TECHNISCHE
UNIVERSITÄT
WIEN

VIENNA
UNIVERSITY OF
TECHNOLOGY

Dissertation

Performance Portable Short Vector Transforms

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
E115 – Institut für Angewandte und Numerische Mathematik

eingereicht an der Technischen Universität Wien
Fakultät für Technische Naturwissenschaften und Informatik

von

Dipl.-Ing. Franz Franchetti
Matrikelnummer 9525993
Hartiggasse 3/602
2700 Wiener Neustadt

Wien, am 7. Jänner 2003

Kurzfassung

In dieser Dissertation wird eine mathematische Methode entwickelt, die automatische Leistungsoptimierung von Programmen zur Berechnung von diskreten linearen Transformationen für Prozessoren mit Multimedia-Vektorerweiterungen (short vector SIMD extensions) ermöglicht, wobei besonderes Gewicht auf die diskrete Fourier-Transformation (DFT) gelegt wird. Die neuentwickelte Methode basiert auf dem Kronecker-Produkt-Formalismus, der erweitert wurde, um die spezifischen Eigenschaften von Multimedia-Vektorerweiterungen abzubilden. Es wurde auch eine speziell angepaßte Cooley-Tukey-FFT-Variante[1] entwickelt, die sowohl für Vektorlängen der Form $N = 2^k$ als auch für allgemeinere Problemgrößen anwendbar ist.

Die neuentwickelte Methode wurde als Erweiterung für Spiral[2] und Fftw[3], die derzeitigen Top-Systeme im Bereich der automatischen Leistungsoptimierung für diskrete lineare Transformationen, getestet. Sie erlaubt es, extrem schnelle Programme zur Berechnung der DFT zu erzeugen, welche die derzeit schnellsten Programme zur Berechnung der DFT auf Intel-Prozessoren mit den Multimedia-Vektorerweiterungen "Streaming SIMD Extensions" (SSE und SSE 2) sind. Sie sind schneller als die entsprechenden Programme aus der manuell optimierten Intel-Softwarebibliothek MKL (Math Kernel Library). Zusätzlich wurden die bisher ersten und einzigen automatisch leistungsoptimierten Programme zur Berechnung der Walsh-Hadamard-Transformation und für zwei-dimensionale Kosinus-Transformationen erzeugt.

Wichtige Resultate dieser Dissertation sind: (*i*) Die Vektorisierung von diskreten linearen Transformationen für Multimedia-Vektorerweiterungen erfordert nichttriviale strukturelle Änderungen, wenn automatische Leistungsoptimierung durchgeführt werden soll. (*ii*) Leistungsportabilität über verschiedene Plattformen und Prozessorgenerationen ist besonders schwierig bei Prozessoren mit Multimedia-Vektorerweiterungen zu erreichen. (*iii*) Aufgrund der Komplexität der Algorithmen für diskrete lineare Transformationen können vektorisierende Compiler keine Programme erzeugen, die eine zufriedenstellende Gleitpunktleistung aufweisen. (*iv*) Aufgrund anderer Designziele können Software-Bibliotheken für klassische Vektorcomputer auf Prozessoren mit Multimedia-Vektorerweiterungen nicht effizient eingesetzt werden.

Die in dieser Dissertation entwickelte Methode basiert auf dem Kronecker-Produkt-Formalismus, angewendet auf diskrete lineare Transformationen (Van Loan [90], Moura et al. [72]). Die numerischen Experimente wurden mit neuentwickelten Erweiterungen zu Fftw (Frigo und Johnson [33]) und Spiral (Moura et al. [72]) durchgeführt. Resultate dieser Dissertation wurden in Franchetti et al. [24, 25, 27, 28, 29, 30] veröffentlicht.

---

[1] FFT ist die Abkürzung von "fast Fourier transform" (Schnelle Fourier-Transformation)

[2] Spiral ist die Abkürzung von "signal processing algorithms implementation research for adaptive libraries" (Implementierungsforschung an Signalverarbeitungsalgorithmen für adaptive Software)

[3] Fftw ist die Abkürzung von "Fastest Fourier Transform in the West" (Schnellste Fourier-Transformation des Westens)

Summary

This thesis provides a mathematical framework for automatic performance tuning of discrete linear transform codes targeted at computers with processors featuring short vector SIMD extensions. Strong emphasis is placed on the discrete Fourier transform (DFT). The mathematical framework of this thesis is based on the Kronecker product approach to describe linear transforms. This approach has been extended to express the specific features of all current short vector SIMD extensions. A special short vector Cooley-Tukey FFT[4] is introduced.

The methods developed in this thesis are applicable as extensions to Spiral[5] and Fftw[6], the current state-of-the-art systems using automatic performance tuning in the field of discrete linear transforms. Application of the new method leads to extremely fast implementations of DFTs on most current short vector SIMD architectures for both non-powers and powers of two.

The automatically generated and optimized codes are currently the fastest codes for Intel machines featuring streaming SIMD extensions (SSE and SSE 2). These codes are faster than Intel's hand tuned math library (MKL) and of course faster than all other implementations freely available for these machines. The codes of this thesis are the first n-way short vector FFTs for both non-powers and powers of two. Moreover, the thesis presents the first automatically performance tuned short vector SIMD implementations for other transforms like Walsh-Hadamard transforms or two-dimensional cosine transforms.

This thesis points out the following major issues: (*i*) SIMD vectorization cannot be achieved easily. Nontrivial mathematical transformation rules are required to obtain automatic performance tuning and thus satisfactory performance for processors featuring 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*) 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.

The framework introduced in this thesis is based on the Kronecker product approach as used in Van Loan [90] and Moura et al. [72]. The experimental results were obtained by extending Fftw (Frigo and Johnson [33]) and Spiral (Moura et al. [72]). Some results of this thesis are presented in Franchetti et al. [24, 25, 27, 28, 29, 30].

---

[4] FFT is the abbreviation of fast Fourier transform.

[5] Spiral is the abbreviation of "signal processing algorithms implementation research for adaptive libraries".

[6] Fftw is the abbreviation of "Fastest Fourier Transform in the West"

Contents

Kurzfassung / Summary 2

Introduction 7

1 Hardware vs. Algorithms 10
  1.1 Discrete Linear Transforms 10
  1.2 Current Hardware Trends 13
  1.3 Performance Implications 15
  1.4 Automatic Performance Tuning 18

2 Standard Hardware 24
  2.1 Processors 24
  2.2 Advanced Architectural Features 27
  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 Motorola's AltiVec Technology 46
  3.4 AMD's 3DNow! 47
  3.5 Vector Computers vs. Short Vector Hardware 48

4 The Mathematical Framework 51
  4.1 Notation 52
  4.2 Extended Subvector Operations 55
  4.3 Kronecker Products 57
  4.4 Stride Permutations 61
  4.5 Twiddle Factors and Diagonal Matrices 65
  4.6 Complex Arithmetic as Real Arithmetic 67
  4.7 Kronecker Product Code Generation 69

5 Fast Algorithms for Linear Transforms 77
  5.1 Discrete Linear Transforms 77
  5.2 The Fast Fourier Transform 83

6 A Portable SIMD API 97
  6.1 Definition of the Portable SIMD API 98

7 Transform Algorithms on Short Vector Hardware 103
7.1 Formal Vectorization of Linear Transforms 104
7.2 FFTs on Short Vector Hardware 114
7.3 A Short Vector Cooley-Tukey Recursion 120
7.4 Short Vector Specific Search 126

8 Experimental Results 129
8.1 A Short Vector Extension for FFTW 129
8.2 A Short Vector Extension for SPIRAL 142

Conclusion and Outlook 164

A Performance Assessment 165
A.1 Short Vector Performance Measures 167
A.2 Empirical Performance Assessment 167

B Short Vector Instruction Sets 175
B.1 The Intel Streaming SIMD Extensions 175
B.2 The Intel Streaming SIMD Extensions 2 180
B.3 The Motorola AltiVec Extensions 183

C The Portable SIMD API 189
C.1 Intel Streaming SIMD Extensions 189
C.2 Intel Streaming SIMD Extensions 2 192
C.3 Motorola AltiVec Extensions 194

D SPIRAL Example Code 199
D.1 Scalar C Code 199
D.2 Short Vector Code 203

E FFTW Example 207
E.1 The FFTW Framework 207
E.2 Scalar C Code 208
E.3 Short Vector Code 209

Table of Abbreviations 212

Bibliography 213

Curriculum Vitae 220

Acknowledgements

I want to thank all persons who supported me to make this work possible.

In particular, I want to thank my Ph. D. advisor Christoph Ueberhuber for many years of a unique working environment, the fruitful cooperation and challenging research, and all the things he taught me. I owe my academic achievements to him. I especially want to express my gratitude for opening the international scientific community for me and teaching me high scientific standards. He gave me the possibility to work at the forefront of scientific research.

I want to thank Herbert Karner for his help, support, and the years of fruitfully working together. I will always remember him.

Additionally, I want to thank Markus Pueschel for his essential input and for giving me the opportunity to spend two summers in Pittsburgh to work with the Spiral team. Important parts of the work presented in this thesis was developed in collaboration with him.

I want to thank Stefan Kral for the valuable discussions and for extending the Fftw codelet generator. Without his help, parts of this work could not have been done.

I want to thank Matteo Frigo and Steven Johnson for their valuable discussions and for giving me the opportunity to access the newest internal versions of Fftw, and for pointing me to the Fftw vector recursion. I want to thank both the Fftw team and the Spiral team for their warm welcome, openness and their cooperation.

Working within Aurora I learned from all the other members, especially from the members of Project 5. I want to thank them for their help and many fruitful discussions. Parts of this work were done in cooperation with the members of Project 5 of Aurora, the Spiral team, and the Fftw team.

I want to thank all members of the Institute for Applied Mathematics and Numerical Analysis at the Vienna University of Technology for their support during all the years I have been working there.

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, here in Austria and in the US, for their friendship and support.

I want to thank my family for always believing in me and I especially want to thank my parents for giving me the opportunity to study and for their support in all the years.

Franz Franchetti

Introduction

The discrete Fourier transform (DFT) plays a central role in the field of scientific computing because DFT methods are an extremely powerful tool to solve various scientific and engineering problems. For example, the DFT is essential for the digital processing of signals or for solving partial differential equations. Other practical uses of DFT methods include geophysical research, vibration analysis, radar and sonar signal processing, speech recognition and synthesis, as well as image processing.

Discrete linear transforms, including the discrete Fourier transform (DFT), the Walsh-Hadamard transform (WHT), and the family of discrete sine and cosine transforms (DSTs, DCTs) are—despite numerical linear algebra algorithms—at the core of most computationally intensive algorithms. Thus, discrete linear transforms are in the center of small scale applications with stringent time constraints (for instance, real time) as well as large scale simulations running on the world's largest supercomputers.

All these transforms are structurally complex and thus lead to complicated algorithms which make it a challenging task to map them to standard hardware efficiently and an even harder problem to exploit special processor features satisfactorily. The unprecedent complexity of today's computer systems implies that performance portable software—software that performs satisfactorily across platforms and hardware generations—can only be achieved by means of automatic empirical performance tuning. It is necessary to apply *search techniques* to find the best implementation for a given target machine. These search techniques have to apply actual run time as cost function, since modelling the machine's behavior accurately enough is impossible on today's computers.

A few years ago major vendors of *general purpose* microprocessors have started to include short vector SIMD (single instruction, multiple data) extensions into their instruction set architecture (ISA) primarily to improve the performance of multi-media applications. Examples of SIMD extensions supporting both integer and floating-point operations include Intel's streaming SIMD extensions (SSE and SSE 2), AMD's 3DNow! as well as its extensions "enhanced 3DNow!" and "3DNow! professional", Motorola's AltiVec extension, and last but not least IBM's Double Hummer floating-point unit for BG/L machines. Each of these ISA extensions is based on the packing of large registers (64 bits or 128 bits) with smaller data types and providing instructions for the parallel operation on these subwords within one register.

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

Processors featuring short vector SIMD instructions are completely different from conventional vector computers. Thus, solutions developed for traditional vector computers are not directly applicable to today's processors featuring short vector SIMD extensions.

By introducing double-precision short vector SIMD extensions, this technology became a major determinant in scientific computing. Conventional scalar code becomes more and more obsolete on machines featuring these extensions as such codes utilize only a fraction of the potential performance. For instance, Intel's Pentium 4 processor featuring the two-way double-precision short vector SIMD extension SSE 2 is currently the processor with the highest *peak performance* (over 6 Gflop/s for double precision and over 12 Gflop/s for single precision) and has become the standard processor in commodity clusters. On the other end of the spectrum, the processors of IBM's BG/L machine—a candidate for the leading position in the Top 500 list—also features a two-way double-precision short vector SIMD extension.

This thesis provides a framework for automatic performance tuning of discrete linear transform codes targeted at computers with processors featuring short vector SIMD extensions, with strong emphasis placed on the discrete Fourier transform.

Synopsis

Chapter 1 introduces discrete linear transforms and discusses he reasons why it is hard to achieve high performance implementations of such algorithms on current computer architectures. The two major automatic performance tuning systems for discrete linear transforms—Spiral and Fftw—are introduced.

In Chapter 2 current hardware trends and advanced hardware features are discussed. The main focus is on CPUs and memory hierarchies.

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

Chapter 4 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 introduced.

Chapter 5 introduces fast algorithms for discrete linear transforms, i. e., matrix-vector products with structured matrices. The special case of 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.

In Chapter 6 a *portable SIMD API* is introduced as a prerequisite for the implementation of the short vector algorithms presented in this thesis.

In Chapter 7 a method for formula-based vectorization of discrete linear transforms is introduced. A large class of discrete linear transforms can be fully vectorized, the remaining transforms can be vectorized at least partially. Various methods to obtain short vector SIMD implementations of DFTs are discussed and the *short vector Cooley-Tukey rule set* is developed which enables high performance short vector SIMD implementations of DFTs.

Chapter 8 shows a number of experimental results. The newly developed formal methods are included into Spiral and Fftw using the portable SIMD API and are tested on various short vector SIMD extensions and architectures. Experimental evidence for the superior performance achievable by using the newly introduced methods is given.

Appendix A discusses the performance assessment of scientific software. Appendix B summarizes the relevant parts of short vector SIMD instruction sets and Appendix C shows the implementation of the portable SIMD API required for the numerical experiments presented in this thesis. Appendix D displays example code obtained using the newly developed short vector SIMD extension for Spiral and Appendix E displays 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 [32], Atlas by Whaley et al. [94], and Spiral by Püschel et al. [80].

1.1 Discrete Linear Transforms

The *discrete Fourier transform* (DFT) is one of the principal algorithmic tools in the field of scientific computing. For example, the DFT is essential in the digital processing of analogous signals. DFT methods are also used in many fields of the computational mathematics, for instance, to solve partial differential equations.

DFT methods are important mathematical tools in geophysical research, vibration analysis, radar and sonar signal processing, speech recognition and synthesis, as well as image processing.

The arithmetic complexity of the DFT was long thought to be $2N^2$ since the DFT boils down to the evaluation of a special matrix-vector product. In 1965 Cooley and Tukey [13] published the *fast Fourier transform* (FFT) algorithm, which reduced the computational complexity of the DFT to $5N \log N$.

The discrete Fourier transform is the most important discrete linear transform. Other examples of discrete linear transforms are the discrete sine and cosine transforms, the wavelet transforms, and the Walsh-Hadamard transform.

1.1.1 Applications

The field of applications for the FFT and the other linear transforms is vast. Several areas where methods based on these transforms play a central role are outlined in this section.

Seismic Observation

The oil and gas industry uses the FFT as a fundamental exploration tool. Using the so-called “seismic reflection method” it is possible to map sub-surface sedimentary rock layers.

A meanwhile historic application of the discrete Fourier transform was to find out, whether a seismic activity was caused by an earthquake or by a nuclear bomb. These events can be distinguished in the frequency domain, because the respective spectra have strongly differing characteristics.

Filtering

Filters are used to attenuate or enhance certain frequency components of a given signal. The most important types are low pass, high pass and band pass filters. But there are even more filter issues that can be addressed with FFTs, for example the complexity reduction of finite impulse responses by providing filter convolution.

Image Processing

Filters can also be applied to two- and more-dimensional digital signals like pictures or movies. They can be smoothed, sharp edges can be enhanced as well as disturbing background noise can be reduced. Especially for highly perturbed medical images (like some X-ray pictures) the FFT can be used to enhance the visibility of certain objects. Furthermore, the spectral representation of a digital picture can be used to gain valuable information for pattern recognition purposes.

The discrete cosine transform (DCT) is a very important transform in image processing. DCTs are used for image compression in the MPEG standards, for motion detection and virtually in any advanced image processing algorithm.

New types of filters are wavelet based and new compression techniques, as introduced in the JPEG2000 standard, utilize the wavelet transform.

Data Communication

The FFT is usually associated with low level aspects of communication, for instance, to understand how a signal behaves when sent through communication channels, amplifiers etc. Especially the degree of distortion can be modelled easily knowing the bandwidth of the channel and the spectrum of the signal. For example, if a signal is composed of certain frequency components but only part of them passes through the channel, adding up only those components passing results in a good approximation of the distorted signal.

Astronomy

Sometimes it is not possible to get all the required information from a “normal” telescope (based on visible light). Radio waves or radar have to be used instead of light. Radio and radar signals are treated just like any other time varying signal and can be processed digitally. For example, the satellite Magellan released in 1989 and sent to earth’s closest planet, Venus, was equipped with modern radar and digital signal processing capabilities and provided excellent data that made possible a computer-generated virtual flight above Venus.

Optics

In optical theory the signal is the oscillatory electric field at a point in space where light passes by. The Fourier transform of the signal is equivalent to breaking up the light into its components by using a prism. The Fourier transform is also used to calculate the diffracted intensity with the experiments of light passing through narrow slits. Even Young’s famous double slit experiment made use of the Fourier transform. These ideas can be applied to all kinds of wave analysis applications like acoustic, X-ray, and microwave diffraction.

Speech Recognition

The field of digital speech processing and recognition is a multi-million Euro business by now. Advances in computational speed and new language models have made speech recognition possible on average PCs.

Speech recognition is a very wide field, consisting of isolated word recognition for specialized fields as medicine as well as connected word and even conversational speech recognition.

Speech recognition is investigated for many reasons like the development of automatic typewriters. The simplicity of use of a speech recognition system and its possible speed as compared with other information input are worth mentioning. Speech is rather independent from other activities involving hands or legs. Many tasks could be performed by computers, if the recognition process was reliable and cheap. For achieving both goals the FFT is an important tool.

FFTs are used to transform the signal, usually after a first filtering process, into the frequency domain to obtain its spectrum. Critical features for perceiving speech by the human ear are mainly included in the spectral information, while the phase information does not play a crucial role.

Not only can the FFT bring the signal into an easier-to-handle form, it is even cheap in time and makes real-time recognition economically affordable.

Further Topics

The discrete Fourier transform is also used for solving partial differential equations and is therefore essential, for example, in computational fluid dynamics. Beside numerical linear algebra the FFT is accounting for most processor cycles. Many large scale applications use the FFT. For instance, climate simulations as conducted on the currently fastest supercomputer—Japan's *Earth Simulator*—are FFT based.

1.2 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.

1.2.1 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 A explains the methodology of run time measurement and performance assessment.

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

1.3.1 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 (see the normalized complexity as function of the problem size in Figure 5.3 on page 91). 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. [9] 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.

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. Performance data were obtained on one processor of an SGI Power Challenge XL (Auer et al. [9]).

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.

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 single-precision FFT routines on an Intel Pentium 4 running at 2.53 GHz. The Pentium 4 features a four-way short vector SIMD extension called streaming SIMD extension (SSE).

SPIRAL-SIMD is an extension of SPIRAL, an architecture adaptive library generator for discrete linear transforms, that generates and optimizes short vector SIMD code. SPIRAL-SIMD is a result of this thesis. The automatically optimized programs generated by SPIRAL-SIMD are currently the fastest FFT codes on an Intel Pentium 4 (apart from $N = 2^{11}$).

Intel MKL (math kernel library) is the hand optimized vendor library for Pentium 4 processors. It utilizes prefetching and SSE.

SIMD-FFT is a short vector SIMD implementation by Rodriguez [82]. Performance has been reported on a Pentium 4 running at 1.4 GHz. As the source code is not publicly available, the performance for 2.53 GHz was estimated by scaling up the reported results. The scaled performance data are an estimate for an upper bound.

SPIRAL with Vectorizing Compiler. The vectorization mode of the Intel

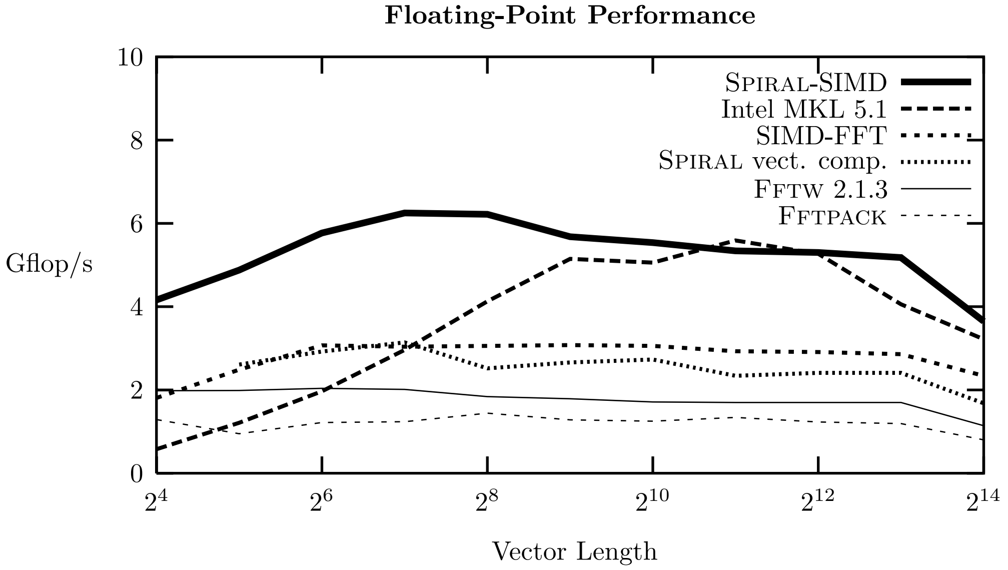


Figure 1.1: Performance of single-precision FFT programs on an Intel Pentium 4 running at 2.53 GHz. Performance is given in *pseudo Gflop/s* (Frigo and Johnson [33]) expressing a weighted inverse of run time that preserves run time relations.

C++ Compiler was activated and applied to the Spiral system. See Section 8.2.6 for details.

FFTW 2.1.3 is the current standard distribution of Fftw. It does not feature any support for short vector SIMD extensions.

The scalar Spiral system delivers the same performance as Fftw 2.1.3 on the Pentium 4. For scalar implementations, Fftw and Spiral are currently the fastest publicly available FFT programs. Both of them are as fast as FFT programs specifically optimized for a given architecture. Thus, Fftw may serve as the baseline in Figure 1.1.

