficient implementation and then to continue by optimizing this formula.

The automatic performance tuning systems SPIRAL and FFTW use a different approach. Both systems find fast implementations by intelligently looking through the search space. SPIRAL uses various search strategies and fully expands the formulas. FFTW uses dynamic programming and restricts its search to the coarse grain structure of the algorithm. The rules are hardcoded into the executor while the fine grain structure is fixed by the codelet generator at compile time.

Chapter 5

Matrix Multiplication

Many mathematical problems involve various matrix calculations. In most cases these calculations can be implemented quite easily. These simple programs may be sufficient for small problems but with increasing problem size they turn out to be slower than one would expect. As a result of inefficient memory management they are unable to take advantage of the potential performance of modern computer systems in a satisfactory way.

The reason for this phenomenon is the high amount of data transfers occurring between different levels of the memory hierarchy if the problem is too large to fit into the (highest level) cache. For example, when performing a matrix multiplication it is important that both factor matrices and the product matrix fit into cache. Otherwise data that is still to be used has to be expunged from the cache in order to free cache space for other data from the main memory.

As matrix multiplication is the most important operation in high performance linear algebra software it is usually carried out using vendor supplied routines optimized for specific computer systems. In most cases this is the relevant BLAS routine, for which standard syntax and semantics have been established, and for which an individually optimized version can be produced using ATLAS (Whaley et al. [115]). Software utilizing optimized BLAS routines is besides being efficient also widely portable.

5.1 The Problem Setting

Example 5.1 illustrates the straight-forward way for implementing a matrix multiplication of an $m \times n$ matrix A and an $n \times p$ matrix B using three nested loops.

Example 5.1 (Naive Matrix Multiplication Implementation)

```
C = 0.  
DO i = 1 to m  
  DO j = 1 to n  
    DO k = 1 to p  
      C(i,j) = C(i,j) + A(i,k)*B(k,j)  
    END DO  
  END DO  
END DO
```

First the element c_{11} is calculated by calculating the inner product of the first row of A and the first column of B . Then j is increased and the first column of B is, after only one access, for the moment not needed any more. But in fact every element of B has to be accessed m times. If the problem is too large to fit into cache this data will be expunged sometime. So for every value of i (which means m times) every element of B has to be loaded into cache, but is accessed only one time. This results in a very large number of cache misses, which leads to a significant performance deterioration. Fig. 5.1 shows (as represented by the code fragment given above) the floating-point performance of the ijk -variant of matrix multiplication (with $m = n = p$).

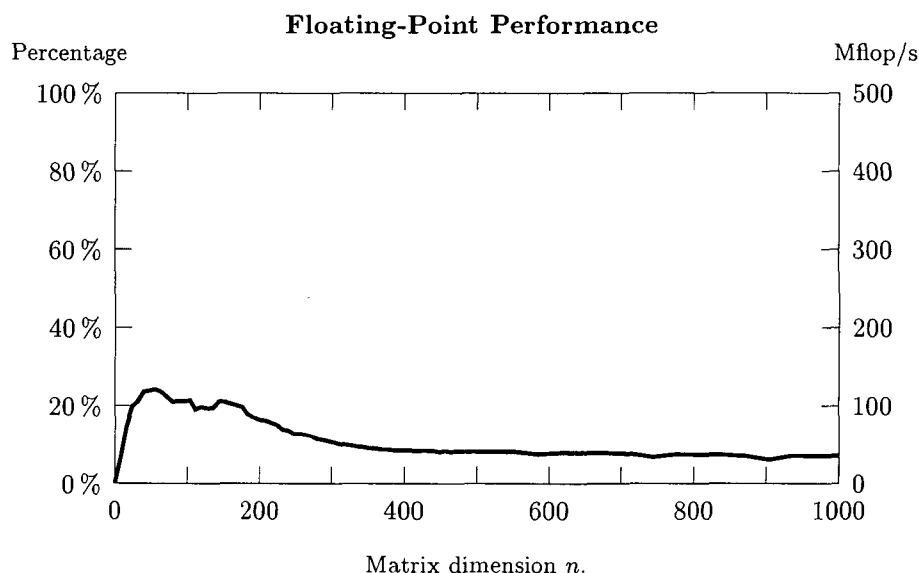


Figure 5.1: Floating-point performance of the ijk -variant of multiplying $n \times n$ matrices on one processor of an SGI Origin 2000.

One technique to overcome this undesirable performance deterioration is blocking, where the calculations are subdivided and applied to small submatrices. The final result is put together from the solutions of the subproblems. Of course a block size should be chosen which allows the subproblems to fit into cache completely. Thus, the block size is a hardware dependent parameter which needs to be set before execution to guarantee an optimal performance. Algorithms with such parameters are called *cache aware*.

The matrix multiplication routine of ATLAS is based on a blocking algorithm. First, the input matrices A and B are copied to block major format, if the problem is large enough to justify the resulting $O(n^2)$ overhead. The blocksize is chosen such that when multiplying one block of A and one block of B resulting in one block of C , there is at least enough space in the L1 cache for the whole A -block, two columns of the B -block and one cache line from the C -block to ensure

maximal L1 cache reuse for block multiplication. The code for this cache-contained matrix multiplication is generated by ATLAS after having empirically determined the optimal blocksize with its feedback loop search engine.

The algorithm wound around the block multiplication works after the same principle. ATLAS allocates space for the whole matrix A , one column of B and one block of C to be copied into, which ensures optimal L2 cache reuse, and calls the generated block multiplication routine for the blocks inside the cache resident area. For further details see Whaley, Petitet and Dongarra [115]. Fig. 5.2 shows the floating-point performance of ATLAS' `dgemm` routine which runs at nearly 80 % of peak performance.

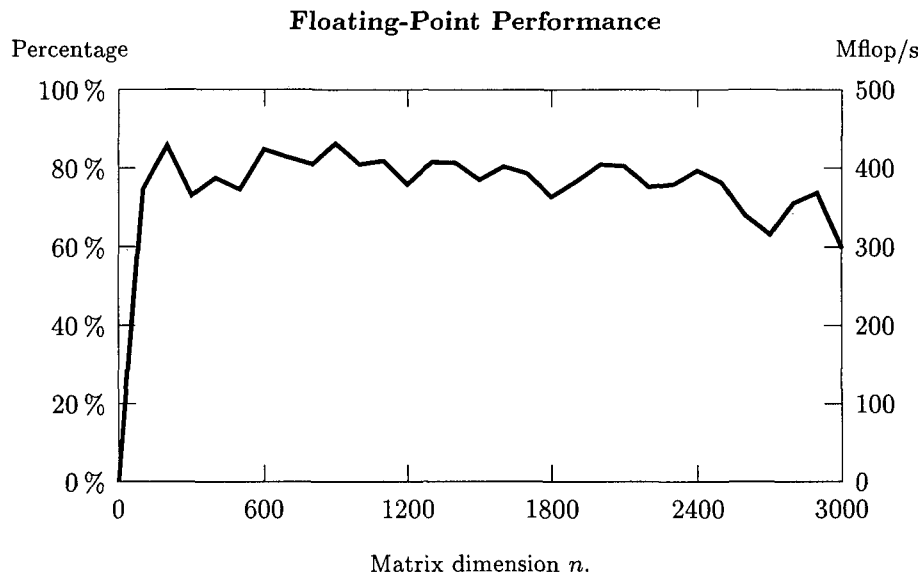


Figure 5.2: Floating-point performance of ATLAS' `dgemm` routine on one processor of an SGI Origin 2000.

Chapter 6

A Portable SIMD API

Short vector SIMD extensions are advanced architectural features. Utilizing the respective instructions to speed up applications introduces another level of complexity and it is not straightforward to produce high performance codes.

The reasons why short vector SIMD instructions are hard to use are the following:

- (i) They are beyond standard (e. g., C) programming.
- (ii) They require an unusually high level of programming expertise.
- (iii) They are usually non-portable.
- (iv) Compilers in general don't (can't) use these features to a satisfactory extent.
- (v) They are changed/extended with every new architecture.
- (vi) It is not clear where and how to use them.
- (vii) There is a potential high payoff (factors of 2, 4, and more) for small and intermediate problem sizes whose solution cannot be accelerated with multi-processor machines but there is also potential speed-up for large scale problems.

As discussed in Chapter 3, a sort of common programming model was established recently. The C language has been extended by new data types according to the available registers and the operations are mapped onto (intrinsic or built-in functions) functions. This way, a portable SIMD API consisting of a set of C macros was defined that can be implemented efficiently on all current architectures and features all necessary operations. Using this programming model, a programmer does not have to deal with assembly language. Register allocation and instruction selection is done by the compiler. However, these interfaces are not standardized neither across different compiler vendors on the same architecture nor across architectures. But for any current short vector SIMD architecture at least one compiler featuring this interface is available.

The machine models introduced in Section 7.4 are based on this portable SIMD API, which serves two main purposes: (i) to abstract from hardware peculiarities, and (ii) to abstract from special compiler features.

The vector straight-line codes generated by the MAP vectorizer (see Chapter 7) which do not use the introduced straight-line code MAP backend (see Chapter 9) use the newly defined portable SIMD API in the scope of this thesis.

Abstracting from Special Machine Features

In the context of this thesis all short vector SIMD extensions feature the functionality required in intermediate level building blocks. However, the implementation of such building blocks depends on special features of the target architecture.

For instance, a complex reordering operation like a permutation has to be implemented using register-register permutation instructions provided by the target architecture. In addition, restrictions like aligned memory access have to be handled. Thus, a set of intermediate building blocks has to be defined which (i) can be implemented on all current short vector SIMD architectures and (ii) enables all discrete linear transforms to be built on top of these building blocks. This set is called the *portable SIMD API*.

Appendix D describes the relevant parts of the instruction sets provided by current short vector SIMD extensions. As the instruction set of IBM's BG/L supercomputer is classified, it has not been included.

Abstracting from Special Compiler Features

All compilers featuring a short vector SIMD C language extension provide the required functionality to implement the portable SIMD API. But syntax and semantics differ from platform to platform and from compiler to compiler. These specifics have to be hidden in the portable SIMD API.

Table 3.1 (on page 40) shows that for any current short vector SIMD extension compilers with short vector SIMD language extensions exist.

Short vector SIMD extensions are advanced architectural features. Utilizing the respective instructions to speed up applications introduces another level of complexity and it is not straightforward to produce high performance codes.

The reasons why short vector SIMD instructions are hard to use are the following: (i) They are beyond standard (e.g., C) programming. (ii) They require an unusually high level of programming expertise. (iii) They are usually non-portable. (iv) Compilers in general don't (can't) use these features to a satisfactory extent. (v) They are changed/extended with every new architecture. (vi) It is not clear where and how to use them.

6.1 Definition of the Portable SIMD API

The portable SIMD API includes macros of four types: (i) data types, (ii) constant handling, (iii) arithmetic operations, and (iv) extended memory operations. An overview of the provided macros is given below. Appendix E contains examples of actual implementations of the portable SIMD API on various platforms.

Data Types

The portable SIMD API introduces three data types, which are all naturally aligned: (i) Real numbers of type `float` or `double` (depending on the extension) have type `simd_real`. (ii) Complex numbers of type `simd_complex` are pairs of `simd_real` elements. (iii) Vectors of type `simd_vector` are vectors of ν elements of type `simd_real`. For two-way short vector SIMD extensions the type

`simd_complex` is equal to `simd_vector`. Table 6.1 summarizes the data types supported by the portable SIMD API.

API type	Elements
<code>simd_real</code>	single or double
<code>simd_complex</code>	a pair of <code>simd_real</code>
<code>simd_vector</code>	native data type, vector length ν , for two-way equal to <code>simd_complex</code>

Table 6.1: Data types provided by the portable SIMD API.

For two-way short vector SIMD extensions the type `simd_complex` is equal to `simd_vector`.

Nomenclature. In the remainder of this section, variables of type `simd_vector` are named `t`, `t0`, `t1` and so forth. Memory locations of type `simd_vector` are named `*v`, `*v0`, `*v1` and so forth. Memory locations of type `simd_complex` are named `*c`, `*c0`, `*c1` and so forth. Memory locations of type `simd_real` are named `*r`, `*r0`, `*r1` and so forth. Constants of type `simd_real` are named `r`, `r0`, `r1` and so forth.

Table 6.1 summarizes the data types supported by the portable SIMD API.

The portable SIMD API includes macros of five types: (i) data types, (ii) constant handling, (iii) arithmetic operations, (iv) reorder operations, and (v) memory access operations. An overview of the provided macros is given below. The portable SIMD API can be extended to arbitrary vector length. Thus, optimization techniques like *loop interleaving* (Gatlin and Carter [41]) can be implemented on top of the portable SIMD API.

In this chapter, the SIMD API is defined for an arbitrary vector length ν . As for the remaining part of this thesis only Intel's SSE 2 and AMD's 3DNow! extensions as well as IBM's double FPU are featured, this universal representation can be disregarded for the sake of simplicity. Only 2-way SIMD extensions will be of further concern.

Constant Handling

The portable SIMD API provides declaration macros for the following types of constants whose values are known at compile time: (i) the zero vector, (ii) homogeneous vector constants (all components have the same value), and (iii) inhomogeneous vector constants (all components may have a different value).

There are three types of constant load operations: (i) load a constant vector (both homogeneous and inhomogeneous) that is known at compile time, (ii) load a constant vector (both homogeneous and inhomogeneous) that is precomputed at run time (but not known at compile time), and (iii) load a precomputed constant

real number and build a homogeneous vector constant with that value. Table 6.2 shows the most important macros for constant handling.

Macro	Type
DECLARE_CONST(name, r)	compile time homogeneous
DECLARE_CONST_2(name, r0, r1)	compile time inhomogeneous
DECLARE_CONST_4(name, r0, r1, r2, r3)	compile time inhomogeneous
LOAD_CONST(name)	compile time
LOAD_CONST_SCALAR(*r)	precomputed homogeneous real
LOAD_CONST_VECT(*v)	precomputed vector
SIMD_SET_ZERO()	compile time homogeneous

Table 6.2: Constant handling operations provided by the portable SIMD API. *r*, *r0*, *r1*, *r2* and *r3* denote variables of type `simd_real`. **r* denotes a memory location holding a value of type `simd_real`. **v* denotes a memory location holding a value of type `simd_vector`.

Arithmetic Operations

The portable SIMD API provides (with and without constant operators) real addition, subtraction, and multiplication operations, the unary minus, four types of fused multiply-add operations, and a complex multiplication. See Table 6.3 for a summary.

Macro	Operation
VEC_ADD(v, v0, v1)	$v = v_0 + v_1$
VEC_SUB(v, v0, v1)	$v = v_0 - v_1$
VEC_ACCA(v, v1, v2, ..., vv)	$v = (v_1.1 + \dots + v_1.v, \dots, v_v.1 + \dots + v_v.v)$
VEC_ACCS(v, v1, v2, ..., vv)	$v = (v_1.1 - \dots - v_1.v, \dots, v_v.1 - \dots - v_v.v)$
VEC_UMINUS(v, v0)	$v = -v_0$
VEC_CHS(v, v0, pos)	$v = (v_0.1, \dots, -v_0.pos, \dots, v_0.v)$
VEC_MUL(v, v0, v1)	$v = v_0 \times v_1$
VEC_MUL_CONST(v, *v0, v1)	$v = v_0 \times v_1$
VEC_MUL_MEM(v, r, v0)	$v.1 = r \times v_0.1, v.2 = r \times v_0.1$
VEC_MADD(v, v0, v1, v2)	$v = v_0 \times v_1 + v_2$
VEC_MSUB(v, v0, v1, v2)	$v = v_0 \times v_1 - v_2$
VEC_NMSUB(v, v0, v1, v2)	$v = -(v_0 \times v_1 - v_2)$
VEC_FMCA(v, *v0, v1, v2)	$v = v_0 \times v_1 + v_2$
VEC_FMCS(v, *v0, v1, v2)	$v = v_0 \times v_1 - v_2$
COMPLEX_MULT(v0, v1, v2, v3, v4, v5)	$v_0 = v_2 \times v_4 - v_3 \times v_5$

Table 6.3: Arithmetic operations provided by the portable SIMD API. *v*, *v0*, *v1*, *v2*, *v3*, *v4*, *v5* and *vv* denote variables of type `simd_vector`. *pos* denotes the position of the desired vector element.

Reorder Operations

The SIMD API supports the vector reorder operations `VEC_COPY`, `VEC_UNPACK`, `VEC_SWAP2` and `VEC_SHUFFLE2`. They are summarized in Table 6.4. The `VEC_SWAP2` and `VEC_SHUFFLE2` operation are defined for 2-way SIMD architectures only.

Macro	Operation
<code>VEC_COPY(v, v0)</code>	$v = v_0$
<code>VEC_UNPACK(v, v1, ..., vν, pos)</code>	$v = (v_1.pos, \dots, v_\nu.pos)$
<code>VEC_SWAP2(v, v0)</code>	$v.1 = v_0.2, v.2 = v_0.1$
<code>VEC_SHUFFLE2(v, v0, v1, pos1, pos2)</code>	$v.1 = v_0.pos1, v.2 = v_0.pos2$

Table 6.4: Vector reorder operations provided by the SIMD API. v , v_0 , v_1 and v_ν denote variables of type `simd_vector`. pos denotes the desired position inside the vector of length ν . The `VEC_SWAP2` and `VEC_SHUFFLE2` instruction are defined for 2-way SIMD extensions exclusively.

Memory Access Operations

The portable SIMD API provides load and store instructions both for whole vectors of type `simd_vector` (`VEC_LOAD`, `VEC_STORE`) as well as for single data elements of type `simd_real` (`SCA_LOAD`, `SCA_STORE`). In the latter case, the locations of the `simd_real` value that is to be loaded/stored inside the source and destination vectors must be specified. Table 6.5 gives the syntactic and semantic details of the memory access operations supported by the SIMD API.

Macro	Operation
<code>VEC_LOAD(v, *v0)</code>	$v = *v_0$
<code>SCA_LOAD(v.pos1, *v0.pos2)</code>	$v.pos1 = *v_0.pos2$
<code>VEC_STORE(*v, v0)</code>	$*v = v_0$
<code>SCA_STORE(*v.pos1, v0.pos2)</code>	$*v.pos1 = v_0.pos2$

Table 6.5: Memory access operations provided by the SIMD API. v and v_0 denote variables of type `simd_vector`. $*v$ and $*v_0$ denote memory locations holding values of type `simd_vector`. $pos1$ and $pos2$ denote the desired position inside the vector of length ν .

Specific 2-way Operations

The portable SIMD API provides macros for operations which are specific to vectors where $\nu = 2$ i.e., 2-way SIMD. Tabel 6.6 shows how ν -way macros translate to specific 2-way macros necessary in the context of this thesis.

ν -way Macro	2-way Macro
VEC_CHS(<i>v</i> , <i>v0</i> , 1)	VEC_CHS_LO(<i>v</i> , <i>v0</i>)
VEC_CHS(<i>v</i> , <i>v0</i> , 2)	VEC_CHS_HI(<i>v</i> , <i>v0</i>)
VEC_MUL_CONST(<i>v</i> , * <i>v0</i> , <i>v1</i>)	VEC_MULCONST2(<i>v</i> , <i>v1</i> , * <i>v0</i>)
VEC_UNPACK(<i>v</i> , <i>v1</i> , <i>v2</i> , 1)	VEC_UNPACK_LO(<i>v</i> , <i>v1</i> , <i>v2</i>)
VEC_UNPACK(<i>v</i> , <i>v1</i> , <i>v2</i> , 2)	VEC_UNPACK_HI(<i>v</i> , <i>v1</i> , <i>v2</i>)
VEC_SHUFFLE2(<i>v</i> , <i>v0</i> , <i>v1</i> , 2, 1)	VEC_SHUFFLE01(<i>v</i> , <i>v0</i> , <i>v1</i>)
VEC_LOAD(<i>v</i> , * <i>v0</i>)	VEC_LOAD_Q(<i>v</i> , * <i>v0</i>)
SCA_LOAD(<i>v</i> .1, * <i>v0</i> .pos2)	VEC_LOAD_D_LO(<i>v</i> .1, * <i>v0</i> .pos2)
SCA_LOAD(<i>v</i> .2, * <i>v0</i> .pos2)	VEC_LOAD_D_HI(<i>v</i> .2, * <i>v0</i> .pos2)
VEC_STORE(* <i>v</i> , <i>v0</i>)	VEC_STORE_Q(* <i>v</i> , <i>v0</i>)
SCA_STORE(* <i>v</i> .pos1, <i>v0</i> .1)	VEC_STORE_D_LO(* <i>v</i> .pos1, <i>v0</i>)
SCA_STORE(* <i>v</i> .pos1, <i>v0</i> .2)	VEC_STORE_D_HI(* <i>v</i> .pos1, <i>v0</i>)

Table 6.6: Specific 2-way operations provided by the SIMD API. *v*, *v0*, *v1* and *v2* denote variables of type `simd_vector`. **v* and **v0* denote memory locations holding values of type `simd_vector`. *pos1* and *pos2* denote the desired position inside the vector of length ν .

Chapter 7

Automatic Vectorization of Straight-Line Code

A few years ago major vendors of *general purpose* microprocessors started to include short vector single instruction, multiple data (SIMD) extensions into their instruction set architectures to enable exploitation of data level parallelism found in multi-media applications. Examples of SIMD extensions supporting both integer and floating-point operations include Intel's SSE, AMD's 3DNow!, and Motorola's AltiVec.

SIMD extensions have the potential to significantly speed up implementations in all areas where (i) performance is crucial, and (ii) the relevant algorithms exhibit fine grain parallelism.

Currently, most short vector extensions can be utilized by either high language extensions, or by vectorizing compilers.

The available SIMD extensions mirror the underlying hardware features by means of data types and intrinsic or built-in functions that can be utilized by hand-coding. However, the scalar code produced by automatic performance tuning software packages (see Chapter 1) often has thousands of lines which makes hand-coding using short vector language extensions unfeasible.

Besides, the usefulness of vectorizing compilers is rather limited because they mostly deal with loop-level parallelism and the underlying scalar code has to be of a special structure to allow for this. Thus, vectorizing compilers are not useful in the vectorization of scalar code emitted by FFTW, SPIRAL, or ATLAS because the code's structure does not allow for loop vectorization.

This chapter introduces the MAP vectorizer that uses a completely different approach to automatically extracts *2-way SIMD parallelism* out of a sequence of operations from static single assignment (SSA) straight-line code while maintaining data locality and utilizing special features of short vector SIMD extensions.

The MAP vectorizer uses a non-deterministic pattern matching system (i) to find two scalar instructions which can be paired into one SIMD instruction defined by the virtual vector architecture's machine model and (ii) to perform a fusion of the corresponding operand variables to 2-way SIMD variables, i. e., cells. The inherent difficulty of this vectorization process results from the requirement to keep the semantics of the scalar computation.

The newly introduced vectorizer boasts the following benefits.

- (i) The total number of instructions is reduced by utilizing short vector in-

structions. In the optimal case, every pair of scalar floating-point instructions is joined into one equivalent SIMD floating-point instruction, cutting the instruction count into half. (ii) The register allocator benefits from the SIMD register file being twice as wide as in the scalar case. (iii) The pairing of memory access instructions on interleaved complex numbers significantly contributes to a reduction of the amount of integer instructions needed for effective address calculation.

7.1 Vectorization of Straight Line Code

The goal of the MAP vectorizer is to transform a scalar computation into short vector code while achieving the best possible utilization of SIMD resources. It produces vectorized code either (i) via a source-to-source transformation delivering macros compliant with the portable SIMD API (see Chapter 6) [32] and additionally providing support for FMA instructions, or (ii) via a source-to-assembly transformation utilizing the MAP backend (see Chapter 9).

The vectorizer aims at vectorizing straight-line code containing arithmetic operations, array access operations and index computations. Such codes cannot be handled by established vectorizing compilers mainly focussing on loop vectorization [50]. Even standard methods for vectorizing straight-line code (Larsen and Amarasinghe [79]) fail in producing high performance vector code as well.

7.2 The Vectorization Approach

The MAP vectorizer uses a backtracking search engine to automatically extract *2-way SIMD parallelism* out of scalar code blocks by fusing pairs of scalar temporary variables into SIMD variables, and by replacing the corresponding scalar instructions by vector instructions as illustrated by Fig. 7.1.

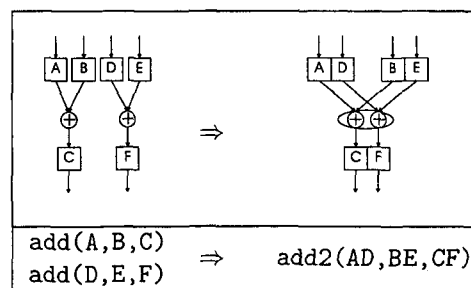


Figure 7.1: 2-way Vectorization. Two scalar add instructions are transformed into a vector `add2` instruction. The result of the vector addition of the fusions AD and BE is stored in CF.

Tuples of scalar variables, i.e., *fusions*, are declared such that every scalar variable appears in exactly one of them. Each fusion is assigned one SIMD variable.

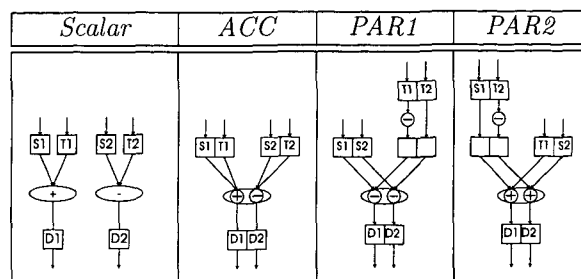


Figure 7.2: SIMD Vectorization of Scalar ADD/SUB. Two scalar instructions, i. e., addition and subtraction, are transformed into semantically equivalent SIMD instructions. Three different ways (i. e., accumulate (*ACC*), parallel 1 (*PAR1*) and parallel 2 (*PAR2*)) to do so exist.

On the set of SIMD variables, SIMD instructions have to perform exactly the same computation as the originally given scalar instructions do on the scalar variables. This is achieved by rules specifying how each pair of scalar instructions is replaced by a sequence of semantically identical SIMD instructions operating on the corresponding SIMD variables.

Rules for pairs of scalar binary instructions allow up to three different, semantically equivalent ways to do so, as illustrated by Fig. 7.2. Locally, this allows vectorizations of different efficiency. Globally, it widens the search space of the vectorizer's backtracking search engine.

The vectorizer's goal is to yield an optimal utilization of SIMD resources. This target and the fact that the backtracking engine uses a *pick first result* strategy to minimize vectorization runtime does not necessarily lead to the best possible vectorization in terms of an overall minimum of SIMD instructions but allows the vectorization of large straight line codes (see Table 7.4 for vectorization runtimes of large FFTW codelets).

The MAP vectorizer utilizes an automatic fallback to different vectorization levels in its vectorization process. Each level provides a set of pairing rules with different pairing restrictions. Thus, more restrictive rules are just applied to highly parallel codes resulting either in efficient vectorization or no vectorization at all. Less restrictive rules address a broader range of codes but result in less efficient vectorization.

This concept assists the searching for good vectorizations needing a minimum of data reorganization, provided such vectorizations exist. The fallback to a less restrictive vectorization level allows to vectorize even codes featuring less parallel parts to some extent.

7.3 Benefits of The MAP Vectorizer

Thus, the vectorizer exhibits the following benefits:

Halving the Instruction Count. Optimally, every pair of scalar floating-point

instructions is joined into one equivalent SIMD floating-point instruction, thus cutting the instruction count into half.

Diminishing Data Spills and Reloads. The wider SIMD register files allow for a more efficient computation. The register allocator indirectly benefits from the register file being twice as wide as in the scalar case.

Accelerating Effective Address Calculations. The pairing of memory access instructions potentially reduces the effort needed for calculating effective addresses by fifty percent.

7.4 Virtual Machine Models

To keep the amount of hardware specific details in the vectorization process as small as possible, *virtual machine models* are utilized. These models are sets of virtual instructions emulating the semantics of operations without incorporating any architecture specific syntactic idioms.

The usage of such models enables portability and extensibility as the vectorizer can be adapted to produce code for any processor architecture supporting 2-way SIMD floating-point instructions easily. Also, future 2-way SIMD instructions, supported by the successors of existing processors, can be easily included into a model's instruction set allowing the vectorizer to extract them.

The virtual machine models used in the vectorizer are abstractions of scalar as well as 2-way SIMD architectures, i. e., the vectorizer transforms virtual scalar to virtual 2-way SIMD instructions. During the optimization process the resulting instructions are rewritten into instructions that are actually available in a specific architecture's instruction set. For that purpose there are specific machine models for (i) AMD K7, (ii) AMD K6, and (iii) Intel P4. While the vectorizer's machine model is target architecture independent, the virtual machine models for AMD's K7, and K6, as well as Intel's P4 are target architecture specific.

Recent hardware development has resulted in a wide range of computer systems having SIMD style instruction set extensions included. These have a lot in common but there are vendor characteristic differences. Even a single vendor enhances and widens the number of SIMD instructions from one hardware generation to the next. For instance, some SIMD architectures (e. g., AMD's K6, K7) include intraoperand style (accumulate) instructions while others, e. g., Intel's Pentium 4 don't. Intel's new SSE 3 SIMD instruction set feature some of them.

Taking this fact into consideration, the following approach is pursued in selecting suitable instructions in the vector code generation process for a specific target architecture.

Principally, all instructions of the vectorizer's virtual machine model are available to the vectorization engine with respect to any addressed target architecture. Even those, for which no equivalent instruction is actually implemented there. Nevertheless, those having an equivalent in a target processor's machine model

are favored. Vector instructions not supported by the target machine model are only used if otherwise no vectorization is possible at all, because they would have to be rewritten into supported instructions in the optimization step directly following vectorization (see Chapter 8).

The Vectorizer's Virtual Machine Model. The instructions supported and therefore extracted by the vectorizer are: (i) load of a SIMD or scalar variable, (ii) store of a SIMD or scalar variable, (iii) swap, unpack, and copy, (iv) multiplication by floating-point constants, (v) unary change sign instructions, (vi) binary parallel instructions, and (vii) binary intraoperand (accumulate) instructions. Table 7.2 illustrates the syntactic and semantic details of the instructions.

Virtual Machine Model of Intel's P4. The virtual machine model of Intel's Pentium 4 supports all instructions of the vectorizer's machine model except the intraoperand (i.e., `accPP2`, `accNN2`, `accNP2`, `accPN2`) instructions, the change sign instructions, and the `swap2` instruction. Additionally, it supports a `shuffleXY2` instruction. For details, see Table 7.2.

Virtual Machine Model of AMD's K6. The virtual machine model of AMD's K6 processor supports all instructions of the vectorizer's machine model except some intraoperand (i.e., `accNN2`, `accNP2`, `accPN2`) instructions, and the `swap2` instruction. For details, see Table 7.2.

Virtual Machine Model of AMD's K7. The virtual machine model of AMD's K7 processor comprises the same virtual instructions as the vectorizer's machine model except the `accPN2` instruction. For details, see Table 7.2.

7.4.1 The Virtual Scalar Machine Model

The instructions defined by the virtual scalar machine model are semantically compliant with SSA code, i.e., they can be directly mapped into C or Fortran code. Besides, the model complies with the output of FFTW's original scalar code generator. SPIRAL and ATLAS codes need to be parsed and translated into a format compatible to the virtual scalar machine model before being processed.

Assumptions Underlying the Scalar Model

Any virtual model for scalar machines has to include the following basic operations: (i) load from, as well as (ii) store to memory instructions, (iii) the unary negation instruction `neg`, instructions for multiplication by a constant, i.e., `mulc`, (iv) binary addition, i.e., `add`, subtraction, i.e., `sub`, and multiplication, i.e., `mult` instructions. Table 7.1 gives examples of such instructions.

Registers. An arbitrarily large number of registers is assumed such that all scalar variables can be treated as registers. Any individual register is called a *scalar cell* which corresponds to a temporary scalar variable `A, B, ...`

Memory $M[i]$, $i = 0, 1, \dots, N-1$ is considered to be an array of scalar cells.

Instructions. There are memory access instructions which load data from a memory location to a register or, respectively, store data from a register to memory. There are unary and binary instructions which, in the first case, have one register and, in the second case have two registers or one register and floating-point constant as their source operands. The result of such an operation is stored in a destination register.

<i>Virtual Scalar Instruction</i>	<i>Effect</i>
<code>load(M[i],A)</code>	$A := M[i]$
<code>store(A,M[i])</code>	$M[i] := A$
<code>mulc(A,(K),B)</code>	$B := A * (K)$
<code>neg(A,B)</code>	$B := -A$
<code>add(A,B,C)</code>	$C := A + B$
<code>sub(A,B,C)</code>	$C := A - B$
<code>mul(A,B,C)</code>	$C := A * B$

Table 7.1: Virtual Scalar Instructions. The first column contains examples of scalar instructions operating on scalar variables A, B, \dots as well as on a floating-point constant K . The second column illustrates the effect of executing these instructions.

7.4.2 The Virtual Vector Machine Model

The virtual vector machine model is based on the portable SIMD API introduced in Chapter 6. It includes virtual instructions needed because of their semantics but not necessarily included in the actually available target architecture instruction set. Thus, these instructions have to be emulated by a sequence of the cheapest available target architecture vector instructions. This is done in a local rewriting step that directly follows the vectorization step in order to keep the vectorization process as simple as possible (see Chapter 8). Moreover, it makes sense to perform the rewriting step not until a vectorization result is available.

To summarize, the virtual vector machine model includes all instructions needed to yield a valid vectorized DAG that allows to be rewritten for any available and future hardware.

Assumptions Underlying the Vector Model

Registers. An arbitrarily large number of registers is assumed. Each register is a 2-way SIMD cell and corresponds to a temporary variable A, B, \dots . The lower part and the higher part of a 2-way SIMD cell can be addressed by $A.l$ and $A.h$, assuming that $A = (A.l, A.h)$.

Memory $M[i]$, $i = 0, 1, \dots, N/2-1$ is considered to be an array of 2-way memory cells. The lower part and the higher part of a 2-way memory cell can be addressed by $M[i].l$ and $M[i].h$, assuming that $M[i] = (M[i].l, M[i].h)$.

Instructions. There are unary and binary instructions available. Unary instructions have one 2-way register or a memory location as source operand. Binary instructions have two source operands which are 2-way registers or a 2-way register and a 2-way floating point constant. The result of any instruction is stored into a 2-way register or into a memory location as well.

The instructions supported by the vector machine model are grouped into seven categories: (i) load of a SIMD or float variable, (ii) store of a SIMD or float variable, (iii) unary swap and binary unpack, (iv) multiplication by floating-point constants, (v) unary change of sign operations, (vi) binary parallel operations, and (vii) binary intraoperand (accumulate) operations. Table 7.2 gives an overview of these instructions.

7.4.3 Scalar and Vector Main Memory Layout

The main memory is physically identical in both, scalar and vector computation. Nevertheless, as already described in Sections 7.4.1 and 7.4.2, the indexing of memory locations is different. While indexing to scalar memory locations always addresses one single data value, indexing a vector memory location has a value tuple (i.e., 2-way memory cell) as its target. Each of the cell components, i.e., the lower and the higher part, can be individually addressed.

Under these assumptions, locations used by memory access instructions from the scalar machine model have to be aliased to their corresponding memory locations usable by vector machine model instructions. The following definitions unambiguously specify these mappings and explicitly ascertain whether memory is accessed in scalar or vector layout. The introduced memory layout will be used throughout this chapter whenever memory access is an issue.

```
#if SINGLE_PREC
    typedef float scalar;
#else
    typedef double scalar;
#endif

typedef struct {
    scalar l, h;
} vector;

union {
    scalar scal[N];
    vector vect[N/2];
} M;

vector A, B, C, D, E, F, G, H, T;
```

Type	Vector Instruction	Effect	Supported
Load Instructions (loadX2)	loadQ2(M[i],A)	A.l := M[i].l, A.h := M[i].h	P4,K6,K7,Vec
	loadDL2(M[i].pos,A)	A.l := M[i].pos; pos := 1 / h	
	loadDH2(M[i].pos,A)	A.h := M[i].pos; pos := 1 / h	
Store Instructions (storeX2)	storeQ2(A,M[i])	M[i].l := A.l, M[i].h := A.h	P4,K6,K7,Vec
	storeDL2(A.pos,M[i])	M[i].l := A.pos; pos := 1 / h	
	storeDH2(A.pos,M[i])	M[i].h := A.pos; pos := 1 / h	
Unary Instructions (unaryX2)	copy2(A,B)	B.l := A.l, B.h := A.h	P4,K6,K7,Vec
	swap2(A,B)	B.l := A.h, B.h := A.l	K7,Vec
	mulc2(A,(K.l,K.h),B)	B.l := A.l*K.l, B.h := A.h*K.h	P4,K6,K7,Vec
	chsL2(A,B)	B.l := -A.l, B.h := A.h	K6,K7,Vec
	chsH2(A,B)	B.l := A.l, B.h := -A.h	
	chsLH2(A,B)	B.l := -A.l, B.h := -A.h	
Binary Instructions (binaryX2)	add2(A,B,C)	C.l := A.l+B.l, C.h := A.h+B.h	P4,K6,K7,Vec
	sub2(A,B,C)	C.l := A.l-B.l, C.h := A.h-B.h	
	mul2(A,B,C)	C.l := A.l*B.l, C.h := A.h*B.h	
	accPP2(A,B,C)	C.l := A.l+A.h, C.h := B.l+B.h	K6,K7,Vec
	accNN2(A,B,C)	C.l := A.l-A.h, C.h := B.l-B.h	K7,Vec
	accNP2(A,B,C)	C.l := A.l-A.h, C.h := B.l+B.h	
	accPN2(A,B,C)	C.l := A.l+A.h, C.h := B.l-B.h	Vec
	unpackL2(A,B,C)	C.l := A.l, C.h := B.l	P4,K6,K7,Vec
	unpackH2(A,B,C)	C.l := A.h, C.h := B.h	
	shuffleXY2(A,B,C)	C.l := A.X, C.h := B.Y X,Y := 1 / h	P4

Table 7.2: Currently Supported Vector Instructions. The first column contains the type of the vector instruction, the second column contains all available vector instructions (containing temporary SIMD variables A,B,...). The third column illustrates the effect of executing these instructions. The fourth column shows which machine models support each instruction.