FFTPACK. The function `cfftf` of the heavily used Fftpack served in this experiment as the non-adaptive standard.

The numerical experiment summarized in Figure 1.1 shows that the architecture adaptive system Spiral-SIMD with support for SSE is most of the time faster than the vendor library. Compiler vectorization accelerates scalar code generated by Spiral significantly but does not deliver top performance. Hand-tuning for SSE as carried out in Simd-Fft leads to good performance behavior but is far from optimum. The industry standard routine `cfftf` of Fftpack is about *six* times slower than the currently fastest program.

The experiments summarized in Auer et al. [9] show that Fftpack is among the fastest scalar programs in the group of publicly available FFT implementations. 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.

1.4 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. 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.

1.4.1 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. [1]).

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.

1.4.2 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.

1.4.3 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. [94]).

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. [12]). 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* (Atlas) project 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. [94]). 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 extension of these systems with the newly developed short vector SIMD vectorization. In cooperation with these two projects the worldwide fastest FFT implementations for Intel processors and very fast implementations for the other short vector SIMD extensions were achieved.

FFTW

Fftw (*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 [31], Frigo and Johnson [32]). 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, 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 5.1 and Haentjens [40]). 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 [33].

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 [90]).

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

Spiral (*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 Moura et al. [72]. Spiral uses a mathematical approach that 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 5 summarizes the discrete linear transforms and Chapter 4 summarizes the mathematical framework. More specifically, Spiral 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 Chapter 4).

- 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 (see Section 4.7).

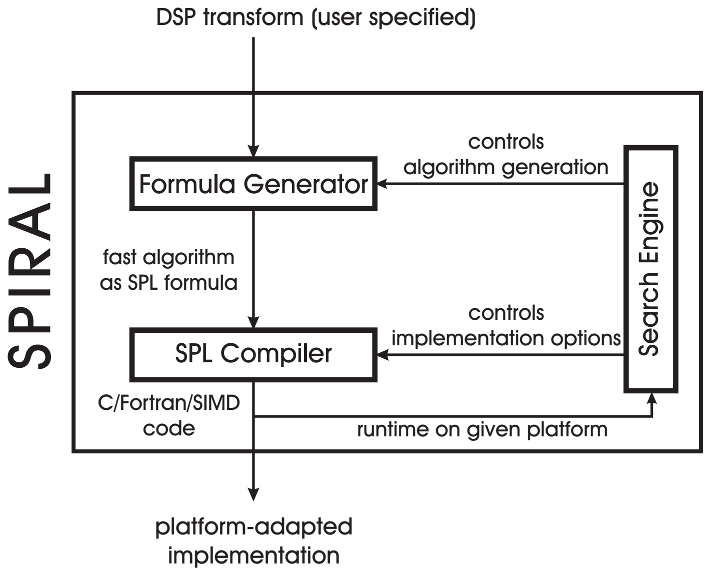


Figure 1.2: Spiral's architecture.

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. [79], Singer and Veloso [84], Xiong et al. [95].

GAP and AREP. SPIRAL's formula generator uses AREP, a package by Egner and Püeschel [22] implemented in the language GAP [88] 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 extensions to both FFTW and SPIRAL to support all current short vector SIMD extensions in their automatic performance tuning process.

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 [36] or Hlavacs and Ueberhuber [41].

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

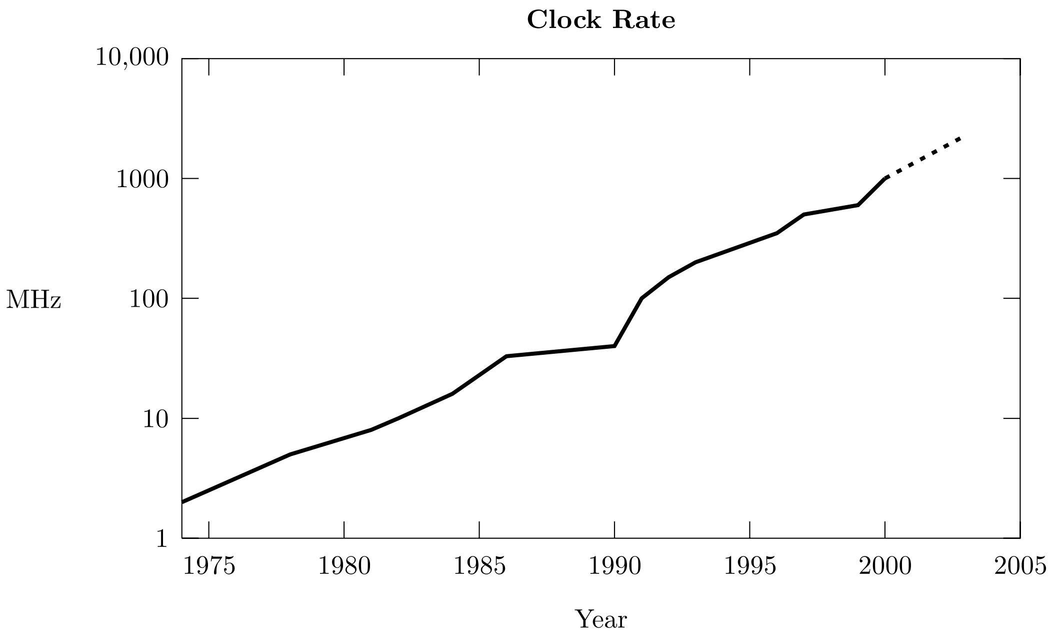


Figure 2.1: Clock rate trend for off-the-shelf CPUs.

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. As these processors still must understand the x86 instruction set, they cut complex instructions into simpler ones and 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.

Register Renaming

One way to increase the number of registers within a processor family is to increase the number of physical registers while keeping the old number of logical ones. This means that programs still use, for instance, only 32 integer registers, while internally, 64 integer registers are available. If several instructions use the same logical register, special circuitry assigns different physical registers to be used instead of the logical one. This technique is called *register renaming*.

Example 2.5 (Register Renaming) The MIPS R10000 is an upgrade of MIPS R6000 processors that had 32 integer and 32 floating-point registers, each 64 bits wide. The R10000 contains 64 integer and 64 floating-point registers that can dynamically be renamed to any of the 32 logical ones.

To do this, the processor internally must maintain a list containing already renamed registers and a list of free registers that can be mapped to any logical register.

Register Bypassing

Competition for registers can stall pipelines and execution units. A technique called *register bypassing* or *register forwarding* enables the forwarding of recently calculated results directly to other execution unit without temporarily storing them in the register file.

Register Windows

A *context switch* is an expensive operation as the current state has to be stored and another state has to be loaded. As the contents of registers also represent the current program state, their contents are stored in memory as well. Constantly storing and loading the contents of a large number of registers takes some time, thus, the idea of *register windows* has emerged. Here, additional register files are used to hold the information of several processes. Instead of loading the whole register file from memory in case of a context switch, the CPU elects the corresponding register window to be the current file and keeps the old one as a back up of the last process.

Example 2.6 (Register Windows) The SPARC architecture is a RISC architecture using register windows. Instead of 32 basic registers, this architecture offers eight overlapping windows of 24 registers each.

Register windows have severe drawbacks. On modern computers, dozens or even hundreds of processes are likely to be executed. Implementing hundreds of register windows would cost too much, a few windows, however, will not improve the system's performance significantly. Thus, the possible speed-up would not pay off the additional effort in designing register windows.

2.2.3 Branch Optimization

Conditional branches are responsible for most pipeline hazards. Whenever a conditional branch enters an instruction pipeline, there is a chance of jumping to a new memory location, yet this cannot be known until the branch instruction is executed. Thus, if the branch is taken, all succeeding instructions that have entered the pipeline are obsolete and must be thrown away. Restarting the pipeline at every conditional branch reduces the system's performance significantly. Therefore, mechanisms for reducing branch penalties had to be found.

Branch Delay Slots

Early pipelined processors used instructions that had been inserted after the branch by the compiler and that were executed in any case whether the branch was taken or not. Such instructions are called *branch delay slots*. However, it is very difficult to find even one useful instruction that can be executed in either branch, to find more than one is almost impossible. As modern superscalar processors issue four or more instructions per cycle, finding four branch delay slots is not practicable.

Speculative Execution

Another way of improving the trouble with conditional branches is to guess whether the branch will be taken or not. In either case, new instructions can be brought from cache or main memory early enough to prevent execution units from stalling. If the guess is correct for most of the branches, the penalty for wrong guesses can be neglected. This idea is extended to *predication* used in the Itanium processor family.

To implement this idea, a small cache called *branch target buffer* is used storing pairs of previously taken branches and the instructions found there. This way, instructions can be brought quickly to execution units. Branches that are scarcely taken do not reside in this buffer, while branches that are often taken and thus have a high probability to be taken again are stored there.

Instructions that have entered execution units based on branch guesses, of course, must not be allowed to be completed until the branch has been resolved. Instead, they must wait in the reorder buffer.

Static Branch Prediction

Another approach uses static information known at compile time for making a guess. Conditional branches often occur at the end of loops, where it is tested whether the loop is executed once more. As loops are often executed thousands or even millions of times, it is highly probable that the branch is taken. Monitoring shows that 90 % of all backward branches and 50 % of all forward branches are taken. Thus, branches that jump back are predicted to be taken while forward branches are not. The information, whether the branch jumps back or forth is often provided by the compiler by setting or clearing a *hint-bit*.

Additionally, in some cases, conditional branches can be resolved by early pipeline stages such as the prefetch logic itself. If, for example, the branch depends on a register to be set to zero, the branch can be resolved earlier.

Example 2.7 (Static Branch Prediction) The Intel Itanium processors use static branch prediction, thus depending on optimized compilers to predict which branch is likely to be taken.

Dynamic Branch Prediction

More sophisticated designs change their predictions dynamically at run time. The POWER PC 604, for example, uses a 512-entry, direct mapped *branch history table* (BHT) mapping four branch-prediction states: *strong taken, weak taken, weak not-taken* and *strong not-taken*. The predicted state of a particular branch instruction is set and modified based on the history of the instruction.

The BHT feeds the *branch target address cache* (BTAC) with both the address of a branch instruction and the target address of the branch. The BTAC—a fully associative, 64-entry cache—stores both the address and the target of previously executed branch instructions. During the fetch stage, this cache is accessed by the fetch logic. If the current fetch address—the address used to get the next instruction from the cache—matches an address in the BTAC then the branch target address associated with the fetch address is used instead of the fetch address to fetch instructions from the cache (Ryan [83]).

Example 2.8 (Dynamic Branch Prediction) The Cyrix 6x86 (M1), Sun UltraSPARC and MIPS R10000 processors use two history bits for branch prediction. Here, the dominant direction is stored, resulting in only one misprediction per branch. Monitoring shows a 86 % hit ratio of this prediction type. The Intel Pentium III uses the more sophisticated Yeh-algorithm requiring 4 bits per entry. This is due to the fact that a mispredicted branch will result in a 13 cycle penalty (in contrast to 3 cycles in the M1). This algorithm achieves a hit ratio between 90 % and 95 %.

2.2.4 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 POWER 4, Intel IA-64 and Motorola POWER PC microprocessors. In the Motorola POWER PC, 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 POWER 4 implement the FMA instruction by integrating the multiplier and the adder into one multiply-add FPU. Therefore in the POWER 4 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 then either multiply or add. The FMA instruction may be used to simulate higher precision cheaply.

2.2.5 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), [70] 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) [52] and AMD processors featuring different 3DNow! versions (Athlon and successors) [2]. 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. 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 2.9 (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.

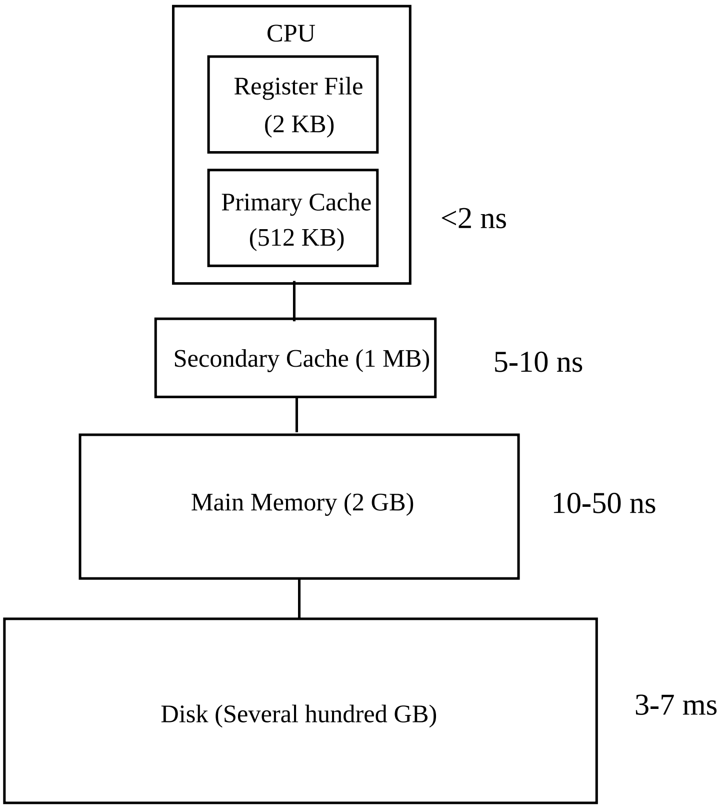


Figure 2.2: The memory hierarchy of a modern computer.

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 POWER 4 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.

Prefetching

One aspect of cache design is to try to keep miss rates low by providing data items when they are needed. One way to achieve this goal is to try to guess, which items are likely to be needed soon and to fetch them from main memory in advance. This technique is called *prefetching*, whereas standard fetching on demand of the processor is called *demand fetching*.

As the flow of machine instructions from memory to the processor is mostly sequential and usually regular data access patterns occur, prefetching can improve the performance of fast RISC processors in an impressive way.

Several multi-media extensions to current processors feature streaming memory access and support for *software initiated prefetching*.

Transaction Lookaside Buffer

On all current computer systems in the scope of this thesis, virtual memory is used. Translating virtual addresses by using tables results in two to three table lookups per translation. By recalling the principles of locality, another feature will make translations much faster: the so-called *translation lookaside buffer* (TLB), resembling a translation cache. TLBs contain a number of recently used translations from virtual to physical memory. Thus, the memory management will first try to find a specified virtual address in the TLB. On a *TLB miss*, the management has consult the virtual page table which is stored in memory. This memory access may lead to cache misses and—in the worst case—page faults.

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 Hummer extension and Intel's SSE 2 can operate on two double-precision numbers in parallel.

The Double Hummer extension which is part of IBM's Blue Gene initiative and will be implemented in BG/L processors is still classified and will therefore be excluded from the following discussion. However, this particular SIMD extension will be 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 multimedia 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 UltraSparc 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 [70, 71]. The features relevant for this thesis are summarized in Appendix B.3.

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 [49, 50, 51] and the C++ compiler manual [52]. The features relevant for this thesis are summarized in Appendix B.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) are introduced in the Pentium 4 processor architecture and include floating-point SIMD instructions, integer SIMD instructions, as well as conversion of packed data between XMM registers and MMX registers.

The newly 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 [49, 50, 51] and the C++ compiler manual [52]. The features relevant for this thesis are summarized in Appendix B.2.

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 [54, 55, 56] and the C++ compiler manual [52].

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 [2] and in the x86-64 manuals [6, 7].

Overview. Table 3.1 gives an overview over the SIMD floating-point capabilities found in current microprocessors.

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.

Vendor	Name	n-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	MS Visual C++ Intel C++ Compiler GNU C Compiler 3.0
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	Hummer^2	2-way	double	BG/L processor	IBM XLCentury

Table 3.1: Short vector SIMD extensions providing floating-point arithmetic found in general purpose microprocessors.

3.1.4 Software Support

Currently, application developers have three common methods for accessing multi-media hardware within in 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 and SSE 2 and 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 and the Vector C compiler. Vectorizing compilers are analyzed in Section 8.2.6.

In the remainder of this chapter the short vector SIMD extensions are discussed in detail.

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 it's predecessor, the Pentium II processor.

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

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 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 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

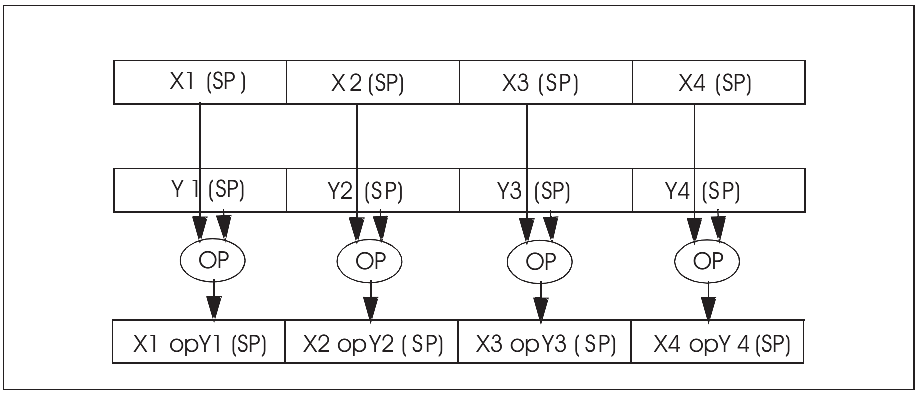


Figure 3.1: Packed SSE operations.

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.

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 sources, `SHUFPS` can return any combination of the four floating-point numbers

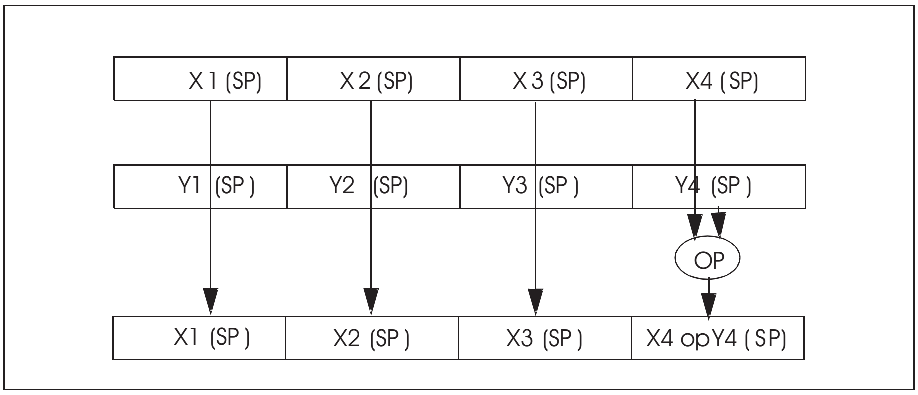


Figure 3.2: Scalar SSE operations.

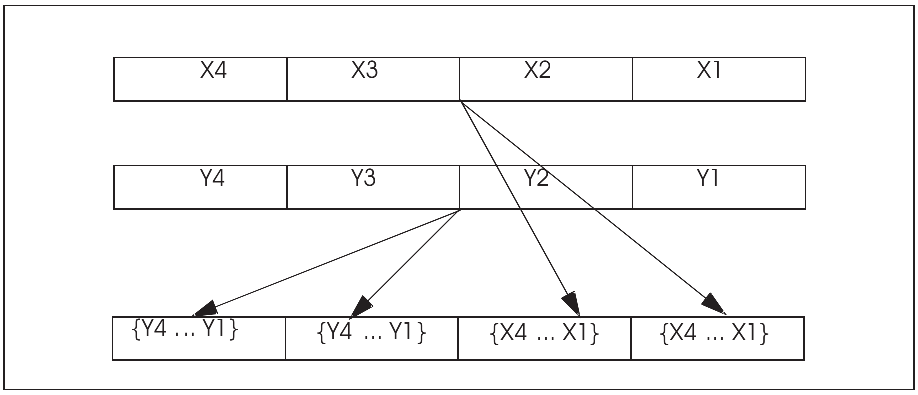


Figure 3.3: Packed shuffle SSE operations.

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, (*ii*) 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 it's 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 [49, 50]. Further information on programming Intel's SSE can be found in the related application notes [44, 46] and the IA-32 optimization manual [47]. Further information on data alignment issues is given in [45].

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 is 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 [49, 50]. Further information on programming Intel's SSE 2 can be found in the IA-32 optimization manuals [47, 48]. Further information on data alignment issues is given in [45].

3.2.3 The Itanium Processor Family

Intel's Itanium processor family (IPF, formerly called IA-64) is a VLIW processor family (introducing the VLIW architecture called EPIC) developed in cooperation with HP. It is targeted at workstations and servers, not necessarily desktop PCs. The first processors of this line are the Itanium and the Itanium 2 processors.

IPF processors have many advanced processor features, including predication, register windows, explicit parallelism, and a large register file. Details of these features can be found in Section 2.2.

In the context of this thesis, the short vector SIMD operations provided by the IPF are the most interesting ones. The IPF features two types of short vector SIMD operations: (*i*) Legacy SSE support, and (*ii*) two-way single-precision SIMD support.

On the processors' hardware, the 64 bit double-precision floating-point registers are split into two-way single-precision vector registers. Any supported double-precision floating-point operation can be executed as a two-way single-precision vector operation. Additionally, IPF hardware features exchange operations similar to the shuffle operations provided by SSE 2.

As IPF processors have to be able to run legacy IA-32 codes which may contain SSE instructions, SSE is supported. But any SSE instruction is emulated by IPF two-way operations. The only language extension to utilize the IPF two-way SIMD operations is available through the Intel C++ SSE intrinsics which are implemented using the IPF SIMD operations. Intel sees the SSE support not as a high-performance API for the IPF SIMD instructions. Thus, currently no native API exists to utilize these instructions from the source code level.

Documentation. IPF is described in the Itanium architecture manuals [54, 55, 56] and the model specific optimization manuals [53, 57]

3.3 Motorola's AltiVec Technology

AltiVec is Motorola's vector processing support that has been added to the POWER PC architecture. The first POWER PC chip that included AltiVec is the MPC 7400 G4. Motorola's AltiVec technology expands the POWER PC architecture through the addition of a 128 bit vector execution unit, which operates concurrently with the existing integer and floating-point units. This new engine provides for highly parallel operations, allowing the simultaneous execution of up to 16 operations in a single clock cycle. The currently newest member of Motorola's Power PC series featuring AltiVec is the MPC 7455 G4.

AltiVec technology offers support for:

- 16-way parallelism for 8 bit signed and unsigned integers and characters,
- 8-way parallelism for 16 bit signed and unsigned integers, and

- 4-way parallelism for 32 bit signed and unsigned integers and IEEE floating-point numbers.

AltiVec technology also includes a separate register file containing 32 entries, each 128 bits wide. These 128 bit wide registers hold the data sources for the AltiVec technology execution units. The registers are loaded and unloaded through vector store and vector load instructions that transfer the contents of a single 128 bit register to and from memory.

AltiVec defines over 160 new instructions. Each AltiVec instruction specifies up to three source operands and a single destination operand. All operands are vector registers, with the exception of the load and store instructions and a few instruction types that provide operands from immediate fields within the instruction.

In the G4 processor, data can not be moved directly between the vector registers and the integer or floating-point registers. Instructions are dispatched to the vector unit in the same fashion as the integer and floating-point units. Since the vector unit is broken down internally into two separate execution units, two AltiVec instructions can be dispatched in the same clock cycle if one is an arithmetic instruction and the other one is a permute instruction.

On AltiVec efficient memory access has to be vector memory access of 16 byte aligned data. All other (unaligned) memory access operations result in high penalties. Such operations have to built from multiple vector access operations and permutations or using multiple expensive single element access operations.

Appendix B.3 describes the AltiVec instructions relevant in the context of this thesis.

Technical details are given in the Motorola AltiVec manuals [70, 71].

3.4 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 [3, 2, 5, 4]

With AMD's next generation processor (codename Hammer [6, 7]), 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.

3.5 Vector Computers vs. Short Vector Hardware

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 (Golub and Van Loan [38], Dongarra et al. [17]). It is therefore a straightforward 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 deskside 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 overlayed 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 superscalar principle by executing several vector operations per time unit (Dongarra et al. [18]).

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 lager 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.

In numerical experiments summarized in Section 8.2.6, a vectorizing compiler was plugged into the Spiral system and the required changes were made. The Spiral/vect system was able to speed up the generated code significantly. For simpler codes, the performance achieved by the vectorizing compiler is close to the results obtained using formula based vectorization as developed in the next chapter (although still inferior). However, it is not possible to achieve the same performance level as reached by the presented approach for more complicated algorithms like FFTs.

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 (Johnson et al. [60]). 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 (Johnson et al. [59]). 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 (Lamson [65]).

Chapter 4

The Mathematical Framework

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 [90].

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. [61]). 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 [76] 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 [90] 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 [87] and Johnson et al. [60] for FFT implementations on classic vector computers and Norton and Silberger [75] on parallel computers with MIMD architecture. Gupta et al. [39] and Pitsianis [77] 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. [60], the SPIRAL project (Moura et al. [72] and Püschel et al. [79]) 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. [60] 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.

4.1 Notation

The notational conventions introduced in the following are used throughout this thesis. 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.

4.1.1 Vector and Matrix Notation

In this thesis, 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 4.1 (Parameterized Matrices) L_8^{64} is a stride permutation matrix of size 64×64 with stride 8 (see Section 4.4), T_2^8 is a complex diagonal matrix of size 8×8 whose entries are given by the parameter "2" (see Section 4.5), 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 4.2 (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 5.1).

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 4.3 (Complex Matrix) The 2-by-3 complex matrix $A \in \mathbb{C}^{2\times3}$ 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.

4.1.2 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*:

$$u = j : k \quad \Leftrightarrow \quad u = (j, j+1, \ldots, k), \quad j \leq k.$$

Example 4.4 (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

$$\begin{aligned} A(u, :) \quad &\Leftrightarrow \quad A(u, 0 : n-1), \\ A(:, v) \quad &\Leftrightarrow \quad A(0 : m-1, v). \end{aligned}$$

Vectors with non-unit increments are specified by the notation

$$u = i : j : k \quad \Leftrightarrow \quad u = (i, i+k, \ldots, j),$$

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 4.5 (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.

4.1.3 Diagonal Matrices

If $d \in \mathbb{C}^n$, then $D = \operatorname{diag}(d) = \operatorname{diag}(d_0, \ldots, d_{n-1}) \in \mathbb{C}^{n\times n}$ is the diagonal matrix

$$D = \begin{pmatrix} d_0 & & & \mathbf{0} \\ & d_1 & & \\ & & \ddots & \\ \mathbf{0} & & & d_{n-1} \end{pmatrix}.$$

Example 4.6 (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

$$\mathrm{I}_n = \begin{pmatrix} 1 & & & \mathbf{0} \\ & 1 & & \\ & & \ddots & \\ \mathbf{0} & & & 1 \end{pmatrix}.$$

4.1.4 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 thesis P is a permutation matrix in most cases.

Example 4.7 (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

$$\mathrm{J}_2 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

leading to

$$A^{\mathrm{J}_2} = \mathrm{J}_2^{-1}\, A\, \mathrm{J}_2 = \begin{pmatrix} a_1 & 0 \\ 0 & a_0 \end{pmatrix}.$$

Property 4.1 (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 4.2 (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

$$A^P P^{-1} = P^{-1}A.$$

4.1.5 Direct Sum of Matrices

Definition 4.1 (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,..., A_{n-1} being *not* necessarily of the same dimension, their direct sum is defined as the block diagonal matrix

$$\bigoplus_{i=0}^{n-1} A_i = A_0 \oplus A_1 \oplus \cdots \oplus A_{n-1} = \begin{pmatrix} A_0 & & & \mathbf{0} \\ & A_1 & & \\ & & \ddots & \\ \mathbf{0} & & & A_{n-1} \end{pmatrix}.$$

4.1.6 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 4.2 (Direct Sum of Vectors) Let y be a vector of length N and x_i be n vectors of lengths m_i:

$$y = \begin{pmatrix} u_0 \\ u_1 \\ \vdots \\ u_{N-1} \end{pmatrix},\ x_0 = \begin{pmatrix} u_0 \\ u_1 \\ \vdots \\ u_{m_0-1} \end{pmatrix},\ x_1 = \begin{pmatrix} u_{m_0} \\ u_{m_0+1} \\ \vdots \\ u_{m_1-1} \end{pmatrix}, \ldots,\ x_{n-1} = \begin{pmatrix} u_{m_{n-2}} \\ u_{m_{n-2}+1} \\ \vdots \\ u_{N-1} \end{pmatrix}.$$

Then the direct sum of x_0, $x_1, \ldots,$ x_{n-1} is defined by

$$y = \bigoplus_{i=0}^{n-1} x_i = x_0 \oplus x_1 \oplus \ldots \oplus x_{n-1} = \begin{pmatrix} u_0 \\ u_1 \\ \vdots \\ u_{N-1} \end{pmatrix} \in \mathbb{C}^N.$$

4.2 Extended Subvector Operations

Most identities introduced in this chapter can be formulated and proved easily using the standard basis.

Definition 4.3 (Standard Basis) Let e_0^N, $e_1^N, \ldots,$ 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

$$\{e_i^N : i = 0, 1, \ldots, N-1\} \tag{4.1}$$

is the standard basis of $\mathbb{C}^N$.

4.2.1 The Read/Write Notation

The *read/write notation* (RW notation) is used in mathematical formulas to represent operations like writes to or reads from a vector at a certain position. Using RW notation it is possible to describe pseudo code on the basis of mathematical formula interpretation without dealing with implementation details.

The prerequisite is the distributive law for matrix-vector products.

Property 4.3 (Distributivity)

$$\sum_{i=0}^{k-1}\Big(A_i x\Big) = \Big(\sum_{i=0}^{k-1} A_i\Big)x.$$

Definition 4.4 (Basic Read Operation) A basic read operation applied to a vector of size N reads out a subvector of size n with stride s at base address b.

$$\mathrm{R}_{b,s}^{N,n} := \left(\begin{array}{c} {e_b^N}^T \\ \hline {e_{b+s}^N}^T \\ \hline \vdots \\ \hline {e_{b+(n-1)s}^N}^T \end{array} \right),$$

where $e_i^N \in \mathbb{C}^{N \times 1}$ is a vector of the standard basis (4.1).

Example 4.8 (Basic Read Operation) For $x \in \mathbb{C}^8$, $y \in \mathbb{C}^4$, $y := \mathrm{R}_{0,2}^{8,4}\, x$ is given by

$$y \;:=\; \mathrm{R}_{0,2}^{8,4}\, x \;=\; \begin{pmatrix} 1 & . & . & . & . & . & . & . \\ . & . & 1 & . & . & . & . & . \\ . & . & . & . & 1 & . & . & . \\ . & . & . & . & . & . & 1 & . \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{pmatrix} \;=\; \begin{pmatrix} x_0 \\ x_2 \\ x_4 \\ x_6 \end{pmatrix}$$

with the zeros represented by dots.

Definition 4.5 (Basic Write Operation) A basic write operation applied to a vector of size N writes a subvector of size n with stride s to base address b:

$$\mathrm{W}_{b,s}^{N,n} := \left(e_b^N \mid e_{b+s}^N \mid \cdots \mid e_{b+(n-1)s}^N \right),$$

where $e_i^N \in \mathbb{C}^{N \times 1}$ is a vector of the standard basis (4.1).

Example 4.9 (Basic Write Operation) For $x \in \mathbb{C}^8$, $y \in \mathbb{C}^4$, $y := \mathrm{W}_{0,1}^{8,4}\, x$ is given by

$$y \;:=\; \mathrm{W}_{0,1}^{8,4}\, x \;=\; \begin{pmatrix} 1 & . & . & . \\ . & 1 & . & . \\ . & . & 1 & . \\ . & . & . & 1 \\ . & . & . & . \\ . & . & . & . \\ . & . & . & . \\ . & . & . & . \end{pmatrix} \begin{pmatrix} x_0 \\ x_2 \\ x_4 \\ x_6 \end{pmatrix} \;=\; \begin{pmatrix} x_0 \\ x_2 \\ x_4 \\ x_6 \\ . \\ . \\ . \\ . \end{pmatrix}$$

with the zeros represented by dots.

4.2.2 Algebraic Properties of Read and Write Operations

Read and write operations have the following algebraic properties.

When using a basic read operation to obtain an intermediate subvector and then again using a read operation to obtain the final subvector, this operation can be expressed by a single basic read operation.

Property 4.4 (Read Multiplicativity)

$$\mathrm{R}_{b_1+b_2s_1,s_1s_2}^{N_1N_2n,n} = \mathrm{R}_{b_2,s_2}^{N_2n,n} \; \mathrm{R}_{b_1,s_1}^{N_1N_2n,N_2n} .$$

The same applies to two consecutive basic write operations.

Property 4.5 (Write Multiplicativity)

$$\mathrm{W}_{b_1+b_2s_1,s_1s_2}^{N_1N_2n,n} = \mathrm{W}_{b_1,s_1}^{N_1N_2n,N_2n} \; \mathrm{W}_{b_2,s_2}^{N_2n,n} .$$

Multiplication of read and write matrices yields an identity matrix.

Property 4.6 (Read Write Identity)

$$\mathrm{I}_n = \mathrm{R}_{b,s}^{mn,n} \; \mathrm{W}_{b,s}^{mn,n} .$$

Property 4.7 (Read Write Identity)

$$\mathrm{I}_{mn} = \mathrm{W}_{b,s}^{mn,n} \; \mathrm{R}_{b,s}^{mn,n} .$$

Transposition of read matrices yields write matrices.

Property 4.8 (Read Write Transposition)

$$(\mathrm{R}_{b,s}^{mn,n})^\top = \mathrm{R}_{b,s}^{mn,n},$$

$$(\mathrm{W}_{b,s}^{mn,n})^\top = \mathrm{R}_{b,s}^{mn,n} .$$

4.3 Kronecker Products

Definition 4.6 (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

$$A \otimes B := \begin{pmatrix} a_{0,0}B & \dots & a_{0,N_1-1}B \\ \vdots & \ddots & \vdots \\ a_{M_1-1,0}B & \dots & a_{M_1-1,N_1-1}B \end{pmatrix} \in \mathbb{C}^{M_1M_2 \times N_1N_2}.$$

Definition 4.7 (Tensor Basis) Set $N = N_1N_2$ and form the set of tensor products

$$e_i^{N_1} \otimes e_j^{N_2}, \quad i = 0, 1, \dots, N_1 - 1, \quad j = 0, 1, \dots, N_2 - 1. \tag{4.2}$$

This set is called *tensor basis*.

Since any element e_k^N of the standard basis (4.1) can be expressed as

$$e_{j+iN_2}^N = e_i^{N_1} \otimes e_j^{N_2}, \quad i = 0, 1, \ldots, N_1 - 1, \quad j = 0, 1, \ldots, N_2 - 1,$$

the tensor basis of Definition 4.7 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 4.8 (Parallel Kronecker Products) Let $A \in \mathbb{C}^{m \times n}$ be an arbitrary matrix and let $\mathrm{I}_k \in \mathbb{C}^{k \times k}$ be the identity matrix. The expression

$$\mathrm{I}_k \otimes A = \begin{pmatrix} A & & & \mathbf{0} \\ & A & & \\ & & \ddots & \\ \mathbf{0} & & & A \end{pmatrix}$$

is called *parallel Kronecker product*.

A parallel Kronecker product can be viewed as a *parallel operation*. Its action on a vector $x = x_0 \oplus x_1 \oplus \cdots \oplus x_{k-1}$ can be performed by computing the action of A on each of the k consecutive segments x_i of x independently.

Example 4.10 (Parallel Kronecker Product) Let $A_2 \in \mathbb{C}^{2 \times 2}$ be an arbitrary matrix and let $\mathrm{I}_3 \in \mathbb{C}^{3 \times 3}$ be the identity matrix. Then

$$y := (\mathrm{I}_3 \otimes A_2) x$$

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

$$\begin{pmatrix} y_0 \\ y_1 \end{pmatrix} := \begin{pmatrix} a_{0,0} & a_{0,1} \\ a_{1,0} & a_{1,1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \end{pmatrix}$$

$$\begin{pmatrix} y_2 \\ y_3 \end{pmatrix} := \begin{pmatrix} a_{0,0} & a_{0,1} \\ a_{1,0} & a_{1,1} \end{pmatrix} \begin{pmatrix} x_2 \\ x_3 \end{pmatrix}$$

$$\begin{pmatrix} y_4 \\ y_5 \end{pmatrix} := \begin{pmatrix} a_{0,0} & a_{0,1} \\ a_{1,0} & a_{1,1} \end{pmatrix} \begin{pmatrix} x_4 \\ x_5 \end{pmatrix}$$

independently.

Definition 4.9 (Vector Kronecker Products) Let $A \in \mathbb{C}^{m \times n}$ be an arbitrary matrix and let $\mathrm{I}_k \in \mathbb{C}^{k \times k}$ be the identity matrix. The expression

$$A \otimes \mathrm{I}_k = \begin{pmatrix} a_{0,0}\,\mathrm{I}_k & \cdots & a_{0,n-1}\,\mathrm{I}_k \\ \vdots & \ddots & \vdots \\ a_{m-1,0}\,\mathrm{I}_k & \cdots & a_{m-1,n-1}\,\mathrm{I}_k \end{pmatrix}$$

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

$$a_{r,0}x_0 + a_{r,1}x_1 + \cdots + a_{r,n-1}x_{n-1}, \quad r = 0, 1, \ldots, m-1$$

are performed. Expressions of the form $A \otimes \mathrm{I}_k$ are called vector operations as the operate on vectors of size k.

Example 4.11 (Vector Kronecker Product) Let $A_2 \in \mathbb{C}^{2 \times 2}$ be an arbitrary matrix and let $\mathrm{I}_3 \in \mathbb{C}^{3 \times 3}$ be the identity matrix. Then

$$y := (A_2 \otimes \mathrm{I}_3)x$$

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:

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \end{pmatrix} := a_{0,0} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} + a_{0,1} \begin{pmatrix} x_3 \\ x_4 \\ x_5 \end{pmatrix}$$

$$\begin{pmatrix} y_3 \\ y_4 \\ y_5 \end{pmatrix} := a_{1,0} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} + a_{1,1} \begin{pmatrix} x_3 \\ x_4 \\ x_5 \end{pmatrix}.$$

4.3.1 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 4.7.

Property 4.9 (Identity) If I_m and I_n are identity matrices, then

$$\mathrm{I}_m \otimes \mathrm{I}_n = \mathrm{I}_{mn}\,.$$

Property 4.10 (Identity) If I_m and I_n are identity matrices, then

$$\mathrm{I}_m \oplus \mathrm{I}_n = \mathrm{I}_{m+n}\,.$$

Property 4.11 (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 4.12 (Transposition) If A, B are arbitrary matrices, then

$$(A \otimes B)^\top = A^\top \otimes B^\top.$$

Property 4.13 (Inversion) If A and B are regular matrices, then

$$(A \otimes B)^{-1} = A^{-1} \otimes B^{-1}.$$

Property 4.14 (Mixed-Product Property) If A, B, C and D are arbitrary matrices, then

$$(A \otimes B)\,(C \otimes D) = A\,C \otimes B\,D,$$

provided the products AC and BD are defined.

A consequence of this property is the following factorization.

Corollary 4.1 (Decomposition) If $A \in \mathbb{C}^{m_1 \times n_1}$ and $B \in \mathbb{C}^{m_2 \times n_2}$, then

$$A \otimes B = A\,\mathrm{I}_{n_1} \otimes \mathrm{I}_{m_2}\,B = (A \otimes \mathrm{I}_{m_2})(\mathrm{I}_{n_1} \otimes B),$$

$$A \otimes B = \mathrm{I}_{m_1}\,A \otimes B\,\mathrm{I}_{n_2} = (\mathrm{I}_{m_1} \otimes B)(A \otimes \mathrm{I}_{n_2}).$$

The mixed-product property can be generalized in two different ways.

Corollary 4.2 (Generalized Mixed-Product Property) For k matrices of appropriate sizes it holds that

$$(A_1 \otimes A_2 \otimes \cdots \otimes A_k)(B_1 \otimes B_2 \otimes \cdots \otimes B_k) = A_1B_1 \otimes A_2B_2 \cdots \otimes A_kB_k,$$

and

$$(A_1 \otimes B_1)(A_2 \otimes B_2) \cdots (A_k \otimes B_k) = (A_1A_2 \cdots A_k) \otimes (B_1B_2 \cdots B_k).$$

Property 4.15 (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 4.16 (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.

4.4 Stride Permutations

Definition 4.10 (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 (4.2) of $\mathbb{C}^{mn}$.

$$\mathrm{L}_n^{mn}(e_i^n \otimes e_j^m) = e_j^m \otimes e_i^n.$$

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 4.3 (Stride Permutation) For a vector $x \in \mathbb{C}^{mn}$ the application of the stride permutation L_n^{mn} results in

$$\mathrm{L}_n^{mn}\, x := \begin{pmatrix} x(0 : (m-1)n : n) \\ x(1 : (m-1)n + 1 : n) \\ \vdots \\ x(n-1 : mn-1 : n) \end{pmatrix}.$$

Definition 4.11 (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 4.12 (Perfect Shuffle Permutation) The permutation $\mathrm{L}_{n/2}^n$, 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 $\mathrm{L}_{n/2}^n$ is denoted in short by Π_n.

Mixed Kronecker Products

In this thesis combinations of tensor products and stride permutations are very important. These constructs 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 is a major issue in this thesis and leads to the short vector Cooley-Tukey FFT.

Definition 4.13 (Right Mixed Kronecker Product) Let $A \in \mathbb{C}^{m \times n}$ be an arbitrary matrix, $\mathrm{I}_k \in \mathbb{C}^{k \times k}$ be the identity matrix, and L_k^{km} be a stride permutation. An expression of the form

$$(\mathrm{I}_k \otimes A)\, \mathrm{L}_k^{mk}$$

is called *right mixed Kronecker product.*

Definition 4.14 (Left Mixed Kronecker Product) Let $A \in \mathbb{C}^{m \times n}$ be an arbitrary matrix, $\mathrm{I}_k \in \mathbb{C}^{k \times k}$ be the identity matrix, and L_k^{mk} be a stride permutation. An expression of the form

$$\mathrm{L}_k^{mk}(A \otimes \mathrm{I}_k)$$

is called *left mixed Kronecker product.*

4.4.1 Algebraic Properties of Stride Permutations

Property 4.17 (Identity)

$$\mathrm{L}_1^n = \mathrm{L}_n^n = \mathrm{I}_n$$

Property 4.18 (Inversion/Transposition) If $N = mn$ the

$$(\mathrm{L}_m^{mn})^{-1} = (\mathrm{L}_m^{mn})^\top = \mathrm{L}_n^{mn}\,.$$

Property 4.19 (Multiplication) If $N = kmn$ then

$$\mathrm{L}_k^{kmn}\, \mathrm{L}_m^{kmn} = \mathrm{L}_m^{kmn}\, \mathrm{L}_k^{kmn} = \mathrm{L}_{km}^{kmn}\,.$$

Example 4.12 (Inversion of the Perfect Shuffle Permutation) The inverse matrix of $\mathrm{L}_2^{2^i}$ is given by the perfect shuffle permutation:

$$(\mathrm{L}_2^{2^i})^{-1} = \mathrm{L}_{2^{i-1}}^{2^i} = \Pi_{2^i}.$$

As already mentioned, the Kronecker product is not commutative. However, with the aid of stride permutations, the order of factors can be reverted.

Theorem 4.1 (Commutation) If $A \in \mathbb{C}^{m_1 \times n_1}$ and $B \in \mathbb{C}^{m_2 \times n_2}$ then

$$\mathrm{L}_{m_1}^{m_1 m_2}(A \otimes B) = (B \otimes A)\, \mathrm{L}_{n_2}^{n_1 n_2}\, .$$

Proof: Johnson et al. [60].

Several special cases are worth noting.