According to the above definitions, memory accesses such as `load(M.scal[i])`, `loadQ(M.vect[j])`, `loadDL(M.vect[j].h)`, and `loadDH(M.vect[j].l)` with $i = 0, 1, \dots, N-1$ and $j = 0, 1, \dots, N/2-1$ are well defined.

7.5 The Vectorization Engine

The vectorization engine expects a scalar DAG (directed acyclic graph) represented by straight line code consisting of virtual scalar instructions in static single assignment (SSA) form as input. In the vectorization engine all virtual scalar instructions are replaced by virtual SIMD instructions. To achieve this goal, the vectorization engine has to find pairs of scalar floating-point instructions (and fusions of their respective operands), each yielding—in the optimal case—one SIMD floating-point instruction. In some cases, additional SIMD instructions may be required to obtain a SIMD construct that is semantically equivalent to the original pair of scalar instructions. The vectorization algorithm described hereby utilizes the infrastructure provided by MAP's vectorization engine. To describe the vectorization algorithm some definitions and explanations for describing the concepts of the vectorization engine, are provided.

Pairing

Pairing rules specify ways of transforming pairs of scalar instructions into a single SIMD instruction or a sequence of semantically equivalent SIMD instructions. A pairing rule often provides several alternatives to do so. The rules used in the vectorizer are classified according to the type of the scalar instruction pair: unary (i.e., multiplication by a constant), binary (i.e., addition and subtraction), and memory access type (i.e., load and store instructions). Pairings of the following instruction combinations are supported: (i) consecutive load/load, (ii) arbitrary load/load, (iii) consecutive store/store, (iv) arbitrary store/store, (v) unary/unary, (vi) binary/binary, (vii) unary/binary, (viii) unary/load, and (ix) load/binary. Not all of the above pairing combination allow for an optimal utilization of SIMD resources.

A *pairing ruleset* comprises various pairing rules. At the moment, the pairing ruleset does not comprise a rule for vectorizing an odd number of scalar store instructions.

Two scalar instructions are vectorized, i.e., *paired*, if and only if neither of them is already paired and the instruction types are matching a pairing rule from the utilized pairing rule set.

Fusion

Two scalar operands S and T are assigned together, i.e., *fused*, to form a SIMD variable of layout $ST = (S, T)$ or $TS = (T, S)$ if and only if they are the corre-

sponding operands of instructions considered for pairing and neither of them is already involved in another fusion. The position of the scalar variables S and T inside a SIMD variable (either as its lower or its higher part) strictly defines the fusion, i.e., $ST \neq TS$, see Fig. 7.2. A special fusion type containing the same scalar variable twice, i.e., $TT = (T, T)$, is needed for some types of code partly containing non-parallel program flow.

An instruction pairing rule forces the fusion layout for the corresponding scalar operands. For two scalar binary instructions, three substantially different layouts for assigning the four operands to two SIMD variables exist, namely (i) accumulate (*ACC*), (ii) parallel 1 (*PAR1*), and (iii) parallel 2 (*PAR2*). Fig. 7.2 illustrates their semantics in more detail.

A fusion $X_{12} = (X_1, X_2)$ is *compatible* to another fusion $Y_{12} = (Y_1, Y_2)$ if and only if $X_{12} = Y_{12}$ or $X_1 = Y_2$ and $X_2 = Y_1$. In the second case, a swap operation is required as an additional "special transformation" to use Y_{12} whenever X_{12} is needed. The number of special transformations is minimized in a separate optimization step to minimize the runtime of the vectorization process. Chapter 8 describes this along other optimization techniques in more detail.

Vectorization Levels

A vectorization level embodies a subset of rules from the overall set of pairing rules. Different subsets belonging to different levels comprise pairing rules of different versatility. Generally, more versatile rule sets lead to less efficient vectorization but allow to operate on codes featuring less parallelism. In contrast, less versatile rule sets are more restrictive in their application and are thus only usable for highly parallel codes. They assure that if a good solution exists at all, it is found resulting in highly performant code.

The vectorization engine utilizes three levels of vectorization in its search process as follows: First, a vectorization is sought that utilizes the most restrictive level, i.e., *full vectorization*. This vectorization level only provides pairing rules for instructions of the same type and allows quadword memory access operations exclusively. If full vectorization is not obtainable, a fallback to a less restrictive vectorization level, i.e., *semi vectorization*, is made. This level provides versatile pairing rules for instructions of mixed type and allows doubleword memory access operations. In the worst case, if neither semi vectorization nor full vectorization is feasible, a fallback is made to a vector implementation of scalar code, i.e., *null vectorization* is applied. Null vectorization rules allow to vectorize any code by leaving half of each SIMD instruction's capacity unused. Even null vectorization results in better performance than using legacy x87 code. Besides, null vectorization is used as default, without trying full and semi vectorization, when it is known in advance that vectorization is impossible because, e.g., of an odd number of scalar store instructions, etc.

Table 7.3 illustrates which rules are available for each vectorization level.

Operations	Vectorization Level		
	semi	full	null
<i>Store/Store</i>	×	×	—
<i>Load/Load</i>	×	×	—
<i>Load/Binary</i>	×	—	—
<i>Binary/Load</i>	×	—	—
<i>Unary/Unary</i>	×	×	—
<i>Unary/Any</i>	×	—	—
<i>Any/Unary</i>	×	—	—
<i>Binary/Binary</i>	×	×	—
<i>Null Vectorization</i>	—	—	×

Table 7.3: Rulesets for Three Vectorization Levels. *Semi vectorization* uses all transformation rule groups except null vectorization rules. *Full vectorization* only applies rules which refer to pairs of the same instruction type, i.e., binary/binary, unary/unary, ... In *null vectorization* there is no vectorization applied at all. Each scalar instruction is straightforwardly transformed into a semantically equivalent SIMD instruction.

7.5.1 The Vectorization Algorithm

MAP's vectorization algorithm implements a depth first search with chronological backtracking. The search space is given by applying rules given by the current vectorization level in an arbitrary order. Depending on how versatile/restrictive the utilized pairing rule set is, there can be many, one or no possible solution at all.

The vectorization algorithm performs the following steps:

Step 1: Initially, no scalar variables are fused and no scalar instructions are paired. The process starts by pairing two arbitrary store instructions and fusing the corresponding source operands. Should the algorithm backtrack without success, it tries possible pairings of store instructions, one after the other.

Step 2: Pick an existing fusion on the vectorization path currently being processed, whose two writing instructions have not yet been paired. As the scalar code is assumed to be in SSA form, there is exactly one instruction that uses each of the fusion's scalar variables as its destination operand. According to the vectorization level and the type of these instructions, an applicable pairing rule is chosen. If all existing fusions have already been processed, i.e., the dependency path has been successfully vectorized from the stores to all affected loads, start the vectorization of another dependency path by choosing two remaining stores. If all stores have been paired and no fusions are left to be processed, a solution has been found and the algorithm terminates.

Step 3: According to the chosen pairing rule, fuse the source operands of the scalar instructions if possible (i.e., none of them is already part of another fusion) or, if a compatible fusion exists use it instead.

<i>Code Name</i>	<i>Loads/Stores</i>	<i>Adds+Subs</i>	<i>Muls</i>	<i>Vectorization Runtime</i>
<code>frc_32</code>	32/32	156	42	<.1s
<code>frc_30</code>	30/30	158	56	5.6s
<code>ftw_17</code>	66/34	328	180	<.1s
<code>fn_128</code>	256/256	2164	660	0.7s
<code>fn_256</code>	512/512	5008	1656	2.2s

Table 7.4: Vectorization Runtimes. The table shows instruction counts as well as vectorization runtimes in seconds for various twiddle complex-to-complex, no-twiddle complex-to-complex, and real-to-halfcomplex FFTW codelets. The vectorization algorithm has been implemented in Objective Caml. The runtimes have been determined using Objective Caml v3.06 with native-code compilation on an 800 MHz AMD K7 processor.

Step 4: Pair the chosen instructions, i. e., substitute them by one or more according SIMD instructions.

Step 5: If a fusion or pairing alternative does not lead to a valid vectorization, choose another pairing rule. If none of the applicable rules leads to a solution, fail and backtrack to the most recent vectorization step.

Steps 2 to 5 are iterated until all scalar instructions are vectorized, possibly requiring new initialization carried out by Step 1 during the search process. If the search process terminates without having found a result, a fallback to the next more general vectorization level is tried, leading to null vectorization in the worst case.

If a given rule set is capable of delivering more than one valid solution, the order in which the pairing rules are tested is relevant for the result. This is used to favor specific kinds of instruction sequences by ranking the corresponding rules before the others. For instance, the vectorization engine is forced first to look for instructions implementing operation semantics directly supported by a given architecture, thus minimizing the number of extracted virtual instructions that have to be rewritten in the optimization step.

Runtime. Table 7.4 shows runtimes of the vectorization algorithm for some representative FFTW codelets. Even large codelet, e. g., $n = 256$, can be vectorized in a few seconds.

Example. Figures 7.3 to 7.5 contain an example for *one* possible solution of the vectorization algorithm. Fig. 7.3 depicts the scalar code, an FFT kernel of size 3, which is the input of MAP's vectorizer. Fig. 7.4 lists all transformations (together with the rule group number from Section 7.6) that yield the vectorization solution shown in Fig. 7.5. Due to the fact that different vectorizations are possible, this is just one example out of all possible solutions.

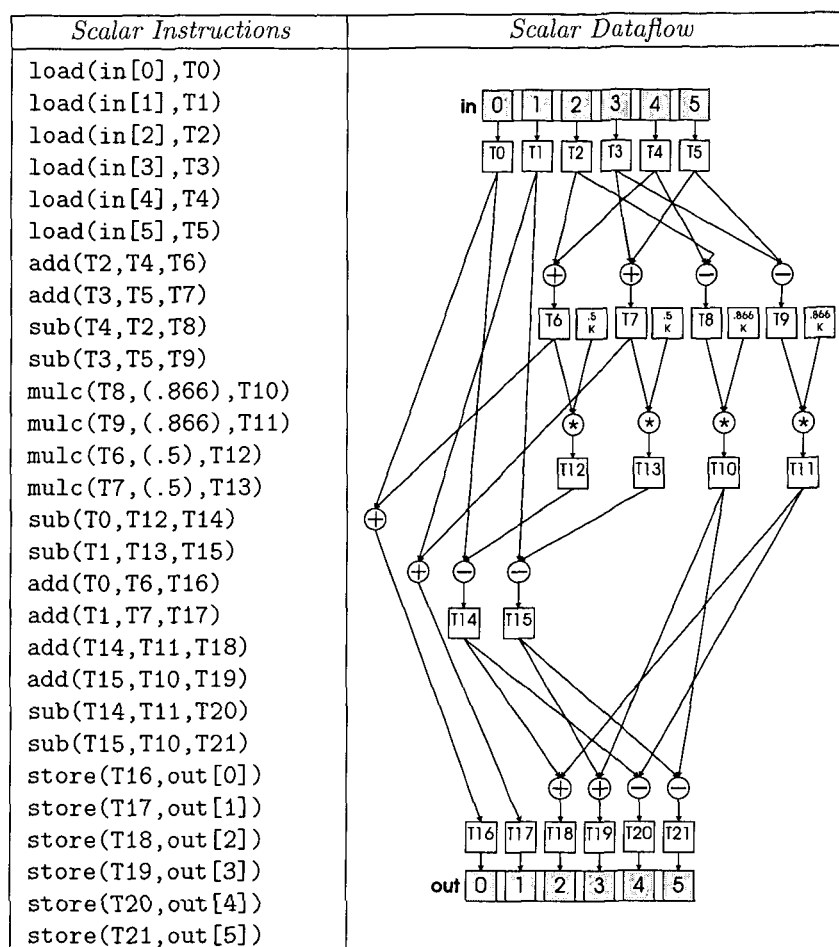


Figure 7.3: (Example) Scalar FFT of Size $N = 3$. The scalar instruction sequence in the left column corresponds to the scalar data flow layout depicted in the right column.

7.5.2 Vectorization Heuristics

The vectorization search space can be pruned by applying additional heuristic schemes that reduce the number of possible pairing partners in load/load and store/store vectorization. This pruning is done in a way to enforce the exploitation of obvious parallelism inherent in the code.

Full vectorization tries several heuristic schemes to vectorize memory access operations, whereas semi vectorization just relies on one of these schemes. Experiments have shown that the application of heuristic schemes significantly reduces the vectorization runtime.

Limiting Pairings for Memory Accesses

The vectorization search space can be pruned by utilizing heuristics to restrict the possible pairings for load/load and store/store vectorization rules (see Sec-

<i>Group</i>	<i>Scalar Sequence</i>	<i>SIMD Sequence</i>
1	load(in[0],T0) load(in[1],T1)	⇒ loadQ2(in2[01],T0T1)
1	load(in[2],T2) load(in[3],T3)	⇒ loadQ2(in2[23],T2T3)
1	load(in[4],T4) load(in[5],T5)	⇒ loadQ2(in2[45],T4T5)
5	add(T2,T4,T6) add(T3,T5,T7)	⇒ add2(T2T3,T4T5,T6T7)
5	sub(T4,T2,T8) sub(T3,T5,T9)	⇒ sub2(T2T3,T4T5,T8T9) mulc2(T8T9,(-1.,1.),T8T9)
3	mulc(T8,(.866),T10) mulc(T9,(.866),T11)	⇒ mulc2(T8T9,(.866,.866),T10T11)
3	mulc(T6,(.5),T12) mulc(T7,(.5),T13)	⇒ mulc2(T6T7,(.5,.5),T12T13)
5	sub(T0,T12,T14) sub(T1,T13,T15)	⇒ sub2(T0T1,T12T13,T14T15)
5	add(T0,T6,T16) add(T1,T7,T17)	⇒ add2(T0T1,T6T7,T16T17)
7	add(T14,T11,T18) add(T15,T10,T19)	⇒ swap2(T10T11,T11T10) add2(T14T15,T11T10,T18T19)
5	sub(T14,T11,T20) sub(T15,T10,T21)	⇒ sub2(T14T15,T11T10,T20T21)
2	store(T16,in[0]) store(T17,in[1])	⇒ storeQ2(T16T17,out2[0])
2	store(T18,in[2]) store(T19,in[3])	⇒ storeQ2(T18T19,out2[1])
2	store(T20,in[4]) store(T21,in[5])	⇒ storeQ2(T20T21,out2[2])

Figure 7.4: (Example) Scalar To Vector Transformation of an FFT of Size $N = 3$. The scalar instructions in the second left column are transformed into the vector instruction sequence in the right column. It will be explained in Section 7.6 that it is not always possible to transform two scalar instructions into one SIMD instruction. The leftmost column shows to which transformation rule group introduced in Section 7.6 the scalar to vector transformation belongs.

tions 7.6.1 and 7.6.2). The details are given in Tables 7.5 and 7.6. The heuristics immediately reject apparently suboptimal vectorization paths and enforce obvious parallelism onto the vectorization process. Full vectorization uses heuristics H0, H1 and H2 to vectorize memory accesses, semi vectorization just relies on heuristic H1.

Experiments have shown that these heuristics significantly reduce vectorization runtime by additionally pruning the vector code search space.

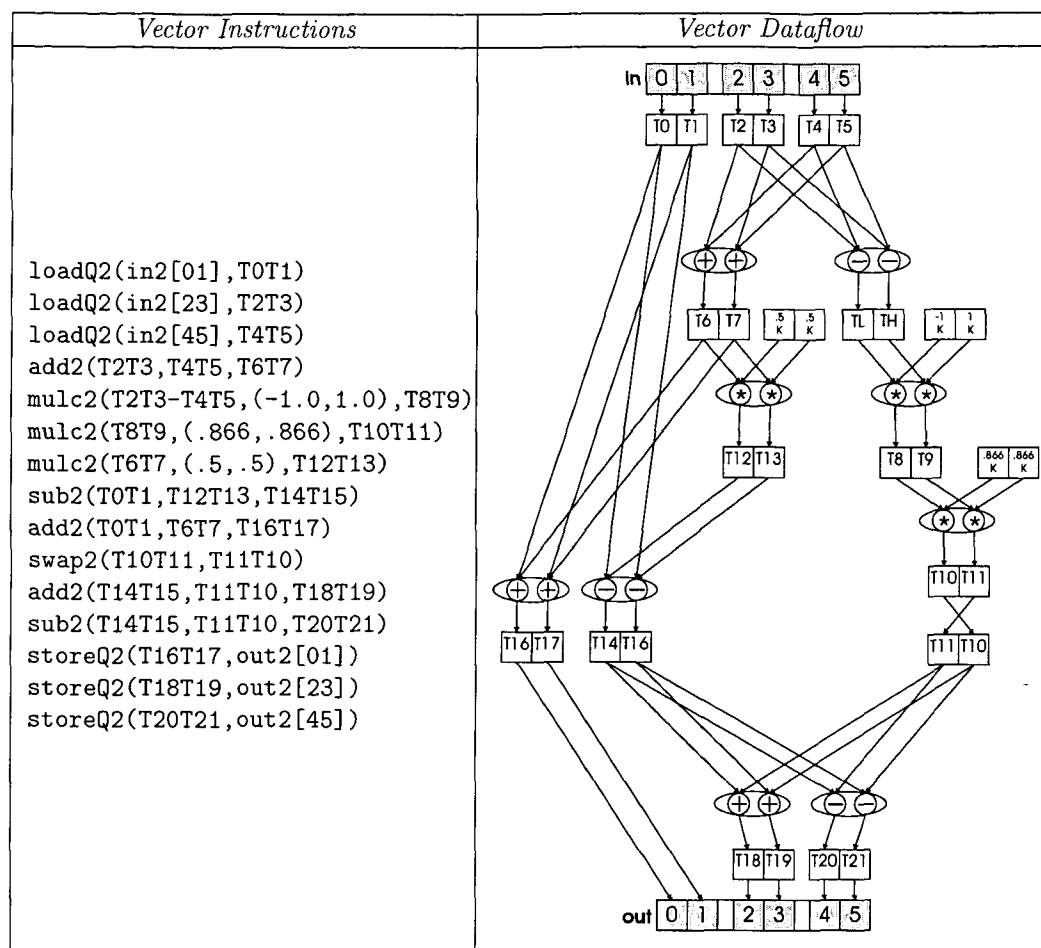


Figure 7.5: Example: 2-way Vector FFT of Size $N = 3$. The vector instruction sequence in the left column corresponds to the data flow layout depicted in the right column. This code is one possible result of the vectorizer when it is fed the scalar instruction sequence from Fig. 7.3.

Order of Application of Pairing Rules

If a given rule set is capable of delivering more than one valid solution, the order in which the pairing rules are tested is relevant for the result. This is used to favor specific kinds of instruction sequences by ranking the corresponding rules before the others. For instance, the vectorization engine is forced first to look for instructions implementing operations directly supported by a given architecture, thus minimizing the number of extracted virtual instructions that have to be rewritten in the optimization step.

For example, MAP's AMD K7 specific ruleset provides the rules in an order to favor intraoperand and swap instructions because they are inherent in this processor's target instruction set. On the other hand, the application order in Intel's Pentium 4 ruleset "tries" to avoid these instructions by only extracting them if no vectorization can be achieved otherwise, i. e., the preferred rules do

Memory Access		H0	H1	H2
Arbitrary		$ i_1 - i_2 = N/2$	$ i_1 - i_2 = N/2$	$ i_1 - i_2 = N/2$
Complex	Re/Re	$i_1 = 0; i_2 = N/2$	$i_1 = 0; i_2 = N/2$	$ i_1 - i_2 = N/4$
		$i_2 = 0; i_1 = N/2$	$i_2 = 0; i_1 = N/2$	
	Re/Im Im/Re	$i_1, i_2 <> 0; i_1 - i_2 = N/4$ or $i_1 = N/4; i_2 = N/4$	$i_1, i_2 <> 0$ $i_1 = i_2$	$i_1 + i_2 = N/2$
	Im/Im	not supported	not supported	not supported

Table 7.5: Heuristics for Store/Store Vectorization. In the leftmost column the possible types of store memory access in MAP are listed. For further details refer to [36]. The following columns show which constraints the indices i_1 and i_2 of the two stores have to satisfy for a pairing of the corresponding instructions to be allowed. N is the size of memory when addressed with scalar memory layout.

Memory Access		H0	H1	H2
Arbitrary		$ i_1 - i_2 = N/2$	$ i_1 - i_2 = N/2$	$ i_1 - i_2 = N/2$
Complex	Re/Re	not restricted	not restricted	not restricted
	Re/Im			
	Im/Re			
	Im/Im			

Table 7.6: Heuristics for Load/Load Vectorization. In the leftmost column the possible types of load memory access in MAP are listed. For further details refer to [36]. The following columns show which constraints the indices i_1 and i_2 of the two loads have to satisfy for a pairing of the corresponding instructions to be allowed. N is the size of memory when addressed with scalar memory layout.

not work. They are rewritten into supported instructions in a separate step after the vectorization process (see Chapter 8).

7.5.3 Handling Different Results of Vectorization

The backtracking search in the vectorization process might yield multiple solutions. Therefore, three strategies are introduced for selecting a result, namely (i) the pick first strategy, (ii) the pick best strategy, and (iii) the pick good strategy.

The *pick first strategy* immediately commits to the first solution. The *pick best strategy* lets some later compiler stage, ideally the last one, evaluate the quality of every obtained vectorized code and commits to the best solution. The *pick good strategy* restricts the set of regarded vectorization results to the first solutions obtained from the vectorizer.

7.6 Pairing Rules

Rules for scalar instruction pairing are fundamental to the pattern matching approach of the vectorization engine as described in the previous section.

The rules used in MAP's code generator can be classified according to the types of the two scalar instructions on which vectorization is performed, i.e., unary, binary and memory access type. Accordingly, eight groups of pairing rules exist: (i) load/load, (ii) store/store, (iii) unary/unary, (iv) load/binary, (v) binary/binary, (vi) unary/any, (vii) reordering of compatible fusions, and (viii) null vectorization. For each group, an introduction to its transformation's semantics as well as a specific code example will be given in the following sections.

7.6.1 Group 1: Load/Load Pairing Rules

This set of rules aims at transforming two scalar loads from memory into a SIMD load storing into a 2-way vector variable. In fact, there are two rules differing in the way they address main memory locations, as well as the way they are implemented in SIMD.

Example Rule	Scalar Sequence	⇒	Vector Sequence
Load from Consecutive Addresses		⇒	
	load(M.scal[4],D1) load(M.scal[5],D2)	⇒	loadQ2(M.vect[2],D1D2)
Load from Arbitrary Addresses		⇒	
	load(M.scal[2],D1) load(M.scal[6],D2)	⇒	loadD2(M.vect[1].1,A.1) loadD2(M.vect[3].1,B.1) unpackL2(A,B,D1D2)

Figure 7.6: (Example) Various Load Pairings. While scalar loads on consecutive addresses can be implemented in SIMD straightforwardly, the scalar loads on arbitrary addresses need to be rearranged. The load on consecutive addresses corresponds to a complex load while the load on arbitrary addresses corresponds to a split load on MAP input arrays.

Load from Consecutive Addresses

This transformation of two scalar loads to one SIMD load only works on two consecutive addresses.

If $\text{addr1} = \text{addr0} + 1$ with $\text{addr0} = 2*i$, $i = 0, 1, \dots, N/2-1$, the scalar loads are transformed into one SIMD load as demonstrated by the upper example in Fig. 7.6. In MAP's input arrays, a load on such consecutive memory cells corresponds to a *complex load*, where the first element is assumed to be the real part and the second element is the imaginary part of a complex number.

Load from Arbitrary Addresses

This type of SIMD load works on any two addresses. Therefore, addr0 and addr1 can address any valid position of the input array. This has the drawback that the two scalar loads from addr0 and addr1 cannot be realized using just one vector load instruction, i. e., `loadQ2`.

Therefore, the data elements are loaded into the lower parts of two temporary vector variables `A.1` and `B.1`, using two `loadD2` instructions. An `unpackL2` operation must be applied on `A.1` and `B.1` before the data can be used for computation. This instruction sequence is illustrated by the lower example in Fig. 7.6. In the case of MAP's input arrays, a load on arbitrary memory cells corresponds to a *split load*, where the first and the second element are both real elements of a real transform array or any combination of real and imaginary parts of a complex array.

7.6.2 Group 2: Store/Store Pairing Rules

This set of rules is targeted at transforming two scalar stores to memory into one SIMD store from a two-way vector variable. Again, there are two rules differing in the style they arrange the data locations in memory, and in the way they are implemented in SIMD.

Store to Consecutive Addresses

This type of transforming two scalar stores into one SIMD store only works if the 2-way vector variable is stored into two consecutive addresses.

If $\text{addr1} = \text{addr0} + 1$ with $\text{addr0} = 2*i$ where $i = 0, 1, \dots, N/2-1$, the scalar stores are transformed into one SIMD store as shown in the upper part of Fig. 7.7. In the case of MAP's output arrays, a store on such consecutive memory cells corresponds to a complex store, where the first element is the real part and the second element the imaginary part of a complex number. This rule is the counterpart to the load to consecutive addresses rule.

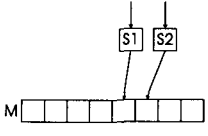
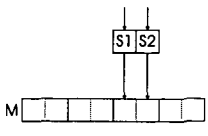
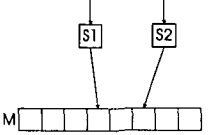
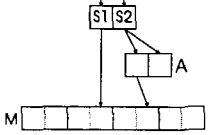
Example Rule	Sequence	\Rightarrow	Sequence'
Store to Consecutive Addresses		\Rightarrow	
	<code>store(S1,M.scal[4])</code> <code>store(S2,M.scal[5])</code>	\Rightarrow	<code>storeQ2(S1S2,M.vect[2])</code>
Store To Arbitrary Addresses		\Rightarrow	
	<code>store(S1,M.scal[3])</code> <code>store(S2,M.scal[5])</code>	\Rightarrow	<code>storeD2(S1S2.1,M.vect[1].h)</code> <code>unpackH2(S1S2,S1S2,A)</code> <code>storeD2(A.1,M[2].h)</code>

Figure 7.7: (Example) Various Store Pairings. This is the counterpart ruleset to the load rules from Fig. 7.6. The store to consecutive addresses corresponds to a complex store while the store to arbitrary addresses corresponds to a split store to MAP's output arrays.

Store to Arbitrary Addresses

This type of transforming two scalar stores into one SIMD store works on any two addresses. Therefore, `addr0` and `addr1` can address any valid position of the output array with the same drawback as its counterpart, i. e., load from arbitrary addresses. Two scalar stores to `addr0` and `addr1` cannot be realized with just one vector quad store instruction, i. e., `storeQ2`. The lower part of the SIMD variable can be directly stored to the corresponding memory location by a double store while the higher part has to be rearranged into a temporary vector variable by an unpack operation before it can be stored into memory from there. This approach is shown in the lower example of Fig. 7.7.

In the case of MAP's output arrays, a store on arbitrary memory cells corresponds to a split store whose function is analogous to split access that has been already described in the load case.

7.6.3 Group 3: Unary/Unary Pairing Rules

These rules are used to transform two scalar unary instructions into one vector instruction.

These transformations are performed straightforwardly as depicted in Fig. 7.8. Fig. 7.9 defines, according to the scalar machine model, how to transform combinations of scalar operations into semantically equivalent SIMD operations. Fig. 7.10 gives an example of a unary/unary transformation.

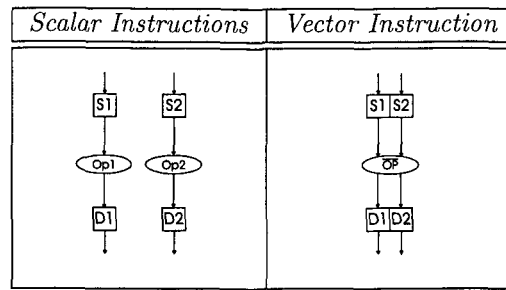


Figure 7.8: Pairing of Two Unary Instructions. Two scalar unary operations $Op1$ and $Op2$ are transformed into one SIMD operation \overline{Op} . The operation transformation is performed according to the unary operation algebra introduced in Fig. 7.9.

Operation 1	Operation 2	\Rightarrow	Operation'
neg	neg	\Rightarrow	chsLH2
neg	mulc(K)	\Rightarrow	mulc2(-1,K)
mulc(K)	neg	\Rightarrow	mulc2(K,-1)
mulc(K1)	mulc(K2)	\Rightarrow	mulc2(K1,K2)

Figure 7.9: Unary/Unary Operation Algebra. Two scalar operations are transformed into one semantically equivalent SIMD operation.

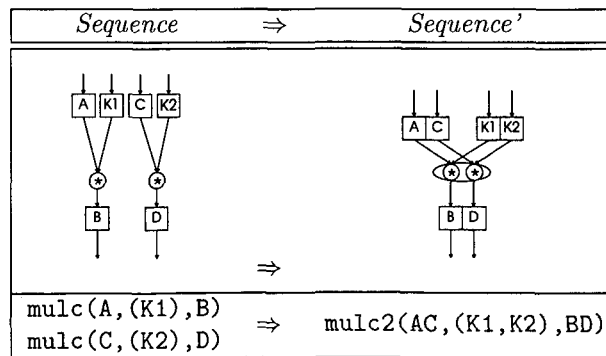


Figure 7.10: (Example) Application of a Unary/Unary Rule. Two scalar `mulc` instructions are transformed into a vector `mulc2` instruction. This is the transformation out of Group 3 which most frequently occurs in FFT kernel vectorization.

7.6.4 Group 4: Load/Binary Pairing Rules

This set of rules is used for transforming scalar load and binary instructions (addition or subtraction) into one vector instruction of intraoperand type.

Fig. 7.11 shows the binding of a load instruction with a binary instruction. Not all possible binary instructions supported by the virtual machine model of the vectorizer are supported in this ruleset. Fig. 7.12 illustrates which instructions are allowed to be paired with the unary load and which SIMD operations are utilized to transform these instructions into a semantically equivalent vector instruction sequence. Fig. 7.13 gives an example showing the application of a load/binary transformation.

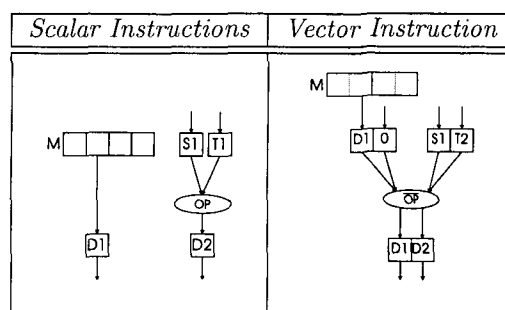


Figure 7.11: Pairing of Load and Binary Instructions. The scalar load and the binary operation Op are transformed into two SIMD instructions of which one is a SIMD double load. The operation \overline{Op} of the intraoperand vector instruction is set according to the algebra defined in Fig. 7.12. Not only the depicted fusion of load/binary but also the mirrored binary/load is allowed.

Operation	\Rightarrow	Operation'
add	\Rightarrow	accPP2
sub	\Rightarrow	accNN2

Figure 7.12: Load/Binary Operation Algebra. Scalar binary operations are transformed into semantically equivalent SIMD operations.

7.6.5 Group 5: Binop/Binop Pairing Rules

This set of rules is used to transform two scalar binary instructions into one single vector instruction or, if necessary, a sequence of vector instructions.

As there are intraoperand and parallel style binary vector instructions provided by the vectorizer's machine model, there are different but computationally equivalent ways of vectorizing scalar binary instructions. This brings about the benefit of widening the vectorization search space.

When pairing binary scalar instructions for vectorization, the respective four input operands have to be fused. There are three different ways (disregarding redundant equivalents) for fusing the four operands to two SIMD cells, namely (i) accumulate (ii) parallel 1, and (iii) parallel 2.

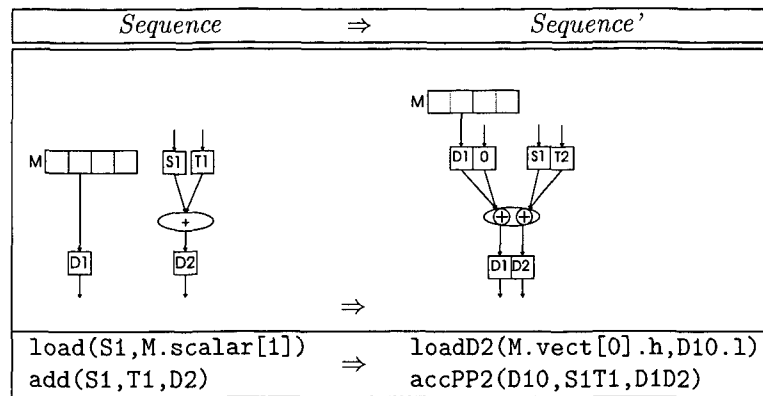


Figure 7.13: (Example) Application of a Load/Binary Rule. Scalar load and add instructions are transformed into one double load into the lower part of a SIMD variable whose higher part is initialized with 0. The accumulate instruction leaves the loaded value unchanged and performs the addition.

The *accumulate binding type* (*ACC*) fuses input operands in a way allowing intraoperand SIMD instructions to be performed. The *parallel 1* (*PAR1*) and the *parallel 2* (*PAR2*) binding type are mirrored cases for fusing operands in a way such that parallel SIMD operations like addition, subtraction and multiplication can be performed.

Fig. 7.14 illustrates the three binding types in more detail. Fig. 7.15 gives an overview of all transformations depending on the binding type. All possible binary instruction pairs are presented together with their transformation into SIMD instruction sequences. Fig. 7.16 depicts an example for computationally equivalent SIMD transformations of a scalar addition and subtraction instruction.

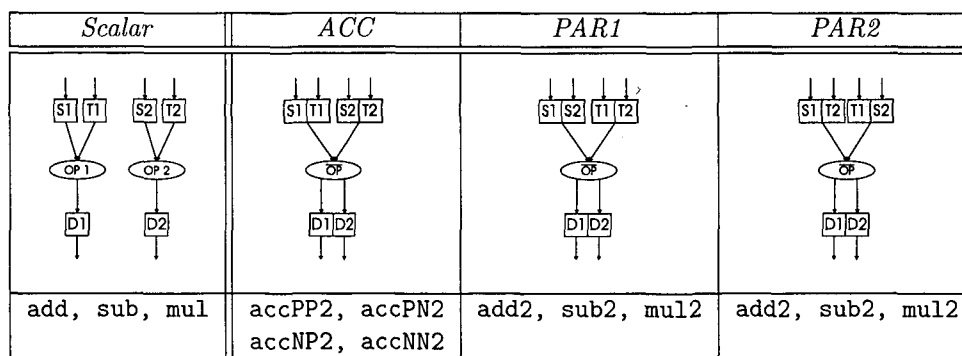


Figure 7.14: Binary Binding Types. The operands *S1*, *T1* and *S2*, *T2* of the scalar binary operations in the leftmost column can be fused in three different ways shown in columns two, three and four. Accumulate style operand fusion allows the scalar operations *Op1* and *Op2* to be transformed into an intraoperand vector operation \overline{Op} . Parallel 1 and Parallel 2 operand fusions allow scalar operations to be transformed into a parallel style operation. The last row shows feasible operations that *Op* can be transformed into.