Corollary 4.4 If $A \in \mathbb{C}^{m \times m}$ and $B \in \mathbb{C}^{n \times n}$ then

$$A \otimes B = \mathrm{L}_{m}^{mn}(B \otimes A)\, \mathrm{L}_{n}^{mn} = (B \otimes A)^{\mathrm{L}_{m}^{mn}}.$$

Application of this relation leads to

$$\begin{aligned} \mathrm{I}_m \otimes B &= \mathrm{L}_{m}^{mn}(B \otimes \mathrm{I}_m)\, \mathrm{L}_{n}^{mn} = (B \otimes \mathrm{I}_m)^{\mathrm{L}_{n}^{mn}}, \\ A \otimes \mathrm{I}_n &= \mathrm{L}_{m}^{mn}(\mathrm{I}_n \otimes A)\, \mathrm{L}_{n}^{mn} = (\mathrm{I}_n \otimes A)^{\mathrm{L}_{n}^{mn}}. \end{aligned}$$

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 $A \otimes \mathrm{I}_n$ into the parallel Kronecker factor $\mathrm{I}_n \otimes A$ and the parallel Kronecker factor $\mathrm{I}_m \otimes B$ into the vector Kronecker factor $B \otimes \mathrm{I}_m$. Continuing this way, it is possible to write

$$\begin{aligned} A \otimes B &= (A \otimes \mathrm{I}_n)(\mathrm{I}_m \otimes B) \\ &= \mathrm{L}_{m}^{mn}(\mathrm{I}_n \otimes A)\, \mathrm{L}_{n}^{mn}(\mathrm{I}_m \otimes B), \end{aligned} \tag{4.3}$$

which can be used to compute the action of $A \otimes B$ as a sequence of two parallel Kronecker factors. It also holds that

$$A \otimes B = (A \otimes \mathrm{I}_n)\, \mathrm{L}_{m}^{mn}(B \otimes \mathrm{I}_m)\, \mathrm{L}_{n}^{mn}, \tag{4.4}$$

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 4.20 If $A \in \mathbb{C}^{m \times m}$ and $B \in \mathbb{C}^{n \times n}$ then

$$A \otimes B = \mathrm{L}_{m}^{nm}(\mathrm{I}_n \otimes A)\, \mathrm{L}_{n}^{mn}(\mathrm{I}_m \otimes B).$$

Property 4.21 If $N = kmn$ then

$$\mathrm{L}_{n}^{kmn} = (\mathrm{L}_{n}^{kn} \otimes \mathrm{I}_m)(\mathrm{I}_k \otimes \mathrm{L}_{n}^{mn}).$$

Property 4.22 If $N = kmn$ then

$$\mathrm{L}^{kmn}_{km} = (\mathrm{I}_k \otimes \mathrm{L}^{mn}_{m})(\mathrm{L}^{kn}_{k} \otimes \mathrm{I}_m).$$

Property 4.23 If $N = kmn$ then

$$(\mathrm{L}^{km}_{m} \otimes \mathrm{I}_n) = (\mathrm{I}_m \otimes \mathrm{L}^{kn}_{k})\, \mathrm{L}^{kmn}_{mn} \quad .$$

Proof: Using Properties 4.18 and 4.22 lead to

$$(\mathrm{L}^{km}_{m} \otimes \mathrm{I}_n) = (\mathrm{I}_m \otimes \mathrm{L}^{km}_{k})(\mathrm{I}_m \otimes \mathrm{L}^{km}_{m})(\mathrm{L}^{km}_{m} \otimes \mathrm{I}_n) = (\mathrm{I}_m \otimes \mathrm{L}^{kn}_{k})\, \mathrm{L}^{kmn}_{mn} . \qquad \square$$

Property 4.24 If $N = kmn$ then

$$(\mathrm{L}^{km}_{m} \otimes \mathrm{I}_n)(\mathrm{I}_k \otimes \mathrm{L}^{mn}_{m})\, \mathrm{L}^{kmn}_{k} = (\mathrm{I}_m \otimes \mathrm{L}^{kn}_{k})(\mathrm{L}^{mn}_{m} \otimes \mathrm{I}_k) \quad .$$

Proof: Using Property 4.23 leads to

$$(\mathrm{I}_m \otimes \mathrm{L}^{kn}_{k})\, \mathrm{L}^{kmn}_{mn}(\mathrm{I}_k \otimes \mathrm{L}^{mn}_{m})\, \mathrm{L}^{kmn}_{k} = (\mathrm{I}_m \otimes \mathrm{L}^{kn}_{k})(\mathrm{L}^{mn}_{m} \otimes \mathrm{I}_k). \qquad \square$$

The following two properties show, how the mixed Kronecker product can be decomposed. Property 4.25 shows the more general case and Property 4.26 shows the full factorization.

Property 4.25

$$(\mathrm{I}_{sk} \otimes A_{ms\times n})\, \mathrm{L}^{skn}_{sk} = \Big(\mathrm{I}_k \otimes \mathrm{L}^{ms^2}_{s}\, (A_{ms\times n} \otimes \mathrm{I}_s)\Big)\, \Big(\mathrm{L}^{kn}_{k} \otimes \mathrm{I}_s\Big)$$

Property 4.26

$$(\mathrm{I}_{sk} \otimes A_{ms\times n})\, \mathrm{L}^{skn}_{sk} = \Big(\mathrm{I}_k \otimes (\mathrm{L}^{ms}_{s} \otimes \mathrm{I}_s)\, \Big(\mathrm{I}_m \otimes \mathrm{L}^{s^2}_{s}\Big)\, (A_{ms\times n} \otimes \mathrm{I}_s)\Big)\, \Big(\mathrm{L}^{kn}_{k} \otimes \mathrm{I}_s\Big)$$

4.4.2 Digit Permutations

The following permutation generalizes the stride permutation.

Definition 4.15 (Digit Permutation) Let $N = N_1 N_2 \cdots N_k$ and let σ be a permutation of the numbers $1, 2, \ldots, k$. Then the digit permutation is defined by

$$\mathrm{L}^{(N_1,\ldots,N_k)}_{\sigma}(e^{N_1}_{i_1} \otimes \cdots \otimes e^{N_k}_{i_k}) = (e^{N_{\sigma(1)}}_{i_{\sigma(1)}} \otimes \cdots \otimes e^{N_{\sigma(k)}}_{i_{\sigma(k)}}).$$

Theorem 4.2 (Permutation) Let A_0, A_1,…,A_k be $N_i \times N_i$ matrices and let σ be a permutation of the numbers $1, 2, \ldots, k$, then

$$A_1 \otimes \cdots \otimes A_k = (\mathrm{L}^{(N_1,\ldots,N_k)}_{\sigma})^{-1}(A_{\sigma(1)} \otimes \cdots \otimes A_{\sigma(k)})\, \mathrm{L}^{(N_1,\ldots,N_k)}_{\sigma} .$$

Proof: Johnson et al. [60].

Digit reversal is a special permutation arising in FFT algorithms.

Definition 4.16 (Digit Reversal Matrix) The k-digit digit reversal permutation matrix

$$\mathrm{R}^{(N_1,N_2,\ldots,N_k)}$$

of size $N = N_1 N_2 \cdots N_k$ is defined by

$$\mathrm{R}^{(N_1,\ldots,N_k)}(e_{i_1}^{N_1} \otimes \cdots \otimes e_{i_k}^{N_k}) = e_{i_k}^{N_k} \otimes \cdots \otimes e_{i_1}^{N_1}.$$

The special case when $N_1 = N_2 = \cdots = N_k = p$ is denoted by R_{p^k}.

Theorem 4.3 The digit reversal matrix R_{p^k} satisfies recursion

$$\mathrm{R}_{p^k} = \prod_{i=2}^{k} (\mathrm{I}_{p^{k-i}} \otimes \mathrm{L}_p^{p^i}).$$

Proof: Johnson et al. [60].

4.5 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 4.17 (Twiddle Factor Matrix) Let $\omega_N = e^{2\pi i/N}$ denote the Nth root of unity. The twiddle factor matrix, denoted by T_m^{mn}, is a diagonal matrix defined by

$$\mathrm{T}_m^{mn}(e_i^m \otimes e_j^n) = \omega_{mn}^{ij}(e_i^m \otimes e_j^n), \quad i = 0, 1, \ldots, m-1,\ j = 0, 1, \ldots, n-1,$$

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

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

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 4.5 (Conjugating Diagonal Matrices) Let

$$D = \operatorname{diag}(d_0, d_1, \dots, d_{N-1})$$

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.,

$$D^{P_\sigma} = P_\sigma^{-1}\, D\, P_\sigma = \operatorname{diag}(d_{\sigma(0)}, d_{\sigma(1)}, \dots, d_{\sigma(N-1)}) = \bigoplus_{i=0}^{N-1} d_{\sigma(i)}.$$

Corollary 4.6 (Conjugating Twiddle Factors) Conjugating T_m^{mn} by L_m^{mn} results in T_n^{mn}, i. e.,

$$(\mathrm{T}_m^{mn})^{\mathrm{L}_m^{mn}} = \mathrm{T}_n^{mn}\,.$$

Tensor bases are a useful tool to compute the actual entries of conjugated twiddle factor matrices.

Property 4.27 (Twiddle Factor $\mathbf{I}_r \otimes \mathbf{T}_m^{mn}$)

$$(\mathrm{I}_r \otimes \mathrm{T}_m^{mn})(e_i^r \otimes e_j^m \otimes e_k^n) = \omega_{mn}^{jk}(e_i^r \otimes e_j^m \otimes e_k^n),$$

$$\mathrm{I}_r \otimes \mathrm{T}_m^{mn} = \bigoplus_{i=0}^{r-1}\bigoplus_{j=0}^{m-1}\bigoplus_{k=0}^{n-1} \omega_{mn}^{jk} = \bigoplus_{i=0}^{r-1}\bigoplus_{j=0}^{m-1} \Omega_{n,j}(\omega_{mn})$$

$(\mathrm{I}_r \otimes \mathrm{T}_m^{mn})^P$ 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 4.13 (Conjugation of Twiddle Factors) Consider the construct

$$(\mathrm{I}_4 \otimes \mathrm{T}_4^8)^{\mathrm{L}_8^{32}} = \mathrm{L}_4^{32}(\mathrm{I}_4 \otimes \mathrm{T}_4^8)\,\mathrm{L}_8^{32}\,.$$

Thus, $\mathrm{I}_4 \otimes \mathrm{T}_4^8$ is conjugated by L_8^{32}. Computation of the result yields

$$\begin{aligned}
(\mathrm{L}_4^{32}(\mathrm{I}_4 \otimes \mathrm{T}_4^8)\,\mathrm{L}_8^{32})(e_i^4 \otimes e_j^4 \otimes e_k^2) &= (\mathrm{L}_4^{32}(\mathrm{I}_4 \otimes \mathrm{T}_4^8))(e_j^4 \otimes e_k^2 \otimes e_i^4) \\
&= \mathrm{L}_4^{32}\,\omega_8^{ki}(e_j^4 \otimes e_k^2 \otimes e_i^4) \\
&= \omega_8^{ki}(e_i^4 \otimes e_j^4 \otimes e_k^2)
\end{aligned}$$

$$\begin{aligned}
\mathrm{L}_4^{32}(\mathrm{I}_4 \otimes \mathrm{T}_4^8)\,\mathrm{L}_8^{32} &= \bigoplus_{i=0}^{3}\bigoplus_{j=0}^{3}\bigoplus_{k=0}^{1} \omega_8^{ik} = \bigoplus_{i=0}^{3}\bigoplus_{j=0}^{3} \Omega_{2,j}(\omega_8) \\
&= \operatorname{diag}(1,1,1,\omega_8,1,\omega_8^2,1,\omega_8^3,1,1,1,\omega_8,1,\omega_8^2,1,\omega_8^3, \\
&\qquad 1,1,1,\omega_8,1,\omega_8^2,1,\omega_8^3,1,1,1,\omega_8,1,\omega_8^2,1,\omega_8^3).
\end{aligned}$$

4.6 Complex Arithmetic as Real Arithmetic

The previous sections dealt with matrices featuring complex entries. In this thesis, most operations are considered to be real, as conventional computer hardware can only handle real numbers directly. Thus $A \in \mathbb{C}^{m \times n}$ has to be transformed into $\overline{A} \in \mathbb{R}^{2m \times 2n}$. This is especially required for vector instructions, as there alignment and stride issues are crucial.

Vector instructions provide only *real* arithmetic of vectors with their elements stored contiguously in memory. Thus, to map formulas to vector code, complex formulas and matrices are to be translated into real ones. The most commonly used data format is the *interleaved complex format* (alternately real and imaginary part). It will be expressed formally as a mapping of formulas. The fact that the complex multiplication $(u + iv) \times (y + iz)$ is equivalent to the real multiplication

$$\begin{pmatrix} u & -v \\ v & u \end{pmatrix} \begin{pmatrix} y \\ z \end{pmatrix}$$

will be used. Thus, the complex matrix-vector multiplication $M\,x \in \mathbb{C}^n$ corresponds to the real operation $\overline{M}\,\tilde{x} \in \mathbb{R}^{2n}$, where $\overline{M}$ arises from M by replacing every entry $u + iv$ by the corresponding (2×2)-matrix above, and $\tilde{x}$ is in interleaved complex format. To evaluate the bar operator $\overline{(\;)}$ of a formula, a set of identities is needed which will be described in the next section.

Other complex data formats can also be expressed in this way. For instance, the *split complex format* (where a complex vector is stored as a vector of real parts followed by a vector of imaginary parts) can be expressed by an operator $\widetilde{(\;)}$ with

$$\widetilde{A} = \overline{A}^{\mathrm{L}^{2N}_{N}}\,.$$

4.6.1 Algebraic Properties of the Bar Operator

In this thesis the machine parameter ν denotes the vector length of a short vector SIMD architecture (see Chapter 3).

Property 4.28 (Distribution) For a matrix product the bar operator distributes over the factors, i. e.,

$$\overline{A\,B} = \overline{A}\,\overline{B}.$$

Property 4.29 (Real Matrix Property) If A is a *real* matrix, then

$$\overline{A} = A \otimes \mathrm{I}_2\,.$$

Property 4.30 (Parallel Kronecker Product) In a parallel Kronecker product, the bar operator can be removed from the identity matrix:

$$\overline{\mathrm{I}_n \otimes A_m} = \mathrm{I}_n \otimes \overline{A_m}.$$

The distribution of the bar operator over vector Kronecker products leaves a degree of freedom, which is crucial for the generation of fast vector code. The next three properties show this degree of freedom.

Property 4.31 (General Vector Kronecker Product) In a vector Kronecker product, the bar operator can be removed from the identity matrix:

$$\overline{A_m \otimes \mathrm{I}_n} = (\overline{A_m} \otimes \mathrm{I}_n)^{\left(\mathrm{I}_m \otimes \mathrm{L}_2^{2n}\right)}.$$

Property 4.32 (Vector Kronecker Product) The bar operator can be removed from the identity matrix:

$$\overline{A_m \otimes \mathrm{I}_\nu} = (\overline{A_m} \otimes \mathrm{I}_\nu)^{\left(\mathrm{I}_m \otimes \mathrm{L}_2^{2\nu}\right)}.$$

Property 4.33 (Vector Kronecker Product) For n being a multiple of ν, the bar operator can be partly removed from the identity matrix:

$$\overline{A_m \otimes \mathrm{I}_n} = (\overline{A_m \otimes \mathrm{I}_{\frac{n}{\nu}}} \otimes \mathrm{I}_\nu)^{\left(\mathrm{I}_{\frac{mn}{\nu}} \otimes \mathrm{L}_2^{2\nu}\right)}.$$

Vector Diagonals

The application of the bar operator to complex diagonals does not feature the same structure as vector Kronecker products. However, this vector Kronecker structure is required by the target hardware. Applying the correct conjugation, the vector Kronecker structure can be obtained.

Definition 4.18 (The Bar-Prime Operator) For a complex matrix

$$D = \operatorname{diag}(c_0, \dots, c_{\nu-1})$$

with $c_i = a_i + ib_i$, the bar-prime operator $\overline{(\ )}'$ is defined by

$$\overline{D}' := \overline{D}^{\mathrm{L}_\nu^{2\nu}} = \begin{pmatrix} a_0 & & & -b_0 & & \\ & \ddots & & & \ddots & \\ & & a_{\nu-1} & & & -b_{\nu-1} \\ b_0 & & & a_0 & & \\ & \ddots & & & \ddots & \\ & & b_{\nu-1} & & & a_{\nu-1} \end{pmatrix}.$$

The construct $\overline{D}'$ has the same structure as

$$A \otimes \mathrm{I}_\nu$$

with $A \in \mathbb{R}^{2\times 2}$—only the actual numbers vary—and thus is a vector operation with vector length ν.

Definition 4.19 (Vector Diagonals) For n being a multiple of ν, the complex diagonal

$$D = \mathrm{diag}(c_0, \ldots, c_{n-1})$$

with $c_i = a_i + ib_i$ is divided into n/ν diagonal matrices $D_i \in \mathbb{C}^{\nu\times\nu}$:

$$D = \bigoplus_{i=0}^{\frac{n}{\nu}-1} D_i.$$

The application of the the bar-prime operator $\overline{(\;)}'$ is defined by

$$\overline{D}' := \bigoplus_{i=0}^{\frac{n}{\nu}-1} \overline{D_i}' = \overline{D}^{\left(\mathrm{I}_{\frac{n}{\nu}} \otimes \mathrm{L}_{\nu}^{2\nu}\right)}. \tag{4.5}$$

A construct matching (4.5) is called *vector diagonal.*

The construct $\overline{D}'$ has the same structure as

$$(\mathrm{I}_{\frac{n}{\nu}} \otimes A) \otimes \mathrm{I}_\nu$$

with $A \in \mathbb{R}^{2\times 2}$—(only the actual numbers vary), and thus is again a vector operation with vector length ν.

Example 4.14 (Twiddle Matrix) To implement the complex twiddle diagonal

$$\mathrm{T}_n^{mn}$$

it is divided into subdiagonals D_i of length ν and Definition 4.19 is applied:

$$\overline{\mathrm{T}}'^{mn}_{n} := \overline{\mathrm{T}}_n^{mn\left(\mathrm{I}_{\frac{mn}{\nu}} \otimes \mathrm{L}_{\nu}^{2\nu}\right)} = \bigoplus_{i=0}^{\frac{mn}{\nu}-1} \overline{D}_i', \quad \nu \mid mn. \tag{4.6}$$

4.7 Kronecker Product Code Generation

Kronecker products and direct sums have a natural interpretation as programs. A Kronecker product formula that constructs a matrix M can be interpreted as the operation

$$y := M\, x$$

with suitable vectors x and y. The formula accounts for the fact that M is a structured matrix whose structure is used to compute the matrix-vector product efficiently. In the following, algorithms are given for important constructs occurring in formulas for discrete linear transforms. Details can be found in Johnson et al. [60] and Moura et al. [72].

The RW Notation

To enable a direct translation of the RW notation into programs, a special subprogram for implementing the formula

$$y := \sum_{i=0}^{n} \mathrm{W}_{b(i),s}^{N,k}\ x, \tag{4.7}$$

whose base address $b(i)$ is a function of the loop iteration, is required. Whenever this construct is used in this thesis, $b(i)$ has the following property.

Property 4.34 (Loop Independence) A sum of type (4.7) can be translated into an loop with independent iterations, if for any $r \in \{0, 1, \ldots, N-1\}$ exactly *one* iteration $i \in \{0, 1, \ldots, n\}$ and a unique offset $j \in \mathbb{N}$ exists such that

$$r = b(i) + js.$$

Thus, the sum in (4.7) acts as *loop*. The kth component of y is computed by a sum where exactly *one* summand is not zero, which follows from Property 4.34. By *storing* the respective kth component of the intermediate result

$$\mathrm{W}_{b(i),s}^{N,k}\ x$$

(where the component is known to be non-zero) in the respective loop iteration i into the kth component of y these additions can be omitted.

A single iteration i of (4.7) can be implemented using the *update* function. For simplicity set $b := b(i)$. All nonzero entries of the vector $\mathrm{W}_{b,s}^{N,k}\ x$ are stored into the respective entries of y while all other entries of y remain unchanged, i. e., only entries with indices $b + js$ are copied.

Algorithm 4.1 ($\mathbf{update(y, W_{b,s}^{N,k}\ x)}$)

```
do $i = 0,\ N-1$
  do $j = 0,\ k-1$
    if $\mathrm{W}_{b,s}^{N,k}(i,j) = 1$ then $y(i) := x(j)$
  end do
end do
```

Using the update function given by Algorithm 4.1, equation (4.7) can be implemented as a loop using the following algorithm.

Algorithm 4.2 ($\mathbf{y := \sum_{i=0}^{n} W_{b(i),s}^{N,k}\ x}$)

```
do $i = 0,\ n$
  update$(y, \mathrm{W}_{b(i),s}^{N,k}\ x)$
end do
```

The implementation of sums as loops by Algorithm 4.2 is used throughout this section to obtain implementations of tensor products and stride permutations.

Direct Sums of Matrices

If $x \in \mathbb{C}^N$, $y \in \mathbb{C}^M$, $A \in \mathbb{C}^{M \times N}$ with $x = x_0 \oplus x_1$ with $x_0 \in \mathbb{C}^{n_0}$, $x_1 \in \mathbb{C}^{n_1}$, $y = y_0 \oplus y_1$ with $y_0 \in \mathbb{C}^{m_0}$, $y_1 \in \mathbb{C}^{m_1}$, and $A = A_0 \oplus A_1$ with $A_0 \in \mathbb{C}^{m_0 \times n_0}$, $A_1 \in \mathbb{C}^{m_1 \times n_1}$ then the following algorithm can be used for computing $y := A\,x$.

Algorithm 4.3 ($\mathbf{y := (A_0 \oplus A_1)\,x}$)

```
  $y_0 := A_0\,x_0$
  $y_1 := A_1\,x_1$
```

Corollary 4.7 (Direct Sum) A direct sum

$$A_0 \oplus A_1 \oplus \cdots \oplus A_{k-1} \quad \text{with} \quad A_i \in \mathbb{C}^{m \times n}$$

can be written as a sum using the read and write operators $\mathrm{R}_{b,s}^{N,n}$ and $\mathrm{W}_{b,s}^{N,m}$:

$$(A_0 \oplus A_1 \oplus \cdots \oplus A_{k-1})\,x = \sum_{i=0}^{k-1} \mathrm{W}_{im,1}^{mk,m}\, A_i\, \mathrm{R}_{in,1}^{nk,n}\; x.$$

Applying Algorithm 4.2 leads to the following algorithm.

Algorithm 4.4 ($\mathbf{y := (\bigoplus_{i=0}^{k-1} A_i)\,x}$)

```
  do $i = 0,\, k-1$
    $t := \mathrm{R}_{in,1}^{nk,n}\; x$
    $t' := A_i\, t$
    update$(y, \mathrm{W}_{im,1}^{mk,m}, t')$
  end do
```

Kronecker products with identity matrices represent loops in a natural way. According to Corollary 4.1 (on page 60) a general Kronecker product can be expressed as two Kronecker products with identity matrices. Thus, any Kronecker product can be factored into a product of a parallel Kronecker factor and a vector Kronecker factor. The implementation of these special Kronecker products is summarized in the following two sections.

Parallel Kronecker Products

Parallel Kronecker products as given by Definition 4.8 can be translated into a loop of independent operations on blocks. The iterations of these loops are independent and can be computed in parallel.

Corollary 4.8 (Parallel Kronecker Products) A parallel Kronecker product

$$\mathrm{I}_k \otimes A \quad \text{with} \quad A \in \mathbb{C}^{m \times n}$$

can be written as a sum using the read and write operators $\mathrm{R}_{b,s}^{N,n}$ and $\mathrm{W}_{b,s}^{N,n}$:

$$(\mathrm{I}_k \otimes A)\, x = \sum_{i=0}^{k-1} \mathrm{W}_{im,1}^{mk,m}\, A\, \mathrm{R}_{in,1}^{nk,n}\; x.$$

Applying Algorithm 4.2 leads to the following algorithm.

Algorithm 4.5 ($\mathbf{y := (I_k \otimes A)\, x}$)

```
  **do** $i = 0,\ k-1$
    $t := \mathrm{R}_{in,1}^{nk,n}\; x$
    $t' := A\, t$
    **update**$(y, \mathrm{W}_{im,1}^{mk,m}, t')$
  **end do**
```

Vector Kronecker Products

Vector Kronecker products as given by Definition 4.9 can be translated into operations on vectors. These operations on vectors can be implemented using a loop where the respective elements in consecutive loop iterations are accessed at unit stride. This is achieved by $b(i) = i$. Such loops are called *vectorizable* and can be implemented efficiently using vector hardware.