<i>Binding</i>	<i>ADD/ADD</i>	<i>ADD/SUB</i>	<i>SUB/ADD</i>	<i>SUB/SUB</i>	<i>MUL/MUL</i>
<i>ACC</i>	accPP2	accNP2 swap2	accNP2	accNN2	—
<i>PAR1</i>	add2	chsL2 sub2	chsL2 add2	sub2	mul2
<i>PAR2</i>	add2	chsH2 add2	chsL2 add2	sub2 chsH2	mul2

Figure 7.15: Binding Types for Two Binary Operations. The columns specify the scalar operations and their transformations into semantically equivalent SIMD instructions. The rows are indexed by the different binding types and therefore indicate which transformations are allowed together with which of the binding types.

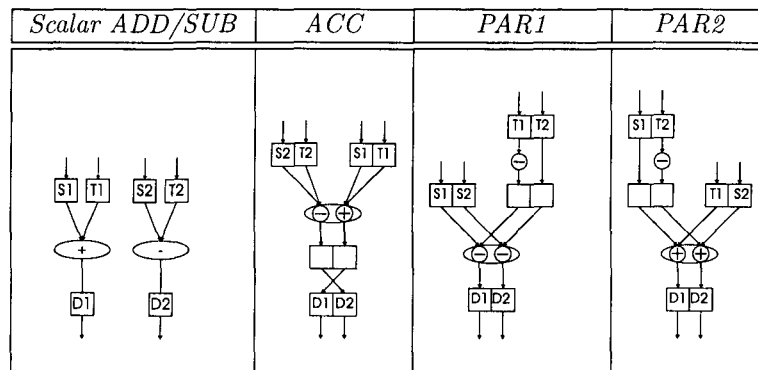


Figure 7.16: (Example) Binary Bindings for ADD/SUB. The ADD/SUB case has been chosen because of its nontrivial transformation from scalar add and sub instructions to a sequence of more than one semantically identical SIMD instructions.

7.6.6 Group 6: Unary/Any Pairing Rules

This set of rules is used to pair a unary instruction with any load or binary instruction. It is characteristic for this set of rules, that the source operand of the unary instruction is always fused with the destination operand of the other instruction. The vector operation is chosen such that it performs a dummy operation on the vector register part which holds the destination of the preceding “any” instruction. Not only a rule for the combination unary/any but also for any/unary is provided. Figures 7.17 and 7.18 illustrate this approach in more detail.

7.6.7 Group 7: Reordering of Compatible Fusions

As stated in Section 7.5, every scalar variable is allowed to be a member of not more than one fusion. Therefore, if two scalar operands A and B already form a SIMD cell fusion of layout $AB = (A, B)$, but a fusion of layout $BA = (B, A)$ is needed, an additional vector swap instruction with AB as its input is generated.

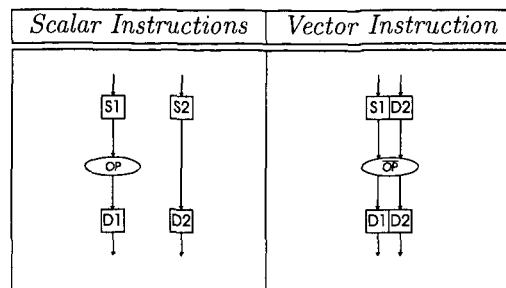


Figure 7.17: Pairing of a Unary Operation with Any Other Operation. It is specific for this pairing type that not the sources $S1$ and $S2$ of the two scalar operations are fused, but the source of the unary operation $S1$ with the destination $D2$ of the “any” operation which remains unchanged by the vector operation \overline{Op} . The source operands may also be fused the other way round, i. e., $D2S1$.

Operation	\Rightarrow	Operation'
neg	\Rightarrow	mulc2(-1,1)
mulc(K)	\Rightarrow	mulc2(K,1)

Figure 7.18: Unary/Any Operation Algebra. The scalar unary operation is transformed into a semantically equivalent SIMD Operation which performs a dummy operation on the corresponding part of the fused SIMD register by simply multiplying one.

This preserves the existing fusion AB , but also provides the desired SIMD cell with switched operands. Two examples are given in Fig. 7.19.

Group 8: Null Vectorization Pairing Rules

This set of rules is used to replace one arbitrary scalar instruction by one equivalent vector instruction. The scalar operands are transformed into SIMD operands by putting the scalar content into the lower part of a SIMD cell. The instruction count is not reduced, thus, no satisfactory utilization of the two-way SIMD infrastructure can be achieved this way. This ruleset is to be used only at the null vectorization level.

Fig. 7.20 depicts all rules for transforming single scalar operations into vectorial ones.

Example Rule	Scalar Sequence	⇒	Vector Sequence
Insert Swap for Load Vectorization			
	<pre>load(M.scal[4], A) load(M.scal[5], B) add(B, D, C) add(A, E, F)</pre>	⇒	<pre>loadQ2(M.vect[2], T1T2) swap2(T1T2, BA) add2(BA, DE, CF)</pre>
Insert Swap for Other Instruction Vectorization			
	<pre>add(A, B, C) add(D, E, F) sub(D, G, H) sub(A, I, J)</pre>	⇒	<pre>swap2(DA, T1T2) add2(T1T2, BE, CF) sub2(DA, GI, HJ)</pre>

Figure 7.19: (Example) Generation of Swaps Needed for Vectorization. All scalar instruction sequences' dataflow is downwards while the vectorization is performed bottom up. In the upper example, the two scalar loads cannot be vectorized into `loadQ2(M.vect[2], AB)`. This happens because A and B are already involved in pairing BA. Therefore, a temporary SIMD variable T1T2 is needed along with `swap2(T1T2, BA)` to preserve the dataflow semantics of the scalar DAG. In the lower example, the scalar add instructions are not vectorized to `add2(AD, BE, CF)` because the pairing DA already exists. The temporary SIMD variable T1T2 is used along with `swap2(DA, T1T2)` to enable the vectorization of the scalar add instructions.

<i>Operation</i>	\Rightarrow	<i>Operation'</i>
load(M[i],A)	\Rightarrow	loadD2(M[i],A2.1)
store(A,M[i])	\Rightarrow	storeD2(A2.1,M[i])
mulc(A,K,B)	\Rightarrow	mulc2(A2,K,1,B2)
neg(A,B)	\Rightarrow	chsL2(A2,B2)
add(A,B,C)	\Rightarrow	add2(A2,B2,C2)
sub(A,B,C)	\Rightarrow	sub2(A2,B2,C2)
mul(A,B,C)	\Rightarrow	mul2(A2,B2,C2)

Figure 7.20: Null Vectorization Rules. In the case of null vectorization, a general agreement is to rearrange content of the scalar registers A,B,... into the lower parts A.1,B.1,... of the vector cells A2,B2,... Under this assumption it is possible to transform the scalar unary, binary and load/store instructions into equivalent SIMD instructions.

Chapter 8

Rewriting and Optimization

During the non-deterministic search process of the vectorizer, it is not clear if a (global) solution of the search process exists at all, as the vectorizer maintains only weak (local) consistency, ensuring that all SIMD instructions extracted so far are compatible with each other.

For that reason, it is better to postpone all further optimization of the SIMD code until the vectorizer can guarantee that a solution exists. Delaying the following optimization steps, i.e., *peephole optimization* and *scheduling*, until one particular solution has been found entails two positive effects: (i) The implementation of the vectorizer is kept simple as vectorization and optimization are not mixed up. (ii) An overall speed-up of the vectorization process is achieved, as no effort is wasted in incrementally optimizing partial (local) solutions that are not a part of a global solution.

Once the vectorizer has found a solution and committed to it, the resulting SIMD SSA code is put into the optimizer that consists of two parts.

The first part is a peephole optimizer that matches a set of transformation rules against the SIMD code, substituting code sequences by optimized, semantically equivalent code, until a fixed point is reached.

The implemented rewriting rules perform both (i) commonplace compiler optimization (carrying out dead code elimination, copy propagation, constant folding, and redundant instruction elimination, a special case of common subexpression elimination) and (ii) use of machine specific idioms.

Moreover, the vectorization engine uses an abstract machine model, whose instructions are not necessarily supported on a specific target architecture (see Section 7.4).

Therefore, it is necessary to optimize with respect to target-architecture independent as well as target-architecture dependent criterions.

The second part of the optimizer topologically sorts the DAG in an attempt to maximize register usage.

The result of that second part of the optimizer, i.e., vectorized, optimized, and scheduled SIMD code, can be put into the MAP backend for compilation to x86 assembly or can be directly emitted to intrinsic code using the portable SIMD API.

The remaining parts of this chapter are organized as follows. Section 8.1 describes the goals targeted by the peephole optimization. Section 8.2 introduces the peephole optimizer, explains how it works, and describes what kind of optimizations

it performs. Section 8.4 shows how the scheduler works and how its output can be further refined in certain cases.

8.1 Optimization Goals

The peephole optimizer tries to maximize the computational performance of the input code by pursuing the following goals.

Reduce the Number of Instructions. The output of the vectorizer usually includes many redundant operations, e. g., SIMD swaps. By applying dead code elimination, constant folding, copy propagation, and by rewriting SIMD specific code patterns, the size of the SIMD code can be reduced significantly.

Shorten the Length of the Critical Path. The length of the critical path ([105]) of a computational task defines the minimum execution time the computation must inevitably take, regardless of the maximum amount of computation allowed to be carried out simultaneously. In certain cases, the length of the critical path of the data dependency graph can be shortened by replacing a dependency on some instruction with a dependency on a predecessor of that particular instruction. Applying rules of this group slightly increases the register pressure.

Reduce the Number of Source Operands. Some transformation rules reduce the number of source operands needed to perform an operation, locally reducing the register pressure.

Exploit Target Specific SIMD Features. These rules only work on machines offering intra-operand SIMD instructions, as they focus on substituting patterns using a parallel SIMD style with sequences of intra-operand SIMD instructions.

Rewrite Unsupported Instructions. As some SIMD instructions, defined by the vectorizer's virtual hardware architecture, are extracted during the vectorizer process which may not be available on a particular target machine, all unsupported instructions need to be rewritten into combinations of instructions actually supported by the target hardware.

8.2 Peephole Optimization

Peephole optimization is a local code rewriting technique. A window (*peephole*) uncovers a (small) set of instructions that is considered for optimization. The instructions dealt with inside a peephole are connected by dependencies [2]. Then, if possible, optimizing transformations are applied to the actually considered set of instructions. More specifically, a combination of instructions that matches a corresponding pattern is improved according to the associated transformation rule. The newly obtained sequence is either shorter or comprises faster but semantically equivalent instructions.

It is a typical characteristic of peephole optimization that new rewriting possibilities arise after carrying out a first set of transformations, i. e., several iterations are needed to yield a satisfactory optimization result. Because of the locality of the approach, peephole optimization usually is not able to eliminate all redundancies in a given code.

8.2.1 The Local Rewriting Process

The core of the peephole optimization module described in this chapter is a rule based local rewriting engine. In the following, definitions needed to describe its basic functionality are given.

Transformations. A transformation adds and/or removes instructions to/from a given sequence of instructions.

Instruction Patterns are used to identify groups of instructions. They either match instruction types (e.g., `unaryX2`, `binaryX2`, `loadX2` or `storeX2`) or specific instructions (e.g., `loadQ2`, `mulc2`, ...).

Rewriting Rules. A rewriting rule is specified by an instruction pattern and the corresponding transformations.

Rule Guards may be used to control the application of a rewriting rule.

Firing Rules. A rule *fires*, if and only if its pattern successfully matches the considered instructions. In this case the corresponding transformations are performed.

Rule Sets. A rule set is an ordered set of distinct rewriting rules.

8.2.2 The Rewriting Engine

Because of the locality of the peephole optimization, certain combinations that could easily be optimized, can not be rewritten.

There are (at least) two different methods addressing this problem. First, the size of the peephole could be enlarged. Secondly, peephole optimization could be split into multiple consecutive passes that include different rules to move instructions into the peephole of some other rules, enabling further optimization.

As the first strategy would immensely increase the number and the complexity of the optimization rules, hindering verification and debugging, MAP's peephole optimizer implements the second strategy.

Therefore, the rewriting engine performs three optimization steps with three different rule sets. Steps one and two optimize within the scope of the vectorizer's machine model. The third and final optimization step optimizes and transforms code from the vectorizer's machine model to the target machine model. Everyone of these steps is carried out as long as improvements to the instruction DAG are yielded.

The rule sets for step one and two are identical, except for the so-called *code motion rule*. While step one's code motion rule moves swap instructions up, step two's code motion rule moves swap instructions down in the DAG. Thus, local instruction sequences allowing for further optimization may be obtained. The up rule and the down rule cannot be combined into a single optimization step, because the up and down code motion would lead to non-termination.

8.2.3 The Order of Rule Application

During the rewriting process, the code is under permanent alteration. Rules not firing in the first place might become applicable through code transformations performed by other rules. Taking this rule "interaction" into consideration is a prerequisite for achieving a good overall optimization result. So the code rewriting process is implemented iteratively. An ordered sequence of rules—provided by a rule set—is specified for any individual optimization step. The rules are tried for their applicability as described in the following:

For a given instruction sequence that is to be rewritten, one rule after the other is matched for its applicability. As soon as a suitable rule is found, the according transformation is applied and the process starts anew with the first rule. The first rules in the rule set are therefore favored and will be checked more frequently. This is continued as long as at least one rule matches and optimizes the instruction DAG.

8.3 Transformation Rules

This section describes the target architecture specific and unspecific transformation rules utilized by the local rewriting system. The first class of transformations is divided into subgroups according to the different optimization goals they are targeted at. The second class has a subgroup for each supported target architecture.

8.3.1 Target Architecture Independent Rules

Three groups of optimization rules, namely (i) substitution, (ii) sign change specific, and (iii) code motion rules, do not depend on the target architecture and can therefore be applied straightforwardly to the vectorizer's output.

The Substitution Rules

This first group of rules is intended to perform *copy propagation* on (i) registers, (ii) stores, (iii) unary, and (iv) binary short vector instructions.

In most cases, copy propagation can be performed straightforwardly as the first three examples in Fig. 8.1 illustrate. The last group of examples in Fig. 8.1

deals with nonstandard register substitutions utilizing special properties of vector operations. These are exclusively algebraic sign identities and are shown in Fig. 8.2. In the following, the substitution rules are described in more detail.

Register and Store Substitution. For an arbitrary unary, binary or store instruction I2 that consumes a temporary variable B, get B's producer instruction I1. If and only if I1 is a copy instruction, replace B by the copy's source operand.

Unary and Binary Substitution. For two identical unary or binary instructions I1 and I2 that have the same source operands, replace I2 by a copy instruction of I1's destination into I2's destination.

Nonstandard Unary Substitution. For two read-after-write dependent unary instructions, replace the second instruction according to the rules given in Fig. 8.2.

Rule Type	Sequence	\Rightarrow	Sequence'
Register	copy2(A,B) chsL2(B,C)	\Rightarrow	copy2(A,B) chsL2(A,C)
	copy2(A,B) add2(B,C,D)	\Rightarrow	copy2(A,B) add2(A,C,D)
Store	copy2(A,B) storeD2(hi,B,M,3)	\Rightarrow	copy2(A,B) storeD2(hi,A,M,3)
	copy2(A,B) storeQ2(B,M,2)	\Rightarrow	copy2(A,B) storeQ2(A,M,2)
Unary	chsh2(A,B) chsh2(A,C)	\Rightarrow	chsh2(A,B) copy2(B,C)
Binary	sub2(A,B,C) sub2(A,B,D)	\Rightarrow	sub2(A,B,C) copy2(C,D)
Nonstandard Unary	chsL2(A,B) chsh2(B,C)	\Rightarrow	chsL2(A,B) chsLH2(A,C)
	chsh2(A,B) mulc2(B,(N1,N2),C)	\Rightarrow	chsh2(A,B) mulc2(A,(N1,-N2),C)
	mulc2(A,(N1,N2),B)	\Rightarrow	mulc2(A,(N1,N2),B)
	mulc2(B,(M1,M2),C)	\Rightarrow	mulc2(A,(N1*M1,N2*M2),C)

Figure 8.1: (Example) Subgroups of Substitution Rules. The register, binary and unary substitution transformations are transformed straightforwardly into *Sequence'*. The nontrivial register operations cannot be transformed that easy as they involve vector algebra specific transformations as described in Fig. 8.2.

The Sign Change Specific Rules

Whenever two scalar binary instructions, one of them implementing a subtraction, the other one an addition operation, are paired in the vectorization process, an additional sign change instruction is unavoidable¹ for implementing a parallel

¹All parallel operations currently available in modern processor instruction sets, e. g., Intel Pentium 4, AMD K7, or AMD K6, just support add/add or sub/sub but no mixed add/sub or

<i>Operation 1</i>	<i>Operation 2</i>	\Rightarrow	<i>Operation'</i>
Swap2	Swap2	\Rightarrow	Identity
chsL2	chsL2	\Rightarrow	Identity
chsL2	chsH2	\Rightarrow	chsLH2
chsL2	chsLH2	\Rightarrow	chsH2
chsH2	chsL2	\Rightarrow	chsLH2
chsH2	chsH2	\Rightarrow	Identity
chsH2	chsLH2	\Rightarrow	chsL2
chsLH2	chsL2	\Rightarrow	chsH2
chsLH2	chsH2	\Rightarrow	chsL2
chsLH2	chsLH2	\Rightarrow	Identity
chsL2	mulc2(N,M)	\Rightarrow	mulc2(-N,M)
chsH2	mulc2(N,M)	\Rightarrow	mulc2(N,-M)
chsLH2	mulc2(N,M)	\Rightarrow	mulc2(-N,-M)
mulc2(N,M)	chsL2	\Rightarrow	mulc2(-N,M)
mulc2(N,M)	chsH2	\Rightarrow	mulc2(N,-M)
mulc2(N,M)	chsLH2	\Rightarrow	mulc2(-N,-M)
mulc2(N1,M1)	mulc2(N2,M2)	\Rightarrow	mulc2(N1*N2,M1*M2)

Figure 8.2: Nonstandard Operand Transformations for Register Substitution. *Operation 2* is transformed into *Operation'* to eliminate the read-after-write dependency between *Operation 1* and *Operation 2*. These transformations are supposed to remove as many register dependencies between operations as possible to allow for later optimization.

add/sub or sub/add vector operation.

Sign change specific transformations remove sign change instructions as well as rewrite sign changes in combinations with other instructions into more optimal code sequences.

Some sign change specific rules are used to find code sequences including sign change instructions where optimization allows to minimize the overall instruction count. Other rules, are used to rewrite sign change instructions into swap instructions which are generally known to be cheaper or at least of the same quality in terms of latency and throughput. The only constraint is that the overall instruction count is not to be increased.

Fig. 8.3 exemplarily shows one characteristic example for each of these rule targets. In Fig. 8.4 all utilizable rules are given showing the basic transform idea without further details about operand registers and their constraints.

The Code Motion Rules

This set of rules is used to topologically shift swap instructions up and down in the considered instruction sequence. It allows for the interchange of swap instructions with unary or binary instructions, which opens appliance possibilities for other rules at first not firing. The code motion rules which interchange a swap

sub/add as 2-way short vector operand combinations.

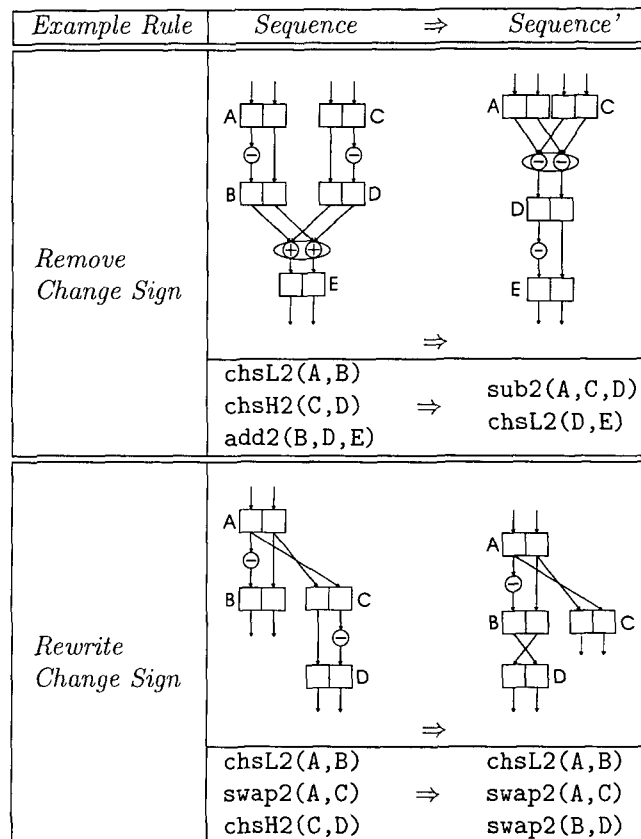


Figure 8.3: (Example) Change Sign Specific Transformations. One example rule for each subgroup of these transformation rules is given.

Sequence	\Rightarrow	Sequence'	Goal
$\text{chsL2}, \text{chsH2}, \text{add2}, \text{add2}$	\Rightarrow	$\text{chsL2}, \text{add2}, \text{sub2}$	instr
$\text{chsH2}, \text{chsL2}, \text{add2}$	\Rightarrow	$\text{sub2}, \text{chsH2}$	instr
$\text{chsL2}, \text{chsH2}, \text{add2}$	\Rightarrow	$\text{sub2}, \text{chsL2}$	instr
$\text{chsL2}, \text{swap2}, \text{chsH2}$	\Rightarrow	$\text{chsL2}, \text{swap2}, \text{swap2}$	swap
$\text{chsH2}, \text{swap2}, \text{chsL2}$	\Rightarrow	$\text{chsH2}, \text{swap2}, \text{swap2}$	swap

Figure 8.4: Rewriting Ruleset for Change Sign Specific Transformations. All combinations of remove and rewrite change sign rule transformations are given together with the goal they aim at. The goal *instr* aims at minimizing the instruction count. *swap* occurs when the associated transformation generates this operation.

instruction with an arbitrary sign change instruction are used in the rulesets for optimization steps one and two as described in Section 8.2.2.

Fig. 8.5 gives an example for the functionality of these code motion transformations. Fig. 8.6 briefly illustrates all available code motion transformations.

Example Rule	Sequence \Rightarrow Sequence'
Binary Code Motion	
	$\text{swap2}(A,B)$ $\text{swap2}(C,D)$ $\text{add2}(B,D,E)$ \Rightarrow $\text{swap2}(A,B)$ $\text{add2}(A,C,D)$ $\text{swap2}(D,E)$
Unary Code Motion	
	$\text{chsL2}(A,B)$ $\text{swap2}(B,C)$ \Rightarrow $\text{swap2}(A,B)$ $\text{chsH2}(B,C)$

Figure 8.5: (Example) Code Motion Transformations. Code motion transformations are intended to change the topological position of swap operations inside the examined instruction sequence.

Sequence	\Rightarrow	Sequence'
$\text{swap2}, \text{swap2}, \text{binop2}$	\Rightarrow	$\text{swap2}, \text{binop2}, \text{swap2}$
$\text{swap2}, \text{binop2}, \text{swap2}$	\Rightarrow	$\text{swap2}, \text{swap2}, \text{binop2}$
$\text{mulc2}, \text{swap2}$	\Rightarrow	$\text{swap2}, \text{mulc2}$
$\text{swap2}, \text{mulc2}, \text{mulc2}$	\Rightarrow	$\text{swap2}, \text{mulc2}, \text{mulc2}$
$\text{chsL2}, \text{swap2}$	\Rightarrow	$\text{swap2}, \text{chsH2}$
$\text{swap2}, \text{chsL2}$	\Rightarrow	$\text{chsH2}, \text{swap2}$

Figure 8.6: Rule Set for Code Motion Transformations. The shift transformations allow for topological position interchange of swap operations with unary *mulconst* operations and all types of binary operations.

8.3.2 Target Architecture Specific Rules

So far rule groups have been introduced whose code transformations were specific to the vectorizer's machine model. These are independent of the final target architecture. The instruction set specific rules address the fact that the vector code utilizing the vectorizer's machine model is not necessarily compliant and/or optimal for every target architecture.

Although the vectorizer tries to avoid the extraction of SIMD instructions that are not supported by the target architecture (e. g., intra-operand instructions on a Intel Pentium 4), such instructions are sometimes needed to prevent the vectorization from failure. Whenever the vectorizer extracts instructions that are not available on the target architecture, the optimizer first tries to improve code sequences containing these instructions by applying target architecture independent optimization rules.

As target architecture independent optimization rules have no specific knowledge about the instructions actually supported by the target machine, they do not convert intra-operand instructions into parallel instructions (or vice versa).

Target architecture specific rules are applied in a final transformation step of MAP's peephole optimization module where the code is rewritten into an architecture compatible form. Code for the (i) AMD K7 (ii) AMD K6, and (iii) Intel P4 instruction sets can be generated. Afterwards, additional optimizations are performed targeted at the peculiarities of architecture specific instructions present in the new vector code.

Specific Rules for AMD's K7

Fig. 8.7 presents examples for transforming code to the *AMD K7 Virtual Machine Model*. Accumulate instructions are generated and optimizations concerning these highly efficient instructions are performed. Fig. 8.8 shows the basic AMD K7 transformation ideas without further operand and constraint details.

Specific Rules for AMD's K6

Fig. 8.9 presents examples for transforming instructions unsupported by the *AMD K6 Virtual Machine Model* into supported ones. Fig. 8.10 shows the basic AMD K6 transformation ideas without further operand and constraint details.

Specific Rules for Intel's Pentium 4

Fig. 8.11 shows example transformations for the *Intel P4 Virtual Machine Model*. Fig. 8.12 shows the basic Intel Pentium 4 transformation ideas without further operand and constraint details.

8.4 The Scheduler

The scheduling process of MAP's optimizer is virtually identical with its counterpart in FRTW 2.1.3. It has been extended from operating on scalar instructions to be equivalently useable on virtual SIMD instructions. In the following, the goals and the basic functionality of the scheduler will be described.

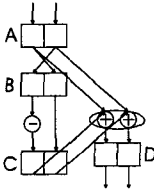
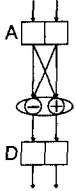
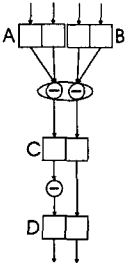
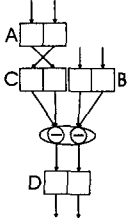
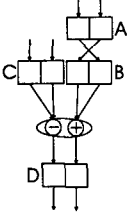
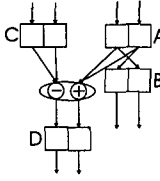
Example Rule	Sequence \Rightarrow Sequence'
<i>Rewrite for Accumulate</i>	 \Rightarrow  $\text{swap2}(A,B)$ $\text{chsL2}(B,C)$ $\text{add2}(A,C,D)$ \Rightarrow $\text{accNP2}(A,A,D)$
<i>Rewrite for Change Sign</i>	 \Rightarrow  $\text{accNN2}(A,B,C)$ $\text{chsL2}(C,D)$ \Rightarrow $\text{swap2}(A,C)$ $\text{accNN2}(C,B,D)$
<i>Copy Propagation</i>	 \Rightarrow  $\text{swap2}(A,B)$ $\text{accNP2}(C,B,D)$ \Rightarrow $\text{swap2}(A,B)$ $\text{accNP2}(C,A,D)$

Figure 8.7: (Example) Transformations for AMD's K7. One example rule for each subgroup of transformation rules is given.

The scheduling phase produces a topologically sorted order of the DAG, i.e., a list of static single assignments, called a *schedule*. This schedule can be executed by a sequential processor. The goal of scheduling is to minimize the variable lifetime in the resulting code to enhance locality, i.e., reduce register pressure by reducing register spills.

The scheduling process is divided into a scheduling phase and into an annotated scheduling phase.

The *scheduling phase* transforms the DAG into a recursive decomposition of serial and parallel sub-DAGs. A serial decomposition specifies that a sub-DAG

<i>Sequence</i>	\Rightarrow	<i>Sequence'</i>	<i>Goal</i>
swap2, accXp2	\Rightarrow	swap2, accXp2	reg
swap2, accpX2	\Rightarrow	swap2, accpX2	reg
swap2, accNP2	\Rightarrow	swap2, accNP2	reg
swap2, swap2, nnacc2, mulc2	\Rightarrow	swap2, swap2, nnacc2, mulc2	reg
accXX2, swap2	\Rightarrow	accXX2	instrcnt
swap2, chsL2, sub2	\Rightarrow	accNP2, swap2	acc
swap2, chsL2, add2	\Rightarrow	accNP2	acc
swap2, chsH2, add2	\Rightarrow	accPN2, swap2	acc
accNN2, chsL2	\Rightarrow	swap2, accNN2	swap
accNN2, chsH2	\Rightarrow	swap2, accNN2	swap

Figure 8.8: Rule Set for AMD K7 Transformations. All AMD K7 specific rules are given together with the goals they are objected at. The goals *reg* and *instrcnt* are targeted at minimizing the number of operand registers and the instruction count. *acc* and *swap* are quoted when the associated transformation generates these operations.

<i>Example Rule</i>	<i>Sequence</i>	\Rightarrow	<i>Sequence'</i>
<i>Rewrite Accumulate</i>			
	accNP2(A,B,C)	\Rightarrow	chsH2(A,T) accPP2(T,B,C)
<i>Rewrite Swap</i>			
	swap2(A,B)	\Rightarrow	unpackL2(A,A,T) unpackH2(A,T,B)

Figure 8.9: (Example) Transformations for AMD's K6. As the AMD K6 does not support swap operations and not all combinations of accumulate operations, they have to be rewritten.

D1 has dependencies to another sub-DAG D2 and therefore D1 has to be executed before D2. In a parallel decomposition the relative execution order of a sub-DAG is not specified.

<i>Sequence</i>	\Rightarrow	<i>Sequence'</i>
swap2	\Rightarrow	unpackL2, unpackH2
swap2	\Rightarrow	unpackH2, unpackL2
accNN2	\Rightarrow	unpackL2, unpackH2, sub2
accNP2	\Rightarrow	unpackH2, accPP2

Figure 8.10: Rule Set for AMD K6 Transformations. This figure contains all rewriting combinations of unsupported operations on the AMD K6.

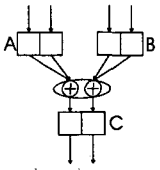
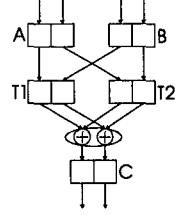
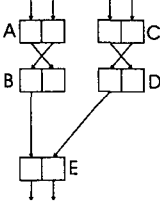
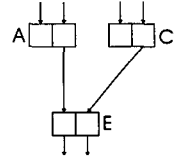
<i>Example Rule</i>	<i>Sequence</i>	\Rightarrow	<i>Sequence'</i>
<i>Rewrite Accumulate</i>		\Rightarrow	
	accPP2(A,B,C)	\Rightarrow	unpackL2(A,B,T1) unpackH2(A,B,T2) add2(T1,T2,C)
<i>Rewrite Swap</i>		\Rightarrow	
	swap2(A,B) swap2(C,D) unpackL2(B,D,E)	\Rightarrow	unpackH2(A,C,E)

Figure 8.11: (Example) Transformations for Intel's Pentium 4. As Intel's Pentium 4 has no special operations for accumulate and swap in its instruction set, these operations have to be rewritten. For each type of unsupported instructions, an example is given.

The *annotated scheduling phase* annotates a serial order onto parallel blocks of the serial-parallel DAG. Parallel blocks using mostly the same set of register variables are scheduled consecutively, i.e., they get an annotation which defines their scheduled order. This order is optimized w.r.t. minimizing variable lifetime. Therefore, the annotated scheduler finds the smallest sub-DAG that encompasses the entire lifespan of the variable. This is necessary for finding nested scopes inside a set of sub-DAGs.

A more detailed description of the scheduler and an example illustrating its functionality can be found in [38].

<i>Sequence</i>	\Rightarrow	<i>Sequence'</i>
swap2, chsH2	\Rightarrow	chsL2, swap2
chsH2, swap2	\Rightarrow	swap2, chsL2
swap2, swap2, unpackL2	\Rightarrow	swap2, swap2, unpackH2
swap2, swap2, unpackH2	\Rightarrow	swap2, swap2, unpackL2
unpackL2, swap2	\Rightarrow	unpackL2
unpackH2, swap2	\Rightarrow	unpackH2
swap2	\Rightarrow	shufpd2
accPP2	\Rightarrow	unpackL2, unpackH2, add2
accNN2	\Rightarrow	unpackL2, unpackH2, sub2
accPN2	\Rightarrow	unpackL2, unpackH2, chsL2, sub2
accNP2	\Rightarrow	unpackL2, unpackH2, chsL2, add2

Figure 8.12: Rule Set for Pentium 4 Transformations. All rewriting combinations of unsupported operations on Intel's Pentium 4.

8.4.1 Improvements of the Scheduler

MAP performs additional reordering (*code motion*) to the output of the scheduler. The goal of code motion is to improve locality of register accesses inside a schedule by shifting instructions up and down.

As the access patterns occurring in SIMD vectorized code may differ significantly from the scalar case, FFTW-GEL uses a set of heuristics to reorder instructions, thus breaking FFTW's schedule.

Moreover, there are usually more unary instructions in SIMD code than in scalar code, because these instructions are needed to do auxiliary operations, e.g., data shuffling and sign change operations.

To improve the output of MAP's scheduler of in these cases, MAP employs a local reordering strategy, trying to reduce the register pressure.

The following definitions are needed in the context of code motion.

Instruction Neighbor. An instruction *X* is neighbored to an instruction *Y* when *X* is directly followed by *Y* in a sub-DAG of the schedule or vice versa.

Instruction Sequence. An instruction sequence $\langle I_1, \dots, I_n \rangle$ consists of *n* neighboring instructions.

Producer Instruction. A producer of a register *R* is an instruction using *R* as destination operand.

Consumer Instruction. A consumer of a register *R* is an instruction using *R* as source operand.

Moving an Instruction Down. Consider an instruction sequence $\langle I_1, I_2 \rangle$ where *I*₁ and *I*₂ are neighbors. Instruction *I*₁ is moved down in the sequence leading to the new sequence $\langle I_2, I_1 \rangle$ if and only if all source operands of the instruction *I*₂ are neither source nor target operands of instruction *I*₁.

Moving an Instruction Up. Consider an instruction sequence $\langle I1, I2 \rangle$ where $I1$ and $I2$ are neighbors. Instruction $I2$ is moved up in the sequence leading to the new sequence $\langle I2, I1 \rangle$ if and only if all source operands of the instruction $I1$ are neither source nor target operands of instruction $I2$.

As the code dealt with is of SSA form, for the reasons described in Appel [10], the code motion process can be simplified. In the up shifts, it is not necessary to check for the destination of $I1$ being source or destination of $I2$. In the down shifts, it is not necessary to check for the destination of $I2$ being source or destination of $I1$.

Code Motion Transformation

The following code motion steps are performed: (i) Loads are moved down. (ii) Stores are moved up. (iii) Unary instructions, whose source operand is not read by any instruction following in the program text, are moved up. (iv) All other unary instructions are moved down. (v) Binary instructions are moved up if both of their source operand are not used in the following program text. This step has turned out to be especially beneficial for SIMD codes mainly consisting of intra-operand instructions.

Chapter 9

Backend Techniques for Straight-Line Code

The MAP backend [74, 76] introduced in this section generates assembly code optimized for short vector SIMD hardware. It is able to exploit special features of automatically generated straight-line codes. The MAP backend is included in the most current version of FFTW, FFTW-GEL and has been connected to SPIRAL and ATLAS. It currently supports assembly code for x86 with 3DNow! or SSE 2.

The automatically generated codes to be translated are large blocks of SSA code with the following properties: (i) there is no program control flow except the basic block's single point of entry and exit, (ii) there is full knowledge of future temporary variable usage, (iii) there are indexed memory accesses possibly with a stride as runtime parameter, and (iv) loads from and stores to data vector elements are performed only once.

Standard backends fail to compile such blocks of SSA code to performant machine code as they are targeted at a broad range of structurally different handwritten codes lacking the necessary domain specific meta information mentioned above. Thus, standard backends fail in register allocation when too many temporary variables are to be assigned to a small number of registers and they also have trouble with efficient calculation of effective addresses.

9.1 Optimization Techniques Used in MAP

The MAP backend performs the following two kinds of optimization.

Low Level Optimization. Low level optimization rests upon the following two properties of the x86 instruction set.

First, most instructions are destructive, i. e., the output of an instruction is written into one of its source registers.

Secondly, the instruction set offers special instructions that constitute a combination of several simple instructions, e. g., a "shift by some constant and add" instruction. Code quality can often be improved by utilizing these composite instructions.

After transforming high-level instructions into a sequence of equivalent x86-style code, the MAP backend applies register allocation based on the farthest first

algorithm for choosing a spill victim. That process is interwoven with the generation of efficient code for the calculation of effective addresses. The latter utilizes a particularly well-suited mix of integer instructions to speed up the access of elements residing in strided arrays.

Ultra Low Level Optimization. Even within a class of processors implementing the same instruction set architecture, individual members may have different execution behavior with regard to instruction latencies, instruction throughput, ports, and (the number and type of) functional units.

Ultra low level optimizations take specific execution properties of one particular processor into account. These properties are provided by processor-specific execution models specifying (i) how many instructions can be issued per clock cycle, (ii) the latency of each instruction, (iii) the execution resources required by each instruction, and (iv) the overall available resources.

The ultra low level optimizations comprise an instruction scheduler and a register reallocator, both of which form a feedback driven optimization loop.

The instruction scheduler is capable of estimating the runtime of the generated code by modeling a super-scalar, in-order approximation of the target processor.

The register reallocator eliminates useless copy instructions, promotes the use of CISC style load-and-use instructions, and renames all logical SIMD registers using a least recently used (LRU) heuristic. The register reallocator can immediately effect code size and register pressure, but more importantly, the renaming of registers can help the instruction scheduler do a better job in the next iteration of the optimization loop.

As long as the code improves with regard to the estimated runtime or the code size, a new iteration of the optimization loop is started.

As soon as the feedback driven optimizations are done, a final pass reorders memory access instructions to avoid address generation interlocks (AGIs).

9.2 Backend Optimization Goals

MAP's backend aims at the optimization of (i) integer arithmetics, and (ii) the code size of vectorized straight line code.

Optimized Integer Arithmetics. All integer calculations done by DSP or matrix multiplication kernels are solely devoted to effective address calculation

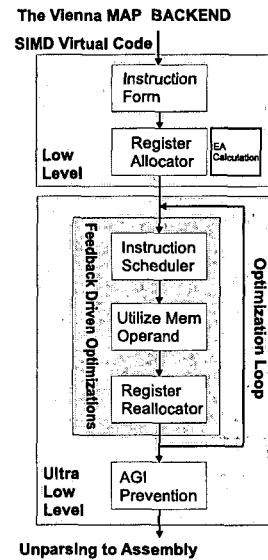


Figure 9.1: The basic structure of MAP's backend. A SIMD DAG generated by MAP's vectorization frontend is subjected to advanced low and ultra low level backend optimization.

needed to access data arrays residing in memory. Whereas general purpose compilers normally do not optimize the respective integer calculations, MAP's backend applies a special technique for generating optimized code to be used in effective address computation. Using this optimization technique, the performance of DSP and linear algebra kernels improves significantly, eliminating the need for several specialized variants of the same kernel working with different fixed strides.

Minimizing Code Size. Although the main reduction of code size is achieved by the MAP vectorizer [75], the backend further reduces the size of a kernel. Code size minimization is of utmost importance because (i) the instruction caches of modern processors are relatively small, and (ii) it allows to generate and use codelets for larger transforms without performance deterioration.

For instance, the Pentium 4's trace cache holds 12K of micro-operations. This is not overwhelmingly much, considering that each x86 instruction demands at least one micro-operation. Instructions using direct memory operands demand even more.

MAP's backend optimization even reduces the code size of bigger basic blocks, helping a processor's instruction cache not to be overstrained. If, for any reason, there is a demand for very big basic blocks (e. g., large FFTW codelets, completely unrolled ATLAS kernels, . . .) , it can be guaranteed that their performance is good or at least acceptable.

The backend uses (i) efficient methods to compute effective addresses, (ii) a register allocator, (iii) an optimizer for in-memory operands, and (iv) a register reallocator for code size reduction.

For instance, the efficient calculation of effective addresses is hinted to reuse already computed addresses as operands for other address computation code. This technique yields the shortest possible instruction sequences in address computation code, i. e., for one effective address translation locally.

The basic block register allocator that utilizes the farthest first algorithm as its spill heuristic, reduces the number of register spills and reloads. Hence, besides a performance improvement, the number of spill and reload instructions is considerably reduced in the final assembly code.

Direct use of in-memory operands helps to discover weaknesses of the register allocator. Loads from memory whose data is used as an instruction's operand are spilled only once. Such register operands are discarded and their use is substituted by equivalent memory operands. The register reallocator does not care about dropped out load instructions for discarded register operands in its subsequent allocation process.

9.3 One Time Optimizations

Architecture specific instruction forms, register allocation, computation of effective addresses, and avoidance of address generation interlocks are optimization

techniques performed only once.

9.3.1 Architecture Specific Instruction Forms

While most instructions of RISC style instruction sets comprise instructions in standard form that take two source registers and one destination register without any additional equality constraints pending, in the CISC style x86 instruction most unary and binary instructions are destructive, i. e., one source register must be used as a destination register.

When translating high-level instructions¹ into x86 instructions, one copy instruction is inserted for every instruction. In all following stages of the backend compilation, x86 style instructions are used.

Example 9.1 demonstrates the rewriting of standard unspecific instruction forms into x86, i. e., in this case Pentium 4, specific instruction forms.

Example 9.1 (Rewriting Instruction Forms) Unary and binary vector instructions of standard instruction forms of a short example code sequence are rewritten into Pentium 4 specific instruction forms to allow for further, architecture and processor close backend optimization.

<i>Standard Instruction Form</i>	<i>x86 Instruction Forms</i>
<code>add2(s1,s2,d);</code>	\longrightarrow <code>p4copy2(s1,d);</code> <code>p4add2(s2,d);</code>
<code>mulc2(s,(0.7,0.5),d);</code>	\longrightarrow <code>p4copy2(s,d);</code> <code>p4mulc2((0.7,0.5),d);</code>

9.3.2 Register Allocation for Straight-Line Code

The register allocator's input code contains vector computation as well as its integer address computation accessing input and output arrays in memory. Thus, registers have to be allocated for both register types assuming two different target register files.

The codes are in SSA form and thus only one textual definition of a variable exists. There is no control flow either. As a consequence of these properties, it is possible to evaluate the effective live span of each temporary variable, and thus, a farthest first algorithm [105] can be used as a spilling scheme.

Whenever a temporary variable needs to be mapped to a logical register, the register allocator uses the following strategy for finding a *spill victim*: (i) Whenever possible, a fresh logical register is chosen. (ii) If there is at least one dead

¹In the higher levels of MAP's code generation, i. e., vectorizer and optimizer, intermediate code is used without taking care for specific instruction forms. As the subsequent low and ultra low level optimization are very close to a specific architecture, intermediate code has to be transformed from standard form into x86 form accordingly.

logical register, i. e., a logical register holding a value that is not referenced in the future, the logical register that has been dead the longest is chosen. Reusing a dead logical register introduces a false dependency. (iii) If there is no dead logical register, the logical register R that holds a value V such that the reference to V lies farther in the future than the references to the values held by all other logical registers is chosen.

General purpose compilers cannot apply the farthest first algorithm in their spilling schemes as they also address codes possibly containing complex control flow structures not allowing to precisely determine the life span of each temporary variable as in straight-line SSA codes.

Experiments carried out with numerical straight-line SSA code have shown that the simple farthest first spilling strategy is superior to the strategies utilized in general purpose C compilers [44].

9.3.3 Optimized Index Computation

Index computations are normally not addressed by advanced optimization techniques as their occurrence in non-DSP codes is negligible. Nevertheless, straight-line code produced by DSP program generators features disproportionately many memory access operations in relation to arithmetic operations. Thus, optimized index computation in such code is crucial for achieving high performance and poses one of MAP's optimization tasks.

The targeted codes operate on arrays of input and output data. Both may not be stored contiguously in memory. Thus, access to element `in[i]` may result in a memory access operation either at address

$$\text{in} + i \cdot \text{sizeof}(\text{in}) \quad \text{or} \quad \text{in} + i \cdot \text{sizeof}(\text{in}) \cdot \text{stride}$$

where `in` and `stride` are parameters passed from the calling function.

Memory accesses with constant indices do not need explicit effective address computation whereas those with variable array strides do.

Guidelines for Effective Address Computation

To achieve an efficient computation of effective addresses, the MAP backend follows some guidelines. (i) General multiplications are avoided. Instead of using costly integer multiplication instructions (`imul`) for general integer multiplication in effective address calculation, it is usually better to use equivalent sequences of simpler instructions (`add`, `sub`, `shift`) instead. Typically, these instructions can be decoded faster, have shorter latencies, and are all fully pipelined. (ii) The integer register file's content is reused as often as possible. This is feasible as integer registers are solely reserved for address computation and the full past and future of required multiples of the stride is known. (iii) Only integer multiplications are rewritten whose result can be obtained using less than n other operations, where

n depends on the latency of the integer multiplication instruction on the target architecture.

The Load Effective Address Instruction

On x86 compatible hardware architectures it is possible to use the powerful `lea` instruction instead of a sequence of (`add`, `sub`, `shift`) instructions as it combines up to two adds and one shift operation into one instruction. The `lea` instruction can be used, for instance, to quickly calculate addresses of array elements. As it is better to utilize instructions with implicit integer computation, as this can further reduce the code size, the `lea` instruction is used in the process of effective address computation in MAP's x86 backend.

Example 9.2 (The LEA Instruction) The complex computation of $\text{eax} = \text{ebx} + \text{ecx} + 8$ corresponds to just one `lea` instruction: `lea eax, [ebx + ecx + 8]` `lea` calculates the value `eax` by performing the addition of a base register `ebx` with an index register `ecx` and some constant displacement, e.g., 8.

In general, the `lea` instruction implements addressing modes of the form

$$\text{Base} + \text{Index} * \text{Scale} + \text{Displacement}$$

like, for instance,

$$\begin{pmatrix} \text{eax} \\ \text{ebx} \\ \text{ecx} \\ \text{edx} \\ \text{esp} \\ \text{ebp} \\ \text{esi} \\ \text{edi} \end{pmatrix} + \left[\begin{pmatrix} \text{eax} \\ \text{ebx} \\ \text{ecx} \\ \text{edx} \\ \text{ebp} \\ \text{esi} \\ \text{edi} \end{pmatrix} * \begin{pmatrix} \text{none} \\ 1 \\ 2 \\ 4 \\ 8 \end{pmatrix} \right] + \begin{pmatrix} \text{none} \\ 8 - \text{bit} \\ 16 - \text{bit} \\ 32 - \text{bit} \end{pmatrix}$$

Base + Index*Scale + Displacement

Base denotes an arbitrary register out of 8 different 32-bit integer registers.

Index is any non-base 32-bit register. It might include some of the base registers.

Scale is a scaling factor being either 0 (none), 1, 2, 4, or 8.

Displacement is either an 8, 16, or 32-bit integer containing an address or an offset. It may happen that there is no displacement at all.

Not all of the four addressing components need to be specified, like in Example 9.2 where the **Scale** factor has been omitted.

The Optimized Index Computation Process

The generation of code sequences for calculating effective addresses is intertwined with the register allocation process.

Whenever the register allocator needs to emit code for the calculation of an effective address, the (locally) shortest possible sequence of `lea` instructions is determined by depth-first iterative deepening (DFID) with a target architecture specific depth limit n which depends on the latency of the integer multiplication instruction on the target architecture². As the shortest sequences tend to eagerly reuse already calculated contents of the integer register file, a replacement policy based on the least recently used (LRU) heuristic is employed for integer registers.

There are different ways of generating `lea` instructions producing a new factor out of existing ones in one step of the iterative deepening approach:

$$d = s1 * \begin{Bmatrix} 1 \\ 2 \\ 4 \\ 8 \end{Bmatrix} + s2 \quad d = s * \begin{Bmatrix} 4 \\ 8 \end{Bmatrix}$$

whereby d is the destination register, and s , $s1$, and $s2$ are registers holding previously generated factors that are already present in the register file.

In the following, Example 9.3 shows an optimal efficient index computation, as only *one* `lea` instruction is needed to compute an effective address.

Example 9.3 (Efficient Computation of $17*stride$) The `lea` instruction is used to compute $17*stride$ from the current entries of the integer register file. In this example all required operands already reside in the register file providing three different ways of computing the result. The entries in the register file and the respective `lea` instructions are displayed below. The factors $1*stride$ and $-1*stride$ have an register file entry by default and can therefore be assumed as initially available. Due to the properties of the considered codes, it is very likely that the needed factor for a `lea` instruction is already present in the integer register file resulting in a maximum factor reuse.

Integer Register File		LEA Instructions	Result
Reg	Entry		
eax	stride*3	lea ecx, [ebx + 4*eax] lea ecx, [edi + 2*esi] lea ecx, [edx + 4*esp]	(5+4*3)*stride
ebx	stride*5		(-1+2*9)*stride
edx	stride*1		(1+4*4)*stride
esp	stride*4		
esi	stride*9		
edi	stride*-1		

²e.g., $n = 3$ operations on an AMD K7, as the general multiplication instruction has a latency of 4 cycles on this processors.

9.3.4 Avoiding Address Generation Interlocks

The memory access operations in the numerical straight-line codes of this thesis do not have WAW dependencies concerning memory instructions as writing to a specific memory location happens only once. Thus, any order of memory instructions obeys the given program semantics. This enables the MAP backend to reorder store instructions to resolve address generation interlocks (AGIs). AGIs occur whenever a memory operation involving some expensive effective address calculation directly precedes a memory operation involving inexpensive address calculation or no address calculation at all. Generally, as there is no knowledge whether there are WAW dependencies between memory accessing instructions or not in the codes, the second access is stalled by the computation of the first access' effective address. Example 9.4 illustrates the prevention of an AGI.

Example 9.4 (Address Generation Interlock) A write operation to `out[17]` is directly preceding a write operation to `out[0]` in the assumed code. As hardware requires an in-order memory access, writing to `out[0]` requires a completed writing to `out[17]` and thus the computation of the effective address of `out[17]`. Both would stall write to `out[0]`. By knowing that there are no WAW dependencies between `out[0]` and `out[17]`, these operations are swapped. The write to `out[0]` is done *concurrently* to computing the effective address of `out[17]` and thus its writing is not stalled.

<code>addr = 17 * stride;</code>		<code>addr = 17 * stride;</code>
<code>out[addr] = temp0;</code>	\longrightarrow	<code>out[0] = temp1;</code>
<code>out[0] = temp1;</code>		<code>out[addr] = temp1;</code>

AGI prevention is performed as MAP's last optimization after the feedback driven optimization loop.

9.4 Feedback Driven Optimizations

Instruction scheduling, direct use of in-memory operands, and register reallocation are executed in a feedback driven optimization loop to further improve the optimization effect. The instruction scheduler serves as a basis for estimating the runtime of the entire basic block. As long as the code's estimated execution time can be improved, the feedback driven optimizations are executed.

A final optimization technique applied to the output of the optimization feedback loop is the reordering of memory access instructions to avoid AGIs.

9.4.1 Basic Block Instruction Scheduling

Instruction scheduling is an optimization technique that aims at the rearrangement of the micro-operations executed in a processor's pipeline to maximize the number of function units operating in parallel and to minimize the time they spend waiting for each other [69]. To maximize a program's overall performance,

it is essential that the respective code is scheduled in a way to take maximum advantage of the pipelines provided by the architecture [91].

The MAP backend's instruction scheduler deals with the instructions of a single basic block, using the local list scheduling described in [91] and [105]. The scheduling algorithm utilizes information about the critical-path lengths of the underlying data dependency graph as a heuristic when selecting an instruction to be issued.

The list scheduling algorithm implemented interacts with an execution model of the target processor to simulate the effects of super-scalar in-order execution of an instruction sequence.

The Processor Execution Model

The MAP backend uses an execution model specifying low level hardware properties for each specific x86 target processor (e. g., Intel Pentium 4 and AMD K7).

For each instruction supported by a particular processor, the execution model specifies (i) how many instructions can be decoded and issued per clock cycle, (ii) the latency of each instruction, (iii) the execution resources required by each instruction, and (iv) the available resources.

The execution model can be seen as an oracle that decides whether a given sequence of n instructions can be issued within the same cycle and executed in parallel by means of super-scalar in-order execution while avoiding resource conflicts³. That oracle is used by MAP's instruction scheduler to resolve *tie situations* occurring in the instruction scheduling process.

For the sake of simplicity, the execution models do not take address generation resources into account. In fact, the models assume that enough address generation units (AGUs) are available at any time. Also, all execution models implemented do not address instruction and data cache misses.

Using Execution Modeling in Instruction Scheduling

The issue priority heuristic of the list scheduling algorithm presented in [91] selects among all instructions currently in ready state the instruction having the longest critical path.

While the MAP backend also uses information about critical path lengths as a global guideline, it uses a target processor specific *execution model* to estimate how many cycles the execution of a given code would take on the target.

As the instruction scheduler of MAP has an actual notion of time, it tries to maximally "fill" each cycle. The issuing of an instruction is delayed until the

³An instruction issue is possible if and only if (i) the latencies of the instructions are optimal in terms of the scheduling heuristic, (ii) n is less or equal the number of processor slots per cycle, and (iii) the required execution units are not conflicting.

results of its predecessors are available. Among all ready instructions that could issue/execute with all other instructions previously chosen for that particular cycle, the instruction scheduler selects the one with the longest critical path and commits to it.

Whenever it is not possible to find a suitable instruction, the scheduler advances to the next cycle, updating its scheduling lists accordingly.

9.4.2 Register Reallocation

Register allocation and instruction scheduling have conflicting goals. As the register allocator tries to minimize the number of register spills, it prefers introducing a false dependency on a dead logical register over spilling a live logical register. The instruction scheduler, however, aims at the maximization of the pipeline usage of the processor, by spreading out the dependent parts of code sequences according to the latencies of the respective instructions.

As the MAP backend performs register allocation before doing instruction scheduling, it is clear that the false dependencies introduced by the register allocator severely reduce the number of possible choices of the instruction scheduler.

To address this problem, the register reallocator tries to lift some (too restrictive) data dependencies introduced by the register allocator (or by a previous pass of register reallocation), enabling a following pass of the instruction scheduler to do a better job. Also, the register reallocator uses information about the spills introduced by the register allocator to minimize the code size by appropriately utilizing CISC style instructions and by optimizing x86 copy instructions.

Preparatory Steps for the Register Reallocator

To assist the following optimizations performed by the register reallocator, the following two preparatory steps are carried out first.

In the first pass, all SIMD loads and reloads, all SIMD copy instructions, all SIMD unary instructions, and all instructions for integer calculation are moved down in the program text, until the moved instruction reaches another instruction with whom it has either an RAW, a WAW, or a WAR dependency, i. e., the respective instructions are moved down as far as possible.

The second pass is identical with the first one, except for the fact that SIMD reloads are not moved down.

Utilizing CISC Style Instructions with In-Memory Operands

The x86 instruction set architecture defines only few logical registers. But, as on other CISC style machines, it is possible to directly use an operand residing in memory without previously loading it into a register.

Using in-memory operands can (i) reduce register pressure locally and (ii) lower the total instruction count, as two separate instructions, i. e., one load and one use instruction, can be merged into one load-and-use instruction.

However, excessively using in-memory operands can both increase the code size and deteriorate performance. In extreme cases using in-memory operands could lead to (i) half empty register files, while (ii) the same data is loaded over and over again.

The MAP backend transforms load-use combinations into load-and-use instructions whenever a memory operand is used exactly once in the following program text.

Helping the Instruction Scheduler

The process of instruction scheduling sheds some light on the false dependencies introduced by the register allocator, showing that some false dependencies impeded the process of instruction scheduling more than others did. Also, as the utilization of CISC style instructions with in-memory operands slightly reduces the register pressure, reconsidering the false dependencies introduced by the register allocator is one of the main goals of the register reallocator.

The MAP backend's register reallocator takes the output of the instruction scheduler and reallocates all logical SIMD registers using a LRU replacement strategy. During that process, all register spills and reloads remain unchanged.

As the register reallocator effectively replaces false dependencies with more favorable ones with regard to the instruction scheduler, it enables another pass of the instruction scheduler to yield a better result.

Eliminating Superfluous Copy Instructions

As the preparation phase of the register reallocator has an impact on the relative order of SIMD spill and SIMD copy instructions, it is sometimes possible to eliminate copy instructions of logical registers.

Whenever the register reallocator encounters a SIMD copy instruction, such that the source register of that instruction is used at most once in the following program text, the copy instruction is eliminated and its destination register is renamed to its source.

Chapter 10

Experimental Results

This chapter presents numerical experiments carried out to demonstrate the applicability and the performance boosting effects of the newly developed MAP vectorizer and backend tools [74, 76].

The MAP vectorizer provides vectorized code either (i) via a source-to-source transformation producing macros compliant with a portable SIMD API (see Chapter 6) and additionally providing support for FMA instructions, or (ii) via a source-to-assembly transformation utilizing the MAP backend currently supporting assembly code for x86 with 3DNow! or SSE 2.

To obtain relevant performance results, the newly developed methods of the MAP vectorizer and backend have been combined with leading-edge self-tuning numerical software—FFTW [35], SPIRAL [99], and ATLAS [115]—resulting in high-performance SIMD implementation of DSP and linear algebra codes. In the following sections, experimental evidence is given for the performance gain unleashed by the MAP vectorizer and backend.

All performance values were obtained using the original timing routines provided by SPIRAL, FFTW and ATLAS. Third-party codes were assessed using these timing routines as well. Details on performance assessment for scientific software can be found in Appendix C and in Gansterer and Ueberhuber [40]. All codes for complex transforms utilize the *interleaved complex* format (alternately real and imaginary part data format) unless noted differently.

10.1 Experimental Layout

It has been demonstrated, that the MAP vectorizer provides one of the fastest FFTs that are currently available on x86 processors featuring 3DNow! and SSE 2. In addition, these methods provide the only FFTs for BlueGene/L's PowerPC 440 FP2 processor taking full advantage of its double FPU [3]. Formal vectorization and the MAP vectorizer leverage the only vectorized implementations for general DSP transforms like DST, 2D-DCT that are automatically tuned. Moreover, the only fully automatically vectorized ATLAS kernels are obtained utilizing the MAP vectorizer.

Instruction statistics show that in most cases the MAP vectorizer reduces the number of arithmetic instructions significantly (up to 50%). In some cases, however, a considerable amount of reorder instructions is required, thus slightly increasing the overall instruction count. On modern processors like the Pentium 4

this is not a critical issue, as floating-point and reordering instructions can be carried out in parallel by specialized execution units providing an overall speed-up anyway.

Furthermore, the MAP backend accelerates automatically generated DSP code by up to 25 %, compared to standard C compilers like the GNU C compiler. This allows code vectorized by the MAP vectorizer to achieve a *superlinear* speed-up value of 2.2 for two-way SIMD extensions. Using the Intel C++ compiler, speed-up values of up to 1.8 are reached using the MAP vectorizer.

10.1.1 Experimental Setup

Numerical experiments on machines featuring different short vector extensions were conducted: (i) a prototype of BlueGene/L's PowerPC 440 FP2 running at 500 MHz featuring a *double FPU*, (ii) a Pentium 4 featuring SSE and SSE 2 running at 1.8 GHz and, (iii) a 800 MHz Athlon Thunderbird featuring 3DNow! professional. 3DNow! and the *double FPU* are two-way vector extensions (for single-precision and double-precision, respectively) providing a theoretical speed-up of two. The *double FPU* additionally provides fused multiply-add (FMA) instructions.

First, the results of the experiments carried out on IBM's BlueGene/L will be presented. Then the two IA-32 machines, i. e., the 1.8 GHz Pentium 4 and the 1.53 GHz Athlon XP 1800+ will be assessed. The latter machines are interesting to experiment with as their processors have different cache architectures and the investigated codes fit into data caches.

FFT performance is displayed in *pseudo Gflop/s*, i. e., $5N \log_2 N/T$ (in nanoseconds) while actual performance is displayed for all other transforms.

10.2 BlueGene/L Experiments

Very recently, experiments were carried out on a prototype of IBM's BlueGene/L (BG/L) top performance supercomputer [25]. In tandem with SPIRAL, the MAP vectorizer was evaluated on this prototype. Performance data of 1D FFTs with vector lengths $N = 2^2, 2^3, \dots, 2^{10}$ were obtained on one single PowerPC 440 FP2 running at 500 MHz.

The following FFT implementations were tested: (i) The best vectorized code found by SPIRAL utilizing formal vectorization, (ii) the best vectorized code found by SPIRAL utilizing the MAP vectorizer, (iii) the best scalar FFT implementation found by SPIRAL (XLC's vectorizer and FMA extraction turned off), (iv) the best vectorized FFT implementation found by SPIRAL using the XLC compiler's vectorizer and FMA extraction turned on, and (v) the mixed-radix FFT implementation provided by the GNU scientific library (GSL). Figs. 10.1 and 10.2 display the respective performance and speed-up data.

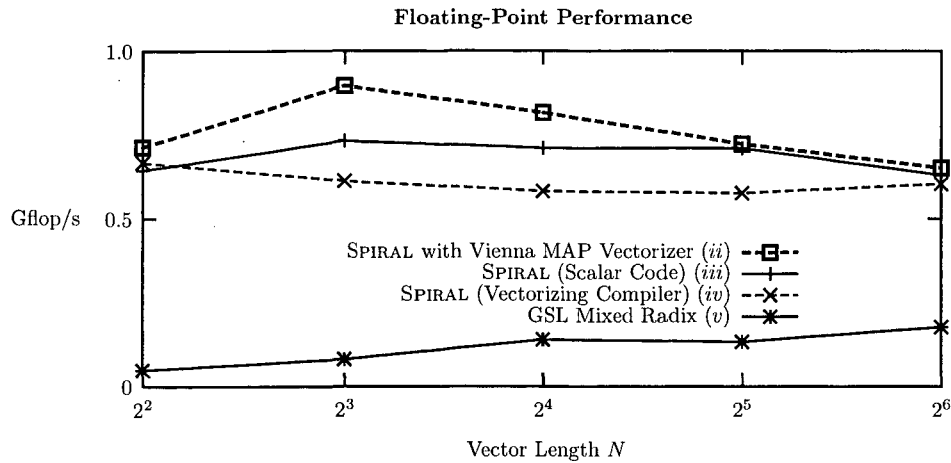


Figure 10.1: Floating-point performance of the MAP vectorizer in tandem with SPIRAL over another vectorized FFT implementations and scalar implementations on BlueGene/L's PowerPC 440 FP2 running at 500 MHz.

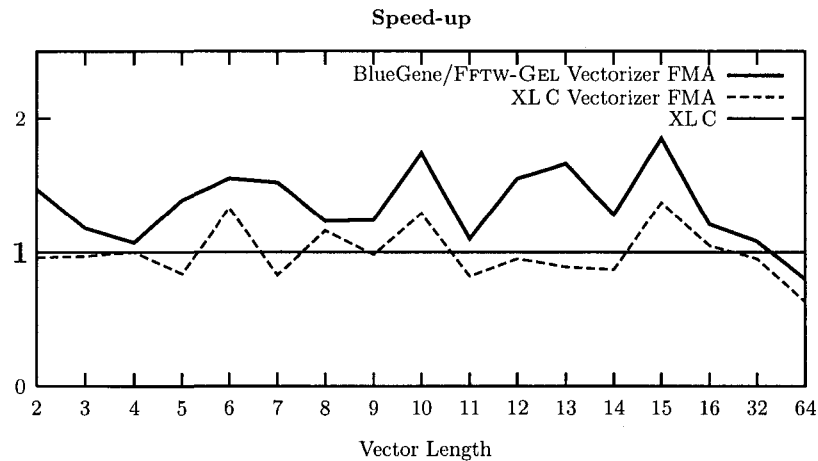


Figure 10.2: Speed-up of (i) the newly developed BlueGene/L MAP vectorizer *with* FMA extraction but without backend optimizations and (ii) FFTW codelets vectorized by XL C *with* FMA extraction, compared to (iii) scalar FFTW codelets using XL C *without* FMAs and *without* vectorization. The experiment has been carried out running no-twiddle codelets.

The best scalar codes found by SPIRAL serve as baseline for the assessment of the various vectorization techniques. The tested codes are very fast scalar implementations that do not utilize FMA instructions. The MAP vectorizer is restricted to problem sizes that can be fully unrolled fitting into instruction cache and the resulting code is such that it can be handled well by the XL C compiler's register allocator. This is the case for problem sizes $N \leq 32 = 2^5$. The third-party GNU GSL FFT library reaches about 30% of the performance of the best scalar SPIRAL generated code and thus shows a very disappointing performance.

XL C's vectorization and FMA extraction produces code which is 15 % slower than scalar XL C without FMA extraction, i. e., XL C's techniques used to vectorize straight-line code does not handle SPIRAL generated FFT codes well.

Fig. 10.2 shows the relative performance (speed-up) of FFTW 2.5.1 no-twiddle codelets vectorized using the MAP vectorizer compared to scalar FFTW codelets and codelets vectorized by IBM's XL C compiler.

IBM's XL C compiler for BlueGene/L using code generation *with* SIMD vectorization and *with* FMA extraction (using the compiler techniques introduced in [79]) sometimes accelerates the code slightly but also slows down the code in some cases. The MAP vectorizer yields speed-up values up to 1.8 for sizes where the XL C compiler's register allocator generates reasonable code. For codes with more than 1,000 lines (vector lengths 16, 32, 64) the performance degrades because of the lack of efficient register allocation.

For automatically generated FFT codes, vectorization of basic blocks using instruction level parallelism sometimes can speed up codes slightly, but in other cases slows down codes significantly. This effect is especially striking when using IBM's XL C compiler for BlueGene/L's PowerPC 440 FP2 processor, as displayed in Figs. 10.1 and 10.2. In all these cases the MAP vectorizer achieves significant speed-up values.

These experiments provide evidence that modern (vectorizing) compilers are not able to generate fast machine code in conjunction with portable libraries.

10.3 Experiments on IA-32 Architectures

This section assesses the MAP vectorizer [72] on IA-32 architectures. Experimental evidence is provided that MAP successfully vectorizes a large class of straight-line codes, including SPIRAL generated codes, FFTW codelets, and ATLAS kernels.

MAP has been investigated on two IA-32 compatible machines: (i) one with an Intel Pentium 4 processor featuring SSE 2 two-way double-precision SIMD extensions and (ii) one with an AMD Athlon Thunderbird featuring 3DNow! professional two-way single-precision SIMD extensions.

All codes were generated using the GNU C compiler 2.95.2 and the GNU assembler 2.9.5.

Performance data are displayed in pseudo Gflop/s, i. e., $5N \log N/T$ (in nanoseconds) for complex-to-complex and $2.5N \log N/T$ for real-to-halfcomplex FFTs.

10.3.1 MAP Vectorization and Backend Applied to FFTW Codelets

MAP was connected to FFTW leading to the AMD specific K7/FFTW-GEL and the Pentium 4 specific P4/FFTW-GEL. In the newest release of FFTW, MAP for AMD machines has been included.

It turned out in the experiments that the maximum speed-up value achievable by vectorization (ignoring other effects like smaller code size, wider register files, etc.) is two on both test machines. However, additional backend optimization leads to further performance improvement of the generated codes.

Both complex-to-complex FFTs and real-to-halfcomplex FFTs for power of two and non-powers of two problem sizes were evaluated using BENCHFFT. Real-to-halfcomplex FFTs are notoriously hard to vectorize (especially for vector lengths being non-powers of two) due to their much more complicated algorithmic structure compared to complex-to-complex FFT algorithms.

Performance on the AMD Athlon. Figs. 10.3 (a)–(d) illustrate the performance of K7/FFTW-GEL using codelets vectorized and assembled by MAP on an Athlon Thunderbird utilizing enhanced 3DNow! which provides two-way single-precision SIMD extensions. Both the vectorizer and the backend were used. Standard C code generated by scalar FFTW, FFTPACK, and GSL demonstrates the performance boosting effect of K7/FFTW-GEL.

More specifically, Figs. 10.3 (a)–(b) display the performance in pseudo Gflop/s of complex-to-complex FFTs. (a) refers to power of two and (b) to non-powers of two vector lengths. Figs. 10.3 (c)–(d) display the corresponding results for real-to-halfcomplex FFTs. (c) refers to power of two and (d) to nonpowers of two vector lengths. In the experiments underlying Figs. 10.3 (a)–(d) the data sets fit into L2 cache.

FFTW is up to two times faster than FFTPACK, the industry standard in non-hardware-adaptive FFT software. Fig. 10.3 (a) shows that for most problem sizes of complex-to-complex power of two FFTs FFTW-GEL is about twice as fast as FFTW with a peak performance of nearly 2 pseudo Gflop/s. In Fig. 10.3 (b), the complex-to-complex non-power of two FFTs of K7/FFTW-GEL are about twice as fast as FFTW with a peak performance of nearly 1.4 pseudo Gflop/s. Fig. 10.3 (c) shows that for most problem sizes of real-to-halfcomplex power of two FFTs FFTW-GEL is about twice as fast as FFTW with a peak performance of nearly 1.5 pseudo Gflop/s. Fig. 10.3 (d) shows that for real-to-halfcomplex non-power of two FFTs FFTW-GEL is about twice as fast as FFTW with a peak performance of nearly 1.2 pseudo Gflop/s.

Performance on the Intel Pentium 4. Figs. 10.4 (a)–(d) illustrate the performance of P4/FFTW-GEL on the Pentium 4. P4/FFTW-GEL utilizes the two-way double-precision SIMD operations provided by SSE 2. Standard C code generated by scalar FFTW and FFTPACK enables the assessment of P4/FFTW-GEL.

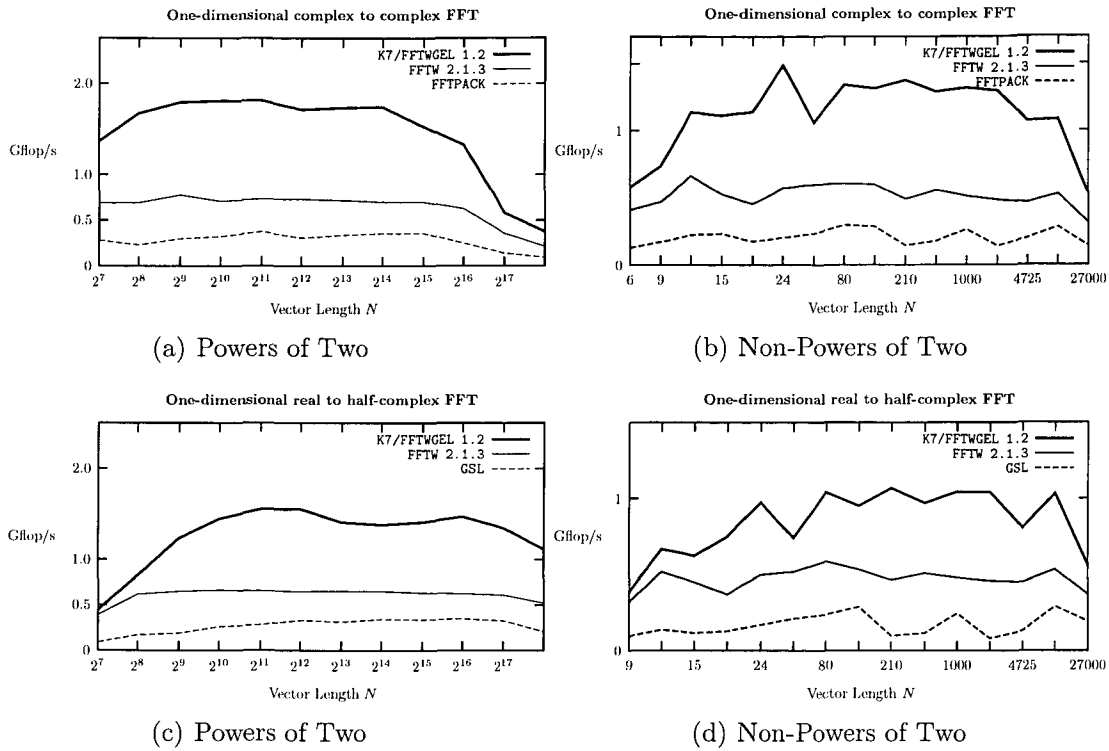


Figure 10.3: Floating-point performance of K7/FFTW-GEL (3DNow!) compared to FFTPACK and FFTW 2.1.3 on a 800 MHz AMD Athlon Thunderbird carrying out complex-to-complex and real-to-halfcomplex FFTs in single-precision both of power of two and nonpowers of two problem sizes.

Figs. 10.4 (a)–(b) display the performance in pseudo Gflop/s of complex-to-complex FFTs. (a) refers to power of two and (b) to non-powers of two transform sizes. Figs. 10.4 (c)–(d) display the runtimes of real-to-halfcomplex FFTs. (c) refers to power of two and (d) to non-powers of two transform sizes. In the experiments underlying Figs 10.4 (a)–(d) the data sets fit into L2 cache.