Corollary 4.9 (Vector Kronecker Product) A vector Kronecker product

$$A \otimes \mathrm{I}_k \quad \text{with} \quad A \in \mathbb{C}^{m\times n}$$

can be written as sum using the read and write operators $\mathrm{R}_{b,s}^{N,n}$ and $\mathrm{W}_{b,s}^{N,n}$:

$$(A \otimes \mathrm{I}_k)\, x = \sum_{i=0}^{k-1} \mathrm{W}_{i,k}^{mk,m}\, A\, \mathrm{R}_{i,k}^{nk,n}\; x.$$

Applying Algorithm 4.2 leads to the following algorithm.

Algorithm 4.6 ($\mathbf{y := (A \otimes I_k)\, x}$)

```
  **do** $i = 0,\ k-1$
    $t := \mathrm{R}_{i,k}^{nk,n}\; x$
    $t' := A\, t$
    **update**$(y, \mathrm{W}_{i,k}^{mk,m}, t')$
  **end do**
```

Stride Permutations

Stride permutations can be translated into two different algorithms. Either (*i*) vector reads and parallel writes are used, or (*ii*) parallel reads and vector writes. Thus, a stride permutation features both vector and parallel characteristics. This leads to difficulties for implementations on both vector and parallel computers.

Corollary 4.10 (Stride Permutation) A stride permutation L_m^{mn} can be written as a sum using the read and write operators $\mathrm{R}_{b,s}^{N,n}$ and $\mathrm{W}_{b,s}^{N,n}$:

$$\mathrm{L}_m^{mn}\ x = \sum_{i=0}^{m-1} \mathrm{W}_{in,1}^{mn,n}\, \mathrm{R}_{i,m}^{mn,n}\ x.$$

Applying Algorithm 4.2 leads to the following algorithm featuring vector reads and parallel writes.

Algorithm 4.7 ($\mathbf{y := L_m^{mn}\, x}$)

```
do $i = 0,\ m-1$
  $t := \mathrm{R}_{i,m}^{mn,n}\ x$
  update$(y, \mathrm{W}_{in,1}^{mn,n}, t)$
end do
```

An alternative implementation is given by the following corollary and the respective algorithm.

Corollary 4.11 (Stride Permutation) A stride permutation L_m^{mn} can be written as a sum using the read and write operators $\mathrm{R}_{b,s}^{N,n}$ and $\mathrm{W}_{b,s}^{N,n}$:

$$\mathrm{L}_m^{mn}\ x = \sum_{i=0}^{n-1} \mathrm{W}_{i,n}^{mn,m}\, \mathrm{R}_{im,1}^{mn,m}\ x.$$

Applying Algorithm 4.2 leads to the following algorithm featuring parallel reads and vector writes.

Algorithm 4.8 ($\mathbf{y := L_m^{mn}\, x}$)

```
do $i = 0,\ n-1$
  $t := \mathrm{R}_{im,1}^{mn,m}\ x$
  update$(y, \mathrm{W}_{i,n}^{mn,m}, t)$
end do
```

A comparison between Algorithms 4.7 and 4.8 and Algorithms 4.5 and 4.6 shows the connection between parallel and vector Kronecker products and stride permutations.

Mixed Kronecker Products

Both left and right mixed Kronecker products can be translated into sums utilizing the methods developed for parallel and vector Kronecker products as well as for stride permutations. These sums can subsequently translated into algorithms for computing the application of mixed tensor products.

Left mixed Kronecker products and right mixed Kronecker products are equivalent, as the application of Corollary 4.4 (on page 63) shows that

$$(\mathrm{I}_k \otimes A)\,\mathrm{L}_k^{mk} = \mathrm{L}_k^{mk}(A \otimes \mathrm{I}_k).$$

The following corollary expresses mixed Kronecker products (both left and the respective right mixed Kronecker products) as sums using the RW notation.

Corollary 4.12 (Mixed Kronecker Product) Left and right mixed Kronecker products can can be written as sums using the read and write operators $\mathrm{R}_{b,s}^{N,n}$ and $\mathrm{W}_{b,s}^{N,n}$:

$$(\mathrm{I}_k \otimes A)\,\mathrm{L}_k^{mk}\; x = \mathrm{L}_k^{mk}(A \otimes \mathrm{I}_k)\, x = \sum_{i=0}^{k-1} \mathrm{W}_{im,1}^{mk,m}\, A\, \mathrm{R}_{i,k}^{mk,m}\; x.$$

Applying Algorithm 4.2 leads to the following algorithm.

Algorithm 4.9 ($\mathbf{y := (I_k \otimes A)L_k^{mk}\, x}$)

```
do $i = 0,\ k-1$
    $t := \mathrm{R}_{i,k}^{mk,m}\; x$
    $t := A\, t$
    update($y, \mathrm{W}_{im,1}^{mk,m}, t$)
end do
```

Algorithm 4.12 has the same access pattern as Algorithm 4.7, i. e., it reads with vector characteristics and writes with parallel characteristics.

4.7.1 The SPL Compiler

The SPIRAL system includes a formula translator called *SPL compiler* which translates SPIRAL's proprietary *signal processing language* (SPL) into C or Fortran code. SPIRAL uses the convention introduced at the beginning of this section and interprets formulas as matrix-vector products with structured matrices. Thus, all SPL programs can be interpreted as programs computing the corresponding matrix-vector products $x \mapsto M\,x$.

Figure 4.1 shows an example SPL program and Appendix D.1 contains both an SPL program and the respective C program generated by the SPL compiler.

SPL Constructs

SPL is used for a computer representation of discrete linear transform algorithms given as formulas. The following are the most important SPL constructs.

General Matrices. Three types of general matrices, supported by SPL, are most important in the context of this thesis: (*i*) generic matrices, (*ii*) generic diagonal matrices, and (*iii*) generic permutations. The respective SPL constructs are the following:

```
(compose
    (tensor
        (F 2)
        (I 2)
    )
    (T 4 2)
    (tensor
        (I 2)
        (F 2)
    )
    (L 4 2)
)
```

Figure 4.1: An algorithm for DFT_4 that computes $y = (\mathrm{DFT}_2 \otimes \mathrm{I}_2)\, \mathrm{T}^4_2 (\mathrm{I}_2 \otimes \mathrm{DFT}_2)\, \mathrm{L}^4_2\, x$ written as an SPL program.

```
(matrix ((a11 ... a1n) ... (am1 ... amn)) ; generic matrix,
(diagonal (a11 ... ann))                  ; generic diagonal matrix,
(permutation (k1 ... kn))                 ; generic permutation matrix.
```

Parameterized matrices. SPL supports all parameterized matrices that are required in the context of this thesis. Examples include (*i*) identity matrices I_n, (*ii*) DFT matrices DFT_n which are no further decomposed and denoted by F_n, (*iii*) stride permutations L^{mn}_n, and (*iv*) twiddle factor matrices T^{mn}_n. The respective SPL constructs are:

```
(I n)                  ; identity matrix,
(F n)                  ; discrete Fourier transform matrix,
(L mn n)               ; stride permutation matrix,
(T mn n)               ; twiddle factor matrix.
```

Matrix operations. Various matrix operations are supported by SPL, including (*i*) the matrix product, (*ii*) the Kronecker product, and (*iii*) the direct sum. The respective SPL constructs are:

```
(compose A1 ... An)        ; matrix product,
(tensor A1 ... An)         ; Kronecker product,
(direct_sum A1 ... An)     ; direct sum.
```

In addition to matrix constructs, SPL provides tags to control the SPL compilation process. For example, there are tags available to control the unrolling strategy or the datatype (real versus complex) to be used (Xiong et al. [95]). For example, the DFT_4 algorithm given in equation (5.1) on page 78 can be written in SPL as represented in Figure 4.1.

SPL Compilation

The SPL compiler translates SPL formulas into optimized C or Fortran code. The code is produced using standard optimization techniques and domain specific optimizations. Some optimizations like loop unrolling can be parameterized to allow SPIRAL's search module (see Figure 1.2 on page 22) to try different implementations of the same formula.

Stage 1. In the first stage, the SPL code is parsed and translated into a binary abstract syntax tree. The internal nodes represent *operators* like `tensor`, `compose`, or `direct_sum`. The leave nodes represent SPL *primitives* like `(I n)`, `(T mn n)`, and `(L mn n)`.

Within this stage, not only the SPL program is parsed, but also the definitions of the supported operators, primitives, symbols, and optimization techniques are loaded. These definitions are part of SPL and can be extended by the user.

All constructs used in leaves of abstract syntax trees are *symbols*. A symbol is a named abstract syntax tree, which is translated into a function call to compute this part of the formula.

Stage 2. In the next stage the abstract syntax tree is translated into an internal serial code representation (called i-code) using pattern matching against built-in templates. For example, any SPL formula matching the template

```
(tensor (I any) ANY)
```

is translated into a loop of the second argument of `tensor`. `any` is a wildcard for an integer (and the number of loop iterations), while `ANY` matches any SPL subformula. The template mechanism is also used to apply important optimizations for constructs like

```
(compose (tensor (I any) ANY) (T any any)).
```

In the optimization stage techniques like (partial) loop unrolling, dead code elimination, constant folding, and constant propagation are applied to improve the i-code. Special attention is paid to the use of temporary variables within the generated code. Optimizations are applied to minimize the dependencies between variables and, if possible, scalars are used instead of arrays.

Stage 3. In the last stage, the optimized i-code is unparsed to the target language C or Fortran. Various methods to handle constants and intrinsic functions are available.

Chapter 5

Fast Algorithms for Linear Transforms

This chapter defines discrete linear transforms, fast algorithms for discrete linear transforms, and summarizes the approach used in this thesis. The approach is based on Kronecker product factorizations of transform matrices and on recursive factorization rules. It mainly follows the methodology introduced by the SPIRAL team, but extends the way complex transforms are described and introduces a new way of describing FFTW's recursion in this context.

5.1 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, all discrete linear transforms are considered. The general method is demonstrated using the two-dimensional cosine transform and the Walsh-Hadamard transform as examples. However, 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 5.1 (Real Discrete Linear Transform) 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 the matrix-vector multiplication

$$y = M\,x.$$

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

Definition 5.2 (Complex Discrete Linear Transform) 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

$$y = M\,x.$$

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

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

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

Example 5.1 (DFT Matrix) The first five DFT matrices are

$$\mathrm{DFT}_1 = (1), \quad \mathrm{DFT}_2 = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \quad \mathrm{DFT}_3 = \begin{pmatrix} 1 & 1 & 1 \\ 1 & e^{-2\pi i/3} & e^{-4\pi i/3} \\ 1 & e^{-4\pi i/3} & e^{-2\pi i/3} \end{pmatrix}$$

$$\mathrm{DFT}_4 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix}, \quad \mathrm{DFT}_5 = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & e^{-2\pi i/5} & e^{-4\pi i/5} & e^{-6\pi i/5} & e^{-8\pi i/5} \\ 1 & e^{-4\pi i/5} & e^{-8\pi i/5} & e^{-2\pi i/5} & e^{-6\pi i/5} \\ 1 & e^{-6\pi i/5} & e^{-2\pi i/5} & e^{-8\pi i/5} & e^{-4\pi i/5} \\ 1 & e^{-8\pi i/5} & e^{-6\pi i/5} & e^{-4\pi i/5} & e^{-2\pi i/5} \end{pmatrix}.$$

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

Definition 5.4 (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

$$y = \mathrm{DFT}_N\, x.$$

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 a small number of the mathematical operators introduced in Chapter 4.

Example 5.2 (DFT_4) Consider a factorization, i. e., a fast algorithm, for DFT_4. Using the mathematical notation from Chapter 4 it follows that

$$\begin{aligned}
\mathrm{DFT}_4 &= \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{pmatrix} \\
&= \left(\begin{array}{cc|cc} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ \hline 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{array}\right) \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & i \end{pmatrix} \cdot \left(\begin{array}{cc|cc} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ \hline 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \end{array}\right) \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad (5.1) \\
&= (\mathrm{DFT}_2 \otimes \mathrm{I}_2) \quad \cdot \quad \mathrm{T}_2^4 \quad \cdot \quad (\mathrm{I}_2 \otimes \mathrm{DFT}_2) \quad \cdot \quad \mathrm{L}_2^4\,.
\end{aligned}$$

T_2^4 denotes a twiddle matrix as defined in Section 4.5, i. e.,

$$\mathrm{T}_2^4 = \mathrm{diag}(1, 1, 1, i).$$

L_2^4 denotes a stride permutation as defined in Section 4.4 which swaps the two middle elements x_1 and x_2 in a four-dimensional vector, i. e.,

$$\begin{pmatrix} x_0 \\ x_2 \\ x_1 \\ x_3 \end{pmatrix} = \mathrm{L}_2^4 \begin{pmatrix} x_0 \\ x_1 \\ x_3 \\ x_3 \end{pmatrix}.$$

Automatic Derivation of Fast Algorithms

In Egner and Püschel [23] a method is 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 (Egner and Püeschel [22]), a library for the computer algebra system GAP [88] used in SPIRAL. AREP is able to factorize transform matrices and to find fast algorithms automatically. In Püschel [78] 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 (Rao and Hwang [81]), as well as the fast Fourier transform (Cooley and Tukey [13]).

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 (Wang [92]).

Example 5.3 (Walsh-Hadamard Transforms) The WHT_N for $N = 2^k$ is given by

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

A particular example of a rule for this transform is

$$\mathrm{WHT}_{2^k} = \prod_{i=1}^{k} \left(\mathrm{I}_{2^{k_1+\cdots+k_{i-1}}} \otimes \mathrm{WHT}_{2^{k_i}} \otimes \mathrm{I}_{2^{k_{i+1}+\cdots+k_t}} \right), \quad k = k_1 + \cdots + k_t. \tag{5.2}$$

Example 5.4 (Discrete Cosine Transforms) The DCT_N for arbitrary N is given by

$$\mathrm{DCT}_N = \big(\cos\left((\ell + 1/2)k\pi/N\right) \mid k, \ell = 0, 1, \ldots, N-1 \big).$$

A corresponding rule is

$$\mathrm{DCT}_{2n} = P_{2n} \left(\mathrm{DCT}_n \oplus S_{2n} \, \mathrm{DCT}_n \; D_{2n} \right) P'_{2n} \left(\mathrm{I}_n \otimes \mathrm{DFT}_2 \right) P''_{2n}.$$

Example 5.5 (Discrete Fourier Transforms) A rule for the DFT_N matrix is given by

$$\mathrm{DFT}_{mn} = \left(\mathrm{DFT}_m \otimes \mathrm{I}_n \right) \mathrm{T}^{mn}_n \left(\mathrm{I}_m \otimes \mathrm{DFT}_n \right) \mathrm{L}^{mn}_m . \tag{5.3}$$

(5.3) is the Cooley-Tukey FFT written in the notation of Chapter 4 (Johnson et al. [60]). It will be discussed in more detail in Section 5.2 on page 83.

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 $M \otimes M$ as indicated by Corollary 4.1. Using the respective property of the tensor product, the rule

$$M \otimes M = \left(M \otimes \mathrm{I}_N \right) \left(\mathrm{I}_N \otimes M \right) \tag{5.4}$$

is obtained.

The set of rules used in SPIRAL is constantly growing. A set of important rules can be found in Püschel et al. [79] and Püschel et al. [80].

Formulas and Base Cases

Eventually a mathematical *formula* is obtained when all transforms are expanded into base cases. When this framework is used to express FFTW's Cooley-Tukey recursion (see Section 5.2.5), the base cases are defined differently from the way they are defined for use with SPIRAL and the newly developed short vector SIMD algorithms. For instance, in FFTW *codelets* which correspond to larger transforms are the base cases while in SPIRAL all formulas are fully expanded. Within the newly developed short vector Cooley-Tukey (see Section 7.3) a new type of base case called *vector terminal* is introduced. However, the general framework of having recursive rules and base cases is intrinsic to all three approaches.

Example 5.6 (Fully Expanded Formula for $\mathbf{WHT_8}$) According to rule (5.2), WHT_8 can fully be expanded into

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

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 5.1 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.

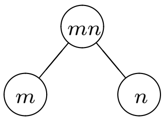


Figure 5.1: Tree representation of a discrete linear transform of size $N = mn$ with one recursion step applied.

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 5.2 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.

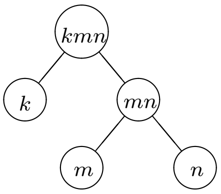


Figure 5.2: Right-expanded tree, two recursive steps.

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 s 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 (5.3) 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 (5.3) 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.

Formula Manipulation

A given formula for a fast linear transform algorithm can be manipulated using mathematical identities. Formula manipulation is used to exhibit the required formula structure throughout this thesis. The goal is to develop a short-vector specific set of formula manipulation rules which can be used to exhibit symbolically vectorizable subexpressions. The identities defined in Chapter 4 are the foundation for such manipulations. In Chapter 7 the short-vector specific manipulation rules are derived. Chapter 8 summarizes the inclusion of the newly developed rules into FFTW and SPIRAL.

Complex Transforms

In the classical approach using Kronecker product factorization, complex transforms are treated as complex valued entities throughout the formula manipulation. Only at the stage of the actual translation into a program a formula is coded using real arithmetic. When using programming languages featuring complex data types the formula is never translated into real arithmetic at source code level.

The actual hardware usually features only real arithmetic, and in case of short vector SIMD extensions, additional restrictions like data alignment and strides of real vectors are introduced.

Thus, throughout this thesis, the bar operator as defined in Section 4.6 is used to obtain the real formulation of complex discrete linear transforms.

Example 5.7 ($\overline{\mathbf{DFT}_4}$) The real version of the complex DFT_4 is given by

$$\overline{\mathrm{DFT}_4} = \left(\begin{array}{cc|cc|cc|cc} 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ \hline 1 & 0 & 0 & 1 & -1 & 0 & 0 & -1 \\ 0 & 1 & -1 & 0 & 0 & -1 & 1 & 0 \\ \hline 1 & 0 & -1 & 0 & 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 & 0 & 1 & 0 & -1 \\ \hline 1 & 0 & 0 & -1 & -1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & -1 & -1 & 0 \end{array}\right).$$

This introduces a new level of detail to the Kronecker product formalism, and allows for an even closer connection between formulas and hardware features.

5.2 The Fast Fourier Transform

In the last section, discrete linear transforms and the discrete Fourier transform were defined using the mathematical notation summarized in Chapter 4. It has been demonstrated that discrete linear transforms feature an intrinsic recursive structure.

In this section, different types of the Cooley-Tukey recursion are discussed and the split radix algorithm is introduced. The difference between conventional iterative algorithms and the recursive approach used by Spiral and Fftw is discussed. Conventional iterative algorithms and vector computer algorithms are summarized.

5.2.1 The Cooley-Tukey Recursion

In 1965 Cooley and Tukey [13] found the fast Fourier transform in a form which is restricted to problem sizes being powers of two. Other authors extended the idea from powers of two to arbitrarily composed numbers. A summary of the historic development can be found in Van Loan [90].

Historically, FFT algorithms were obtained by applying breakdown rules recursively and then manipulating the resulting formulas to obtain the respective iterative algorithms. In the context of this thesis, the rules are more important than the iterative algorithms. The respective FFT rules are named after the algorithm where they first occurred.

Due to the degree of freedom introduced by Corollary 4.4, four different versions of the Cooley-Tukey recursion rule exist. For both tensor products in Theorem 5.1, one can choose whether the identity matrix is the left or the right factor. As these recursion steps can be utilized in different ways, all Cooley-Tukey algorithms—both iterative and recursive ones—can be expressed using these four basic rules. Some of these algorithms require additional formula manipulation. Section 5.2.3 discusses the difference between the iterative and the recursive approach and gives examples of well-known iterative algorithms.

Decimation in Time FFTs

The following theorem leads to the so-called *decimation in time* (DIT) FFT algorithm when applied recursively. In the resulting formula the expressions in parenthesis are to be regrouped to form computational stages.

Theorem 5.1 (DIT Cooley-Tukey Rule) For $mn \geq 2$

$$\mathrm{DFT}_{mn} = (\mathrm{DFT}_m \otimes \mathrm{I}_n)\, \mathrm{T}_n^{mn} (\mathrm{I}_m \otimes \mathrm{DFT}_n)\, \mathrm{L}_m^{mn}\,.$$

Decimation in Frequency FFTs

The following theorem leads to the so-called *decimation in frequency* (DIF) FFT algorithm when applied recursively. In the resulting formula the expressions in parenthesis are to be regrouped to form computational stages.

Theorem 5.2 (DIF Cooley-Tukey Rule) For $mn \geq 2$

$$\mathrm{DFT}_{mn} = \mathrm{L}_n^{mn} (\mathrm{I}_m \otimes \mathrm{DFT}_n)\, \mathrm{T}_n^{mn} (\mathrm{DFT}_m \otimes \mathrm{I}_n).$$

4-Step FFTs

The next theorem leads to the so-called *four-step* or *vector* FFT algorithm. This rule usually is applied only once to exhibit the vector Kronecker product structure. The resulting sub-problems are usually not computed using this rule.

Theorem 5.3 (Vector Cooley-Tukey Rule) For $mn \geq 2$

$$\mathrm{DFT}_{mn} = (\mathrm{DFT}_m \otimes \mathrm{I}_n)\, \mathrm{T}_n^{mn}\, \mathrm{L}_m^{mn} (\mathrm{DFT}_n \otimes \mathrm{I}_m).$$

Based on this theorem a cache oblivious FFT algorithm can be obtained as described in Frigo et al. [34].

6-Step FFTs

The following theorem leads to the so-called *six-step* or *parallel* FFT algorithm. The rule is applied only once to exhibit the parallel Kronecker product structure. The resulting subproblems are usually not computed using this rule.

Theorem 5.4 (Parallel Cooley-Tukey Rule) For $mn \geq 2$

$$\mathrm{DFT}_{mn} = \mathrm{L}_m^{mn} (\mathrm{I}_n \otimes \mathrm{DFT}_m)\, \mathrm{L}_n^{mn}\, \mathrm{T}_n^{mn} (\mathrm{I}_m \otimes \mathrm{DFT}_n)\, \mathrm{L}_m^{mn}\,.$$

This rule was developed by Bailey [10]. A second recursive application of this rule is the prerequisite for parallel one-dimensional FFT algorithms which overlap communication and computation (Franchetti et al. [26], and Karner and Ueberhuber [63]).

5.2.2 Cooley-Tukey Vector Recursion

A recursion rule that enables operations on short vectors is given by the Cooley-Tukey vector recursion. This recursion rule is included in Fftw 2.1.3 as an experimental option. It turned out to be of vital importance for the short vector SIMD adaptation within Fftw and for short vector SIMD implementations of FFTs in general.