Fig. 10.4 (a) shows that speed-ups of up to 2.2 have been achieved for complex FFTs of powers of two (including the performance boost originating from the backend). This is leading to FFTs running at 2.2 pseudo Gflop/s on a 1.8 GHz Pentium 4, utilizing two-way SIMD extensions.

The performance of the code is best within L1 cache and decreases outside L1. This happens to be the case only for vector lengths $N = 2^9$ and 2^{10} due to the size of the Pentium 4's very fast data cache.

Fig. 10.4 (b) shows that for complex-to-complex non-power of two FFTs FFTW-GEL is about twice as fast as FFTW 2.1.3 with a peak performance of nearly 1.2 pseudo Gflop/s. Fig. 10.4 (c) shows that the real-to-halfcomplex FFTs of powers of two produced by FFTW already have a remarkable peak performance of nearly 1 pseudo Gflop/s. Nevertheless, P4/FFTW-GEL is up to 30 % faster

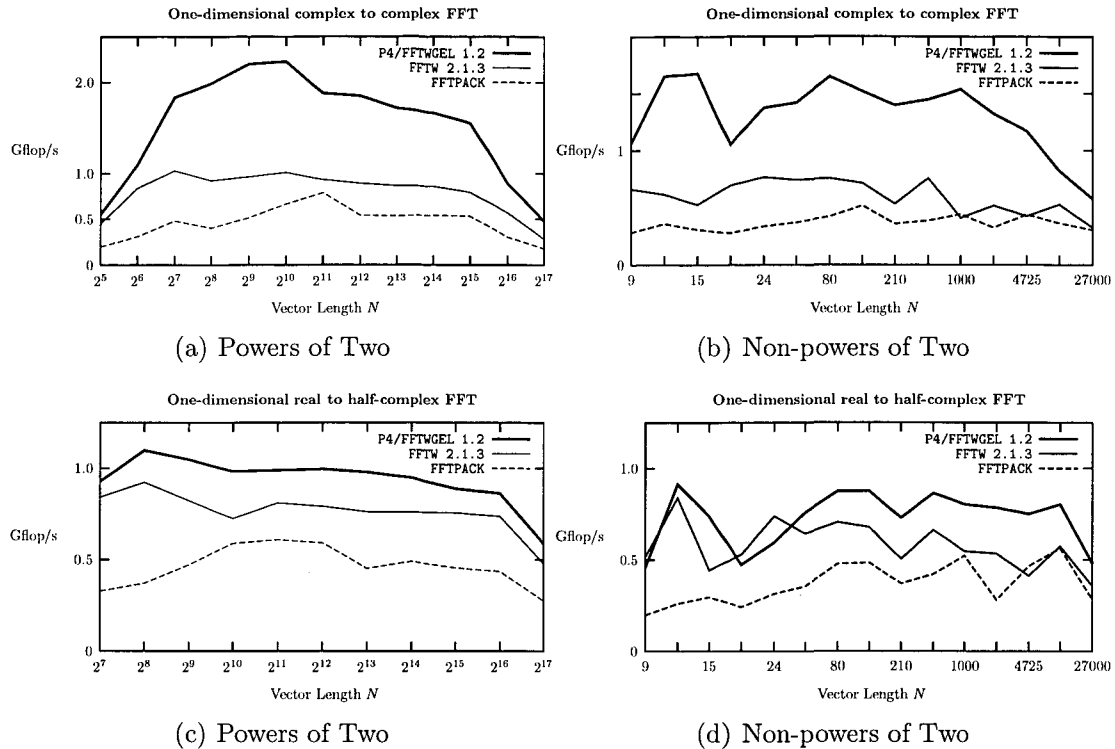


Figure 10.4: Floating-point performance of P4/FFTW-GEL (SSE 2) compared to FFTW (double-precision) on an Intel Pentium 4 running at 1.8 GHz carrying out complex-to-complex and real-to-halfcomplex FFTs having problem sizes of both power of two and nonpowers of two.

than FFTW 2.1.3. The performance improvement is primarily due to backend optimization as vectorization cannot yield a significant improvement in this case. Fig. 10.4 (d) illustrates that for real-to-halfcomplex non-powers of two FFTs, FFTW-GEL is up to 60 % faster than FFTW 2.1.3.

Reduction of the Number of Stack Accessing Instructions. Fig. 10.5 shows the relative count of instructions accessing stack variables, e.g., spill, reload, or load-and-use instructions for FFTW codelets of various size. For codelet sizes $N = 4, \dots, 256$, the MAP backend produces assembly code superior to the C code obtained with FFTW.

This leads to the proposition, that the farthest first policy used as spilling scheme for the register allocation in the MAP backend is superior to the ones used in mainstream C compilers, e.g., gcc, the Intel C compiler, or deccc, particularly for FFTWcodelets and code having a similar structure, e.g., ATLAS linear algebra kernels [44].

Reduction of the Number of Integer Operations. Using the technique for calculating effective addresses described in [71], the total number of integer instructions can be reduced significantly. Fig. 10.6 illustrates the relative reduction

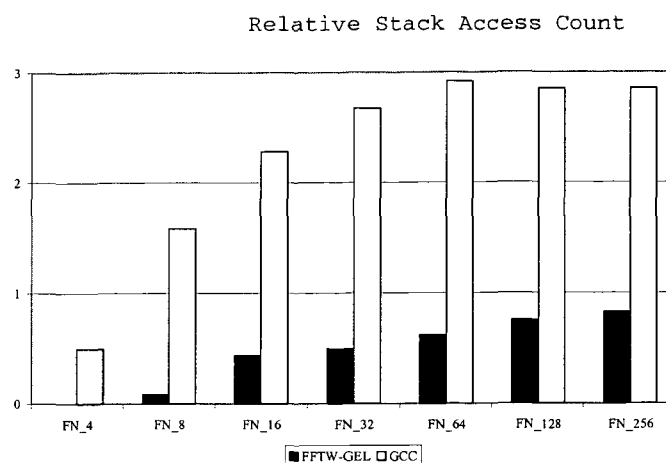


Figure 10.5: Stack access count of MAP vectorized and backend optimized FFTW-GEL codelets compared to standard FFTW codelets, normalized by $N \log_2 N$.

of integer operations achieved by MAP's optimization techniques. The amount of integer operations for effective address calculation is at least halved for the codelet sizes between $N = 4, \dots, 256$ of this assessment study.

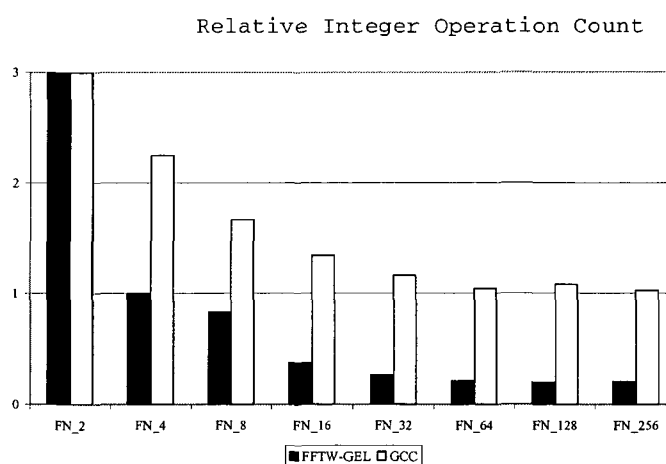


Figure 10.6: Relative integer operation count for FFTW codelets vectorized using the MAP vectorizer and assembled by the MAP backend. The integer operations of effective address computation are significantly reduced.

10.3.2 MAP Vectorization and Backend Applied to SPIRAL Generated Codes

The MAP vectorizer was investigated on a Pentium 4 running at 1.8 GHz using DFTs, DCT, DSTs and WHTs generated by SPIRAL. Fig. 10.7 shows that the MAP vectorizer significantly reduces the number of arithmetic instructions. The vectorization process introduces a significant number of data reordering operations. However, these newly introduced operations can be executed in parallel with the arithmetic operations carried out on the Pentium 4 processor.

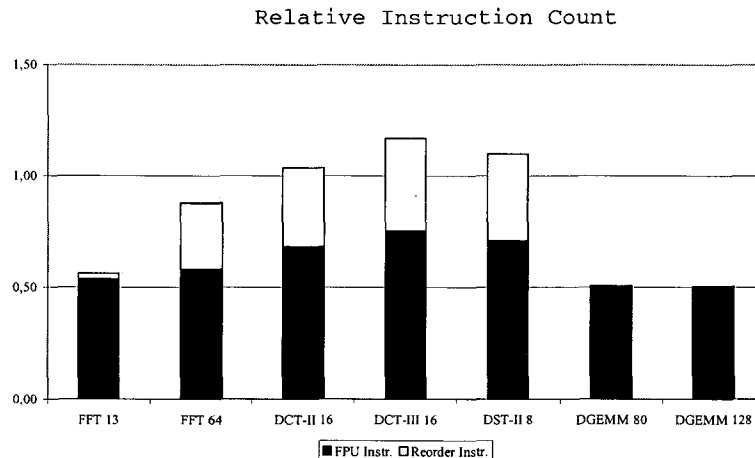


Figure 10.7: Relative instruction count of various DSP transforms and linear algebra routines. The number of arithmetic instructions is significantly reduced by the MAP vectorizer.

To provide evidence that register allocation in standard compilers cannot handle large straight-line code, the MAP backend was compared with the Intel C++ compiler's backend using DFTs (unit stride memory access) of size 4, 8, ..., 64 that require up to 1,300 floating-point instructions. DFT codes were generated and optimized by SPIRAL using (i) the MAP vectorizer and the MAP backend and (ii) the MAP vectorizer outputting C code with intrinsics. In the second case, the Intel C++ compiler's backend was utilized. As largest code in this experiment, DFT₆₄ features approximately 2000 lines of SSA straight-line code. This experiment only assesses register allocation and instruction scheduling, as unit-stride code versions were used.

The experiments show that code generated using the the MAP backend maintains its performance level even for large problem sizes while the performance for $N = 64$ or $N = 128$ FFT code compiled by Intel's C++ compiler degrades by 25% as illustrated in Figs. 10.9 and 10.8. For small problem sizes, when no register spills occur as the number of temporary variables is small, there is no significant difference between the P4/FTW-GEL backend and Intel's C++ com-

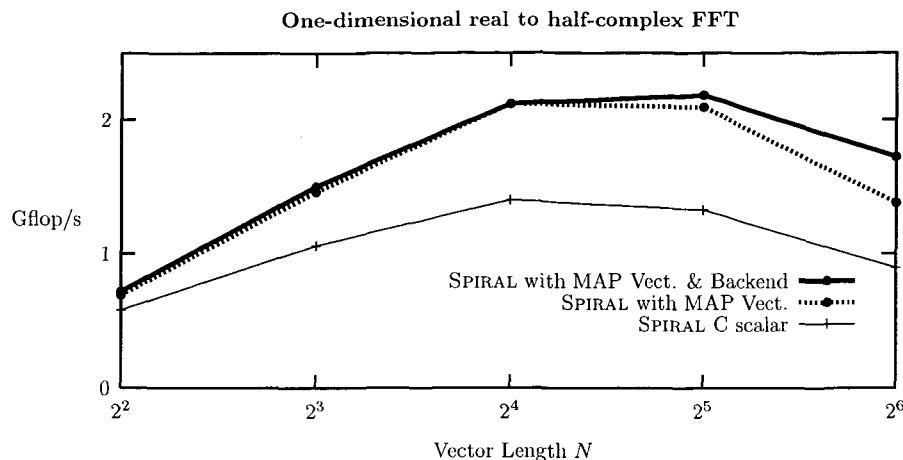


Figure 10.8: Performance of the Intel C++ compiler compared to the MAP backend for a SPIRAL generated one dimensional FFT on an Intel Pentium 4 with 1.8 GHz.

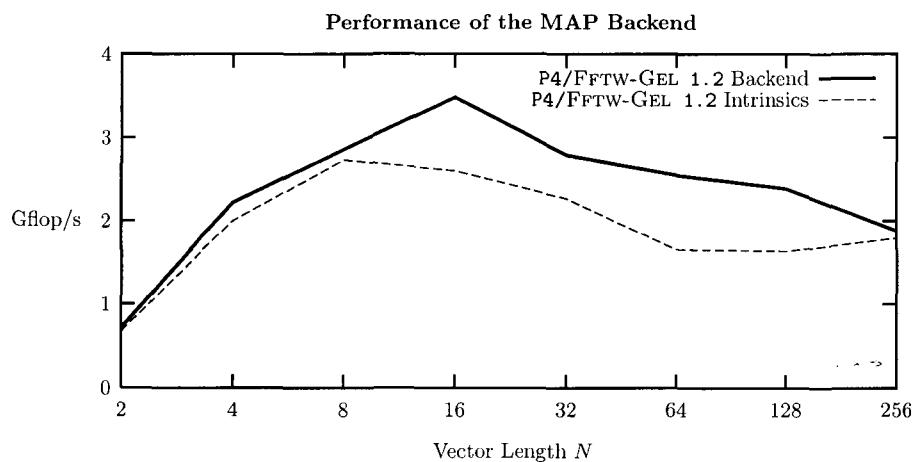


Figure 10.9: Floating-point performance of P4/FFTW-GEL (SSE 2) with backend compared to P4/FFTW-GEL (SSE 2) without backend using intrinsics and the Intel C++ compiler. The experiment has been carried out on a 2.6 GHz Pentium 4 using on real FFTs. Performance data are displayed in pseudo-Gflop/s, i. e., $5N \log N/T$ (nanoseconds).

piler backend. For $N = 256$ the performance of the P4/FFTW-GEL backend decreases as the number of unavoidable spills prevails.

10.3.3 MAP Vectorization of ATLAS Kernels

The MAP vectorizer is able to deal with ATLAS kernels leading to the same instruction count (arithmetic operations, data reorder operations, loads and stores) as the semi-automatically produced (hand-coded) SSE 2 and 3DNow! kernels contributed to the ATLAS project by various people. However, it can automatically

vectorize linear algebra code for kernel sizes that were not yet hand-coded by contributors but would be required for top performance when machine parameters like cache sizes change. Fig. 10.7 shows that the number of arithmetic instructions in `dgemm` kernels is halved by the MAP vectorizer and hardly any reorder instructions are needed.

Conclusion

This thesis presents special purpose compilation techniques targeting the automatic 2-way SIMD vectorization of straight-line DSP and linear algebra code. The combination of these methods with other optimization techniques described in this thesis—peephole optimization and compilation to assembly including various backend optimizations—has been demonstrated to provide outstanding performance for the compiled codes on all supported target architectures.

The most prominent automatic performance tuning systems—ATLAS, FFTW, and SPIRAL—hardware independently provide high performance scalar implementations of numerical algorithms. SIMD short vector hardware support has been provided, at the most, by hand written kernels utilizing SIMD instructions. The newly introduced compiler technology helps to achieve the same level of performance as do hand-tuned vendor libraries while accomplishing performance portability.

In conjunction with ATLAS, FFTW and SPIRAL, SIMD vectorized high performance implementations of FFTs, general DSP transforms, and BLAS kernels have been obtained. Some of the techniques described in this thesis have been included in the current release of the industry standard numerical library FFTW and will become part of IBM's numerical library for BlueGene/L supercomputers.

The MAP vectorizer of this thesis leveraged the so far only vectorized implementation of automatically tuned DCTs, DSTs, and multidimensional DSP transforms. Moreover, the MAP vectorizer has produced the only *fully automatically vectorized* ATLAS kernels so far.

The MAP vectorizer, relying on the techniques introduced in this thesis, provides the fastest FFTs currently available on x86 architectures featuring 3DNow! (AMD Athlon) or SSE 2 (Intel Pentium 4) SIMD extensions.

IBM is currently developing its BlueGene/L supercomputers which aim at the top rank in the TOP500 list. The methods of this thesis rendered possible the only currently existing FFT software for BlueGene/L's PowerPC 440 FP2 processors taking full advantage of their double FPU.

Appendix A

The Kronecker Product Formalism

This chapter introduces the formalisms of Kronecker products (tensor products) and stride permutations, which are the foundations of most algorithms for discrete linear transforms. This includes various FFT algorithms, the Walsh-Hadamard transform, different sine and cosine transforms, wavelet transforms as well as all multidimensional linear transform.

Kronecker products allow to derive and modify algorithms on the structural level instead of using properties of index values in the derivation process. The Kronecker product framework provides a rich algebraic structure which captures most known algorithms for discrete linear transforms. Both iterative as well as recursive algorithms are captured. Most proofs in this section are omitted. They can be found in Van Loan [109].

The Kronecker product formalism has a long and well established history in mathematics and physics, but until recently it has gone virtually unnoticed by computer scientists. This is changing because of the strong connection between certain Kronecker product constructs and advanced computer architectures (Johnson et al. [67]). Through this identification, the Kronecker product formalism has emerged as a powerful tool for designing parallel algorithms.

In this chapter, Kronecker products and their algebraic properties are introduced from a point of view well suited to algorithmic and programming needs. It will be shown that mathematical formulas involving Kronecker product operations are easily translated into various programming constructs and how they can be implemented on vector machines. The unifying approach is required to allow automatic performance tuning for all discrete linear transforms.

In 1968, Pease [94] was the first who utilized Kronecker products for describing FFT algorithms. So it was possible to express all required operations on the matrix level and to obtain considerably clearer structures. Van Loan [109] used this technique for a state-of-the-art presentation of FFT algorithms. In the twenty-five years between the publications of Pease and Van Loan, only a few authors used this powerful technique: Temperton [106] and Johnson et al. [65] for FFT implementations on classic vector computers and Norton and Silberger [93] on parallel computers with MIMD architecture. Gupta et al. [45] and Pitsianis [95] used the Kronecker product formalism to synthesize FFT programs.

The Kronecker product approach to FFT algorithm design antiquates more conventional techniques like signal flow graphs. Signal flow graphs rely on the spatial symmetry of a graph representation of FFT algorithms, whereas the Kronecker product exploits matrix algebra. Following the idea of Johnson et al. [65],

the SPIRAL project (Moura et al. [88] and Püschel et al. [97]) provides the first automatic performance tuning system for the field of discrete linear transforms. One foundation of SPIRAL is the work of Johnson et al. [65] which is extended to cover general discrete linear transforms.

The Kronecker product approach makes it easy to modify a linear transform algorithm by exploiting the underlying algebraic structure of its matrix representation. This is in contrast to the usual signal flow approach where no well defined methodology for modifying linear transform algorithms is available.

A.1 Notation

The notational conventions introduced in the following are used throughout this chapter. Integers denoting problem sizes are referred to by capital letters M , N , etc. Loop indices and counters are denoted by lowercase letters i , j , etc. General integers are denoted by k , m , n , etc. as well as r , s , t , etc.

A.1.1 Vector and Matrix Notation

In this chapter, vectors of real or complex numbers will be referred to by lowercase letters x , y , z , etc., while matrices appear as capital letters A , B , C , etc.

Parameterized matrices (where the size and/or the entries depend on the actual parameters) are denoted by upright capital letters and their parameters.

Example A.1 (Parameterized Matrices) L_8^{64} is a stride permutation matrix of size 64×64 with stride 8 (see Section A.4), T_2^8 is a complex diagonal matrix of size 8×8 whose entries are given by the parameter “2” (see Section A.5), and I_4 is an identity matrix of size 4×4 .

Discrete linear transform matrices are denoted by an abbreviation in upright capital letters and a parameter that denotes the problem size.

Example A.2 (Discrete Linear Transforms) WHT_N denotes a Walsh-Hadamard transform matrix of size $N \times N$ and DFT_N denotes a discrete Fourier transform matrix of size $N \times N$ (see Section 4.1).

Row and column indices of vectors and matrices start from *zero* unless otherwise stated.

The vector space of complex n -vectors is denoted by \mathbb{C}^n . Complex m -by- n matrices are denoted by $\mathbb{C}^{m \times n}$.

Example A.3 (Complex Matrix) The 2-by-3 complex matrix $A \in \mathbb{C}^{2 \times 3}$ is expressed as

$$A = \begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \end{pmatrix}, \quad a_{00}, \dots, a_{12} \in \mathbb{C}.$$

Rows and columns are indexed from zero.

A.1.2 Submatrix Specification

Submatrices of $A \in \mathbb{C}^{m \times n}$ are denoted by $A(u, v)$, where u and v are *index vectors* that define the rows and columns of A used to construct the respective submatrix.

Index vectors can be specified using the *colon notation*:

$$u = j : k \quad \Leftrightarrow \quad u = (j, j+1, \dots, k), \quad j \leq k.$$

Example A.4 (Submatrix) $A(2 : 4, 3 : 7) \in \mathbb{C}^{3 \times 5}$ is the 3-by-5 submatrix of $A \in \mathbb{C}^{m \times n}$ (with $m \geq 4$ and $n \geq 7$) defined by the rows 2, 3, and 4 and the columns 3, 4, 5, 6, and 7.

There are special notational conventions when all rows or columns are extracted from their parent matrix. In particular, if $A \in \mathbb{C}^{m \times n}$, then

$$\begin{aligned} A(u, :) &\Leftrightarrow A(u, 0 : n-1), \\ A(:, v) &\Leftrightarrow A(0 : m-1, v). \end{aligned}$$

Vectors with non-unit increments are specified by the notation

$$u = i : j : k \quad \Leftrightarrow \quad u = (i, i+k, \dots, j),$$

where $k \in \mathbb{Z} \setminus \{0\}$ denotes the increments. The number of elements specified by this notation is

$$\max \left(\left\lfloor \frac{j-i+k}{k} \right\rfloor, 0 \right).$$

Example A.5 (Non-unit Increments) Let $A \in \mathbb{C}^{m \times n}$, then

$$A(0 : m-1 : 2, :) \in \mathbb{C}^{\lfloor \frac{m+1}{2} \rfloor \times n}$$

is the submatrix with the even-indexed rows of A , whereas $A(:, n-1 : 0 : -1) \in \mathbb{C}^{m \times n}$ is A with its columns in reversed order.

A.1.3 Diagonal Matrices

If $d \in \mathbb{C}^n$, then $D = \text{diag}(d) = \text{diag}(d_0, \dots, d_{n-1}) \in \mathbb{C}^{n \times n}$ is the diagonal matrix

$$D = \begin{pmatrix} d_0 & & & 0 \\ & d_1 & & \\ & & \ddots & \\ 0 & & & d_{n-1} \end{pmatrix}.$$

Example A.6 (Identity Matrix) The $n \times n$ identity matrix I_n is a parameterized matrix where the parameter n defines the size of the square matrix and is given by

$$I_n = \begin{pmatrix} 1 & & & 0 \\ & 1 & & \\ & & \ddots & \\ 0 & & & 1 \end{pmatrix}.$$

A.1.4 Conjugation

If $A \in \mathbb{C}^{n \times n}$ is an arbitrary matrix and $P \in \mathbb{C}^{n \times n}$ is an invertible matrix then the conjugation of A by P is defined as

$$A^P = P^{-1}AP.$$

In this chapter P is a permutation matrix in most cases.

Example A.7 (Conjugation of a Matrix) The 2×2 diagonal matrix

$$A = \begin{pmatrix} a_0 & 0 \\ 0 & a_1 \end{pmatrix}$$

is conjugated by the 2×2 anti-diagonal

$$J_2 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

leading to

$$A^{J_2} = J_2^{-1} A J_2 = \begin{pmatrix} a_1 & 0 \\ 0 & a_0 \end{pmatrix}.$$

Property A.1 (Conjugation) For any $A \in \mathbb{C}^{n \times n}$ and $P \in \mathbb{C}^{n \times n}$ being an invertible matrix it holds that

$$PA^P = AP.$$

Property A.2 (Conjugation) For any $A \in \mathbb{C}^{n \times n}$ and $P \in \mathbb{C}^{n \times n}$ being an invertible matrix it holds that

$$A^P P^{-1} = P^{-1} A.$$

A.1.5 Direct Sum of Matrices

Definition A.1 (Direct Sum of Matrices) The direct sum of two matrices A and B is given by

$$A \oplus B = \begin{pmatrix} A & \mathbf{0} \\ \mathbf{0} & B \end{pmatrix},$$

where the $\mathbf{0}$'s denote blocks of zeros of appropriate size.

Given n matrices A_0, A_1, \dots, A_{n-1} being *not* necessarily of the same dimension, their direct sum is defined as the block diagonal matrix

$$\bigoplus_{i=0}^{n-1} A_i = A_0 \oplus A_1 \oplus \dots \oplus A_{n-1} = \begin{pmatrix} A_0 & & & \mathbf{0} \\ & A_1 & & \\ & & \ddots & \\ \mathbf{0} & & & A_{n-1} \end{pmatrix}.$$

A.1.6 Direct Sum of Vectors

Vectors are usually regarded as elements of the vector space \mathbb{C}^N and not as matrices in $\mathbb{C}^{N \times 1}$ or $\mathbb{C}^{1 \times N}$. Thus the direct sum of vectors is a vector. The direct sum of vectors can be used to decompose a vector into subvectors as required in various algorithms.

Definition A.2 (Direct Sum of Vectors) Let y be a vector of length N and x_i be n vectors of lengths m_i :

$$y = \begin{pmatrix} u_0 \\ u_1 \\ \vdots \\ u_{N-1} \end{pmatrix}, \quad x_0 = \begin{pmatrix} u_0 \\ u_1 \\ \vdots \\ u_{m_0-1} \end{pmatrix}, \quad x_1 = \begin{pmatrix} u_{m_0} \\ u_{m_0+1} \\ \vdots \\ u_{m_1-1} \end{pmatrix}, \quad \dots, \quad x_{n-1} = \begin{pmatrix} u_{m_{n-2}} \\ u_{m_{n-2}+1} \\ \vdots \\ u_{N-1} \end{pmatrix}.$$

Then the direct sum of x_0, x_1, \dots, x_{n-1} is defined by

$$y = \bigoplus_{i=0}^{n-1} x_i = x_0 \oplus x_1 \oplus \dots \oplus x_{n-1} = \begin{pmatrix} u_0 \\ u_1 \\ \vdots \\ u_{N-1} \end{pmatrix} \in \mathbb{C}^N.$$

A.2 Extended Subvector Operations

Most identities introduced in this chapter can be formulated and proved easily using the standard basis.

Definition A.3 (Standard Basis) Let $e_0^N, e_1^N, \dots, e_{N-1}^N$ denote the vectors in \mathbb{C}^N with a 1 in the component given by the subscript and 0 elsewhere. The set

$$\{e_i^N : i = 0, 1, \dots, N-1\} \quad (\text{A.1})$$

is the standard basis of \mathbb{C}^N .

A.3 Kronecker Products

Definition A.4 (Kronecker or Tensor Product) The Kronecker product (tensor product) of the matrices $A \in \mathbb{C}^{M_1 \times N_1}$ and $B \in \mathbb{C}^{M_2 \times N_2}$ is the block structured matrix

$$A \otimes B := \begin{pmatrix} a_{0,0}B & \dots & a_{0,N_1-1}B \\ \vdots & \ddots & \vdots \\ a_{M_1-1,0}B & \dots & a_{M_1-1,N_1-1}B \end{pmatrix} \in \mathbb{C}^{M_1 M_2 \times N_1 N_2}.$$

Definition A.5 (Tensor Basis) Set $N = N_1 N_2$ and form the set of tensor products

$$e_i^{N_1} \otimes e_j^{N_2}, \quad i = 0, 1, \dots, N_1 - 1, \quad j = 0, 1, \dots, N_2 - 1. \quad (\text{A.2})$$

This set is called *tensor basis*.

Since any element e_k^N of the standard basis (A.1) can be expressed as

$$e_{j+iN_2}^N = e_i^{N_1} \otimes e_j^{N_2}, \quad i = 0, 1, \dots, N_1 - 1, \quad j = 0, 1, \dots, N_2 - 1,$$

the tensor basis of Definition A.5 ordered by choosing j to be the fastest running parameter is the standard basis of \mathbb{C}^N . In particular, the set of tensor products of the form

$$x^{N_1} \otimes y^{N_2}$$

spans \mathbb{C}^N , $N = N_1 N_2$.

The following two special cases of Kronecker products involving identity matrices are of high importance.

Definition A.6 (Parallel Kronecker Products) Let $A \in \mathbb{C}^{m \times n}$ be an arbitrary matrix and let $I_k \in \mathbb{C}^{k \times k}$ be the identity matrix. The expression

$$I_k \otimes A = \begin{pmatrix} A & & 0 \\ & A & \\ & & \ddots \\ 0 & & & A \end{pmatrix}$$

is called *parallel Kronecker product*.

A parallel Kronecker product can be viewed as a *parallel operation*. Its action on a vector $x = x_0 \oplus x_1 \oplus \dots \oplus x_{k-1}$ can be performed by computing the action of A on each of the k consecutive segments x_i of x independently.

Example A.8 (Parallel Kronecker Product) Let $A_2 \in \mathbb{C}^{2 \times 2}$ be an arbitrary matrix and let $I_3 \in \mathbb{C}^{3 \times 3}$ be the identity matrix. Then

$$y := (I_3 \otimes A_2)x$$

is given by

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{pmatrix} := \begin{pmatrix} a_{0,0} & a_{0,1} & & & & \\ a_{1,0} & a_{1,1} & & & & \\ & & a_{0,0} & a_{0,1} & & \\ & & a_{1,0} & a_{1,1} & & \\ & & & & a_{0,0} & a_{0,1} \\ & & & & a_{1,0} & a_{1,1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix}.$$

This matrix-vector product can be realized by splitting up the input vector $x \in \mathbb{C}^6$ into three subvectors of length 2 and performing the respective matrix-vector products

$$\begin{aligned} \begin{pmatrix} y_0 \\ y_1 \end{pmatrix} &:= \begin{pmatrix} a_{0,0} & a_{0,1} \\ a_{1,0} & a_{1,1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} \\ \begin{pmatrix} y_2 \\ y_3 \end{pmatrix} &:= \begin{pmatrix} a_{0,0} & a_{0,1} \\ a_{1,0} & a_{1,1} \end{pmatrix} \begin{pmatrix} x_2 \\ x_3 \end{pmatrix} \\ \begin{pmatrix} y_4 \\ y_5 \end{pmatrix} &:= \begin{pmatrix} a_{0,0} & a_{0,1} \\ a_{1,0} & a_{1,1} \end{pmatrix} \begin{pmatrix} x_4 \\ x_5 \end{pmatrix} \end{aligned}$$

independently.

Definition A.7 (Vector Kronecker Products) Let $A \in \mathbb{C}^{m \times n}$ be an arbitrary matrix and let $I_k \in \mathbb{C}^{k \times k}$ be the identity matrix. The expression

$$A \otimes I_k = \begin{pmatrix} a_{0,0} I_k & \cdots & a_{0,n-1} I_k \\ \vdots & \ddots & \vdots \\ a_{m-1,0} I_k & \cdots & a_{m-1,n-1} I_k \end{pmatrix}$$

is called *vector Kronecker product*.

A vector Kronecker product can be viewed as a *vector operation*. To compute its action on a vector $x = x_0 \oplus x_1 \oplus \cdots \oplus x_{n-1}$, the n vector operations

$$a_{r,0}x_0 + a_{r,1}x_1 + \cdots + a_{r,n-1}x_{n-1}, \quad r = 0, 1, \dots, m-1$$

are performed. Expressions of the form $A \otimes I_k$ are called vector operations as they operate on vectors of size k .

Example A.9 (Vector Kronecker Product) Let $A_2 \in \mathbb{C}^{2 \times 2}$ be an arbitrary matrix and let $I_3 \in \mathbb{C}^{3 \times 3}$ be the identity matrix. Then

$$y := (A_2 \otimes I_3)x$$

is given by

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{pmatrix} := \begin{pmatrix} a_{0,0} & & a_{0,1} & & & \\ & a_{0,0} & & a_{0,1} & & \\ & & a_{0,0} & & a_{0,1} & \\ a_{1,0} & & & a_{1,1} & & \\ & a_{1,0} & & a_{1,1} & & \\ & & a_{1,0} & & a_{1,1} & \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix}.$$

This matrix-vector product can be computed by splitting up the vector $x \in \mathbb{C}^6$ into two subvectors of length 3 and performing single scalar multiplications with these subvectors:

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \end{pmatrix} := a_{0,0} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} + a_{0,1} \begin{pmatrix} x_3 \\ x_4 \\ x_5 \end{pmatrix}$$

$$\begin{pmatrix} y_3 \\ y_4 \\ y_5 \end{pmatrix} := a_{1,0} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} + a_{1,1} \begin{pmatrix} x_3 \\ x_4 \\ x_5 \end{pmatrix}.$$

A.3.1 Algebraic Properties of Kronecker Products

Most of the following Kronecker product identities can be demonstrated to hold by computing the action of both sides on the tensor basis given by Definition A.5.

Property A.3 (Identity) If I_m and I_n are identity matrices, then

$$I_m \otimes I_n = I_{mn}.$$

Property A.4 (Identity) If I_m and I_n are identity matrices, then

$$I_m \oplus I_n = I_{m+n}.$$

Property A.5 (Associativity) If A, B, C are arbitrary matrices, then

$$(A \otimes B) \otimes C = A \otimes (B \otimes C).$$

Thus, the expression $A \otimes B \otimes C$ is unambiguous.

Property A.6 (Transposition) If A, B are arbitrary matrices, then

$$(A \otimes B)^T = A^T \otimes B^T.$$

Property A.7 (Inversion) If A and B are regular matrices, then

$$(A \otimes B)^{-1} = A^{-1} \otimes B^{-1}.$$

Property A.8 (Mixed-Product Property) If A, B, C and D are arbitrary matrices, then

$$(A \otimes B)(C \otimes D) = AC \otimes BD,$$

provided the products AC and BD are defined.

A consequence of this property is the following factorization.

Corollary A.1 (Decomposition) If $A \in \mathbb{C}^{m_1 \times n_1}$ and $B \in \mathbb{C}^{m_2 \times n_2}$, then

$$A \otimes B = A I_{n_1} \otimes I_{m_2} B = (A \otimes I_{m_2})(I_{n_1} \otimes B),$$

$$A \otimes B = I_{m_1} A \otimes B I_{n_2} = (I_{m_1} \otimes B)(A \otimes I_{n_2}).$$

The mixed-product property can be generalized in two different ways.

Corollary A.2 (Generalized Mixed-Product Property) For k matrices of appropriate sizes it holds that

$$(A_1 \otimes A_2 \otimes \cdots \otimes A_k)(B_1 \otimes B_2 \otimes \cdots \otimes B_k) = A_1 B_1 \otimes A_2 B_2 \cdots \otimes A_k B_k,$$

and

$$(A_1 \otimes B_1)(A_2 \otimes B_2) \cdots (A_k \otimes B_k) = (A_1 A_2 \cdots A_k) \otimes (B_1 B_2 \cdots B_k).$$

Property A.9 (Distributive Law) If A, B , and C are arbitrary matrices, then

$$(A + B) \otimes C = (A \otimes C) + (B \otimes C),$$

$$A \otimes (B + C) = (A \otimes B) + (A \otimes C).$$

The Kronecker product is not commutative. This non-commutativity is mainly responsible for the richness of the Kronecker product algebra, and naturally leads to a distinguished class of permutations, the stride permutations. An important consequence of this lack of commutativity can be seen in the relationship between Kronecker products and direct sums of matrices.

Property A.10 (Left Distributive Law) It holds that

$$(A \oplus B) \otimes C = (A \otimes C) \oplus (B \otimes C).$$

The *right* distributive law does *not* hold.

A.4 Stride Permutations

Definition A.8 (Stride Permutation) For a vector $x \in \mathbb{C}^{mn}$ with

$$x = \sum_{k=0}^{mn-1} x_k e_k^{mn} \quad \text{with} \quad e_k^{mn} = e_i^n \otimes e_j^m, \quad \text{and} \quad x_k \in \mathbb{C},$$

the stride permutation L_n^{mn} is defined by its action on the tensor basis (A.2) of \mathbb{C}^{mn} .

$$L_n^{mn}(e_i^n \otimes e_j^m) = e_j^m \otimes e_i^n.$$

The permutation operator L_n^{mn} sorts the components of x according to their index modulo n . Thus, components with indices equal to $0 \bmod n$ come first, followed by the components with indices equal to $1 \bmod n$, and so on.

Corollary A.3 (Stride Permutation) For a vector $x \in \mathbb{C}^{mn}$ the application of the stride permutation L_n^{mn} results in

$$L_n^{mn} x := \begin{pmatrix} x(0 : (m-1)n : n) \\ x(1 : (m-1)n + 1 : n) \\ \vdots \\ x(n-1 : mn-1 : n) \end{pmatrix}.$$

Definition A.9 (Even-Odd Sort Permutation) The permutation L_2^n , n being even, is called an even-odd sort permutation, because it groups the even-indexed and odd-indexed components together.