Theorem 5.5 (Cooley-Tukey Vector Recursion) For $N = kmn \geq 2$ the construct $(\mathrm{I}_m \otimes \mathrm{DFT}_n)\,\mathrm{L}_m^{mn}$ in Theorem 5.1 can be factored alternatively into

$$\begin{aligned}(\mathrm{I}_m \otimes \mathrm{DFT}_{kn})\,\mathrm{L}_m^{mkn} \;&=\; \mathrm{I}_m \otimes (\mathrm{DFT}_k \otimes \mathrm{I}_n)\,\mathrm{T}_n^{kn} \\ &\quad\; (\mathrm{L}_m^{mk} \otimes \mathrm{I}_n)(\mathrm{I}_k \otimes \underbrace{(\mathrm{I}_m \otimes \mathrm{DFT}_n)\,\mathrm{L}_m^{mn}}_{(a)})(\mathrm{L}_k^{kn} \otimes \mathrm{I}_m).\end{aligned}$$

Proof: Application of Theorem 5.1 to

$$(\mathrm{I}_k \otimes \mathrm{DFT}_{mn})\,\mathrm{L}_k^{kmn}$$

leads to

$$\underbrace{(\mathrm{I}_{mk} \otimes \mathrm{DFT}_n)}_{(1)}(\mathrm{I}_m \otimes \mathrm{L}_k^{kn})\,\mathrm{L}_m^{mkn}\,.$$

Applying Property 4.9 to construct (1) leads to

$$\begin{aligned}\mathrm{I}_{mk} \otimes \mathrm{DFT}_n \;&=\; (\mathrm{I}_k \otimes \mathrm{I}_m \otimes \mathrm{DFT}_n)^{\mathrm{L}_k^{mk} \otimes \mathrm{I}_n} \\ &=\; (\mathrm{L}_m^{mk} \otimes \mathrm{I}_n)(\mathrm{I}_k \otimes \mathrm{I}_m \otimes \mathrm{DFT}_n)(\mathrm{L}_k^{mk} \otimes \mathrm{I}_n)(\mathrm{I}_m \otimes \mathrm{L}_k^{kn})\,\mathrm{L}_m^{kkn}\,.\end{aligned}$$

Now Property 4.24 leads to

$$(\mathrm{L}_m^{mk} \otimes \mathrm{I}_n)(\mathrm{I}_k \otimes \mathrm{I}_m \otimes \mathrm{DFT}_n)(\mathrm{I}_k \otimes \mathrm{L}_m^{mn})(\mathrm{L}_k^{kn} \otimes \mathrm{I}_m)$$

which proves the theorem. □

Theorem 5.5 is "pushing" the factor I_m down the recursion to DFT_n. Construct (a) is of the same type as the original construct and thus the rule can be applied recursively. All tensor products except those in construct (a) are of vector type with vector lengths m and n and all stride permutations except the one in construct (a) occur as factors in such vector tensor products.

5.2.3 Iterative vs. Recursive FFT Algorithms

This section outlines two basic strategies in organizing FFT programs. For the implementation of an FFT algorithm there exist two divergent strategies for successively applying one of the theorems introducing the Cooley-Tukey rules.

Recursive Methods: The multiplication by the DFT matrix is performed by calling program modules which compute the subproblems according to the chosen rule. The child problems are further decomposed by recursively calling the same program again and again until the DFTs are small enough to be executed directly by optimized leaf routines. This approach is used by SPIRAL and FFTW.

Iterative Methods: The FFT code contains the entire matrix decomposition explicitly and manages all tasks directly. Thus, all recursive decomposition steps are flatted and the computation is done stagewise leading to conventional triple loop FFT implementations (Van Loan [90]) which perform the DFT computation stagewise and each stage requires an additional pass through the data vector. The respective formulas are given by Theorems 5.6 through 5.10.

Recursive FFTs

The approach to discrete linear transforms used in this thesis follows the foundations of the SPIRAL system and applies the recursion idea to all discrete linear transforms, including the DFT.

While divide-and-conquer is a standard formulation for FFTs in introductory texts, almost all non-adaptive high performance FFTs use an iterative implementation. This is due to the widespread opinion that recursive divide-and-conquer algorithms are too expensive. Traditionally, the required function calls are among the computationally most expensive instructions.

However, an intriguing feature of divide-and-conquer algorithms is that they should run well on computers with deep memory hierarchies without the need for blocking or tiling. Each successive "divide" step in the divide-and-conquer process generates subproblems that touch successively smaller portions of data. For any level of the memory hierarchy, there is a level of division below which all the data touched by the subproblem will fit into that level of the memory hierarchy. Therefore, a divide-and-conquer algorithm can be viewed as an algorithm that is blocked for all levels of the memory hierarchy.

In the recent development of computer systems any memory access became more and more expensive when compared to function calls. FFTW finally broke from the tradition of iterative implementations of FFTs for the sake of higher performance and implement a recursive hardware adaptive FFT computation.

SPIRAL applies recursion to all discrete linear transforms. It generates and optimizes one library function for each transform and problem size. As outlined in this chapter, Spiral intrinsically produces recursive codes. However, for better performance the recursion is partly unrolled while the data access pattern is preserved.

Iterative FFTs

All conventional (non-adaptive) FFT algorithms have an iterative structure. The first FFT algorithm published by Cooley and Tukey was a right-expanded radix-2 factorized FFT, its decomposition strategy was clear and explicitly implemented. For decades the execution of an FFT was seen as a sequence of computational stages; each stage corresponding to one factor of the products which define algorithms in the modern notation.

Typically, a Cooley-Tukey FFT algorithm for data vectors of length $N = 2^n$ is implemented in form of a *triple-loop*.

While the design and programming of such implementations is rather easy, performance may not be optimal because vector lengths vary from stage to stage and therefore the cache usage is suboptimal. Yet, as long as the input vector fits into the cache memory entirely the iterative strategy was superior because there is no overhead due to additional program organization which was expensive on earlier computer generations. On current computer systems such a clear statement is not possible any more.

An iterative algorithm obtained by repeated application of the DIT recursion rule with equal sizes is the single radix DIT algorithm.

Theorem 5.6 (Single-Radix DIT Factorization) For $N = p^n$

$$\mathrm{DFT}_{p^n} = \Big[\prod_{j=1}^{n} (\mathrm{I}_{p^{j-1}} \otimes \mathrm{DFT}_p \otimes \mathrm{I}_{p^{n-j}})(\mathrm{I}_{p^{j-1}} \otimes \mathrm{T}^{p^{n-j+1}}_{p^{n-j}})\Big]\, \mathrm{R}_{p^n}, \tag{5.5}$$

with

$$\mathrm{R}_{p^n} := \prod_{j=1}^{n-1} (\mathrm{I}_{p^{n-j-1}} \otimes \mathrm{L}^{p^{j+1}}_{p}) \,. \tag{5.6}$$

The permutation matrix R_{p^n} is said to be a *bit reversal matrix* as it orders the entries of the data vector numbered in the base of the radix (e. g., binary numbers for a radix-2 factorization) according to those numbers obtained by reversing the order of the digits of the position numbers (Van Loan [90]). This matrix is an example of a digit permutations introduced matrix (see Section 4.4).

Example 5.8 (Radix-2 DIT FFT Algorithm) A commonly used FFT algorithm for vector lengths $N = 2^k$ is the radix-2 DIT algorithm given by

$$\mathrm{DFT}_{2^n} = \Big[\prod_{j=1}^{n} (\mathrm{I}_{2^{j-1}} \otimes \mathrm{DFT}_2 \otimes \mathrm{I}_{2^{n-j}})(\mathrm{I}_{2^{j-1}} \otimes \mathrm{T}^{2^{n-j+1}}_{2^{n-j}})\Big]\, \mathrm{R}_{2^n} \,.$$

An iterative algorithm obtained by repeated application of the DIF recursion rule with the same sizes is the single radix DIF algorithm.

Theorem 5.7 (Single-Radix DIF Factorization) For $N = p^n$

$$\mathrm{DFT}_{p^n} = \mathrm{R}_{p^n}\Big[\prod_{j=1}^{n}(\mathrm{I}_{p^{n-j}} \otimes \mathrm{T}^{p^j}_{p^{j-1}})(\mathrm{I}_{p^{n-j}} \otimes \mathrm{DFT}_p \otimes \mathrm{I}_{p^{j-1}})\Big]\,. \tag{5.7}$$

If—in contrast to the single radix algorithms above—different values are used for the breakdown rule, an iterative mixed radix DIT algorithm is obtained.

Theorem 5.8 (Mixed-Radix DIT Factorization) For $N = p_1 p_2 \cdots p_n \geq 2$ and $p_0 := 1, p_{n+1} := 1$

$$\mathrm{DFT}_{p_1\cdots p_n} = \Big[\prod_{j=1}^{n}(\mathrm{I}_{p_0\cdots p_{j-1}} \otimes \mathrm{DFT}_{p_j} \otimes \mathrm{I}_{p_{j+1}\cdots p_{n+1}})(\mathrm{I}_{p_0\cdots p_{j-1}} \otimes \mathrm{T}^{p_j\cdots p_n}_{p_{j+1}\cdots p_{n+1}})\Big]\widetilde{\mathrm{R}}_{p_1\cdots p_n}$$

with

$$\widetilde{\mathrm{R}}_{p_1\cdots p_n} = \prod_{j=1}^{n-1}(\mathrm{I}_{p_0\cdots p_{n-j-1}} \otimes \mathrm{L}^{p_{n-j}\cdots p_n}_{p_{n-j}})\,.$$

The Stockham Algorithm

Any iterative Cooley-Tukey FFT algorithm is composed of a *computation phase* which is linear algebra-like and an *order phase* which does not execute arithmetic operations but just performs data reordering.

Several strategies have been developed to achieve structural adaptations of the FFT's computation phase to vector processors. Yet the order phase makes things difficult. The reordering due to the bit-reversal matrix can neither be applied in-place nor be vectorized efficiently. Consequently it decreases decisively the speedup obtained by vectorization of the computation phase.

Consequently, the adaptation of FFT algorithms to vector processors concentrates on developing methods to avoid an explicit order phase. To achieve this goal, the data must be ordered within the computational stages step by step, just in the manner to produce an ordered output.

The single-radix Stockham DIT algorithm (Swarztrauber [86], Van Loan [90]) is an example for FFT algorithms suitable for conventional vector computers.

Theorem 5.9 (Single-Radix Stockham DIT Factorization) For $N = p^n$, the Stockham DIT factorization (also called Stockham I factorization) is given by

$$\mathrm{DFT}_N = \prod_{j=1}^{n}(\mathrm{DFT}_p \otimes \mathrm{I}_{p^{n-1}})(\mathrm{T}^{p^{n-j+1}}_{p^{n-j}} \otimes \mathrm{I}_{p^{j-1}})(\mathrm{L}^{p^{n-j+1}}_{p} \otimes \mathrm{I}_{p^{j-1}})\,. \tag{5.8}$$

Transposition of the Stockham DIT factorization leads to the Stockham DIF FFT variant.

Theorem 5.10 (Single-Radix Stockham DIF Factorization) For $N = p^n$, the Stockham DIF factorization (also called Stockham II factorization) is given by

$$\mathrm{DFT}_N = \prod_{j=1}^{n} (\mathrm{L}^{p^j}_{p^{j-1}} \otimes \mathrm{I}_{p^{n-j}})(\mathrm{T}^{p^j}_{p^{j-1}} \otimes \mathrm{I}_{p^{n-j}})(\mathrm{DFT}_p \otimes \mathrm{I}_{p^{n-1}}) \,. \tag{5.9}$$

Vector Computer FFT Algorithms and Short Vector Extensions

The two Stockham FFT algorithms and the 4-step FFT algorithm have been designed specifically for conventional vector computers. In principle, these algorithms could be used also on processors featuring short vector SIMD extensions, but they have several drawbacks there.

Complex Arithmetic. These algorithms are formulated using complex matrices. However, as outlined in Section 5.1, it is necessary to reformulate complex transforms using real matrices and formulas to capture the level of details required for short vector SIMD extensions.

Vector Length and Stride. All three algorithms are optimized for *long vectors*. The Stockham algorithms are optimized for *fixed stride* but not for *unit stride* memory access. Accordingly, these algorithms do not produce good performance when running on short vector SIMD extensions.

Iterative Algorithms. The very nature of the two Stockham FFT algorithms and the 4-step FFT algorithm is an iterative one which conflicts with the requirements of Spiral and Fftwto support adaptivity.

Thus, algorithms specifically designed for conventional vector computers are not suitable for modern short vector SIMD extensions.

5.2.4 The Split-Radix Recursion

Split-radix FFT algorithms for transform lengths $N = 2^n$ have been introduced in the early eighties (Duhamel [19, 20], Duhamel and Hollmann [21], Sorensen et al. [85], Vetterli and Duhamel [91]).

The split-radix recursion rule differs from the Cooley-Tukey recursion, as it breaks down a larger transform into *three* smaller transforms, two of them being of half the size of the third. This rule leads to the lowest arithmetic complexity among all FFT algorithms and on current architectures it is used mainly for smaller kernel routines or recursion steps near to the end of the recursion. The split-radix FFT algorithm is recursive.

Split-radix algorithms use a particular splitting strategy which is based in principle on the Cooley-Tukey splitting. However, split-radix algorithms are based on a clever synthesis of FFT decompositions according to different radices which makes it impossible to obtain them using the classical Cooley-Tukey factorization. It rather follows the principle that independent parts of the FFT algorithm should be computed independently. Each part should use the optimum computational scheme, regardless of what scheme is used for other parts of the algorithm.

The split-radix approach reduces the number of arithmetic operations required to calculate the FFT in comparison to the Cooley-Tukey algorithm. The split-radix decomposition is constructed to produce as many trivial twiddle-factors as possible. Thus, in particular the number of multiplication operations is greatly reduced.

As empirical performance assessment shows, the number of arithmetic operations is no longer an adequate performance measure on current computer architectures. The split-radix algorithm is competitive with recursive Cooley-Tukey algorithms only for very small problem sizes or in kernel routines where the whole working set easily fits into cache or even in the register file.

The following theorem formulates the Split-radix DIF rule in the mathematical notation introduced in Chapter 4.

Theorem 5.11 (Split-Radix-2/4 DIF Rule) For $N = 2^n \geq 4$

$$\mathrm{DFT}_N = \mathrm{P}_N^{(2/4)}\ \mathrm{DFT}_N^{(2/4)}\ \mathrm{T}_N^{(2/4)}\ \mathrm{A}_N^{(2/4)}$$

with

$$\begin{aligned}
\mathrm{P}_N^{(2/4)} &= \mathrm{L}_{N/2}^{N}\,(\mathrm{I}_{N/2} \oplus \mathrm{L}_{N/4}^{N/2})\,,\\
\mathrm{DFT}_N^{(2/4)} &= \mathrm{DFT}_{N/2} \oplus (\mathrm{I}_2 \otimes \mathrm{DFT}_{N/4})\,,\\
\mathrm{T}_N^{(2/4)} &= \mathrm{I}_{N/2} \oplus \left(\Omega_{4,N/4} \oplus (\Omega_{4,N/4})^3\right)\,,\\
\mathrm{A}_N^{(2/4)} &= \left(\mathrm{I}_{N/2} \oplus (\mathrm{DFT}_2\ \Omega_{2,2} \otimes \mathrm{I}_{N/4})\right)(\mathrm{DFT}_2 \otimes \mathrm{I}_{N/2})\,,\\
\Omega_{m,n} &= \mathrm{diag}(1, \omega_{N/m}, \omega_{N/m}^2, \ldots, \omega_{N/m}^{n-1})\,.
\end{aligned}$$

Only the matrix $\mathrm{T}_N^{(2,4)}$ contains nontrivial twiddle-factors as $\Omega_{2,2} = \mathrm{diag}(1, -i)$.

5.2.5 The Cooley-Tukey Recursion in FFTW

Although the FFTW framework uses the Cooley-Tukey algorithm as specified in Theorem 5.1,

$$\mathrm{DFT}_{mn} = (\mathrm{DFT}_m \otimes \mathrm{I}_n)\ \mathrm{T}_n^{mn}\ (\mathrm{I}_m \otimes \mathrm{DFT}_n)\ \mathrm{L}_m^{mn},$$

a specific interpretation of the final formulas is required to achieve the order of the computation as it is done by FFTW.

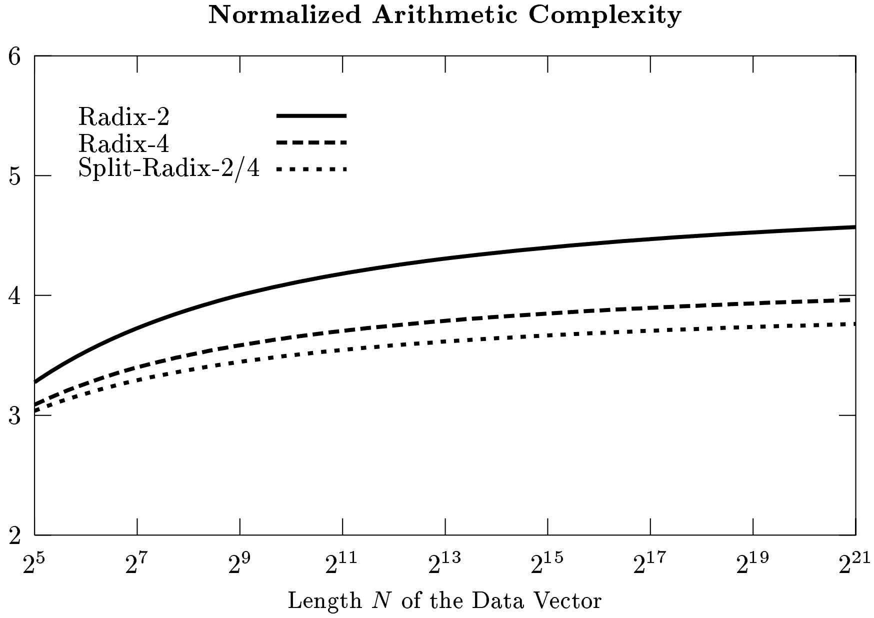


Figure 5.3: Comparison of the normalized arithmetic complexity ($\pi_R/N \log N$) of radix-2, radix-4, and split-radix-2/4 FFT algorithms (multiplications by trivial twiddle-factors can be avoided and are thus not included).

Fftw is the prototypical hardware adaptive recursive implementation of the Cooley-Tukey algorithm. Theorem 5.1 can also be written as

$$\mathrm{DFT}_{mn} = \left(\sum_{i=0}^{n-1} \mathrm{W}_{i,n}^{mn,m}\, \mathrm{DFT}_m\, \mathrm{R}_{i,n}^{mn,m}\right) \mathrm{T}_n^{mn} \left(\sum_{i=0}^{m-1} \mathrm{W}_{in,1}^{mn,n}\, \mathrm{DFT}_n\, \mathrm{R}_{i,m}^{mn,n}\right) \qquad (5.10)$$

using Corollaries 4.10, 4.8, and 4.9. Fftw does not use the Kronecker product formalism but uses the formulation as recursive algorithm.

The following algorithm describes the formulation used by Fftw using the framework of Chapter 4. Thus a connection between the Kronecker product formalism and the algorithmic formulation is established.

Algorithm 5.1 ($\mathbf{y := DFT_{mn}\, x}$)

```
do $i = 0, m - 1$
    $t_1 := \mathrm{R}_{i,m}^{mn,n}\ x$
    $t_1 := \mathrm{DFT}_n\ t_1$
    update($t', \mathrm{W}_{in,1}^{mn,n}, t_1$)
end do
$t' := \mathrm{T}_n^{mn}\ t'$
do $i = 0, n - 1$
    $t_2 := \mathrm{R}_{i,n}^{mn,m}\ t'$
    $t_2 := \mathrm{DFT}_m\ t_2$
    update($y, \mathrm{W}^{mn,m}_{i,n}, t_2$)
  end do
```

To apply the short vector SIMD vectorization derived in Chapter 7 to FFTW, it is required to use the formal connection between the Kronecker product formalism, the RW notation, and the algorithmic description. Throughout this chapter this connection is used, always starting from Kronecker products and finally leading to the actual code.

A major property of FFTW is its restriction to rightmost trees. Practical experiments show that rightmost trees perform well as they achieve memory locality (Haentjens [40]). Restriction to rightmost trees leads to only two different types of basic blocks required by FFTW: (*i*) *twiddle codelets* and (*ii*) *no-twiddle codelets*.

This section shows how the FFTW recursion, the twiddle codelets and the no-twiddle codelets interact to compose the Cooley-Tukey FFT algorithm.

FFTW's Codelets

Definition 5.5 (No-twiddle Codelet) An FFTW no-twiddle codelet of size m performs a DFT_m on m elements of the data vector of size N that are specified by the input and output base b_i and b_o, and the input and output access stride s_i and s_o:

$$\mathcal{F}^{\mathrm{NT}\ b_i,s_i,b_o,s_o}_{m,N} = \mathrm{W}^{N,m}_{b_o,s_o}\,\mathrm{DFT}_m\,\mathrm{R}^{N,m}_{b_i,s_i}\,.$$

Due to the different input and output strides, a no-twiddle codelet has to be computed out-of-place.

Definition 5.6 (Twiddle Codelet) An FFTW twiddle codelet of size m is defined by

$$\mathcal{F}^{\mathrm{TW}\ b,s,d}_{m,N,k} = \sum_{i=0}^{k-1} \mathrm{W}^{N,m}_{b+id,s}\,\mathrm{DFT}_m\,\mathrm{T}^{mk}_{k,i}\,\mathrm{R}^{N,m}_{b+id,s}\,.$$

It performs the operation

$$(\mathrm{DFT}_m \otimes \mathrm{I}_k)\,\mathrm{T}^{mk}_k = \sum_{i=0}^{k-1} \mathrm{W}^{mk,m}_{i,k}\,\mathrm{DFT}_m\,\mathrm{T}^{mk}_{k,i}\,\mathrm{R}^{mk,m}_{i,k} \tag{5.11}$$

with $\mathrm{T}^{mk}_{k,i}$ being defined by

$$\bigoplus_{i=0}^{k-1} \mathrm{T}^{mk}_{k,i} = \left(\mathrm{T}^{mk}_k\right)^{\mathrm{L}^{mk}_k}\,.$$

Corollaries 4.7 and 4.9, as well as Properties 4.4 and 4.5 provide the distributivity required for equation (5.11). Thus, a twiddle codelet operates on a subvector of size mk of a vector of size N that is accessed from base b at stride s and distance d, and applies scaling by T_k^{mk} combined with k DFTs of size m. A twiddle codelet can be computed in-place.

FFTW's Cooley-Tukey Recursion

FFTW's Cooley-Tukey recursion implements Theorem 5.1, i. e., Algorithm 5.1. In each recursion step, the parameter m is chosen such that DFT_m and the twiddle factors can be computed using twiddle codelets, and DFT_n is further decomposed recursively. Finally, DFT_n and the stride permutations are computed using no-twiddle codelets.

This section describes FFTW's Cooley-Tukey recursion using the Kronecker product formalism and RW notation.