Definition A.10 (Perfect Shuffle Permutation) The permutation $L_{n/2}^n$, n being even, is called a perfect shuffle permutation, since its action on a deck of cards could be the shuffling of two equal piles of cards so that the cards are interleaved one from each pile.

The perfect shuffle permutation $L_{n/2}^n$ is denoted in short by Π_n .

Mixed Kronecker Products

Combinations of tensor products and stride permutations have both vector and parallel characteristics like stride permutations and additionally feature arithmetic operations like parallel and vector Kronecker products.

The factorization of these constructs leads to the short vector Cooley-Tukey FFT.

Definition A.11 (Right Mixed Kronecker Product) Let $A \in \mathbb{C}^{m \times n}$ be an arbitrary matrix, $I_k \in \mathbb{C}^{k \times k}$ be the identity matrix, and L_k^{km} be a stride permutation. An expression of the form

$$(I_k \otimes A) L_k^{mk}$$

is called *right mixed Kronecker product*.

Definition A.12 (Left Mixed Kronecker Product) Let $A \in \mathbb{C}^{m \times n}$ be an arbitrary matrix, $I_k \in \mathbb{C}^{k \times k}$ be the identity matrix, and L_k^{mk} be a stride permutation. An expression of the form

$$L_k^{mk}(A \otimes I_k)$$

is called *left mixed Kronecker product*.

A.4.1 Algebraic Properties of Stride Permutations

Property A.11 (Identity)

$$L_1^n = L_n^n = I_n$$

Property A.12 (Inversion/Transposition) If $N = mn$ the

$$(L_m^{mn})^{-1} = (L_m^{mn})^\top = L_n^{mn}.$$

Property A.13 (Multiplication) If $N = kmn$ then

$$L_k^{kmn} L_m^{kmn} = L_m^{kmn} L_k^{kmn} = L_{km}^{kmn}.$$

Example A.10 (Inversion of the Perfect Shuffle Permutation) The inverse matrix of $L_2^{2^i}$ is given by the perfect shuffle permutation:

$$(L_2^{2^i})^{-1} = L_{2^{i-1}}^{2^i} = \Pi_{2^i}.$$

As already mentioned, the Kronecker product is not commutative. However, with the aid of stride permutations, the order of factors can be reverted.

Theorem A.1 (Commutation) If $A \in \mathbb{C}^{m_1 \times n_1}$ and $B \in \mathbb{C}^{m_2 \times n_2}$ then

$$L_{m_1}^{m_1 m_2}(A \otimes B) = (B \otimes A) L_{n_2}^{n_1 n_2}.$$

Proof: Johnson et al. [65].

Several special cases are worth noting.

Corollary A.4 If $A \in \mathbb{C}^{m \times m}$ and $B \in \mathbb{C}^{n \times n}$ then

$$A \otimes B = L_m^{mn}(B \otimes A) L_n^{mn} = (B \otimes A) L_m^{mn}.$$

Application of this relation leads to

$$\begin{aligned} I_m \otimes B &= L_m^{mn}(B \otimes I_m) L_n^{mn} = (B \otimes I_m) L_n^{mn}, \\ A \otimes I_n &= L_m^{mn}(I_n \otimes A) L_n^{mn} = (I_n \otimes A) L_n^{mn}. \end{aligned}$$

Stride permutations interchange parallel and vector Kronecker factors. The readdressing prescribed by L_n^{mn} on input and L_m^{mn} on output turns the vector Kronecker factor $A \otimes I_n$ into the parallel Kronecker factor $I_n \otimes A$ and the parallel Kronecker factor $I_m \otimes B$ into the vector Kronecker factor $B \otimes I_m$. Continuing this way, it is possible to write

$$\begin{aligned} A \otimes B &= (A \otimes I_n)(I_m \otimes B) \\ &= L_m^{mn}(I_n \otimes A) L_n^{mn}(I_m \otimes B), \end{aligned} \quad (\text{A.3})$$

which can be used to compute the action of $A \otimes B$ as a sequence of two parallel Kronecker factors. It also holds that

$$A \otimes B = (A \otimes I_n) L_m^{mn}(B \otimes I_m) L_n^{mn}, \quad (\text{A.4})$$

which can be used to compute the action of $A \otimes B$ as a sequence of two vector Kronecker factors. The stride permutations intervene between computational stages, providing a mathematical language for describing the readdressing.

Occasionally it will be necessary to permute the factors in a tensor product of more than two factors.

Frequently used properties which can be traced back to those before are stated in the following.

Property A.14 If $A \in \mathbb{C}^{m \times m}$ and $B \in \mathbb{C}^{n \times n}$ then

$$A \otimes B = L_m^{nm}(I_n \otimes A) L_n^{mn}(I_m \otimes B).$$

Property A.15 If $N = kmn$ then

$$L_n^{kmn} = (L_n^{kn} \otimes I_m)(I_k \otimes L_n^{mn}).$$

Property A.16 If $N = kmn$ then

$$L_{km}^{kmn} = (I_k \otimes L_m^{mn})(L_k^{kn} \otimes I_m).$$

Property A.17 If $N = kmn$ then

$$(L_m^{km} \otimes I_n) = (I_m \otimes L_k^{kn}) L_{mn}^{kmn}.$$

Proof: Using Properties A.12 and A.16 lead to

$$(L_m^{km} \otimes I_n) = (I_m \otimes L_k^{km})(I_m \otimes L_m^{km})(L_m^{km} \otimes I_n) = (I_m \otimes L_k^{kn})L_{mn}^{kmn}. \quad \square$$

Property A.18 If $N = kmn$ then

$$(L_m^{km} \otimes I_n)(I_k \otimes L_m^{mn})L_k^{kmn} = (I_m \otimes L_k^{kn})(L_m^{mn} \otimes I_k).$$

Proof: Using Property A.17 leads to

$$(I_m \otimes L_k^{kn})L_{mn}^{kmn}(I_k \otimes L_m^{mn})L_k^{kmn} = (I_m \otimes L_k^{kn})(L_m^{mn} \otimes I_k). \quad \square$$

The following two properties show, how the mixed Kronecker product can be decomposed. Property A.19 shows the more general case and Property A.20 shows the full factorization.

Property A.19

$$(I_{sk} \otimes A_{ms \times n})L_{sk}^{skn} = \left(I_k \otimes L_s^{ms^2} (A_{ms \times n} \otimes I_s) \right) (L_k^{kn} \otimes I_s)$$

Property A.20

$$(I_{sk} \otimes A_{ms \times n})L_{sk}^{skn} = \left(I_k \otimes (L_s^{ms} \otimes I_s) \left(I_m \otimes L_s^{s^2} \right) (A_{ms \times n} \otimes I_s) \right) (L_k^{kn} \otimes I_s)$$

A.4.2 Digit Permutations

The following permutation generalizes the stride permutation.

Definition A.13 (Digit Permutation) Let $N = N_1 N_2 \cdots N_k$ and let σ be a permutation of the numbers $1, 2, \dots, k$. Then the digit permutation is defined by

$$L_{\sigma}^{(N_1, \dots, N_k)}(e_{i_1}^{N_1} \otimes \cdots \otimes e_{i_k}^{N_k}) = (e_{i_{\sigma(1)}}^{N_{\sigma(1)}} \otimes \cdots \otimes e_{i_{\sigma(k)}}^{N_{\sigma(k)}}).$$

Theorem A.2 (Permutation) Let A_0, A_1, \dots, A_k be $N_i \times N_i$ matrices and let σ be a permutation of the numbers $1, 2, \dots, k$, then

$$A_1 \otimes \cdots \otimes A_k = (L_{\sigma}^{(N_1, \dots, N_k)})^{-1} (A_{\sigma(1)} \otimes \cdots \otimes A_{\sigma(k)}) L_{\sigma}^{(N_1, \dots, N_k)}.$$

Proof: Johnson et al. [65].

Digit reversal is a special permutation arising in FFT algorithms.

Definition A.14 (Digit Reversal Matrix) The k -digit digit reversal permutation matrix

$$R^{(N_1, N_2, \dots, N_k)}$$

of size $N = N_1 N_2 \cdots N_k$ is defined by

$$R^{(N_1, \dots, N_k)}(e_{i_1}^{N_1} \otimes \cdots \otimes e_{i_k}^{N_k}) = e_{i_k}^{N_k} \otimes \cdots \otimes e_{i_1}^{N_1}.$$

The special case when $N_1 = N_2 = \cdots = N_k = p$ is denoted by R_{p^k} .

Theorem A.3 The digit reversal matrix R_{p^k} satisfies recursion

$$R_{p^k} = \prod_{i=2}^k (I_{p^{k-i}} \otimes L_p^{p^i}).$$

Proof: Johnson et al. [65].

A.5 Twiddle Factors and Diagonal Matrices

An important class of matrices arising in FFT factorizations are diagonal matrices whose diagonal elements are roots of unity. Such matrices are called twiddle factor matrices.

This section collects useful properties of diagonal matrices, especially twiddle factor matrices.

Definition A.15 (Twiddle Factor Matrix) Let $\omega_N = e^{2\pi i/N}$ denote the N th root of unity. The twiddle factor matrix, denoted by T_m^{mn} , is a diagonal matrix defined by

$$T_m^{mn}(e_i^m \otimes e_j^n) = \omega_{mn}^{ij}(e_i^m \otimes e_j^n), \quad i = 0, 1, \dots, m-1, \quad j = 0, 1, \dots, n-1,$$

$$T_m^{mn} = \bigoplus_{i=0}^{m-1} \bigoplus_{j=0}^{n-1} \omega_{mn}^{ij} = \bigoplus_{i=0}^{m-1} \Omega_{n,i}(\omega_{mn}),$$

where $\Omega_{n,k}(\alpha) = \text{diag}(1, \alpha, \dots, \alpha^{n-1})^k$.

The following corollary shows how to conjugate diagonal matrices with a permutation matrix. It holds for all diagonal matrices, but is particularly useful when calculating twiddle factors in FFT algorithms.

Corollary A.5 (Conjugating Diagonal Matrices) Let

$$D = \text{diag}(d_0, d_1, \dots, d_{N-1})$$

be an arbitrary $N \times N$ diagonal matrix and P_σ the permutation matrix according to the permutation σ of $(0, 1, \dots, N-1)$. Conjugating D by P_σ results in a new diagonal matrix whose diagonal elements are permuted by σ , i. e.,

$$D^{P_\sigma} = P_\sigma^{-1} D P_\sigma = \text{diag}(d_{\sigma(0)}, d_{\sigma(1)}, \dots, d_{\sigma(N-1)}) = \bigoplus_{i=0}^{N-1} d_{\sigma(i)}.$$

Corollary A.6 (Conjugating Twiddle Factors) Conjugating T_m^{mn} by L_m^{mn} results in T_n^{mn} , i. e.,

$$(T_m^{mn})^{L_m^{mn}} = T_n^{mn}.$$

Tensor bases are a useful tool to compute the actual entries of conjugated twiddle factor matrices.

Property A.21 (Twiddle Factor $I_r \otimes T_m^{mn}$)

$$(I_r \otimes T_m^{mn})(e_i^r \otimes e_j^m \otimes e_k^n) = \omega_{mn}^{jk}(e_i^r \otimes e_j^m \otimes e_k^n),$$

$$I_r \otimes T_m^{mn} = \bigoplus_{i=0}^{r-1} \bigoplus_{j=0}^{m-1} \bigoplus_{k=0}^{n-1} \omega_{mn}^{jk} = \bigoplus_{i=0}^{r-1} \bigoplus_{j=0}^{m-1} \Omega_{n,j}(\omega_{mn})$$

$(I_r \otimes T_m^{mn})^P$ is the form of twiddle factor matrices as found in FFT algorithms. The following example shows how to compute with twiddle factors in this form.

Example A.11 (Conjugation of Twiddle Factors) Consider the construct

$$(I_4 \otimes T_4^8)^{L_8^{32}} = L_4^{32} (I_4 \otimes T_4^8) L_8^{32}.$$

Thus, $I_4 \otimes T_4^8$ is conjugated by L_8^{32} . Computation of the result yields

$$\begin{aligned} (L_4^{32} (I_4 \otimes T_4^8) L_8^{32})(e_i^4 \otimes e_j^4 \otimes e_k^2) &= (L_4^{32} (I_4 \otimes T_4^8))(e_j^4 \otimes e_k^2 \otimes e_i^4) \\ &= L_4^{32} \omega_8^{ki}(e_j^4 \otimes e_k^2 \otimes e_i^4) \\ &= \omega_8^{ki}(e_i^4 \otimes e_j^4 \otimes e_k^2) \end{aligned}$$

$$\begin{aligned} L_4^{32} (I_4 \otimes T_4^8) L_8^{32} &= \bigoplus_{i=0}^3 \bigoplus_{j=0}^3 \bigoplus_{k=0}^1 \omega_8^{ik} = \bigoplus_{i=0}^3 \bigoplus_{j=0}^3 \Omega_{2,j}(\omega_8) \\ &= \text{diag}(1, 1, 1, \omega_8, 1, \omega_8^2, 1, \omega_8^3, 1, 1, 1, \omega_8, 1, \omega_8^2, 1, \omega_8^3, \\ &\quad 1, 1, 1, \omega_8, 1, \omega_8^2, 1, \omega_8^3, 1, 1, 1, \omega_8, 1, \omega_8^2, 1, \omega_8^3). \end{aligned}$$

Appendix B

Compiler Techniques

B.1 Register Allocation and Memory Accesses

To access variables residing in physical registers is highly desirable because (i) register accesses are generally much faster than memory accesses, and (ii) most instructions in current hardware have been devised to carry out register-to-register operations. But there is an obstacle to be overcome: Physical registers are a very scarce and precious resource. As computational data initially resides in memory, the two main tasks to be tackled are (i) the minimization of data traffic between the few but fast registers and the larger but much slower memory, and (ii) the maximization of the reuse of data residing in registers.

These optimization tasks are crucial for improving the performance of assembly code. They are addressed by special register allocation techniques.

B.1.1 Register Allocation

The purpose of register allocation is the assignment of an arbitrary but finite number of temporary variables to a hardware limited number of physical registers during program execution. In the following, some definitions are given to enable a detailed description of the register allocation process.

Variable Conflicts. When assigned to the same physical register, a temporary variable is in conflict with another temporary variable if one of them is used both before and after the other one's usage within a short period of time.

Register Spilling. A register spill occurs when there are more temporary variables to be loaded than the target hardware's number of free registers. Spill code has to be generated to transfer or *spill* the content of a physical register to a specified stack location in order to make room for the desired data. The register allocator's heuristic strategy determines the best register to be spilled, the "victim", depending on its further usage.

Registers whose content is not accessed anymore can be overwritten without being kept on the stack and therefore are preferred to be such victims.

Register Reloading. The reload of a register content always occurs after a preceding spill. The spilled content is reloaded into register from the relevant stack location.

Typically, there are far more temporary variables than physical registers. Therefore many temporary variables are to be assigned to one physical register which

evokes variable conflicts and thus imposes a negative impact on a code's performance. The register allocator's goal, besides the tasks already stated above, is to make these assignments such that conflicts and resulting register spills are avoided as much as possible.

As the preceding higher optimization steps like vectorizer and Optimizer I made all integer address computation for accessing array elements in implicit form, i.e., not carrying out explicit address computation in the code, register allocation is carried out as low-level optimization in order to take integer address computation code into account in the assignment process of integer registers.

B.2 Instruction Scheduling

Instruction scheduling is a low-level optimization technique aiming at the rearrangement of the pipeline executed instructions's micro-operations to maximize the number of function units operating in parallel and to minimize the time they spend waiting for each other [69]. To accommodate a pipeline's delay or latency that is necessary between an instruction and its dependent successor, adjacent instructions have to be made independent, e.g., to be transformed into one instruction which writes a register and into another one which reads from it. If such a reordering of instructions is not carried out, pipelines would have to stall on unresolved dependencies resulting in unsatisfying pipeline utilization. To maximize a program's overall performance, it is essential that the respective code is scheduled in a way to take best advantage of the pipelines provided by the architecture [91].

B.2.1 Constraints for Instruction Scheduling

It is a highly demanding task to find a good execution order of instructions while (i) preserving data dependencies, and (ii) utilizing execution units, and (iii) taking multiple instruction issue properties into account. Considering this, instruction scheduling algorithms must identify the instructions which can be executed in parallel.

Data Dependencies

If two instructions read from and write to one variable, there may be a data dependency between them. Three different types of data dependencies, i.e., (i) true, (ii) anti, and (iii) output will be described in detail using the following exemplary instruction sequence:

```
I1  t0 = x[3];  
I2  t1 = 0.7071067811865476*t0;  
I3  t0 = t3 - t4;
```

True Dependency—RAW. A true dependency exists between instructions I1 and I2. I2 has t0 as one of its source operands which is written by the preceding instruction I1. To resolve true dependencies, each consumer of a temporary variable has to be issued after its producer, i. e., *read after write* (RAW).

Anti Dependency—WAR. An anti dependency exists between I2 and I3. I2 has t0 as source operand and the following instruction I3 writes t0. To resolve anti dependencies, a new producer of a temporary variable has to be issued after it has been read by its consumers before, i. e., *write after read* (WAR).

Output Dependency—WAW. An output dependency exists between I1 and I3. The order in which producers I1 and I3 write the temporary variable t0 has to be preserved. To resolve output dependencies, a new producer of a temporary variable has to be issued after its preceding producer has written it before, i. e., *write after write* (WAW).

Anti dependencies and output dependencies are also referred to as *false dependencies* as they arise due to the reuse of the same register variables.

Appendix C

Performance Assessment

The assessment of scientific software requires the use of numerical values, which may be determined analytically or empirically, for the quantitative description of performance. As to which parameters are used and how they will be interpreted depends on *what* is to be assessed. The techniques discussed in this appendix have a strong impact on the experimental results presented in Chapter 10. A detailed discussion of performance assessment for both serial and parallel scientific software can be found in Gansterer and Ueberhuber [40].

The user of a computer system who waits for the solution of a particular problem is mainly interested in the time it takes for the problem to be solved. This time depends on two parameters, workload and performance:

$$\text{time} = \frac{\text{workload}}{\text{performance}_{\text{effective}}} = \frac{\text{workload}}{\text{performance}_{\text{peak}} \times \text{efficiency}} .$$

The computation time is therefore influenced by the following quantities:

1. The amount of work (*workload*) which has to be done. This depends on the nature and complexity of the problem as well as on the properties of the algorithm used to solve it. For a given problem complexity, the workload is a characteristic of the algorithm. The workload (and hence the required time) may be reduced by improving the algorithm.
2. *Peak performance* characterizes the computer hardware independently of particular application programs. The procurement of new hardware with high peak performance usually results in reduced time requirements for the solution of the same problem.
3. *Efficiency* is the percentage of peak performance achieved for a given computing task. It tells the user as to what share of the potential peak performance is actually exploited and thus measures the quality of the implementation of an algorithm. Efficiency may be increased by optimizing the program.

The correct and comprehensive performance assessment requires answers to a whole complex of questions: What limits are imposed, independently of specific programming techniques, by the hardware? What are the effects of the different variants of an algorithm on performance? What are the effects of specific programming techniques? What are the effects on efficiency of an optimizing compiler?

CPU Time

The fact that only a small share of computer resources are spent on a particular *job* in a multiuser environment is taken into account by measuring *CPU time*. This quantity specifies the amount of time the processor actually was engaged in solving a particular problem and neglects the time spent on other jobs or on waiting for input/output.

CPU time itself is divided into user CPU time and system CPU time. *User CPU time* is the time spent on executing an application program and its linked routines. *System CPU time* is the time consumed by all system functions required for the execution of the program, such as accessing virtual memory pages in the backing store or executing I/O operations.

Peak Performance

An important hardware characteristic, the *peak performance* P_{\max} of a computer, specifies the maximum number of floating-point (or other) operations which can theoretically be performed per time unit (usually per second).

The peak performance of a computer can be derived from its cycle time T_c and the maximum number N_c of operations which can be executed during a clock cycle:

$$P_{\max} = \frac{N_c}{T_c}.$$

If P_{\max} refers to the number of floating-point operations executed per second, then the result states the *floating-point peak performance*. It is measured in

flop/s (*floating-point operations per second*)

or Mflop/s (10^6 flop/s), Gflop/s (10^9 flop/s), or Tflop/s (10^{12} flop/s). Unfortunately, the fact that there are different classes of floating-point operations, which take different amounts of time to be executed, is neglected far too often (Ueberhuber [108]).

Notation (flop/s) Some authors use the notation flops, Mflops, Gflops etc. instead of flop/s, Mflop/s, Gflop/s etc.

It is most obvious that no program, no matter how efficient, can perform better than peak performance on a particular computer. In fact, only specially optimized parts of a program may come close to peak performance. One of the reasons for this is that, in practice, address calculations, memory operations, and other operations which do not contribute directly to the result are left out of the operation count. Thus, peak performance may be looked upon as a kind of *speed of light* for a computer.

C.1 Short Vector Performance Measures

Short vector SIMD operation is small-scale parallelism. The speed-up of computations depends on the problem to solve and how it fits on the target architecture. In the context of this thesis, the *speed-up* is the most important measure of how efficient the available parallelism is used. The speed-up describes, how many times a vectorized program is executed faster on an n -way short vector SIMD processors than on the scalar FPU.

Definition C.1 (Speed-up) Suppose, program A can be vectorized in the way that it can be computed on an n -way short vector SIMD processors. T_1 denotes the time, the *scalar* program needs when using 1 processor, T_n denotes the time of the n -way vectorized program. Then, the speed-up is defined to be:

$$S_n = \frac{T_1}{T_n}.$$

In most cases, S_n will be higher than 1. However, sometimes, when the problem is not vectorizable efficiently, vectorization overhead will cause T_n to be higher than T_1 .

Speed-up $S_n := T_1/T_n$

Speed-up is the ratio between the run time T_1 of the scalar algorithm (prior to vectorization) and the run time of the vectorized algorithm utilizing an n -way short vector SIMD processor.

Efficiency $E_n := S_n/n \leq 1$

Concurrent efficiency is a metric for the utilization of a short vector SIMD processor's capacity. The closer E_n gets to 1, the better use is made of the potentially n -fold power of an n -way short vector SIMD processor system.

C.2 Empirical Performance Assessment

In contrast to analytical performance assessment obtained from the technical data from the computer system, empirical performance assessment is based on experiments and surveys conducted on given computer systems or abstract models (using simulation).

Temporal Performance

In order to compare different algorithms for solving a given problem on a single computer, either the *execution time*

$$T := t_{\text{end}} - t_{\text{start}}$$

itself or its inverse, referred to as *temporal performance* $P_T := T^{-1}$, can be used. To that end, the execution time is measured. The workload is normalized by $\Delta W = 1$, since only *one* problem is considered.

This kind of assessment is useful for deciding which algorithm or which program solves a given problem fastest. The execution time of a program is the main performance criterion for the user. He only wants to know how long he has to wait for the solution of *his* problem. For him, the most powerful and most efficient algorithm is determined by the shortest execution time or the largest temporal performance. From the user's point of view the workload involved and other details of the algorithm are usually irrelevant.

Empirical Floating-Point Performance

The *floating-point performance* characterizes the workload completed over the time span T as the number of floating-point operations executed in T :

$$P_F [\text{flop/s}] = \frac{W_F}{T} = \frac{\text{number of executed floating-point operations}}{\text{time in seconds}}.$$

This empirical quantity is obtained by measuring executed programs in real life situations. The results are expressed in terms of Mflop/s, Gflop/s or Tflop/s as with analytical performance indices.

Floating-point performance is more suitable for the comparison of different machines than instruction performance, because it is based on *operations* instead of *instructions*. This is because the number of instructions related to a program differs from computer to computer but that the number of floating-point operations will be more or less the same.

Floating-point performance indices based simply on counting floating-point operations may be too inaccurate unless a distinction is made between the different classes of floating-point operations and their respective number of required clock cycles. If these differences are neglected, a program consisting only of floating-point *additions* will have considerably better floating-point performance than a program consisting of the same number of floating-point *divisions*. On the POWER processor, for instance, a floating-point division takes around twenty times as long as a floating-point addition.

C.2.1 Interpretation of Empirical Performance Values

In contrast to peak performance, which is a hardware characteristic, the empirical floating-point performance analysis of computer systems can only be made with real programs, i. e., algorithm implementations. However, it would be misleading to use floating-point performance as an absolute criterion for the assessment of *algorithms*.

A program which achieves higher floating-point performance does not necessarily achieve higher temporal performance, i. e., shorter overall execution times.

In spite of a better (higher) flop/s value, a program may take *longer* to solve the problem if a larger workload is involved. Only for programs with equal workload can the floating-point performance indices be used as a basis for assessing the quality of different *implementations*.

For the benchmark assessment of computer systems, empirical floating-point performance is also suitable and is frequently used (for instance in the LINPACK benchmark or the SPEC¹ benchmark suite).

Pseudo Flop/s

In case of FFT algorithms an algorithm specific performance measure is used by some authors. The arithmetic complexity of $5N \log_2 N$ operations for a FFT transform of size N is assumed. This is an upper bound for the FFT computation and motivated by the fact that different FFT algorithms have slightly different operation counts ranging between $3N \log_2 N$ and $5N \log_2 N$ when all trivial twiddle factors are eliminated (see Appendix A.5). As a complication, some implementations do not eliminate all trivial twiddle factors and the actual number has to be counted. Thus, pseudo flop/s, $(5N \log N)/T$ (a scaled inverse of run time), is a easier comparable performance measure for FFTs and an upper bound for the actual performance (Frigo and Johnson [35]).

Empirical Efficiency

Sometimes it is of interest to obtain information about the degree to which a program and its compiler exploit the potential of a computer. To do so, the ratio between the empirical floating-point performance and the peak performance of the computer is considered.

This *empirical efficiency* is usually significantly lower than 100 %, a fact which is in part due to simplifications in the model for peak performance.

C.2.2 Run Time Measurement

The run-time is the second important performance index (in addition to the workload). To determine the performance of an algorithm, its run-time has to be measured. One has to deal with the resolution of the system clock. Then, most of the time not the whole program, but just some relevant parts of it have to be measured. For example, in FFT programs the initialization step usually is not included, as it takes place only once, and the transform is executed many times.

The following fragment of a C program demonstrates how to determine user and system CPU time as well as the overall *elapsed time* using the predefined subroutine `times`.

¹SPEC is the abbreviation of *Systems Performance Evaluation Cooperative*.

```

#include <sys/times.h>
...
/* period is the granularity of the subroutine times */
period = (float) 1/sysconf(_SC_CLK_TCK);
...
start_time = times(&begin_cpu_time);
/* begin of the examined section */
...
/* end of the examined section */
end_time   = times(&end_cpu_time);
user_cpu   = period*(end_cpu_time.tms_utime
                    - begin_cpu_time.tms_utime);
system_cpu = period*(end_cpu_time.tms_stime
                    - begin_cpu_time.tms_stime);
elapsed    = period*(end_time - start_time);

```

The subroutine `times` provides timing results as multiples of a specific period of time. This period depends on the computer system and must be determined with the UNIX standard subroutine `sysconf` before `times` is used. The subroutine `times` itself must be called immediately before and after that part of the program to be measured. The argument of `times` returns the accumulated user and system CPU times, whereas the current time is returned as the function value of `times`. The difference between the respective begin and end times finally yields, together with scaling by the predetermined period of time, the actual execution times.

Whenever the execution time is smaller than the resolution of the system clock, different solutions are possible: (i) Performance counters can be used to determine the exact number of cycles required, and (ii) the measured part of a program can be executed many times and the overall time is divided by the number of runs.

This second approach has a few drawbacks. The resulting time may be too optimistic, as first-time cache misses will only occur once and the pipeline might be used too efficiently when executing subsequent calls. A possible solution is to empty the cache with special instructions.

Calling in-place FFT algorithms repeatedly has the effect that in subsequent calls the output of the previous call is the input of the next call. This can result, when repeating this process very often, in excessive floating-point errors up to overflow conditions. This would not be a drawback by itself, if processors handled those exceptions as fast as normal operations. But some processors handle them with a great performance loss, making the timing results too pessimistic.

The solution to this problem is calling the program with special vectors (zero vector, eigenvectors) or restoring the first input between every run. The second solution leads to higher run times, but measuring this tiny fraction and subtracting it finally yields the correct result.

C.2.3 Workload Measurement

In order to determine the empirical efficiency of a program, it is necessary to determine the arithmetic complexity. This can be done either “analytically” by using formulas for the arithmetic complexity or “empirically” by counting executed floating-point operations on the computer systems used. Estimating the number of floating-point operations analytically has the disadvantage that real implementations of algorithms often do not achieve the complexity bounds given by analytical formulas.

In order to determine the number of executed floating-point operations, a special feature of modern microprocessors can be used: the *Performance Monitor Counter* (PMC). PMCs are hardware counters able to count various types of events, such as cache misses, memory coherence operations, branch mispredictions, and several categories of issued and graduated instructions. In addition to characterizing the workload of an application by counting the number of floating-point operations, PMCs can help application developers for gaining deeper insight into application performance and for pinpointing performance bottlenecks.

PMCs were first used extensively on Cray vector processors, and appear in some form in all modern microprocessors, such as the MIPS R10000 [39, 83, 84, 117], Intel IA-32 processors and Itanium processor family [54, 57, 82], IBM POWER PC family [111], DEC Alpha [19], and HP PA-8x00 family [48]. Most of the microprocessor vendors provide hardware developers and selected performance analysts with documentation on counters and counter-based performance tools.

Hardware Performance Counters

Performance monitor counters (PMCs) offer an elegant solution to the counting problem. They have many *advantages*:

- (i) They can count any event of any program.
- (ii) They provide exact numbers.
- (iii) They can be used to investigate arbitrary parts of huge programs.
- (iv) They do not affect program speed or results, or the behavior of other programs.
- (v) They can be used in multi-tasking environments to measure the influence of other programs.
- (vi) They are cheap to use in resources and time.

They have *disadvantages* as well: (i) Only a limited number of events can be counted, typically two. When counting more events, the counts have to be multiplexed and they are not exact any more. (ii) Extra instructions have to be inserted, re-coding and re-compilation is necessary. (iii) Documentation is sometimes insufficient and difficult to obtain. (iv) Usage is sometimes difficult and tricky. (v) The use of the counters is different on any architecture.

Performance Relevant Events. All processors, which provide performance counters, count different types of events. There is no standard for implementing such counters, and many events are named differently. But one will find most of

the important events on any implementation.

Cycles: Cycles needed by the program to complete. This event type depends heavily on the underlying architecture. It can be used, for instance, to achieve high resolution timing.

Graduated Instructions, Graduated Loads, Graduated Stores: Instructions, loads and stores completed.

Issued Instructions, Issued Loads, Issued Stores: Instructions started, but not necessarily completed. The number of issued loads is usually far higher than the number of graduated ones, while issued and graduated stores are almost the same.

Primary Instruction Cache Misses: Cache misses of the primary instruction cache. High miss counts can indicate performance deteriorating loop structures.

Secondary Instruction Cache Misses: Cache misses of the secondary instruction cache. Usually, this count is very small and is therefore not a crucial performance indicator.

Primary Data Cache Misses: One of the most crucial performance factors is the number of primary data cache misses. It is usually far higher than the number of instruction cache misses.

Secondary Data Cache Misses: When program input exceeds a certain amount of memory, this event type will dominate even the primary data cache misses.

Graduated Floating-point Instructions: Once used as main performance indicator, the floating-point count is together with precise timing results still one of the most relevant indicators of the performance of a numerical program.

Mispredicted Branches: Number of mispredicted branches. Affects memory and cache accesses, and thus can be a reason for high memory latency.

Platform Independent Interfaces

Every processor architecture has its own set of instructions to access its performance monitor counters. But the basic PMC operations, i. e., selecting the type of event to be measured, starting and pausing the counting process, and reading the final value of the counters, are the same on every computer system.

This lead to the development of application programming interfaces (APIs) that unify the access behavior to the PMCs on different operating systems. Basically, the APIs provide the same library calls on every operating system—then

the called functions transparently access the underlying hardware with the processor's specific instruction set.

This makes the usage of PMCs easier, because the manufacturers interfaces to the counters were designed by engineers, whose main goal was to gain raw performance data without worrying of an easy-to-use interface.

Then, PMCs finally can be accessed the same way on every architecture. This makes program packages, which include PMC access, portable and their code gets more readable.

PAPI. The PAPI project is part of the PTools effort of the Parallel Tools Consortium of the computer science department of the University of Tennessee [90]. PAPI provides two interfaces to PMCs, a high-level and a low-level interface. Recently, a GUI tool was introduced to measure events for running programs without recompilation.

The high-level interface will meet most demands of the most common performance evaluation tasks, while providing a simple and easy-to-use interface. Only a small set of operations are defined, like the ability to start, stop and read specific events. A user with more sophisticated needs can rely on the low-level and fully programmable interface in order to access even seldomly used PMC functions.

From the software point of view, PAPI consists of two layers. The upper layer provides the machine independent entry functions—the application programming interface.

The lower layers exports an independent interface to hardware dependent functions and data structures. These functions access the substrate, which can be the operating system, a kernel extension or assembler instructions. Of course, this layer heavily relies on the underlying hardware and some functions are not available on every architecture. In this case, PAPI tries to emulate the missing functions.

A good example for such an emulation is PAPI's capability of *multiplexing* several hardware events. Multiplexing is a PMC feature common to some processors as the MIPS R10000, which allows for counting more events than the usual two. This is done by switching all events to be counted periodically and estimating the final event counts based on the partial counts and the total time elapsed. The counts are not exact any more, but in one single program run more than the usual two events can be measured.

On processors which do not provide this functionality, PAPI emulates it.

Another PAPI feature is its counter overflow control. This is usually not provided by hardware registers alone and can produce highly misleading data. PAPI implements 64 bit counters to provide a portable implementation of this advanced functionality. Another feature is asynchronous user notification when a counter value exceed some user defined values. This makes histogram generation easy, but even allows for advanced real-time functionality far beyond mere performance

evaluation.

PCL. The Performance Counter Library (PCL) is a common interface for accessing performance counters built into modern microprocessors in a portable way. PCL was developed at the Central Institute for Applied Mathematics (ZAM) at the Research Centre Juelich [13]. PCL supports query for functionality, start and stop of counters, and reading the current values of counters. Performance counting can be done in user mode, system mode, or user-or-system mode.

PCL supports nested calls to PCL functions to allow hierarchical performance measurements. However, nested calls must use exactly the same list of events. PCL functions are callable from C, C++, Fortran, and Java. Similar to PAPI, PCL defines a common set of events across platforms for accesses to the memory hierarchy, cycle and instruction counts, and the status of functional units then translates these into native events on a given platform where possible. PAPI additionally defines events related to SMP cache coherence protocols and to cycles stalled waiting for memory access.

Unlike PAPI, PCL does not support software multiplexing or user-defined overflow handling. The PCL API is very similar to the PAPI high-level API and consists of calls to start a list of counters and to read or stop the counter most recently started.

Appendix D

Short Vector Instruction Set

This appendix summarizes the intrinsic API provided for SSE 2 by the Intel C++ compiler.

The semantics of the intrinsics provided by the compilers is expressed using the C language, but the displayed code is pseudo code. Vector elements are denoted using braces {}.

D.1 The Intel Streaming SIMD Extensions 2

This section summarizes the relevant part of the SSE 2 API provided by the Intel C++ compiler and the Microsoft Visual C compiler. The GNU C compiler 3.x provides a *built-in function* interface with the same functionality. The SSE 2 instruction set is described in the IA-32 manuals [55, 56].

Short Vector Data Types

A new data type is introduced. The 128 bit data type `__m128d` maps a XMM register in two-way double-precision mode. Variables of type `__m128d` are 128 bit wide and 16 byte aligned. `__m128` is a vector of two double variables. Although these components cannot be accessed directly in code, in the pseudo code the components of variable `__m128d var` will be accessed by `var{0}` and `var{1}`.

Components of variables of type `__m128d` can only be accessed by using double variables. To ensure the correct alignment of double variables, the extended attribute `__declspec(align(16))` for qualifying storage-class information has to be used.

```
__declspec(align(16)) double[2] var = {1.0, 2.0};  
__m128d *pvar = &var;
```

Arithmetic Operations

The arithmetic operations are implemented using intrinsic functions. For each supported arithmetic instruction a corresponding function is defined. In the context of this thesis only vector addition, vector subtraction and pointwise vector multiplication is required.