Theorem 5.12 (FFTW's Cooley-Tukey Recursion) The recursion used in FFTW applies the Cooley-Tukey DIT rule

$$\mathrm{DFT}_{mn} = (\mathrm{DFT}_m \otimes \mathrm{I}_n)\ \mathrm{T}_n^{mn}\ (\mathrm{I}_m \otimes \mathrm{DFT}_n)\ \mathrm{L}_m^{mn}$$

to a subvector of size mn of a data vector of size N. The input is read from the base b_i with stride s_i and written to the base b_o at stride s_o. The recursion is given by

$$\mathrm{W}_{b_o,s_o}^{N,mn}\, \mathrm{DFT}_{mn}\, \mathrm{R}_{b_i,s_i}^{N,mn} = \mathcal{F}_{m,N,n}^{\mathrm{TW}\ b_o,ns_o,s_o} \sum_{i=0}^{m-1} \Big(\underbrace{\mathrm{W}_{b_o+ins_o,s_o}^{N,n}\, \mathrm{DFT}_n\, \mathrm{R}_{b_i+is_i,ms_i}^{N,n}}_{(a)} \Big).$$

Construct (a) is of the same structure as the original problem and thus the recursion can be applied again if $n = rs$. Once n becomes small enough, construct (a) can be computed using a no-twiddle codelet, i. e.,

$$\mathcal{F}_{n,N}^{\mathrm{NT}\ bi+is_i,ms_i,b_o+ins_o,s_o} = \mathrm{W}_{b_o+ins_o,s_o}^{N,n}\, \mathrm{DFT}_n\, \mathrm{R}_{b_i+is_i,ms_i}^{N,n}\,.$$

Proof: First, the Cooley-Tukey recursion (Theorem 5.1) is applied:

$$\mathrm{W}_{b_o,s_o}^{N,mn}\, \mathrm{DFT}_{mn}\, \mathrm{R}_{b_i,s_i}^{N,mn} = \mathrm{W}_{b_o,s_o}^{N,mn} (\mathrm{DFT}_m \otimes \mathrm{I}_n)\ \mathrm{T}_n^{mn}\ (\mathrm{I}_m \otimes \mathrm{DFT}_n)\ \mathrm{L}_m^{mn}\, \mathrm{W}_{b_o,s_o}^{N,mn}$$

Next, the tensor products, the stride permutation and the twiddle factor matrix are decomposed using Corollaries 4.7, 4.8, and 4.9, as well as Properties 4.4 and 4.5. This leads to

$$\mathrm{W}_{b_o,s_o}^{N,mn} \left(\sum_{i=0}^{n-1} \mathrm{W}_{i,n}^{mn,m}\, \mathrm{DFT}_m\, \mathrm{T}_{n,i}^{mn}\, \mathrm{R}_{i,n}^{mn,m} \right) \left(\sum_{i=0}^{m-1} \mathrm{W}_{in,1}^{mn,n}\, \mathrm{DFT}_n\, \mathrm{R}_{i,m}^{mn,n} \right) \mathrm{R}_{b_o,s_o}^{N,mn}\,.$$

Now Property 4.6 is used and thus

$$\mathrm{I}_{mn} = \mathrm{R}_{b_o,s_o}^{N,mn}\, \mathrm{W}_{b_o,s_o}^{N,mn}$$

is inserted between the two major factors. As both $\mathrm{R}^{N,mn}_{b_o,s_o}$ and $\mathrm{W}^{N,mn}_{b_o,s_o}$ as well as $\mathrm{W}^{N,mn}_{b_o,s_o}$ and $\mathrm{W}^{N,mn}_{b_o,s_o}$ are independent of i, these factors can be moved into the sums leading to the first factor

$$\sum_{i=0}^{n-1} \Big(\mathrm{W}^{N,mn}_{b_o,s_o}\, \mathrm{W}^{mn,m}_{i,n}\, \mathrm{DFT}_m\, \mathrm{T}^{mn}_{n,i}\, \mathrm{R}^{mn,m}_{i,n}\, \mathrm{R}^{N,mn}_{b_o,s_o} \Big)$$

and the second factor

$$\sum_{i=0}^{m-1} \Big(\mathrm{W}^{N,mn}_{b_o,s_o}\, \mathrm{W}^{mn,n}_{in,1}\, \mathrm{DFT}_n\, \mathrm{R}^{mn,n}_{i,m}\, \mathrm{R}^{N,mn}_{b_o,s_o} \Big)\,.$$

Applying Properties 4.4 and 4.5 leads to the following expression for the whole computation.

$$\sum_{i=0}^{n-1} \Big(\mathrm{W}^{N,m}_{b_o+is_o,ns_o}\, \mathrm{DFT}_m\, \mathrm{T}^{mn}_{n,i}\, \mathrm{R}^{N,m}_{b_o+is_o,ns_o} \Big) \sum_{i=0}^{m-1} \Big(\mathrm{W}^{N,n}_{b_o+ins_o,s_o}\, \mathrm{DFT}_n\, \mathrm{R}^{N,n}_{b_i+is_i,ms_i} \Big)\,.$$

Identifying the left sum with the definition of the twiddle codelet leads to

$$\mathcal{F}^{\mathrm{TW}\ b_o,ns_o,s_o}_{m,N,n} \sum_{i=0}^{m-1} \Big(\mathrm{W}^{N,n}_{b_o+ins_o,s_o}\, \mathrm{DFT}_n\, \mathrm{R}^{N,n}_{b_i+is_i,ms_i} \Big)$$

which proves the theorem. □

Appendix E.1 shows the Fftw framework in pseudo code. There the executor implements the Cooley-Tukey recursion rule. The codelets do the actual computation. Note the close connection between the formal derivation presented in this section and the actual code.

Example 5.9 (FFTW's Recursion for N = rstu) Suppose a DFT computation using three twiddle codelet stages and one no-twiddle stage. Using Theorem 5.1 for $m = r$ and $n = stu$ leads to

$$\mathrm{DFT}_{rstu} = (\mathrm{DFT}_r \otimes \mathrm{I}_{stu})\, \mathrm{T}^{rstu}_{stu} (\mathrm{I}_r \otimes \mathrm{DFT}_{stu})\, \mathrm{L}^{rstu}_r\,.$$

Applying Theorem 5.1 again for $m = s$ and $n = tu$ leads to

$$\mathrm{DFT}_{rstu} = (\mathrm{DFT}_r \otimes \mathrm{I}_{stu})\, \mathrm{T}^{rstu}_{stu} (\mathrm{I}_r \otimes \big((\mathrm{DFT}_s \otimes \mathrm{I}_{tu})\, \mathrm{T}^{stu}_{tu} (\mathrm{I}_s \otimes \mathrm{DFT}_{tu})\, \mathrm{L}^{stu}_s\big)\, \mathrm{L}^{rstu}_r\,.$$

Applying Theorem 5.1 a last time with $m = t$ and $n = u$ leads to the final expansion:

$$\begin{aligned} \mathrm{DFT}_{rstu} \quad = \quad & (\mathrm{DFT}_r \otimes \mathrm{I}_{stu})\, \mathrm{T}^{rstu}_{stu} (\mathrm{I}_r \otimes \big((\mathrm{DFT}_s \otimes \mathrm{I}_{tu})\, \mathrm{T}^{stu}_{tu} \\ & (\mathrm{I}_s \otimes \big((\mathrm{DFT}_t \otimes \mathrm{I}_u)\, \mathrm{T}^{tu}_u (\mathrm{I}_t \otimes \mathrm{DFT}_u)\, \mathrm{L}^{tu}_t\big)\, \mathrm{L}^{stu}_s\,\big)\, \mathrm{L}^{rstu}_r\,. \end{aligned}$$

The same expansion is now derived using Fftw's recursion rule and the codelets' definition. The result is an expression featuring three nested loops. By choosing the innermost loop index

	Twiddle $m = r$	Twiddle $m = s$	Twiddle $m = t$	No-twiddle $m = u$
N	$rstu$	$rstu$	$rstu$	$rstu$
mn	$rstu$	stu	tu	
m	r	s	t	u
n	stu	tu	tu	
b_i	0	i	$i + jr$	$i + jr + krs$
s_i	1	r	rs	rst
b_o	0	$istu$	$istu + jtu$	$istu + jtu + ku$
s_o	1	1	1	1

Table 5.1: Recursion steps applying FFTW's Cooley-Tukey rule (Theorem 5.12) three times to $N = rstu$ with the respective loop indices i, j, and k.

running fastest and the outermost loop index running slowest (the natural choice according to the loop structure) FFTW's recursion is shown unrolled:

$$\begin{aligned}
\mathrm{DFT}_{rstu} \quad = \quad & \mathcal{F}^{\mathrm{TW}\ 0,stu,1}_{r,rstu,stu} \sum_{i=0}^{r-1} \Big(\mathcal{F}^{\mathrm{TW}\ istu,tu,1}_{s,rstu,tu} \\
& \sum_{j=0}^{s-1} \Big(\mathcal{F}^{\mathrm{TW}\ istu+jtu,u,1}_{t,rstu,u} \\
& \sum_{j=0}^{t-1} \Big(\mathcal{F}^{\mathrm{NT}\ i+jr+krs,rst,istu+jtu+ku,1}_{u,rstu} \Big)\Big)\Big).
\end{aligned}$$

FFTW's planner has the possibility to choose from different factorizations

$$N = n_0 n_1 \cdots n_k$$

where both k and n_i can be chosen. The best choice is found using dynamic programming and run time measurement.

The Cooley-Tukey Vector Recursion

FFTW 3.1.2 features an experimental implementation of the Cooley-Tukey vector recursion as given by Theorem 5.5.

This section describes FFTW's Cooley-Tukey vector recursion using the Kronecker product formalism and the RW notation.

Theorem 5.13 (FFTW's Cooley-Tukey Vector Recursion) FFTW's Cooley-Tukey vector recursion computes

$$\begin{aligned}
(\mathrm{I}_m \otimes \mathrm{DFT}_{k_1 k_2 n})\, \mathrm{L}^{m k_1 k_2 n}_{m} \quad = \quad & \mathrm{I}_m \otimes (\mathrm{DFT}_{k_1} \otimes \mathrm{I}_{k_2 n})\, \mathrm{T}^{k_1 k_2 n}_{k_2 n} \\
& (\mathrm{L}^{m k_1}_{m} \otimes \mathrm{I}_{k_2 n}) \\
& (\mathrm{I}_{k_1} \otimes (\mathrm{I}_m \otimes \mathrm{DFT}_{k_2 n})\, \mathrm{L}^{m k_2 n}_{m}) \\
& (\mathrm{L}^{k_1 k_2 n}_{k_1} \otimes \mathrm{I}_m).
\end{aligned}$$

Once $k_2 = 1$ is reached, the recursion stops and the respective leaf is

$$(\mathrm{I}_m \otimes \mathrm{DFT}_n)\,\mathrm{L}_m^{mn} = \mathrm{L}_m^{mn}(\mathrm{DFT}_n \otimes \mathrm{I}_m).$$

Proof: Applying Theorem 5.5 with $m = m$, $k = k_1$ and $n = k_2 n$ leads to the recursion rule and Property 4.4 accounts for the leaf rule. □

Thus, any operations apart from the leaf operations are carried out on vectors of length m and n. The construct I_m originating from the topmost splitting is pushed down to DFT_n.

The vector recursion can be expressed in terms of twiddle codelets and no-twiddle codelets analogous to the standard Cooley-Tukey recursion.

Example 5.10 (FFTW's Vector Recursion for N = rstu) Suppose a DFT computation using three twiddle codelet stages and one no-twiddle stage. Using Theorem 5.1 leads to

$$\mathrm{DFT}_{rstu} = (\mathrm{DFT}_r \otimes \mathrm{I}_{stu})\,\mathrm{T}_{stu}^{rstu}(\mathrm{I}_r \otimes \mathrm{DFT}_{stu})\,\mathrm{L}_r^{rstu}\,.$$

Applying Theorem 5.13 for $m = r$, $k_1 = s$, $k_2 = t$, and $n = u$ to the right construct leads to

$$\begin{aligned}\mathrm{DFT}_{rstu} \quad = \quad & (\mathrm{DFT}_r \otimes \mathrm{I}_{stu})\,\mathrm{T}_{stu}^{rstu}\\ & \big(\mathrm{I}_r \otimes (\mathrm{DFT}_s \otimes \mathrm{I}_{tu})\,\mathrm{T}_{tu}^{stu}\big)\\ & (\mathrm{L}_r^{rs} \otimes \mathrm{I}_{tu})\\ & (\mathrm{I}_s \otimes (\mathrm{I}_r \otimes \mathrm{DFT}_{tu})\,\mathrm{L}_r^{rtu})\\ & (\mathrm{L}_s^{stu} \otimes \mathrm{I}_r).\end{aligned}$$

Applying Theorem 5.13 a second time with $m = r$, $k_1 = t$, $k_2 = 1$, and $n = u$ leads to

$$\begin{aligned}\mathrm{DFT}_{rstu} \quad = \quad & (\mathrm{DFT}_r \otimes \mathrm{I}_{stu})\,\mathrm{T}_{stu}^{rstu}\\ & \big(\mathrm{I}_r \otimes (\mathrm{DFT}_s \otimes \mathrm{I}_{tu})\,\mathrm{T}_{tu}^{stu}\big)\\ & (\mathrm{L}_r^{rs} \otimes \mathrm{I}_{tu})\\ & (\mathrm{I}_s \otimes \big(\mathrm{I}_r \otimes (\mathrm{DFT}_t \otimes \mathrm{I}_u)\,\mathrm{T}_u^{tu}\big)\,(\mathrm{L}_r^{rt} \otimes \mathrm{I}_u)(\mathrm{I}_t \otimes (\mathrm{I}_r \otimes \mathrm{DFT}_u)\,\mathrm{L}_r^{ru})(\mathrm{L}_t^{tu} \otimes \mathrm{I}_r))\\ & (\mathrm{L}_s^{stu} \otimes \mathrm{I}_r).\end{aligned}$$

Applying the leaf transformation of Theorem 5.13 leads to the final expansion:

$$\begin{aligned}\mathrm{DFT}_{rstu} \quad = \quad & (\mathrm{DFT}_r \otimes \mathrm{I}_{stu})\,\mathrm{T}_{stu}^{rstu}\\ & \big(\mathrm{I}_r \otimes (\mathrm{DFT}_s \otimes \mathrm{I}_{tu})\,\mathrm{T}_{tu}^{stu}\big)\\ & (\mathrm{L}_r^{rs} \otimes \mathrm{I}_{tu})\\ & (\mathrm{I}_s \otimes \big(\mathrm{I}_r \otimes (\mathrm{DFT}_t \otimes \mathrm{I}_u)\,\mathrm{T}_u^{tu}\big)\,(\mathrm{L}_r^{rt} \otimes \mathrm{I}_u)(\mathrm{I}_t \otimes \mathrm{L}_r^{ru}(\mathrm{DFT}_u \otimes \mathrm{I}_r))(\mathrm{L}_t^{tu} \otimes \mathrm{I}_r))\\ & (\mathrm{L}_s^{stu} \otimes \mathrm{I}_r).\end{aligned}$$

Except from L_r^{ru}, any expression in this formula is either a (*i*) twiddle factor, (*ii*) of the form $A \otimes \mathrm{I}_r$, or (*iii*) of the form $A \otimes \mathrm{I}_u$. FFTW computes this formula using twiddle and no-twiddle codelets exhibiting a maximum of memory access locality.

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 the 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.

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. Note, that for IPF currently only SSE is supported via compilers. Thus, IPF's SIMD instructions cannot be utilized in native mode within C codes yet.

Careful analysis of the instructions required by the code generated for discrete linear transforms within the context of this thesis allowed to define a set of C macros—a portable SIMD API—that can be implemented efficiently on all current architectures and features all necessary operations. The main restriction turned out to be that across all short vector SIMD extensions only *naturally aligned vectors* can be loaded and stored highly efficient. All other memory access operations lead to performance degradation and possibly to prohibitive performance characteristics.

All codes generated in the scope of this thesis use the newly defined portable SIMD API. This portable SIMD API serves two main purposes: (*i*) to abstract from hardware peculiarities, and (*ii*) to abstract from special compiler features.

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 B describes the relevant parts of the instruction sets provided by current short vector SIMD extensions.

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.

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 C contains examples of actual implementations of the portable SIMD API on various platforms. All examples of such macros displayed in this section suppose a two-way or four-way short vector SIMD extension. The portable SIMD API can be extended to arbitrary vector length ν. Thus, optimization techniques like *loop interleaving* (Gatlin and Carter [37]) can be implemented on top of the portable SIMD API.

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`.

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.

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

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

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 have different components).

Three types of constant load operations have bee introduced: (*i*) load a constant (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.

Arithmetic Operations

The portable SIMD API provides real addition, multiplication, and subtraction operations, the unary minus, two types of fused multiply-add operations, and a complex multiplication. Both variants that either modify a parameter or that return the result exist. See Table 6.3 for a summary.

Macro	**Operation**
`VEC_ADD_P(v, v0, v1)`	$v = v_0 + v_1$
`VEC_SUB_P(v, v0, v1)`	$v = v_0 - v_1$
`VEC_UMINUS_P(v, v0)`	$v = -v_0$
`VEC_MUL_P(v, v0, v1)`	$v = v_0 \times v_1$
`VEC_MADD_P(v, v0, v1, v2)`	$v = v_0 \times v_1 + v_2$
`VEC_NMSUB_P(v, v0, v1, v2)`	$v = -(v_0 \times v_1 - v_2)$

`COMPLEX_MULT(v0, v1, v2,`
`v3, v4, v5)`
	$v_0 = v_2 \times v_4 - v_3 \times v_5$
	$v_1 = v_2 \times v_5 + v_3 \times v_4$

`VEC_ADD(v0, v1)`	return $(v_0 + v_1)$
`VEC_SUB(v0, v1)`	return $(v_0 - v_1)$
`VEC_UMINUS(v0)`	return $-v_0$
`VEC_MUL(v0, v1)`	return $(v_0 \times v_1)$
`VEC_MADD(v0, v1, v2)`	return $v_0 \times v_1 + v_2$
`VEC_NMSUB(v0, v1, v2)`	return $-(v_0 \times v_1 - v_2)$

Table 6.3: Arithmetic operations provided by the portable SIMD API.

Extended Memory Operations

The portable SIMD API features three types of memory operations. All vector reordering operations are part of the memory access operations as all permutations are joined with load or store operations. The portable SIMD API provides three types of memory access operations which are described in the following: (*i*) plain vector load and store, (*i*) vector memory access plus permutation, and (*iii*) generic memory access plus permutation. The semantics of all load operations is to load data from memory locations which are not necessarily aligned nor vectors into a set of vector variables. The semantics of all store operations is to store a set of vector variables at specific memory locations which are not necessarily aligned nor vectors.

Tables 6.4, 6.5, and 6.6 show the most important macros for memory access.

Vector Memory Access. The macros `VEC_LOAD` and `VEC_STORE` load or store naturally aligned vectors from or to memory, respectively. These are the most efficient operations for memory access.

Vector Memory Access plus Permutation. A basic set of permutations of n vector variables is supported. For load operations, n vectors are loaded from memory, permuted accordingly and then stored into n vector variables. The store operations first permute and then store vector variables respectively. The supported permutations are

$$\mathrm{I}_\nu \quad \mathrm{J}_\nu \quad \mathrm{L}^{2\nu}_2 \quad \mathrm{L}^{2\nu}_\nu \quad \mathrm{L}^{\nu^2}_\nu \quad \mathrm{L}^{\nu}_2 \otimes \mathrm{I}_2$$

which lead to

$$\mathrm{I}_2 \quad \mathrm{I}_4 \quad \mathrm{J}_2 \quad \mathrm{J}_4 \quad \mathrm{L}^4_2 \quad \mathrm{L}^8_2 \quad \mathrm{L}^8_4 \quad \mathrm{L}^{16}_4 \quad \mathrm{L}^4_2 \otimes \mathrm{I}_2$$

in the case of two-way and four-way short vector SIMD extensions. The case I_ν is the standard vector memory access. The load and store macros are defined according to these permutations. This leads to macros like `LOAD_L_4_2`, `STORE_J_4`, and `LOAD_L_16_4`.

Generic Memory Access plus Permutation. These macros are a generalization of the vector memory access macros. The implementation of general permutations require these macros which are not directly supported by all short vector SIMD extensions.

Instead of accessing whole vectors, these macros imply memory access at the level of real or complex numbers. Depending on the underlying hardware, these operations may require scalar, subvector or vector memory access. The performance of such permutations depends strongly on the target architecture.

For instance, on SSE, properly aligned memory access for complex numbers does not degrade performance very much. For two-way short vector SIMD extensions complex numbers are native vectors. However, on AltiVec these memory operations feature prohibitive performance characteristics as such a macro can result in many vector memory accesses and permutation operations.

These operations lead to macros like `LOAD_L_4_2_R` which loads four real numbers from arbitrary locations and then performs a L^4_2 operation, or `STORE_8_4_C` which performs a L^8_4 and then stores pairs of real numbers properly aligned at arbitrary locations.

Macro	**Access**	**Permutation**
`LOAD_VECT(t, *v)`	vector	I_ν
`LOAD_J(t, *v)`	vector	J_ν
`LOAD_L_4_2(t0, t1, *v0, *v1)`	vector	L^4_2
`LOAD_R_2(t, *r0, *r1)`	real	implicit
`STORE_VECT(*v, t)`	vector	I_ν
`STORE_J(*v, t)`	vector	J_ν
`STORE_L_4_2(*v0, *v1, t0, t1)`	vector	L^4_2
`LOAD_R_2(*r0, *r1, t)`	real	implicit

Table 6.4: Load and store operations supported by the portable SIMD API for two-way short vector SIMD extensions.

Macro	**Access**	**Permutation**
`LOAD_VECT(t, *v)`	vector	I_ν
`LOAD_J(t, *v)`	vector	J_ν
`LOAD_L_4_2(t, *v)`	vector	L^4_2
`LOAD_L_8_2(t0, t1, *v0, *v1)`	vector	L^8_2
`LOAD_L_8_4(t0, t1, *v0, *v1)`	vector	L^8_4
`LOAD_L_16_4(t0, t1, t2, t3, *v0, *v1, *v2, *v3)`	vector	L^{16}_4
`LOAD_C(t, *c0, *c1)`	complex	implicit
`LOAD_J_C(t, *c0, *c1)`	complex	implicit + J_4
`LOAD_L_4_2_C(t, *c0, *c1)`	complex	implicit + L^4_2
`LOAD_L_8_2_C(t0, t1, *c0, *c1, *c2, *c3)`	complex	implicit + L^8_2
`LOAD_L_8_4_C(t0, t1, *c0, *c1, *c2, *c3)`	complex	implicit + L^8_4
`LOAD_R_4(t, *r0, *r1, *r2, *r3)`	real	implicit

Table 6.5: Load operations supported by the portable SIMD API for four-way short vector SIMD extensions.

Macro	**Access**	**Permutation**
`STORE_VECT(*v, t)`	vector	I_ν
`STORE_J(*v, t)`	vector	J_ν
`STORE_L_4_2(*v, t)`	vector	L^4_2
`STORE_L_8_2(*v0, *v1, t0, t1)`	vector	L^8_2
`STORE_L_8_4(*v0, *v1, t0, t1)`	vector	L^8_4
`STORE_L_16_4(*v0, *v1, *v2, *v3, t0, t1, t2, t3)`	vector	L^{16}_4
`STORE_C(*c0, *c1, t)`	complex	implicit
`STORE_J_C(*c0, *c1, t)`	complex	implicit + J_4
`STORE_L_4_2_C(*c0, *c1, t)`	complex	implicit + L^4_2
`STORE_L_8_2_C(*c0, *c1, *c2, *c3, t0, t1)`	complex	implicit + L^8_2
`STORE_L_8_4_C(*c0, *c1, *c2, *c3, t0, t1)`	complex	implicit + L^8_4
`STORE_R_4(*r0, *r1, *r2, *r3, t)`	real	implicit

Table 6.6: Store operations supported by the portable SIMD API for four-way short vector SIMD extensions.