The Pointwise Addition `_mm_add_pd`. The intrinsic function `_mm_add_pd` abstracts the addition of two XMM registers in two-way double-precision mode.

```

__m128d _mm_add_pd(__m128d a, __m128d b)
{
    __m128d c;
    c{0} = a{0} + b{0};
    c{1} = a{1} + b{1};
    return c;
}

```

The Pointwise Subtraction `_mm_sub_pd`. The intrinsic function `_mm_sub_pd` abstracts the subtraction of two XMM registers in two-way double-precision mode.

```

__m128d _mm_sub_pd(__m128d a, __m128d b)
{
    __m128d c;
    c{0} = a{0} - b{0};
    c{1} = a{1} - b{1};
    return c;
}

```

The Pointwise Multiplication `_mm_mul_pd`. The intrinsic function `_mm_mul_pd` abstracts the pointwise multiplication (Hadamard product) of two XMM registers in two-way double-precision mode.

```

__m128d _mm_mul_pd(__m128d a, __m128d b)
{
    __m128d c;
    c{0} = a{0} * b{0};
    c{1} = a{1} * b{1};
    return c;
}

```

Vector Reordering Operations

SSE 2 features three vector reordering operations: `_mm_shuffle_pd`, `_mm_unpacklo_pd`, and `_mm_unpackhi_pd`. They have to be used to build the required permutations. All three operations feature certain limitations and thus no general recombination can be done utilizing only a single permutation instruction. These intrinsics recombine elements from their two arguments of type `__m128d` into one result of type `__m128d`.

The Shuffle Operation `_mm_shuffle_pd`. This operation is the most general permutation supported by SSE 2. The first two elements of the result variable can be any element of the first parameter and the second two elements of the result variable can be any element of the second parameter. The choice is done according to the third parameter. The SSE 2 API provides the macro `_MM_SHUFFLE2` to encode these choices into `i`.

```

__m128d _mm_shuffle_pd(__m128d a, __m128d b, int i)
{
    __m128d c;
    c{0} = a{i & 1};
    c{1} = b{(i>>1) & 1};
    return c;
}

```

The Unpack Operation `_mm_unpacklo_pd`. This operation is required for easy unpacking of complex numbers and additionally appears in different contexts in SSE 2 codes. The first two elements of the result variable are the zeroth element of the input variables and the second half is filled by the first elements of the input variables.

```

__m128d _mm_unpacklo_pd(__m128d a, __m128d b)
{
    __m128d c;
    c{0} = a{0};
    c{1} = b{0};
    return c;
}

```

The Unpack Operation `_mm_unpackhi_pd`. This operation is required for easy unpacking of complex numbers and additionally appears in different contexts in SSE 2 codes. The first two elements of the result variable are the second element of the input variables and the second half is filled by the third elements of the input variables.

```

__m128d _mm_unpackhi_pd(__m128d a, __m128d b)
{
    __m128d c;
    c{0} = a{1};
    c{1} = b{1};
    return c;
}

```

Memory Access Functions

Although SSE 2 also features unaligned memory access (introducing a penalty), in this thesis only aligned memory access is used. SSE 2 features aligned access for 128 bit quantities (a full XMM register).

128 bit Memory Operations. The SSE API provides intrinsic functions for loading and storing XMM registers in two-way double-precision mode. The target memory location has to be 16 byte aligned. XMM loads and stores are implicitly inserted when `__m128d` variables which do not reside in registers are used.

```

__m128d _mm_load_pd(__m128d *p)
{
    __m128d c;
    c{0} = *p{0};
    c{1} = *p{1};
    return c;
}

void _mm_store_pd(__m128d *p, __m128d a)
{
    *p{0} = a{0};
    *p{1} = a{1};
}

```

Initialization Operations. The SSE 2 API provides intrinsic functions for initializing `__m128` variables. The intrinsic `_mm_setzero_pd` sets all components to zero while `_mm_set1_pd` sets all components to the same value and `_mm_set_pd` sets each component to a different value.

```

__m128d _mm_setzero_ps()
{
    __m128 c;
    c{0} = 0.0;
    c{1} = 0.0;
    return c;
}

__m128d _mm_set1_ps(double f)
{
    __m128 c;
    c{0} = f;
    c{1} = f;
    return c;
}

void _mm_set_pd(double f1, double f0)
{
    __m128 c;
    c{0} = f0;
    c{1} = f1;
    return c;
}

```

Appendix E

The Portable SIMD API

This appendix contains the definition of the portable SIMD API for SSE 2, both for the Intel C compiler and the Microsoft Visual C compiler.

This appendix contains the definition of the portable SIMD API for SSE 2—for the Intel C compiler and the Microsoft Visual C compiler—and for BlueGene/L's PowerPC 440 FP2 as provided by the IBM VisualAge 6.0 for BG/L XLC compiler.

E.1 Intel Streaming SIMD Extensions 2

```
/*
    map_sse2.h

    MAP runtime header for SSE 2
    Intel C++ compiler
*/

#include "emmintrin.h"
#include "rdtsc.h"

#pragma warning (disable:167)
#pragma warning (disable:144)

#define USE_XOR

/* -- Spiral prototypes ----- */

#define SCALAR_PTR(ptr) double *(ptr)

#define DECLARE_SCALAR_ARRAY_ALIGNED(var, i) \
    __declspec(align(16)) double var[i]

#define DECLARE_SPIRAL_FUNC(name, y, x) \
    void (name)(SCALAR_PTR(y), SCALAR_PTR(x))

#define DECLARE_SPIRAL_INIT_FUNC(name) \
    void (name)(void)

#define SPIRAL_FUNC(name, y, x) \
    void (name)(SCALAR_PTR(y), SCALAR_PTR(x))

#define SPIRAL_INIT_FUNC(name) \
```

```

    void (name)(void)

/* -- Data type ----- */

#define DECLARE_VEC(vec) __m128d vec

/* -- Constant handling ----- */

#define DECLARE_CONST(c,v) \
    static const __declspec(align(16)) double (c)[2]={v,v}
#define DECLARE_CONST_2(c, v0, v1) \
    static const __declspec(align(16)) double (c)[2]={v0, v1}

/* -- Arithmetic operations ----- */

#define VEC_ADD(c,a,b) \
    (c) = _mm_add_pd(a,b)
#define VEC_SUB(c,a,b) \
    (c) = _mm_sub_pd(a,b)
#define VEC_MUL(c,a,b) \
    (c) = _mm_mul_pd(a,b)
#define VEC_MUL_CONST(c,a,b) \
    (c) = _mm_mul_pd(_mm_load_pd(a), b)
#define VEC_MUL_MEM(c,a,b) \
    (c) = _mm_mul_pd(_mm_set1_pd(a), b)
#define VEC_MADD(d,a,b,c) \
    (d) = _mm_add_pd(_mm_mul_pd(a,b),c)
#define VEC_MSUB(d,a,b,c) \
    (d) = _mm_sub_pd(_mm_mul_pd(a,b),c)

#define VEC_MULCONST2(trg, cst, src) \
    (trg) = _mm_mul_pd(src, _mm_load_pd(cst))

/* -- Initialization operations ----- */

#ifdef USE_XOR
#define VEC_CHS_HI(trg, src) \
    (trg) = _mm_xor_pd(src, _mm_set_pd(-0.0,0.0))
#define VEC_CHS_LO(trg, src) \
    (trg) = _mm_xor_pd(src, _mm_set_pd(0.0,-0.0))
#else
#define VEC_CHS_HI(trg, src) \
    (trg) = _mm_mul_pd(src, _mm_set_pd(-1.0,1.0))
#define VEC_CHS_LO(trg, src) \
    (trg) = _mm_mul_sd(src, _mm_set_pd(1.0,-1.0))
#endif

/* -- Reordering operations ----- */

#define VEC_UNPACK_HI(trg, src1, src2) \
    (trg) = _mm_unpackhi_pd(src1, src2)
#define VEC_UNPACK_LO(trg, src1, src2) \

```

```

        (trg) = _mm_unpacklo_pd(src1, src2)
/* swap -> shuffle 01 */
#define VEC_SHUFFLE01(trg, src1, src2) \
        (trg) = _mm_shuffle_pd(src1, src2, _MM_SHUFFLE2(0, 1));

/* -- Load operations ----- */

#define VEC_LOAD_Q(src, trg) \
        (trg) = _mm_load_pd((double*)(src))

#define VEC_LOAD_D_LO(trg, src) \
        (trg) = _mm_loadh_pd (trg, (double*)(src));
#define VEC_LOAD_D_HI(trg, src) \
        (trg) = _mm_loadl_pd (trg, (double*)(src));

/* -- Store Operations ----- */

#define VEC_STORE_Q(src, trg) \
        _mm_store_pd((double*)(trg), src)

#define VEC_STORE_D_LO(trg, src) \
        _mm_storeh_pd((double*)trg, src)
#define VEC_STORE_D_HI(trg, src) \
        _mm_storel_pd((double*)trg, src)

```

E.2 BG/L Double FPU SIMD Extensions

```

/*      spl_bgl_XLC.h

        SPL SIMD runtime header for BG/L Double FPU
        IBM VisualAge 6.0 for BG/L XL C compiler

*/

#ifndef __SPL_BGL_XL\C_H
#define __SPL_BGL_XL\C_H

/* -- Data type ----- */

/*
    if SPLIT_LOAD is defined, lfd and lfscdx are used instead of
    lfpdx for INTERLEAVED and TRANSPOSED LOADS
*/

#define SPLIT_LOAD

#define DECLARE_VEC(vec) _Complex double vec

#define DECLARE_SCALAR(s) double s

```

```

#define SCALAR_PTR(ptr) double *(ptr)

#define DECLARE_SCALAR_ARRAY_ALIGNED(var, i) \
    double var[i];__alignx(16,var)

/* -- Spiral prototypes ----- */

#define DECLARE_SPIRAL_FUNC(name, y, x) \
    void (name)(SCALAR_PTR(y), SCALAR_PTR(x))

#define DECLARE_SPIRAL_INIT_FUNC(name) void (name)(void)

#define SPIRAL_FUNC(name, y, x) \
    void (name)(SCALAR_PTR(y), SCALAR_PTR(x))

#define SPIRAL_INIT_FUNC(name) void (name)(void)

/* -- Constant handling ----- */

#define DECLARE_CONST(c,v) \
    static const _Complex double __align(16) c = (v) + __I * (v)

#define DECLARE_CONST_2(c, v0, v1) \
    static const _Complex double __align(16) c = (v0) + __I * (v1)

/* -- Arithmetic operations ----- */

#define VEC_ADD(c,a,b) \
    (c) = __fpadd(a,b)

#define VEC_SUB(c,a,b) \
    (c) = __fpsub(a,b)

#define VEC_MUL(c,a,b) \
    (c) = __fpmul(a,b)

#define VEC_MADD(d,a,b,c) \
    (d) = __fpmadd(c,a,b)

#define VEC_MSUB(d,a,b,c) \
    (d) = __fpmsub(c,a,b)

#define VEC_UMINUS(b, a) (b) = -(a)

#define VEC_MUL_CONST(c,a,b)\ {\
    DECLARE_VEC(ar);\
    VEC_LOAD_Q(&(a), ar);\
    VEC_MUL(c,ar,b);\
}

#define VEC_MUL_MEM(c,a,b)\ {\
    DECLARE_VEC(ar);\

```



```

        ar = __cmplx((a), (a));\
        VEC_MUL(c,ar,b);\
    }

#define VEC_MULCONST2(c,a,b)\ {\
    DECLARE_VEC(ar);\
    VEC_LOAD_Q(&(a), ar);\
    VEC_MUL(c,ar,b);\
}

#define VEC_FMCA(d,a,b,c)\ {\
    DECLARE_VEC(ar);\
    VEC_LOAD_Q(&(a), ar);\
    VEC_MADD(d,ar,b,c);\
}

#define VEC_FMCS(d,a,b,c)\ {\
    DECLARE_VEC(ar);\
    VEC_LOAD_Q(&(a), ar);\
    VEC_MSUB(d,ar,b,c);\
}

/* -- Initialization operations ----- */

#define VEC_CHS_HI(trg, src) \
    (trg)=__cmplx(__creal(src),-__cimag(src))

#define VEC_CHS_LO(trg, src) \
    (trg)=__cmplx(-__creal(src),__cimag(src))

/* -- Reordering operations ----- */

#define VEC_UNPACK_HI(trg, src1, src2) \
    (trg)=__cmplx(__cimag(src1),__cimag(src2))

#define VEC_UNPACK_LO(trg, src1, src2) \
    (trg)=__cmplx(__creal(src1),__creal(src2))

#define VEC_SHUFFLE01(trg, src1, src2) \
    (trg)=__cmplx(__cimag(src1),__creal(src2))

#define VEC_SWAP(trg, src) \
    (trg) = __fxmr(src)

#define C99_TRANSPOSE(d1, d2, s1, s2)\ {\
    d1 = __cmplx (__creal(s1), __creal(s2));\
    d2 = __cmplx (__cimag(s1), __cimag(s2));\
}

/* -- Load operation ----- */

#define VEC_LOAD_Q(src, trg) \

```

```
(trg) = __lfpd((double *)(src))

/* -- Store Operation ----- */

#define VEC_STORE_Q(reg, mem) \
    __stfpd((double *)(mem), reg)

#endif
```

Appendix F

SPIRAL Example Code

This appendix displays a scalar and a short vector SIMD code example for the DFT_{16} which was obtained by utilizing the scalar SPIRAL version and the newly developed short vector SIMD extension for SPIRAL. In addition, the respective SPL program is displayed.

F.1 Scalar C Code

This section shows the scalar single-precision code of a DFT_{16} generated and adapted on an Intel Pentium 4.

Scalar C Program

```
void DFT_16(float *y, float *x)
{
    float f91;
    float f92;
    float f93;
    ...
    float f230;

    f91 = x[2] - x[30];
    f92 = x[3] - x[31];
    f93 = x[2] + x[30];
    f94 = x[3] + x[31];
    f95 = x[4] - x[28];
    f96 = x[5] - x[29];
    f97 = x[4] + x[28];
    f98 = x[5] + x[29];
    f99 = x[6] - x[26];
    f100 = x[7] - x[27];
    f101 = x[6] + x[26];
    f102 = x[7] + x[27];
    f103 = x[8] - x[24];
    f104 = x[9] - x[25];
    f105 = x[8] + x[24];
    f106 = x[9] + x[25];
    f107 = x[10] - x[22];
    f108 = x[11] - x[23];
    f109 = x[10] + x[22];
    f110 = x[11] + x[23];
    f111 = x[12] - x[20];
```

```
f112 = x[13] - x[21];
f113 = x[12] + x[20];
f114 = x[13] + x[21];
f115 = x[14] - x[18];
f116 = x[15] - x[19];
f117 = x[14] + x[18];
f118 = x[15] + x[19];
f119 = x[0] - x[16];
f120 = x[1] - x[17];
f121 = x[0] + x[16];
f122 = x[1] + x[17];
f123 = f93 - f117;
f124 = f94 - f118;
f125 = f93 + f117;
f126 = f94 + f118;
f127 = f97 - f113;
f128 = f98 - f114;
f129 = f97 + f113;
f130 = f98 + f114;
f131 = f101 - f109;
f132 = f102 - f110;
f133 = f101 + f109;
f134 = f102 + f110;
f135 = f121 - f105;
f136 = f122 - f106;
f137 = f121 + f105;
f138 = f122 + f106;
f139 = f125 - f133;
f140 = f126 - f134;
f141 = f125 + f133;
f142 = f126 + f134;
f143 = f137 - f129;
f144 = f138 - f130;
f145 = f137 + f129;
f146 = f138 + f130;
y[16] = f145 - f141;
y[17] = f146 - f142;
y[0] = f145 + f141;
y[1] = f146 + f142;
f151 = 0.7071067811865476 * f139;
f152 = 0.7071067811865476 * f140;
f153 = f135 - f151;
f154 = f136 - f152;
f155 = f135 + f151;
f156 = f136 + f152;
f157 = 0.7071067811865476 * f127;
f158 = 0.7071067811865476 * f128;
f159 = f119 - f157;
f160 = f120 - f158;
f161 = f119 + f157;
f162 = f120 + f158;
f163 = f123 + f131;
```

```
f164 = f124 + f132;
f165 = 1.3065629648763766 * f123;
f166 = 1.3065629648763766 * f124;
f167 = 0.9238795325112866 * f163;
f168 = 0.9238795325112866 * f164;
f169 = 0.5411961001461967 * f131;
f170 = 0.5411961001461967 * f132;
f171 = f165 - f167;
f172 = f166 - f168;
f173 = f167 - f169;
f174 = f168 - f170;
f175 = f161 - f173;
f176 = f162 - f174;
f177 = f161 + f173;
f178 = f162 + f174;
f179 = f159 - f171;
f180 = f160 - f172;
f181 = f159 + f171;
f182 = f160 + f172;
f183 = f91 + f115;
f184 = f92 + f116;
f185 = f91 - f115;
f186 = f92 - f116;
f187 = f99 + f107;
f188 = f100 + f108;
f189 = f107 - f99;
f190 = f108 - f100;
f191 = f185 - f189;
f192 = f186 - f190;
f193 = f185 + f189;
f194 = f186 + f190;
f195 = 0.7071067811865476 * f191;
f196 = 0.7071067811865476 * f192;
f197 = f183 - f187;
f198 = f184 - f188;
f199 = 1.3065629648763766 * f183;
f200 = 1.3065629648763766 * f184;
f201 = 0.9238795325112866 * f197;
f202 = 0.9238795325112866 * f198;
f203 = 0.5411961001461967 * f187;
f204 = 0.5411961001461967 * f188;
f205 = f199 - f201;
f206 = f200 - f202;
f207 = f201 + f203;
f208 = f202 + f204;
f209 = f95 - f111;
f210 = f96 - f112;
f211 = f95 + f111;
f212 = f96 + f112;
f213 = 0.7071067811865476 * f211;
f214 = 0.7071067811865476 * f212;
f215 = f213 - f103;
```

```

f216 = f214 - f104;
f217 = f213 + f103;
f218 = f214 + f104;
f219 = f205 - f217;
f220 = f206 - f218;
f221 = f205 + f217;
f222 = f206 + f218;
f223 = f195 - f209;
f224 = f196 - f210;
f225 = f195 + f209;
f226 = f196 + f210;
f227 = f215 - f207;
f228 = f216 - f208;
f229 = f215 + f207;
f230 = f216 + f208;
y[30] = f177 + f222;
y[31] = f178 - f221;
y[2] = f177 - f222;
y[3] = f178 + f221;
y[28] = f155 + f226;
y[29] = f156 - f225;
y[4] = f155 - f226;
y[5] = f156 + f225;
y[26] = f181 + f230;
y[27] = f182 - f229;
y[6] = f181 - f230;
y[7] = f182 + f229;
y[24] = f143 + f194;
y[25] = f144 - f193;
y[8] = f143 - f194;
y[9] = f144 + f193;
y[22] = f179 - f228;
y[23] = f180 + f227;
y[10] = f179 + f228;
y[11] = f180 - f227;
y[20] = f153 + f224;
y[21] = f154 - f223;
y[12] = f153 - f224;
y[13] = f154 + f223;
y[18] = f175 + f220;
y[19] = f176 - f219;
y[14] = f175 - f220;
y[15] = f176 + f219;
}

```

F.2 Short Vector Code

This section shows the two-way short vector SIMD code for a DFT_{16} vectorized by the MAP Vectorizer for an Intel Pentium 4. The respective C program using the portable SIMD API is displayed.

```

#include "map_sse2.h"

DECLARE_SPIRAL_FUNC(DFT_16,y,x);
DECLARE_SPIRAL_INIT_FUNC(init_DFT_16);

DECLARE_CONST(VEC_CONST1, -1.00000000000000000000000000000000,
                      +1.00000000000000000000000000000000);
DECLARE_CONST(VEC_CONST2, +0.70710678100000000000000000000000,
                      +0.70710678100000000000000000000000);
DECLARE_CONST(VEC_CONST3, +0.92387953200000000000000000000000,
                      +0.38268343200000000000000000000000);
DECLARE_CONST(VEC_CONST4, +0.38268343200000000000000000000000,
                      +0.92387953200000000000000000000000);

SPIRAL_FUNC(DFT_16,y,x){

    DECLARE_VEC(t1018);
    DECLARE_VEC(t1019);
    DECLARE_VEC(t1120);
    ...
    DECLARE_VEC(t1271);

    VEC_LOAD_Q(x+16, t1024);
    VEC_LOAD_Q(x+0, t1023);
    VEC_ADD(t1122, t1023, t1024);
    VEC_SUB(t1018, t1023, t1024);
    VEC_LOAD_Q(x+24, t1030);
    VEC_LOAD_Q(x+8, t1029);
    VEC_ADD(t1123, t1029, t1030);
    VEC_SUB(t1028, t1029, t1030);
    VEC_SHUFFLE01(t1019, t1028, t1028);
    VEC_CHS_LO(t1020, t1019);
    VEC_SUB(t1117, t1122, t1123);
    VEC_ADD(t1191, t1122, t1123);
    VEC_LOAD_Q(x+28, t1046);
    VEC_LOAD_Q(x+12, t1045);
    VEC_SUB(t1042, t1045, t1046);
    VEC_ADD(t1131, t1045, t1046);
    VEC_MULCONST2(t1034, VEC_CONST2, t1042);
    VEC_LOAD_Q(x+4, t1040);
    VEC_SHUFFLE01(t1038, t1040, t1040);
    VEC_LOAD_Q(x+20, t1041);
    VEC_SHUFFLE01(t1039, t1041, t1041);
    VEC_SUB(t1035, t1038, t1039);
    VEC_ADD(t1128, t1038, t1039);
    VEC_MULCONST2(t1033, VEC_CONST2, t1035);
    VEC_SHUFFLE01(t1193, t1128, t1128);
    VEC_SHUFFLE01(t1129, t1131, t1131);
    VEC_ADD(t1192, t1131, t1193);
    VEC_SUB(t1118, t1128, t1129);
    VEC_CHS_LO(t1119, t1118);
    VEC_UNPACK_LO(t1245, t1033, t1034);

```

```
VEC_UNPACK_HI(t1244, t1033, t1034);
VEC_SHUFFLE01(t1047, t1033, t1033);
VEC_CHS_LO(t1243, t1244);
VEC_SUB(t1031, t1245, t1243);
VEC_UNPACK_LO(t1260, t1047, t1034);
VEC_UNPACK_HI(t1259, t1047, t1034);
VEC_CHS_LO(t1258, t1259);
VEC_ADD(t1032, t1258, t1260);
VEC_LOAD_Q(x+22, t1093);
VEC_LOAD_Q(x+6, t1092);
VEC_ADD(t1146, t1092, t1093);
VEC_SUB(t1091, t1092, t1093);
VEC_SHUFFLE01(t1085, t1091, t1091);
VEC_LOAD_Q(x+30, t1097);
VEC_LOAD_Q(x+14, t1096);
VEC_SUB(t1086, t1096, t1097);
VEC_ADD(t1147, t1096, t1097);
VEC_CHS_LO(t1087, t1086);
VEC_SUB(t1143, t1146, t1147);
VEC_ADD(t1200, t1146, t1147);
VEC_MULCONST2(t1135, VEC_CONST2, t1143);
VEC_ADD(t1075, t1085, t1087);
VEC_SUB(t1168, t1085, t1087);
VEC_LOAD_Q(x+2, t1069);
VEC_SHUFFLE01(t1067, t1069, t1069);
VEC_LOAD_Q(x+18, t1070);
VEC_SHUFFLE01(t1068, t1070, t1070);
VEC_LOAD_Q(x+26, t1074);
VEC_LOAD_Q(x+10, t1073);
VEC_SUB(t1063, t1073, t1074);
VEC_ADD(t1142, t1073, t1074);
VEC_SHUFFLE01(t1140, t1142, t1142);
VEC_CHS_LO(t1064, t1063);
VEC_ADD(t1139, t1067, t1068);
VEC_SUB(t1062, t1067, t1068);
VEC_SUB(t1136, t1139, t1140);
VEC_ADD(t1197, t1139, t1140);
VEC_MULCONST2(t1134, VEC_CONST2, t1136);
VEC_ADD(t1052, t1062, t1064);
VEC_SUB(t1166, t1062, t1064);
VEC_MULCONST2(t1164, VEC_CONST3, t1166);
VEC_SHUFFLE01(t1172, t1166, t1166);
VEC_MULCONST2(t1170, VEC_CONST3, t1172);
VEC_MULCONST2(t1165, VEC_CONST4, t1168);
VEC_SHUFFLE01(t1173, t1168, t1168);
VEC_MULCONST2(t1171, VEC_CONST4, t1173);
VEC_UNPACK_LO(t1231, t1164, t1165);
VEC_UNPACK_HI(t1230, t1164, t1165);
VEC_ADD(t1162, t1230, t1231);
VEC_UNPACK_LO(t1242, t1170, t1171);
VEC_UNPACK_HI(t1240, t1170, t1171);
VEC_SUB(t1163, t1242, t1240);
```



```
VEC_SHUFFLE01(t1150, t1134, t1134);
VEC_UNPACK_HI(t1250, t1134, t1135);
VEC_CHS_LO(t1249, t1250);
VEC_UNPACK_LO(t1251, t1134, t1135);
VEC_SUB(t1132, t1251, t1249);
VEC_UNPACK_HI(t1265, t1150, t1135);
VEC_UNPACK_LO(t1266, t1150, t1135);
VEC_CHS_LO(t1264, t1265);
VEC_ADD(t1133, t1264, t1266);
VEC_MULCONST2(t1099, VEC_CONST4, t1075);
VEC_MULCONST2(t1051, VEC_CONST3, t1075);
VEC_SHUFFLE01(t1100, t1052, t1052);
VEC_MULCONST2(t1050, VEC_CONST4, t1052);
VEC_MULCONST2(t1098, VEC_CONST4, t1100);
VEC_UNPACK_LO(t1236, t1098, t1099);
VEC_UNPACK_HI(t1234, t1098, t1099);
VEC_SUB(t1049, t1236, t1234);
VEC_UNPACK_LO(t1229, t1050, t1051);
VEC_UNPACK_HI(t1228, t1050, t1051);
VEC_ADD(t1048, t1228, t1229);
VEC_SHUFFLE01(t1225, t1197, t1197);
VEC_SHUFFLE01(t1198, t1200, t1200);
VEC_SUB(t1187, t1197, t1198);
VEC_ADD(t1224, t1200, t1225);
VEC_CHS_LO(t1188, t1187);
VEC_SUB(t1186, t1191, t1192);
VEC_ADD(t1223, t1191, t1192);
VEC_ADD(t1176, t1186, t1188);
VEC_SUB(t1201, t1186, t1188);
VEC_STORE_Q(t1176, y+8);
VEC_STORE_Q(t1201, y+24);
VEC_ADD(t1220, t1223, t1224);
VEC_SUB(t1227, t1223, t1224);
VEC_STORE_Q(t1220, y+0);
VEC_STORE_Q(t1227, y+16);
VEC_ADD(t1159, t1018, t1020);
VEC_SUB(t1006, t1018, t1020);
VEC_UNPACK_LO(t1269, t1032, t1031);
VEC_UNPACK_HI(t1268, t1032, t1031);
VEC_CHS_LO(t1267, t1268);
VEC_ADD(t1160, t1267, t1269);
VEC_UNPACK_LO(t1233, t1163, t1162);
VEC_UNPACK_HI(t1237, t1162, t1163);
VEC_UNPACK_HI(t1232, t1163, t1162);
VEC_ADD(t1218, t1232, t1233);
VEC_UNPACK_LO(t1239, t1162, t1163);
VEC_SUB(t1156, t1239, t1237);
VEC_CHS_LO(t1157, t1156);
VEC_ADD(t1217, t1159, t1160);
VEC_SUB(t1155, t1159, t1160);
VEC_ADD(t1215, t1217, t1218);
VEC_SUB(t1219, t1217, t1218);
```

```

    VEC_STORE_Q(t1215, y+2);
    VEC_STORE_Q(t1219, y+18);
    VEC_ADD(t1153, t1155, t1157);
    VEC_SUB(t1174, t1155, t1157);
    VEC_STORE_Q(t1153, y+10);
    VEC_STORE_Q(t1174, y+26);
    VEC_SUB(t1105, t1117, t1119);
    VEC_ADD(t1211, t1117, t1119);
    VEC_UNPACK_LO(t1263, t1132, t1133);
    VEC_UNPACK_HI(t1271, t1133, t1132);
    VEC_CHS_LO(t1270, t1271);
    VEC_UNPACK_HI(t1262, t1132, t1133);
    VEC_UNPACK_LO(t1272, t1133, t1132);
    VEC_ADD(t1212, t1270, t1272);
    VEC_CHS_LO(t1261, t1262);
    VEC_ADD(t1106, t1261, t1263);
    VEC_CHS_LO(t1107, t1106);
    VEC_ADD(t1103, t1105, t1107);
    VEC_SUB(t1151, t1105, t1107);
    VEC_STORE_Q(t1103, y+12);
    VEC_STORE_Q(t1151, y+28);
    VEC_ADD(t1209, t1211, t1212);
    VEC_SUB(t1214, t1211, t1212);
    VEC_STORE_Q(t1209, y+4);
    VEC_STORE_Q(t1214, y+20);
    VEC_UNPACK_LO(t1257, t1031, t1032);
    VEC_UNPACK_HI(t1256, t1031, t1032);
    VEC_CHS_LO(t1255, t1256);
    VEC_ADD(t1007, t1255, t1257);
    VEC_CHS_LO(t1008, t1007);
    VEC_UNPACK_LO(t1248, t1048, t1049);
    VEC_UNPACK_HI(t1253, t1049, t1048);
    VEC_CHS_LO(t1252, t1253);
    VEC_UNPACK_HI(t1247, t1048, t1049);
    VEC_UNPACK_LO(t1254, t1049, t1048);
    VEC_SUB(t1206, t1254, t1252);
    VEC_CHS_LO(t1246, t1247);
    VEC_SUB(t1003, t1248, t1246);
    VEC_CHS_LO(t1004, t1003);
    VEC_SUB(t1002, t1006, t1008);
    VEC_ADD(t1205, t1006, t1008);
    VEC_ADD(t1000, t1002, t1004);
    VEC_SUB(t1101, t1002, t1004);
    VEC_STORE_Q(t1000, y+14);
    VEC_STORE_Q(t1101, y+30);
    VEC_ADD(t1203, t1205, t1206);
    VEC_SUB(t1208, t1205, t1206);
    VEC_STORE_Q(t1203, y+6);
    VEC_STORE_Q(t1208, y+22);
}
SPIRAL_INIT_FUNC(init_DFT_16){};

```

Appendix G

FFTW Example Code

This appendix displays a scalar and the respective short vector SIMD no-twiddle codelet of size four.

G.1 Scalar C Code

This section shows a standard FFTW no-twiddle codelet of size 4.

```
void fftw_no_twiddle_4 (const fftw_complex * input, fftw_complex
                        *output, int istride, int ostride)
{
    fftw_real tmp3;
    fftw_real tmp11;
    fftw_real tmp9;
    fftw_real tmp15;
    fftw_real tmp6;
    fftw_real tmp10;
    fftw_real tmp14;
    fftw_real tmp16;
    {
        fftw_real tmp1;
        fftw_real tmp2;
        fftw_real tmp7;
        fftw_real tmp8;
        tmp1 = c_re (input[0]);
        tmp2 = c_re (input[2 * istride]);
        tmp3 = (tmp1 + tmp2);
        tmp11 = (tmp1 - tmp2);
        tmp7 = c_im (input[0]);
        tmp8 = c_im (input[2 * istride]);
        tmp9 = (tmp7 - tmp8);
        tmp15 = (tmp7 + tmp8);
    }
    {
        fftw_real tmp4;
        fftw_real tmp5;
        fftw_real tmp12;
        fftw_real tmp13;
        tmp4 = c_re (input[istride]);
        tmp5 = c_re (input[3 * istride]);
        tmp6 = (tmp4 + tmp5);
        tmp10 = (tmp4 - tmp5);
        tmp12 = c_im (input[istride]);
```

```

    tmp13 = c_im (input[3 * istride]);
    tmp14 = (tmp12 - tmp13);
    tmp16 = (tmp12 + tmp13);
}
c_re (output[2 * ostride]) = (tmp3 - tmp6);
c_re (output[0]) = (tmp3 + tmp6);
c_im (output[ostride]) = (tmp9 - tmp10);
c_im (output[3 * ostride]) = (tmp10 + tmp9);
c_re (output[3 * ostride]) = (tmp11 - tmp14);
c_re (output[ostride]) = (tmp11 + tmp14);
c_im (output[2 * ostride]) = (tmp15 - tmp16);
c_im (output[0]) = (tmp15 + tmp16);
}

```

G.2 Short Vector Code

The following sections show a two-way SIMD vectorized FFTW no-twiddle codelet of size 4.

MAP provides the vectorized FFTW codelet either (i) via a source-to-assembly transformation utilizing the MAP backend (see G.2.1) and (ii) via a source-to-source transformation producing macros compliant with the portable SIMD API (see G.2.2).

G.2.1 FFTW Codelet Transformed with the MAP Vectorizer and Backend

To generate a no-twiddle codelet of size 4 using the MAP Vectorizer and the MAP Backend, `map` is called using

```
map -notwiddle 4 -output-2-asm
```

resulting in the following code:

```

.section .rodata
    .balign 64
    chs_lo: .double -1.0, +1.0
.text
    .balign 64
    .globl fftw_no_twiddle_4
    .type  fftw_no_twiddle_4, @function
fftw_no_twiddle_4:
    /* promise simd cell size = 16 */
    movl 12(%esp), %ecx
    movl 4(%esp), %eax
    sall $4, %ecx
    movapd (%eax), %xmm1
    leal (%ecx,%ecx,2), %edx
    movapd (%eax,%ecx), %xmm4

```

```

    movapd (%eax,%ecx,2), %xmm0
    movl 16(%esp), %ecx
    movapd (%eax,%edx), %xmm3
    movl 8(%esp), %edx
    /* simd data load/store barrier */
    movapd %xmm1, %xmm2
    sall $4, %ecx
    movapd %xmm4, %xmm5
    subpd %xmm0, %xmm1
    leal (%ecx,%ecx,2), %eax
    addpd %xmm0, %xmm2
    movapd %xmm1, %xmm7
    movapd %xmm2, %xmm6
    subpd %xmm3, %xmm5
    addpd %xmm3, %xmm4
    shufpd $9, %xmm5, %xmm5
    addpd %xmm4, %xmm6
    subpd %xmm4, %xmm2
    mulsd chs_lo, %xmm5
    movapd %xmm6, (%edx)
    movapd %xmm2, (%edx,%ecx,2)
    subpd %xmm5, %xmm7
    addpd %xmm5, %xmm1
    movapd %xmm7, (%edx,%ecx)
    movapd %xmm1, (%edx,%eax)
    ret

.section .rodata

    .globl fftw_no_twiddle_4_desc
    .balign 32
fftw_no_twiddle_4_desc:
    /* descr_str_ptr */      .long .LC0
    /* codelet_ptr */       .long fftw_no_twiddle_4
    /* n */                  .long 4
    /* cdir_int */          .long -1
    /* ctype_int */         .long 0
    /* signature */         .long 89
    /* num_twiddles */      .long 0
    /* twiddle_order */     .long 0
.LC0:    .string "fftw_no_twiddle_4"

```

G.2.2 FFTW Codelet Transformed with the MAP Vectorizer

To generate a no-twiddle codelet of size 4 using the MAP Vectorizer and the MAP Backend, map has to be called using

```
map -notwiddle 4 -output-2-c
```

resulting in the following C code (using the portable SIMD API described in Appendix E):

```

DECLARE_CONST(VECT_CONST1, -1.00000000000000000000000000000000,
                  +1.00000000000000000000000000000000);

void fftw_no_twiddle_4(const fftw_complex *inC,
                      fftw_complex *outC,
                      int istride, int ostride)
{
    const fftw_simd2 *input = (const fftw_simd2 *) inC;
    fftw_simd2 *output = (fftw_simd2 *) outC;

    DECLARE_VEC(t1001);
    DECLARE_VEC(t1002);
    DECLARE_VEC(t1003);
    ...
    DECLARE_VEC(t1014);

    VEC_LOAD_Q(input+istride*2, t1004);
    VEC_LOAD_Q(input+0, t1003);
    VEC_ADD(t1001, t1003, t1004);
    VEC_SUB(t1009, t1003, t1004);
    VEC_LOAD_Q(input+istride*3, t1006);
    VEC_LOAD_Q(input+istride, t1005);
    VEC_SUB(t1013, t1005, t1006);
    VEC_SHUFFLE01(t1010, t1013, t1013);
    VEC_ADD(t1002, t1005, t1006);
    VEC_CHS_LO(t1011, t1010);
    VEC_SUB(t1000, t1001, t1002);
    VEC_ADD(t1007, t1001, t1002);
    VEC_STORE_Q(t1000, output+ostride*2);
    VEC_STORE_Q(t1007, output+0);
    VEC_SUB(t1008, t1009, t1011);
    VEC_ADD(t1014, t1009, t1011);
    VEC_STORE_Q(t1008, output+ostride);
    VEC_STORE_Q(t1014, output+ostride*3);

    fftw_codelet_desc fftw_no_twiddle_4_desc = {
        "fftw_no_twiddle_4",
        (void (*)(void)) fftw_no_twiddle_4,
        4,
        FFTW_FORWARD,
        FFTW_NOTW,
        89,
        0,
        (const int *) 0,
    };
};

```

Appendix H

ATLAS Example Code

This appendix displays a scalar and the respective short vector SIMD ATLAS kernel of size 4×4 .

H.1 Scalar C Code

This section shows a standard ATLAS matmul kernel of size 4×4 .

```
void ATL_dJIK4x4x4NN4x4x4_a1_b1
(const int M, const int N, const int K, const double alpha,
 const double *A, const int lda,
 const double *B, const int ldb, const double beta,
 double *C, const int ldc)

/*
 * matmul with TA=N, TB=N, MB=4, NB=4, KB=4,
 * lda=4, ldb=4, ldc=4, mu=2, nu=2, ku=4
 */
{
  const double *stM = A + 4;
  const double *stN = B + 16;
  #define incAk 16
  const int incAm = 2 - 16, incAn = -4;
  #define incBk 4
  const int incBm = -4, incBn = 8;
  #define incCm 2
  #define incCn 4
  double *pC0=C;
  const double *pA0=A;
  const double *pB0=B;
  register int k;
  register double rA0, rA1;
  register double rB0, rB1;
  register double rC0_0, rC1_0, rC0_1, rC1_1;
  do /* N-loop */
  {
    do /* M-loop */
    {
      rC0_0 = *pC0;
      rC1_0 = pC0[1];
      rC0_1 = pC0[4];
      rC1_1 = pC0[5];
      rA0 = *pA0;
```

```

    rB0 = *pB0;
    rA1 = pA0[1];
    rB1 = pB0[4];
    rC0_0 += rA0 * rB0;
    rC1_0 += rA1 * rB0;
    rC0_1 += rA0 * rB1;
    rC1_1 += rA1 * rB1;
    rA0 = pA0[4];
    rB0 = pB0[1];
    rA1 = pA0[5];
    rB1 = pB0[5];
    rC0_0 += rA0 * rB0;
    rC1_0 += rA1 * rB0;
    rC0_1 += rA0 * rB1;
    rC1_1 += rA1 * rB1;
    rA0 = pA0[8];
    rB0 = pB0[2];
    rA1 = pA0[9];
    rB1 = pB0[6];
    rC0_0 += rA0 * rB0;
    rC1_0 += rA1 * rB0;
    rC0_1 += rA0 * rB1;
    rC1_1 += rA1 * rB1;
    rA0 = pA0[12];
    rB0 = pB0[3];
    rA1 = pA0[13];
    rB1 = pB0[7];
    rC0_0 += rA0 * rB0;
    rC1_0 += rA1 * rB0;
    rC0_1 += rA0 * rB1;
    rC1_1 += rA1 * rB1;
    pA0 += incAk;
    pB0 += incBk;
    *pC0 = rC0_0;
    pC0[1] = rC1_0;
    pC0[4] = rC0_1;
    pC0[5] = rC1_1;
    pC0 += incCm;
    pA0 += incAm;
    pB0 += incBm;
}
while(pA0 != stM);
pC0 += incCn;
pA0 += incAn;
pB0 += incBn;
}
while(pB0 != stN);
}

#ifdef incAm
    #undef incAm
#endif
#endif

```



```

        #undef incAn
    #endif #ifdef incAk
        #undef incAk
    #endif #ifdef incBm
        #undef incBm
    #endif #ifdef incBn
        #undef incBn
    #endif #ifdef incBk
        #undef incBk
    #endif #ifdef incCm
        #undef incCm
    #endif #ifdef incCn
        #undef incCn
    #endif #ifdef incCk
        #undef incCk
    #endif #ifdef Mb
        #undef Mb
    #endif #ifdef Nb
        #undef Nb
    #endif #ifdef Kb
        #undef Kb
    #endif

```

H.2 Short Vector Code

MAP provides the vectorized ATLAS kernel either (i) via a source-to-assembly transformation utilizing the MAP backend (see H.2.1) and (ii) via a source-to-source transformation producing macros compliant with the portable SIMD API (see G.2.2).

H.2.1 ATLAS Kernel Transformed with the MAP Vectorizer and Backend

To automatically generate an ATLAS vector matmul kernel of size 4×5 using the MAP Vectorizer and the MAP Backend, map has to be called using

```
map -input-atlas-c atl_dmm4x4.c -output-2-asm
```

resulting in the following code:

```

/*
 *
 * ATLAS Level 3 BLAS
 * unrolling: mu=2, nu=2, ku=4
 * leading dims : lda=4, ldb=4, ldc=4
 * inner block: mb=4, nb=4, kb=4
 *
 */

```

```

.text
    .balign 64
    .globl _ATL_dJIK4x4x4NN4x4x4_a1_b1
    .def _ATL_dJIK4x4x4NN4x4x4_a1_b1; .scl 2; .type 32; .endef
_ATL_dJIK4x4x4NN4x4x4_a1_b1:
    subl $16, %esp
    movl 40(%esp), %eax
    movl 48(%esp), %ecx
    movl %ebx, 12(%esp)
    movl %esi, 8(%esp)
    movl 64(%esp), %edx
    leal 128(%eax), %ebx
    leal 128(%ecx), %esi
    .p2align 4,,7
.NMLOOP:
    /* promise simd cell size = 16 */
    movapd (%eax), %xmm1
    movapd 32(%eax), %xmm3
    movapd (%ecx), %xmm0
    movapd 16(%eax), %xmm7
    movapd 32(%ecx), %xmm5
    movapd 48(%ecx), %xmm6
    movapd %xmm1, %xmm2
    movapd %xmm3, %xmm4
    mulpd %xmm0, %xmm2
    mulpd %xmm0, %xmm4
    movapd %xmm7, %xmm0
    mulpd %xmm5, %xmm1
    mulpd %xmm6, %xmm0
    mulpd %xmm3, %xmm5
    movapd 16(%ecx), %xmm3
    addpd %xmm0, %xmm1
    movapd 48(%eax), %xmm0
    addl $64, %eax
    /* simd data load/store barrier */
    addl $0, %ecx
    mulpd %xmm3, %xmm7
    mulpd %xmm0, %xmm6
    mulpd %xmm3, %xmm0
    addpd %xmm7, %xmm2
    movapd %xmm1, %xmm7
    addpd %xmm6, %xmm5
    movapd %xmm2, %xmm3
    addpd %xmm0, %xmm4
    unpckhpd %xmm5, %xmm1
    unpckhpd %xmm4, %xmm3
    unpcklpd %xmm5, %xmm7
    unpcklpd %xmm4, %xmm2
    addpd %xmm7, %xmm1
    addpd %xmm2, %xmm3
    movapd %xmm1, 32(%edx)

```

```

movapd %xmm3, (%edx)
addl $16, %edx
cmpl %ebx, %eax
jne .NMLLOOP
addl $32, %edx
addl $-128, %eax
addl $64, %ecx
cmpl %esi, %ecx
jne .NMLLOOP
movl 12(%esp), %ebx
movl 8(%esp), %esi
addl $16, %esp
ret

```

H.2.2 Atlas Kernel Transformed with the MAP Vectorizer

To automatically generate an ATLAS vector matmul kernel of size 4×4 using the MAP Vectorizer, `map` has to be called using

```
map -input-atlas-c atl_dmm4x4.c -output-2-c
```

resulting in the following C code (using the portable SIMD API described in Appendix E):

```

#include "map_sse2.h"

void ATL_dJIK4x4x4NN4x4x4_a1_b1
    (const int M, const int N, const int K, const double alpha,
     const double *A, const int lda, const double *B, const int ldb,
     const double beta, double *C, const int ldc)
{
    const double *stM = A + 4;
    const double *stN = B + 16;
    #define incAk 16
    const int incAm = 2 - 16, incAn = -4;
    #define incBk 4
    const int incBm = -4, incBn = 8;
    #define incCm 2
    #define incCn 4
    double *pC0=C;
    const double *pA0=A;
    const double *pB0=B;
    register int k;
    DECLARE_VEC(t1324);
    DECLARE_VEC(t1363);
    DECLARE_VEC(t1365);
    ...
    DECLARE_VEC(t1403);

    do /* N-loop */
    {

```

```

do /* M-loop */
{
    VEC_LOAD_Q(pB0+4, t1368);
    VEC_LOAD_Q(pA0+0, t1367);
    VEC_MUL(t1365, t1367, t1368);
    VEC_LOAD_Q(pB0+0, t1396);
    VEC_MUL(t1394, t1367, t1396);
    VEC_LOAD_Q(pA0+4, t1373);
    VEC_MUL(t1398, t1373, t1396);
    VEC_MUL(t1371, t1368, t1373);
    VEC_LOAD_Q(pB0+6, t1370);
    VEC_LOAD_Q(pA0+2, t1369);
    VEC_MUL(t1366, t1369, t1370);
    VEC_ADD(t1363, t1365, t1366);
    VEC_LOAD_Q(pB0+2, t1397);
    VEC_MUL(t1395, t1369, t1397);
    VEC_ADD(t1392, t1394, t1395);
    VEC_LOAD_Q(pA0+6, t1374);
    VEC_MUL(t1399, t1374, t1397);
    VEC_ADD(t1393, t1398, t1399);
    VEC_MUL(t1372, t1370, t1374);
    VEC_ADD(t1364, t1371, t1372);
    VEC_UNPACK_LO(t1401, t1363, t1364);
    VEC_UNPACK_HI(t1400, t1363, t1364);
    VEC_ADD(t1324, t1400, t1401);
    VEC_STORE_Q(t1324, pC0+4);
    VEC_UNPACK_LO(t1403, t1392, t1393);
    VEC_UNPACK_HI(t1402, t1392, t1393);
    VEC_ADD(t1375, t1402, t1403);
    VEC_STORE_Q(t1375, pC0+0);
    pA0 += incAk;
    pB0 += incBk;
    pC0 += incCm;
    pA0 += incAm;
    pB0 += incBm;
}
while(pA0 != stM);
pC0 += incCn;
pA0 += incAn;
pB0 += incBn;
}
while(pB0 != stN);
}

#ifdef incAm
    #undef incAm
#endif
#ifdef incAn
    #undef incAn
#endif
#ifdef incAk
    #undef incAk

```

```
#endif
#ifdef incBm
    #undef incBm
#endif
#ifdef incBn
    #undef incBn
#endif
#ifdef incBk
    #undef incBk
#endif
#ifdef incCm
    #undef incCm
#endif
#ifdef incCn
    #undef incCn
#endif
#ifdef incCk
    #undef incCk
#endif
#ifdef Mb
    #undef Mb
#endif
#ifdef Nb
    #undef Nb
#endif
#ifdef Kb
    #undef Kb
#endif
#endif
```

Table of Abbreviations

AGI	Address generation interlock
AGU	Address generation unit
API	Application programming interface
AEOS	Automatical empirical optimization of software
BLAS	Basic linear algebra subprograms
CISC	Complex instruction set computer
CPI	Cycles per instruction
CPU	Central processing unit
DAG	Directed acyclic graph
DCT	Discrete cosine transform
DFID	Depth-first iterative deepening
DFT	Discrete Fourier transform
DRAM	Dynamic random access memory
DSP	Digital signal processing, digital signal processor
FFT	Fast Fourier transform
FMA	Fused multiply-add
FPU	Floating-point unit
GUI	Graphical user interface
ISA	Instruction set architecture
LRU	Least recently used
PMC	Performance monitor counter
RAM	Random access memory
RAW	Read after write
ROM	Read-only memory
RISC	Reduced instruction set computer
SIMD	Single instruction, multiple data
SPL	Signal processing language
SRAM	Static random access memory
SSA	Static single assignment
SSE	Streaming SIMD extension
TLB	Transaction lookaside buffer
VLIW	Very long instruction word
WAR	Write after read
WAW	Write after write
WHT	Walsh-Hadamard transform

Bibliography

- [1] D. Aberdeen, J. Baxter: *Emerald: a fast matrix-matrix multiply using Intel's SSE instructions*. *Concurrency and Computation: Practice and Experience* 13 (2001)(2), pp. 103–119.
- [2] A. V. Aho, R. Sethi, J. D. Ullman: *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., 1986.
- [3] G. Almasi, R. Bellofatto, J. Brunheroto, C. Causcaval, J. G. Castanos, L. Ceze, P. Crumley, C. C. Erway, J. Gagliano, D. Lieber, X. Martorell, J. E. Moreira, A. Sanomiya, K. Strauss: *An Overview of the BlueGene/L System Software Organization*. In *Proceedings of the Euro-Par '03 Conference on Parallel and Distributed Computing LNCS 2790*.
- [4] AMD Corporation: *3DNow! Technology Manual*, 2000.
- [5] AMD Corporation: *3DNow! Instruction Porting Guide Application Note*, 2002.
- [6] AMD Corporation: *AMD Athlon Processor x86 Code Optimization Guide*, 2002.
- [7] AMD Corporation: *AMD Extensions to the 3DNow! and MMX Instruction Sets Manual*, 2002.
- [8] AMD Corporation: *AMD x86-64 Architecture Programmers Manual, Volume 1: Application Programming*. Order Number 24592, 2002.
- [9] AMD Corporation: *AMD x86-64 Architecture Programmers Manual, Volume 2: System Programming*. Order Number 24593, 2002.
- [10] A. W. Appel: *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.
- [11] Apple Computer: *vDSP Library*, 2001.
<http://developer.apple.com/>
- [12] M. Auer, R. Benedik, F. Franchetti, H. Karner, P. Kristöfel, R. Schachinger, A. Slateff, W. U. C. *Performance Evaluation of FFT Routines — Machine Independent Serial Programs*. AURORA Technical Report TR1999-05, Institute for Applied Mathematics and Numerical Analysis, Vienna University of Technology, 1999.

- [13] R. Berrendorf, H. Ziegler: PCL, 2002.
<http://www.fz-juelich.de/zam/PCL>
- [14] J. Bilmes, K. Asanović, C. Chin, J. Demmel: *Optimizing Matrix Multiply Using PHiPAC: a Portable, High-Performance, ANSI C Coding Methodology*. In *Proceedings of the 1997 International Conference on Supercomputing*, ACM Press, New York, pp. 340–347.
- [15] CodePlay Software Limited: VECTOR C, 2002.
www.codeplay.com
- [16] J. W. Cooley, J. W. Tukey: *An Algorithm for the Machine Calculation of Complex Fourier Series*. Math. Comp. 19 (1965), pp. 297–301.
- [17] R. Crandall, J. Klivington: *Supercomputer-Style FFT Library for the Apple G4*. Advanced Computation Group, Apple Computer, 2002.
- [18] D. DeVries: *A Vectorizing SUIF Compiler*, 1997.
citeseer.nj.nec.com/devries97vectorizing.html
- [19] Digital Equipment Corporation: PFM—*The 21064 Performance Counter Pseudo-Device*. DEC OSF/1 Manual pages, 1995.
- [20] J. J. Dongarra, I. S. Duff, D. C. Sorensen, H. A. van der Vorst: *Numerical Linear Algebra for High-Performance Computing*. SIAM Press, Philadelphia, 1998.
- [21] J. J. Dongarra, F. G. Gustavson, A. Karp: *Implementing Linear Algebra Algorithms for Dense Matrices on a Vector Pipeline Machine*. SIAM Review 26 (1984), pp. 91–112.
- [22] S. Egner, M. Püschel: *The AREP WWW Home Page*, 2000.
www.ece.cmu.edu/~smart/arep/arep.html
- [23] S. Egner, M. Püschel: *Symmetry-Based Matrix Factorization*. Journal of Symbolic Computation, to appear.
- [24] F. Franchetti: *Performance Portable Short Vector Transforms*. Ph.D. thesis, Institute for Applied Mathematics and Numerical Analysis, Vienna University of Technology, 2003.
- [25] F. Franchetti, S. Kral, J. Lorenz, M. Püschel, C. W. Ueberhuber: *Automatically Optimized FFT Codes for the BlueGene/L Supercomputer*. Submitted to 6th International Meeting on High Performance Computing for Computational Science.

- [26] F. Franchetti, S. Kral, J. Lorenz, C. W. Ueberhuber: *Domain Specific Compiler Techniques*. In *IEEE Proceedings Special Issue on Program Generation, Optimization, and Platform Adaptation*.
- [27] F. Franchetti, M. Püschel: *Automatic Generation of SIMD DSP Code*. AURORA Technical Report TR2001-17, Institute for Applied Mathematics and Numerical Analysis, Vienna University of Technology, 2001.
- [28] F. Franchetti, M. Püschel: *A SIMD Vectorizing Compiler for Digital Signal Processing Algorithms*. In *In Proceeding of the International Parallel and Distributed Processing Symposium (IPDPS'02)*, pp. 20–26.
- [29] F. Franchetti, M. Püschel: *Short Vector Code Generation and Adaptation for DSP Algorithms*. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing. Conference Proceedings (ICASSP '03)*, IEEE Comput. Soc. Press, Los Alamitos, USA, Vol. 2, pp. 537–540.
- [30] F. Franchetti, M. Püschel: *Short Vector Code Generation for the Discrete Fourier Transform*. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS'03)*, Comp. Society Press, Los Alamitos, USA.
- [31] F. Franchetti, M. Püschel: *Top Performance SIMD FFTs*. AURORA Technical Report TR2003-31, Institute for Applied Mathematics and Numerical Analysis, Vienna University of Technology, 2003.
- [32] F. Franchetti, C. W. Ueberhuber: *An Abstraction Layer for SIMD Extensions*. AURORA Technical Report TR2003-06, Institute for Applied Mathematics and Numerical Analysis, Vienna University of Technology, 2003.
- [33] M. Frigo: *A Fast Fourier Transform Compiler*. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, ACM Press, New York, pp. 169–180.
- [34] M. Frigo, S. G. Johnson: *The Fastest Fourier Transform in the West*. Tech. Report MIT-LCS-TR-728, MIT Laboratory for Computer Science, Cambridge, 1997.
- [35] M. Frigo, S. G. Johnson: *FFTW: An Adaptive Software Architecture for the FFT*. In *ICASSP 98*, Vol. 3, pp. 1381–1384.
- [36] M. Frigo, S. G. Johnson: *FFTW: An adaptive software architecture for the FFT*. In *Proceedings of the ACM SIGPLAN '99 conference on Programming language design and implementation*, pp. 169–180.

- [37] M. Frigo, S. G. Johnson: *The Design and Implementation of FFTW*. IEEE Special Issue on Program Generation, Optimization, and Platform Adaptation (2003).
- [38] M. Frigo, S. Kral: *The Advanced FFT Program Generator GENFFT*. AURORA Technical Report TR2001-03, Institute for Applied Mathematics and Numerical Analysis, Vienna University of Technology, 2001.
- [39] M. Galles, E. Williams: *Performance Optimizations, Implementation, and Verification of the SGI Challenge Multiprocessor*. In Proceedings of the 27th Annual Hawaii International Conference on System Sciences, 1994.
- [40] W.N. Gansterer, C.W. Ueberhuber: *Hochleistungsrechnen mit HPF*. Springer-Verlag, Berlin Heidelberg New York Tokyo, 2001.
- [41] K.S. Gatlin, L. Carter: *Faster FFTs via Architecture-Cognizance*. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'00)*, IEEE Comp. Society Press, Los Alamitos, CA, pp. 249–260.
- [42] G.H. Golub, C.F. Van Loan: *Matrix Computations*, 2nd edn. Johns Hopkins University Press, Baltimore, 1989.
- [43] T.G. Group: *The GAP (Groups, Algorithms and Programming) WWW Home Page*. www-gap.dcs.st-and.ac.uk/~gap/.
- [44] J. Guo, M.J. Garzarán, D. Padua: *The Power of Belady's Algorithm in Register Allocation for Long Basic Blocks*. In *Proceedings of the LCPC 2003*.
- [45] S.K.S. Gupta, Z. Li, J.H. Reif: *Synthesizing Efficient Out-of-Core Programs for Block Recursive Algorithms using Block-Cyclic Data Distributions*. Technical Report TR-96-04, Dept. of Computer Science, Duke University, Durham, USA, 1996.
- [46] G. Haentjens: *An Investigation of Cooley-Tukey Decompositions for the FFT*. Master thesis, Electrical and Computer Engineering Department, Carnegie Mellon University, Pittsburgh, PA, 2000.
- [47] H. Hlavacs, C.W. Ueberhuber: *High-Performance Computers – Hardware, Software and Performance Simulation*. SCS-Europe, Ghent, 2002.
- [48] D. Hunt: *Advanced Performance Features of the 64-bit PA8000*. COMP-CON'95, 1995.
- [49] Intel Corporation: *AP-808 Split Radix Fast Fourier Transform Using Streaming SIMD Extensions*, 1999.

- [50] Intel Corporation: *AP-833 Data Alignment and Programming Issues for the Streaming SIMD Extensions with the Intel C/C++ Compiler*, 1999.
- [51] Intel Corporation: *AP-833 Data Alignment and Programming Issues for the Streaming SIMD Extensions with the Intel C/C++ Compiler*, 1999.
- [52] Intel Corporation: *AP-931 Streaming SIMD Extensions—LU Decomposition*, 1999.
- [53] Intel Corporation: *Intel Architecture Optimization—Reference Manual*, 1999.
- [54] Intel Corporation: *Intel Architecture Software Developer's Manual*, 1999.
- [55] Intel Corporation: *Intel Architecture Software Developer's Manual—Volume 1: Basic Architecture*, 1999.
- [56] Intel Corporation: *Intel Architecture Software Developer's Manual—Volume 2: Instruction Set Reference*, 1999.
- [57] Intel Corporation: *Intel Architecture Software Developer's Manual—Volume 3: System Programming*, 1999.
- [58] Intel Corporation: *Desktop Performance and Optimization for Intel Pentium 4 Processor*, 2002.
- [59] Intel Corporation: *Intel Itanium Architecture Software Developer's Manual Vol. 1 rev. 2.1: Application Architecture*, 2002.
- [60] Intel Corporation: *Intel Itanium Architecture Software Developer's Manual Vol. 2 rev. 2.1: System Architecture* (2002).
- [61] Intel Corporation: *Intel Itanium Architecture Software Developer's Manual Vol. 3 rev. 2.1: Instruction Set Reference*, 2002.
- [62] Intel Corporation: *Math Kernel Library*, 2002.
<http://www.intel.com/software/products/mkl>
- [63] Intel Corporation: *Prescott New Instructions Software Developer's Guide*. App. Note, 2004.
- [64] Intel Corporation: *The Microarchitecture of the Intel Pentium 4 Processor on 90nm Technology*. Intel Technology Journal 8 (2004)(1).
- [65] J. Johnson, R. W. Johnson, D. Rodriguez, R. Tolimieri: *A Methodology for Designing, Modifying, and Implementing Fourier Transform Algorithms on Various Architectures*. Circuits Systems Signal Process 9 (1990), pp. 449–500.

- [66] J. R. Johnson, R. W. Johnson, C. P. Marshall, J. E. Mertz, D. Pryor, J. H. Weckel: *Data Flow, the FFT, and the CRAY T3E*. In *Proc. of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*.
- [67] R. W. Johnson, C. H. Huang, J. Johnson: *Multilinear Algebra and Parallel Programming*. *J. Supercomputing* 5 (1991), pp. 189–217.
- [68] S. Joshi, P. Dubey: *Radix-4 FFT Implementation Using SIMD Multi-Media Instructions*. In *Proceedings of the ICASSP 99*, pp. 2131–2135.
- [69] N. P. Jouppi, D. W. Wall: *Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines*. WRL Research Report 7, Digital Western Research Laboratory Palo Alto, California, 1989.
- [70] S. Kral, F. Franchetti, J. Lorenz, C. Ueberhuber: *Practical Assessment of SIMD Vectorization*. Technical Report AURORA TR2003-12, Institute for Applied Mathematics and Numerical Analysis, Vienna University of Technology, 2003.
- [71] S. Kral, F. Franchetti, J. Lorenz, C. W. Ueberhuber: *Backend Optimization for Straight Line Code*. AURORA Technical Report TR2003-11, Institute for Applied Mathematics and Numerical Analysis, Vienna University of Technology, 2003.
- [72] S. Kral, F. Franchetti, J. Lorenz, C. W. Ueberhuber: *Compiler Technology for the SIMD Vectorization of Straight Line Code*. AURORA Technical Report TR2003-07, Institute for Applied Mathematics and Numerical Analysis, Vienna University of Technology, 2003.
- [73] S. Kral, F. Franchetti, J. Lorenz, C. W. Ueberhuber: *Optimization Techniques for SIMD Vectorized Straight Line Code*. AURORA Technical Report TR2003-10, Institute for Applied Mathematics and Numerical Analysis, Vienna University of Technology, 2003.
- [74] S. Kral, F. Franchetti, J. Lorenz, C. W. Ueberhuber: *SIMD Vectorization of Straight Line FFT Code*. In *Proceedings of the Euro-Par '03 Conference on Parallel and Distributed Computing LNCS 2790*, pp. 251–260.
- [75] S. Kral, F. Franchetti, J. Lorenz, C. W. Ueberhuber: *SIMD Vectorization Techniques for Straight Line Code*. AURORA Technical Report TR2003-02, Institute for Applied Mathematics and Numerical Analysis, Vienna University of Technology, 2003.
- [76] S. Kral, F. Franchetti, J. Lorenz, C. W. Ueberhuber, P. Wurziinger: *FFT Compiler Techniques*. 13th International Conference on Compiler Construction.

- [77] A. Krall, S. Lelait: *Compilation Techniques for Multimedia Processors*. International Journal of Parallel Programming 28 (2000)(4), pp. 347–361.
citeseer.ist.psu.edu/krall00compilation.html
- [78] S. Lamson: SCIPORT, 1995.
<http://www.netlib.org/scilib/>
- [79] S. Larsen, S. Amarasinghe: *Exploiting Superword Level Parallelism with Multimedia Instruction Sets*. ACM SIGPLAN Notices 35 (2000)(5), pp. 145–156.
- [80] R. Leupers, S. Bashford: *Graph-based code selection techniques for embedded processors*. ACM Transactions on Design Automation of Electronic Systems. 5 (2000)(4), pp. 794–814.
citeseer.nj.nec.com/leupers00graph.html
- [81] M. Lorenz, L. Wehmeyer, T. Drger: *Energy aware Compilation for DSPs with SIMD instructions*, 2003.
citeseer.nj.nec.com/lorenz02energy.html
- [82] T. Mathisen: *Pentium Secrets*. Byte 7 (1994), pp. 191–192.
- [83] MIPS Technologies Inc. : *R10000 Microprocessor Technical Brief*, 1994.
- [84] MIPS Technologies Inc. : *Definition of MIPS R10000 Performance Counter*, 1997.
- [85] Motorola Corporation: *AltiVec Technology Programming Environments Manual*, 1998.
- [86] Motorola Corporation: *AltiVec Technology Programming Interface Manual*, 1998.
- [87] J.M.F. Moura, J. Johnson, R. Johnson, D. Padua, V. Prasanna, M. Püschel, M.M. Veloso: *SPIRAL: Automatic Library Generation and Platform-Adaptation for DSP Algorithms*, 1998.
<http://www.ece.cmu.edu/~spiral>.
- [88] J.M.F. Moura, J. Johnson, R.W. Johnson, D. Padua, V. Prasanna, M. Püschel, M.M. Veloso: *SPIRAL: Portable Library of Optimized Signal Processing Algorithms*, 1998. <http://www.ece.cmu.edu/spiral>.
- [89] J.M.F. Moura, J. Johnson, D. Padua, M. Püschel, M. Veloso: *SPIRAL. Special Issue on Program Generation, Optimization, and Platform Adaptation* (2003).
- [90] P. Mucci, S. Browne, G. Ho, C. Deane: *PAPI*, 2002.
<http://icl.cs.utk.edu/projects/papi>

- [91] S. S. Muchnick: *Advanced Compiler Design and Implementation*. Morgan Kaufman Publishers, San Francisco, 1997.
- [92] I. Nicholson: LIBSIMD, 2002.
<http://libsimd.sourceforge.net/>
- [93] A. Norton, A. J. Silberger: *Parallelization and Performance Analysis of the Cooley-Tukey FFT Algorithm for Shared-Memory Architectures*. IEEE Trans. Comput. 36 (1987), pp. 581–591.
- [94] M. C. Pease: *An Adaptation of the Fast Fourier Transform for Parallel Processing*. Journal of the ACM 15 (1968), pp. 252–264.
- [95] N. P. Pitsianis: *The Kronecker Product in Optimization and Fast Transform Generation*. Phd thesis, Department of Computer Science, Cornell University, 1997.
- [96] M. Püschel: *Decomposing Monomial Representations of Solvable Groups*. Symbolic Computation 34 (2002)(6), pp. 561–596.
- [97] M. Püschel, B. Singer, M. Veloso, J. M. F. Moura: *Fast Automatic Generation of DSP Algorithms*. In *Proceedings of the ICCS 2001*, Springer, LNCS 2073, pp. 97–106.
- [98] M. Püschel, B. Singer, J. Xiong, J. Moura, J. Johnson, D. Padua, M. Veloso, R. Johnson: *SPIRAL: A Generator for Platform-Adapted Libraries of Signal Processing Algorithms*. Journal of High Performance Computing and Applications, *submitted*, 2002.
- [99] M. Püschel, B. Singer, J. Xiong, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, R. W. Johnson: *SPIRAL: A Generator for Platform-Adapted Libraries of Signal Processing Algorithms*. To appear in Journal of High Performance Computing and Applications.
- [100] K. R. Rao, J. J. Hwang: *Techniques & Standards for Image, Video and Audio Coding*. Prentice Hall PTR, 1996.
- [101] P. Rodriguez: *A Radix-2 FFT Algorithm for Modern Single Instruction Multiple Data (SIMD) Architectures*. In *Proceedings of the ICASSP 2002*.
- [102] See homepage for details: *ATLAS homepage*. [Http://math-atlas.sourceforge.net/](http://math-atlas.sourceforge.net/).
- [103] B. Singer, M. Veloso: *Stochastic Search for Signal Processing Algorithm Optimization*. In *Proceedings of the Supercomputing 2001*.

- [104] N. Sreeraman, R. Govindarajan: *A Vectorizing Compiler for Multimedia Extensions*. International Journal of Parallel Programming 28 (2000), pp. 363–400.
- [105] Y.N. Srikant, P. Shankar: *The Compiler Design Handbook*. CRC Press LLC, Boca Raton London New York Washington D.C., 2003.
- [106] C. Temperton: *Fast Mixed-Radix Real Fourier Transforms*. J. Comput. Phys. 52 (1983), pp. 340–350.
- [107] The GAP Group: *GAP—Groups, Algorithms, and Programming, Version 4.2*, 2000.
www-gap.dcs.st-and.ac.uk/~gap
- [108] C.W. Ueberhuber: *Numerical Computation*. Springer-Verlag, Berlin Heidelberg New York Tokyo, 1997.
- [109] C. Van Loan: *Computational Frameworks for the Fast Fourier Transform*, Vol. 10 of *Frontiers in Applied Mathematics*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, 1992.
- [110] Z. Wang: *Fast Algorithms for the Discrete W Transform and for the Discrete Fourier Transform*. IEEE Trans. on Acoustics, Speech, and Signal Processing ASSP-32 (1984)(4), pp. 803–816.
- [111] E. H. Welbon, C. C. Chan-Nui, D. J. Shippy, D. A. Hicks: *POWER 2 Performance Monitor. POWER PC and POWER 2: Technical Aspects of the New IBM RISC System/6000*. IBM Corporation SA23-2737 (1994), pp. 55–63.
- [112] R. C. Whaley, J. Dongarra: *Automatically Tuned Linear Algebra Software*. Technical Report UT-CS-97-366, University of Tennessee, 1997. URL: <http://www.netlib.org/lapack/lawns/lawn131.ps>.
- [113] R. C. Whaley, J. Dongarra: *Automatically Tuned Linear Algebra Software*. In *SuperComputing 1998: High Performance Networking and Computing*. CD-ROM Proceedings. **Winner, best paper in the systems category**. URL: http://www.cs.utk.edu/~rwhaley/papers/atlas_sc98.ps.
- [114] R. C. Whaley, J. Dongarra: *Automatically Tuned Linear Algebra Software*. In *Ninth SIAM Conference on Parallel Processing for Scientific Computing*. CD-ROM Proceedings.
- [115] R. C. Whaley, A. Petitet, J. J. Dongarra: *Automated Empirical Optimization of Software and the ATLAS Project*. Parallel Computing 27 (2001)(1–2), pp. 3–35. Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000 (www.netlib.org/lapack/lawns/lawn147.ps).

- [116] J. Xiong, J. Johnson, R. Johnson, D. Padua: *SPL: A Language and Compiler for DSP Algorithms*. In *Proceedings of the PLDI 2001*, pp. 298–308.
- [117] M. Zagha, B. Larson, S. Turner, M. Itzkowitz: *Performance Analysis Using the MIPS R10000 Performance Counters*. In *Proceedings of the Supercomputing'96*, IEEE Computer Society Press, 1996.

CURRICULUM VITAE

Name: Juergen Lorenz

Title: Dipl.-Ing.

Date and Place of Birth: 8 September 1977, Neunkirchen, Lower Austria

Nationality: Austria

Home Address: Talgasse 19, A-2620 Neunkirchen, Austria

Affiliation

Institute for Analysis and Scientific Computing
Vienna University of Technology (TU Wien)
Wiedner Hauptstrasse 8-10/101, A-1040 Vienna
Phone: +43 1 58801 11524
Fax: +43 1 58801 11599
E-mail: juergen.lorenz@aurora.anum.tuwien.ac.at

Education

1995	High School Diploma (<i>Matura</i>)
1995 – 2002	Studies in Computer Sciences at the Vienna University of Technology
2002	Dipl.-Ing. (Computer Science)
2002 – 2004	PhD studies

Employment

1998	Summer internship with JENOPTIK Systemhaus Wr. Neustadt
2002 –	Research Assistant at the Institute for Analysis and Scientific Computing (TU Wien), funded by the SFB AURORA

Selected Project Experience

2002 –	Participation in the SFB AURORA
--------	---------------------------------

Publications

1. F. Franchetti, S. Kral, J. Lorenz, M. Püschel, C.W. Ueberhuber: *Automatically Optimized FFT Codes for the BlueGene/L Supercomputer*. Submitted to 6th International Meeting on High Performance Computing for Computational Science.
2. F. Franchetti, S. Kral, J. Lorenz, C.W. Ueberhuber: *Domain Specific Compiler Techniques*. In *IEEE Proceedings Special Issue on Program Generation, Optimization, and Platform Adaptation*.
3. F. Franchetti, J. Lorenz, C.W. Ueberhuber: *Low Communication FFTs*. Tech Report AURORA TR2002-27, Institute for Applied Mathematics and Numerical Analysis, Vienna University of Technology, 2002.
4. F. Franchetti, J. Lorenz, C.W. Ueberhuber: *Latency Hiding Parallel FFTs*. AURORA Technical Report TR2002-30, Institute for Applied Mathematics and Numerical Analysis, Vienna University of Technology, 2002.
5. S. Kral, F. Franchetti, J. Lorenz, C.W. Ueberhuber: *Backend Optimization for Straight Line Code*. AURORA Technical Report TR2003-11, Institute for Applied Mathematics and Numerical Analysis, Vienna University of Technology, 2003.
6. S. Kral, F. Franchetti, J. Lorenz, C.W. Ueberhuber: *Compiler Technology for the SIMD Vectorization of Straight Line Code*. AURORA Technical Report TR2003-07, Institute for Applied Mathematics and Numerical Analysis, Vienna University of Technology, 2003.
7. S. Kral, F. Franchetti, J. Lorenz, C.W. Ueberhuber: *SIMD Vectorization of Straight Line FFT Code*. In *Proceedings of the Euro-Par '03 Conference on Parallel and Distributed Computing LNCS 2790*, pp. 251-260.
8. S. Kral, F. Franchetti, J. Lorenz, C.W. Ueberhuber: *SIMD Vectorization Techniques for Straight Line Code*. AURORA Technical Report TR2003-02, Institute for Applied Mathematics and Numerical Analysis, Vienna University of Technology, 2003.
9. S. Kral, F. Franchetti, J. Lorenz, C.W. Ueberhuber, P. Wurzinger: *FFT Compiler Techniques*. 13th International Conference on Compiler Construction, to appear.
10. F. Franchetti, S. Kral, J. Lorenz, C.W. Ueberhuber: *Domain Specific Compiler Techniques*. In *IEEE Proceedings, Special Issue on Program Generation, Optimization, and Platform Adaptation*, to appear.