Chapter 7

Transform Algorithms on Short Vector Hardware

This chapter shows how discrete linear transform formulas can be transformed into other equivalent formulas which (*i*) can be implemented efficiently using short vector SIMD extensions, and (*ii*) are structurally close to the original formulas. Specifically, constructs are identified that allow for an implementation on *all* current short vector SIMD extensions. In addition, a set of rules is given that allows to formally transform constructs appearing in discrete linear transforms into versions that match the requirements of short vector SIMD extensions. Such constructs are said to be *vectorized*.

This set of rules is designed to be included into the SPIRAL system to enable SPIRAL to generate efficient vector code. Thus, the formal approach and the automatic performance tuning provided by SPIRAL is used and extended by the newly developed formal rules and the associated implementation guidelines.

Furthermore, the important case of DFTs is handled in more detail leading to a set of short vector Cooley-Tukey rules that allow high-performance short vector SIMD implementations of FFTs and solve the intrinsic problems present in all other approaches to vectorize FFTs for short vector SIMD extensions.

This set of short vector Cooley-Tukey rules accounts for high-performance implementations of DFTs within the SPIRAL framework. The extension of SPIRAL with the general formal vectorization and the DFT specific short vector Cooley-Tukey rules as well as the respective numerical experiments are presented in Section 8.1. In addition, the short vector Cooley-Tukey rules are the foundation of a short vector SIMD extension to FFTW which delivers about the same performance level as the short vector Cooley-Tukey FFT implementation within SPIRAL. The extension of FFTW and the respective numerical experiments are presented in Section 8.2.

Related Work. Floating-point short vector SIMD extensions are relatively new features of modern microprocessors. They are the successors of integer short vector SIMD extensions. A radix-4 FFT implementation for the NEC V80R DSP processor featuring a four-way integer short vector SIMD extension has been presented some years ago (Joshi and Dubey [62]). A portable Fortran implementation of the proprietary Cray SCILIB library targeted at traditional vector computers is the SCIPORT library (Lamson [65]).

Apple Computers Inc. included the vDSP vendor library into their operating system MAC OS X. This library features a DFT implementation which supports the MPC 74xx G4 AltiVec extension [8]. This implementation based on the Stockham algorithm is described in Crandall and Klivington [15]. Intel's math kernel library (MKL) provides support for SSE, SSE 2, and the Itanium processor family [58]. An SSE split-radix FFT implementation is given in an Intel application note [44].

An ongoing effort to develop a portable library which will utilize short vector SIMD extensions is libSIMD (Nicholson [74]). SIMD-FFT (Rodriguez [82]) is a radix-2 FFT implementation for SSE and AltiVec. Fftw-gel is a short vector SIMD extension for Fftw 2.1.3 supporting 3DNow! and SSE 2 (Kral [64]).

7.1 Formal Vectorization of Linear Transforms

This section outlines how formulas that describe algorithms for discrete linear transforms can be vectorized symbolically using formula manipulation techniques such that the resulting formula can be implemented on short vector SIMD extensions efficiently. In principle, this result could be used create short vector SIMD implementations of discrete linear transforms without advanced code generation techniques. Formulas obtained using the newly developed techniques would serve as starting point for a traditional implementation and optimization cycle where formulas serve as starting point and then implementations are manually derived and subsequently optimized further. However, as the main focus of this work is automatic performance tuning and code generation, both formal rules and implementation guidelines are developed. The major goal is to provide the required theoretical framework to enable automatic code generation for discrete linear transforms within Fftw and Spiral.

7.1.1 Vectorizable Formulas

The key problem to solve is to identify the formula constructs that can be vectorized and to find an efficient implementation of the required building blocks. The presented approach is based on the vectorization of the basic construct

$$A \otimes \mathrm{I}_\nu, \quad \nu \text{ being the vector length,} \tag{7.1}$$

and A denoting an arbitrary formula. This construct can be naturally implemented by replacing in a scalar implementation of A all scalar operations by the corresponding vector operations.

Extending from (7.1), the most general normalized construct that can be vectorized by the presented approach (i. e., a construct that can be implemented using exclusively macros provided by the portable SIMD API) is

$$\prod_{i=1}^{k} P_i D_i (A_i \otimes \mathrm{I}_\nu) E_i Q_i, \tag{7.2}$$

with arbitrary matrices A_i, permutation matrices P_i, Q_i, and matrices D_i, E_i that are either real diagonals or vector diagonals matching equation (4.5). For example, all DFT and WHT algorithms arising from rules (5.3) and (5.2), respectively, and two-dimensional transforms given by equation (5.4) as well as any multi-dimensional transform, can be normalized to formulas matching equation (7.2). Thus, all these algorithms be completely vectorized using the approach presented in this chapter.

Two prerequisites are required to obtain a vectorized implementation of a formula.

- *Symbolic vectorization rules* allow to normalize a formula, using manipulation rules, to exhibit maximal subformulas that match (7.2).
- *Code generation guidelines* describe how to translate the vectorizable subformulas matching (7.2) into efficient vectorized code built on top of the portable SIMD API; the remainder of the formula is implemented in standard C code.

Efficient utilization of short vector SIMD extensions requires unit-stride data access, but other access patterns are predominating in discrete linear transform algorithms. An important example are subformulas of the form $\mathrm{I}_n \otimes A$. A similar problem arises from the interleaved data format found in complex transforms. In the context of this thesis, all such problems are solved by applying identities from Chapter 4 and by using formula manipulation that formally modify these expressions to make them match equation (7.2) by introducing permutations P_i and Q_i.

To allow high performance implementations it is necessary to introduce such permutations P_i, and Q_i that are supported efficiently by the portable SIMD API. The efficient support of permutations depends both on the type of permutation and on the underlying short vector SIMD architecture. A permutation can be negligible concerning run time, but it may also slow down the whole discrete linear transform algorithm dramatically. Implementing the newly derived formula manipulation rules in an automatic performance tuning system favors permutations that are cheap in terms of run time on the target platform. Additionally, a class of permutations is defined that can be realized efficiently on all short vector SIMD extensions and includes the permutations occurring in the considered transforms.

It is important to note that the approach presented in this chapter utilizes high-level structural information about discrete linear transform algorithms. This information is available in the formula representation, but not in a C code representation of the algorithm. For this reason, a general purpose vectorizing compiler fails, e. g., to vectorize the construct $\mathrm{I}_\nu \otimes A$, even though it is completely vectorized using the methods of this chapter.

7.1.2 Symbolic Vectorization

First, all vectorizable constructs have to be extracted. The core elements of the normalized formulas are symbols.

Definition 7.1 (Symbol) A symbol S_i is defined as by

$$\mathrm{S}_i = P_i D_i (A_i \otimes \mathrm{I}_\nu) E_i Q_i$$

with permutation matrices P_i and Q_i and real diagonals or vector diagonals D_i and E_i and an arbitrary real matrix A_i.

Symbols defined by Definition 7.1 are the basic construct to be vectorized. Using the identities summarized in Chapter 4 an expression of the form

$$F = \prod_{i=1}^{n} (R_i \oplus \mathrm{S}_i \oplus T_i) U_i, \quad \nu \text{ divides the size of } R_i \tag{7.3}$$

has to be obtained. The constructs R_i, T_i, and U_i are arbitrary formulas and S_i is a symbol. In practice this normalization can be achieved manually or by means of symbolic manipulation utilizing computer algebra systems or functional languages. For instance, SPIRAL utilizes the computer algebra system GAP and FFTW's codelet generator `genfft` is written in the functional language OCAML [66].

The constructs R_i, T_i, and U_i of equation (7.3) have to be be translated into standard C code as they do not match equation (7.2). For example, a formula that has no vectorizable parts degenerates to

$$F = U_1,$$

while in a completely vectorizable formula R_i and T_i vanish, U_i is in this case the identity, and thus F matches equation (7.2).

Note that the normalization given by equation (7.3) is not unique. When normalizing a given formula, it is important to preserve the structure of the original formula as much as possible to keep the data access patterns as close to the original one as possible and minimize the interference with an automatic performance tuning system.

Important subformulas that become symbols S_i when applying the transformation identities summarized in Chapter 4 include

$$A \otimes B, \quad A \otimes \mathrm{I}_k, \quad \mathrm{I}_k \otimes A, \quad \overline{A \otimes \mathrm{I}_k}, \quad \text{and} \quad \overline{\mathrm{I}_k \otimes A},$$

as can be seen from the manipulation rules summarized in Sections 4.2.2, 4.3.1, 4.4.1, and 4.6.1.

Furthermore, products $Q_i P_{i+1}$ of adjacent permutations may entirely or partially cancel out leading to identity matrices. An important example of such an cancellation is the real version of a complex formula. In such formulas Properties 4.18 and 4.33 can be often applied leading to the cancellation

$$(\mathrm{I}_k \otimes \mathrm{L}^{2\nu}_{\nu})(\mathrm{I}_k \otimes \mathrm{L}^{2\nu}_{2}) = \mathrm{I}_{2k\nu}\,.$$

The constructs $\mathrm{I}_k \otimes \mathrm{L}^{2\nu}_{\nu}$ and $\mathrm{I}_k \otimes \mathrm{L}^{2\nu}_{2}$ become a product $Q_i P_{i+1}$ during the normalization due to the identities summarized in Section 4.6.1.

After the normalization, the symbols S_i can be treated as independent formulas for which vector code is generated. The remaining formula consisting of the respective R_i, T_i, and U_i is translated into standard (scalar) C code where the symbols S_i become subroutines that are implemented utilizing the portable SIMD API and thus become vectorized code.

Example 7.1 (Symbolic Vectorization) This example shows the symbolic vectorization of DFT_{16}. A normalized formula for DFT_{16} is given by

$$\mathrm{DFT}_{16} = (\mathrm{DFT}_4 \otimes \mathrm{I}_4)\ \mathrm{T}^{16}_4\ (\mathrm{I}_4 \otimes \mathrm{DFT}_4)\ \mathrm{L}^{16}_4 \tag{7.4}$$

with

$$\mathrm{DFT}_4 = (\mathrm{DFT}_2 \otimes \mathrm{I}_2)\, \mathrm{T}^4_2\, (\mathrm{I}_2 \otimes \mathrm{DFT}_2)\, \mathrm{L}^4_2\,. \tag{7.5}$$

The formula is first transformed into its real representation using the bar operator $\overline{(\ )}$ and the identities summarized in Section 4.6.1. Permutations are inserted to make the formula match equation (7.3) with the constraint that the constructs R_i, T_i are as small as possible, and U_i is as close to I_k as possible. The inserted permutations are factored using identities from Section 4.4.1 such that they match the permutations that can be implemented efficiently as outlined in the next section.

The target architecture is assumed to feature a four-way short vector SIMD extension ($\nu = 4$). Formula manipulation leads to

$$\begin{aligned}\overline{\mathrm{DFT}}_{16} = &\Big(\big(\mathrm{I}_4 \otimes \mathrm{L}^8_4\big)\ \big(\overline{\mathrm{DFT}}_4 \otimes \mathrm{I}_4\big)\ \overline{\mathrm{T}}'^{16}_4 \Big)\\ &\Big(\big(\mathrm{I}_4 \otimes \mathrm{L}^8_2\big)\ \big(\mathrm{L}^{16}_4 \otimes \mathrm{I}_2\big)\ \big(\mathrm{I}_4 \otimes \mathrm{L}^8_4\big)\ \big(\overline{\mathrm{DFT}}_4 \otimes \mathrm{I}_4\big)\ \big(\mathrm{I}_4 \otimes \mathrm{L}^8_2\big) \Big).\end{aligned} \tag{7.6}$$

The achieved formula is factored into two symbols S_1 and S_2

$$\overline{\mathrm{DFT}}_{16} = \mathrm{S}_1\ \mathrm{S}_2\,.$$

The first symbol is given by

$$\mathrm{S}_1 = \big(\mathrm{I}_4 \otimes \mathrm{L}^8_4\big)\ \big(\overline{\mathrm{DFT}}_4 \otimes \mathrm{I}_4\big)\ \overline{\mathrm{T}}'^{16}_4$$

with

$$\begin{aligned}
P_1 &= \big(\mathrm{I}_4 \otimes \mathrm{L}^8_4\big)\\
D_1 &= \mathrm{I}_{32}\\
A_1 &= \overline{\mathrm{DFT}}_4\\
E_1 &= \overline{\mathrm{T}}'^{16}_4 = \overline{\mathrm{T}}^{16}_4{}^{\left(\mathrm{I}_4 \otimes \mathrm{L}^8_4\right)}\\
Q_1 &= \mathrm{I}_{32}\\
R_1 &= \mathrm{I}_0\\
T_1 &= \mathrm{I}_0\\
U_1 &= \mathrm{I}_{32}
\end{aligned}$$

and the second one by

$$\mathrm{S}_2 = \left(\mathrm{I}_4 \otimes \mathrm{L}_2^8\right) \left(\mathrm{L}_4^{16} \otimes \mathrm{I}_2\right) \left(\mathrm{I}_4 \otimes \mathrm{L}_4^8\right) \left(\overline{\mathrm{DFT}}_4 \otimes \mathrm{I}_4\right) \left(\mathrm{I}_4 \otimes \mathrm{L}_2^8\right)$$

with

$$\begin{aligned} P_2 &= \left(\mathrm{I}_4 \otimes \mathrm{L}_2^8\right) \left(\mathrm{L}_4^{16} \otimes \mathrm{I}_2\right) \left(\mathrm{I}_4 \otimes \mathrm{L}_4^8\right) \\ D_2 &= \mathrm{I}_{32} \\ A_2 &= \overline{\mathrm{DFT}}_4 \\ E_2 &= \mathrm{I}_{32} \\ Q_2 &= \mathrm{I}_4 \otimes \mathrm{L}_2^8 \\ R_2 &= \mathrm{I}_0 \\ T_2 &= \mathrm{I}_0 \\ U_2 &= \mathrm{I}_{32}\,. \end{aligned}$$

Thus, the transform DFT_{16} is symbolically vectorized. Two symbols S_1 and S_2 are required and all parameters are defined.

7.1.3 Types of Permutations

The key to high performance for short vector SIMD implementations of symbolically vectorized formulas is the implementation of the permutations P_i and Q_i in Definition 7.1. These permutations can be classified with respect to their performance relevant properties leading to different classes of permutations featuring different types factorizations. Considering the translation of formulas into code according to Section 4.7 makes the required memory access explicit.

Three classes of permutations are required to both support the general case and high-performance implementations: (*i*) permutations that can be implemented utilizing exclusively *vector* memory access, (*ii*) permutations requiring memory access of *pairs* of real numbers, and (*iii*) permutationst that require access of *single* real numbers. The implementation of the permutations P_i and Q_i crucially depends on their factorization and thus on their membership in one of the three classes.

These classes are mirrored in the portable SIMD API by three types of combined load or store and permute macros, respectively, that are provided by the portable SIMD API. If new permutations are required, the portable SIMD API can easily be extended, as any permutation falls into one of the three supported classes.

In the remainder of this section, the three classes of macros are discussed and the respective factorization is given.

Vector Memory Access Operations

These macros feature only vector memory access and in-register permutations. The permutations that can be handled using this class of macros are given by

$$(U \otimes \mathrm{I}_\nu)(\mathrm{I}_k \otimes W)(V \otimes \mathrm{I}_\nu), \quad \nu \text{ divides the size of } W, \tag{7.7}$$

where U, V, and W are permutation matrices. Equation (7.7) allows for loop code generation if the permutations U and V can be translated into loop code, as tensor products can be translated into loops. Thus, the class of permutations can be extended to support more general permutations if completely unrolled implementations can be used. For instance, the SPL compiler produces completely unrolled code for small subformulas. Then $\mathrm{I}_k \otimes W$ is replaced by a direct sum of different permutation matrices W_i as given by

$$(U \otimes \mathrm{I}_\nu)(\bigoplus_{i=0}^{k-1} W_i)(V \otimes \mathrm{I}_\nu), \quad \nu \text{ divides the size of } W_i, \tag{7.8}$$

Example 7.2 (Vector Memory Access Operations) Equation (7.7) includes all permutations needed for a vectorized implementation of the DFTs, WHTs, and two-dimensional transforms. For instance, in Example 7.1 all permutations can be factored according to equation (7.7) leading to

$$\begin{aligned}\overline{\mathrm{DFT}}_{16} = &\Big((\mathrm{I}_4 \otimes \mathrm{L}^8_4)\, (\overline{\mathrm{DFT}}_4 \otimes \mathrm{I}_4)\, \overline{\mathrm{T}}'^{16}_4 \Big) \\ &\Big((\mathrm{L}^8_4 \otimes \mathrm{I}_4)\, (\mathrm{I}_2 \otimes \mathrm{L}^{16}_4)\, (\mathrm{L}^8_2 \otimes \mathrm{I}_4)\, (\overline{\mathrm{DFT}}_4 \otimes \mathrm{I}_4)\, (\mathrm{I}_4 \otimes \mathrm{L}^8_2) \Big)\end{aligned} \tag{7.9}$$

with

$$\begin{aligned} P_1 &= \mathrm{I}_4 \otimes \mathrm{L}^8_4 \\ Q_2 &= \mathrm{I}_4 \otimes \mathrm{L}^8_2 \\ P_2 &= (\mathrm{I}_4 \otimes \mathrm{L}^8_2)(\mathrm{L}^{16}_4 \otimes \mathrm{I}_2)(\mathrm{I}_4 \otimes \mathrm{L}^8_4) \\ &= (\mathrm{L}^8_4 \otimes \mathrm{I}_4)(\mathrm{I}_2 \otimes \mathrm{L}^{16}_4)(\mathrm{L}^8_2 \otimes \mathrm{I}_4). \end{aligned}$$

By factoring P_2, the desired form is achieved. Now all permutations can be implemented using `LOAD_L_8_2`, `STORE_L_8_4`, and `STORE_L_16_4`.

Any permutation covered by equations (7.7) and (7.8) can be implemented using the appropriate macros provided by the portable SIMD API.

For permutations P_i the respective permutation U and for permutations Q_i the respective permutation V is an index transformation for vector load and vector store operations. Thus these permutations are handled by the value of the parameters `*v`$_i$.

For permutations P_i the respective permutation V and for permutations Q_i the respective permutation U is handled by renaming temporary variables and thus by the value of the parameters `t`$_i$. Thus, the input and output parameters for the portable SIMD API macros implement the permutations U and V. The permutation W is implemented transparently within the respective macros.

Table 7.1 summarizes the most important macros of the portable SIMD API that are vector memory access operations.

Subvector Memory Access Operations

The second important type of permutations operates on pairs of real numbers. This includes all permutations of complex numbers using the interleaved complex format and the bar operator $\overline{(\ )}$. This class of permutations is given by all

Macro	**Permutation W**
`LOAD_VECT(t, *v)`	I_ν
`LOAD_J(t, *v)`	J_ν
`LOAD_L_4_2(t, *v)`	L_2^4
`LOAD_L_8_2(t0, t1, *v0, *v1)`	L_2^8
`LOAD_L_8_4(t0, t1, *v0, *v1)`	L_4^8

`LOAD_L_16_4(t0, t1, t2, t3,`
`*v0, *v1, *v2, *v3)`
	L_4^{16}

`STORE_VECT(*v, t)`	I_ν
`STORE_J(*v, t)`	J_ν
`STORE_L_4_2(*v, t)`	L_2^4
`STORE_L_8_2(*v0, *v1, t0, t1)`	L_2^8
`STORE_L_8_4(*v0, *v1, t0, t1)`	L_4^8

`STORE_L_16_4(*v0, *v1, *v2, *v3,`
`t0, t1, t2, t3)`
	L_4^{16}

Table 7.1: Combined vector memory access and permutation macros supported by the portable SIMD API for four-way extensions.

permutations

$$P_i = (U \otimes \mathrm{I}_2)(\mathrm{I}_k \otimes W)(V \otimes \mathrm{I}_\nu) \tag{7.10}$$

$$Q_i = (U \otimes \mathrm{I}_\nu)(\mathrm{I}_k \otimes W)(V \otimes \mathrm{I}_2) \tag{7.11}$$

where U, V, and W are permutation matrices. The size of W has to be a multiple of ν. For permutations P_i the respective permutation U and for permutations Q_i the respective permutation V is an index transformation for load and store operations of pairs and thus these permutations are handled by the value of $\texttt{*c}_i$. For permutations P_i the respective permutation V and for permutations Q_i the respective permutation U is handled by renaming temporary variables and thus by the value of the parameters $\texttt{t}_i$. Thus, the input and output parameters for the portable SIMD API macros implement the permutations U and V. The permutation W is implemented transparently within the corresponding macros. Again, if unrolled code is acceptable, the class of supported permutations is extended by changing

$$\mathrm{I}_k \otimes W \qquad \text{into} \qquad \bigoplus_{k=0}^{k-1} W_i .$$

For two-way vector extensions, this class is the same as the vector memory access operations. On four-way short vector SIMD extensions, however, the performance of permutations requiring pairwise memory access crucially depends on the target architecture.

Using SSE these permutations can be implemented efficiently as SSE features memory access of naturally aligned pairs of single-precision floating-point numbers and these memory access instructions do not degrade performance. On the

AltiVec extension, load operations for such pairs have to be built from vector access operations and expensive general permutations. Storing pairs has to be done using permutations and expensive single element stores. Thus, on AltiVec, pairwise data access operations have to be avoided.

Example 7.3 (Subvector Memory Access Operations) In Example 7.1 the permutations P_i and Q_i can be chosen differently leading to a subvector memory access instead of a vector memory access. Equation (7.6) can alternatively be transformed into

$$\begin{aligned}\overline{\mathrm{DFT}}_{16} = & \Big((\mathrm{I}_4 \otimes \mathrm{L}^8_4)\,(\overline{\mathrm{DFT}}_4 \otimes \mathrm{I}_4)\,\overline{\mathrm{T}}'^{16}_4\,(\mathrm{I}_4 \otimes \mathrm{L}^8_2)\,(\mathrm{L}^{16}_4 \otimes \mathrm{I}_2) \Big) \\ & \Big((\mathrm{I}_4 \otimes \mathrm{L}^8_4)\,(\overline{\mathrm{DFT}}_4 \otimes \mathrm{I}_4)\,(\mathrm{I}_4 \otimes \mathrm{L}^8_2) \Big)\end{aligned}$$

leading to

$$\begin{aligned} P_1 &= \mathrm{I}_4 \otimes \mathrm{L}^8_4 \\ Q_1 &= (\mathrm{I}_4 \otimes \mathrm{L}^8_2)(\mathrm{L}^{16}_4 \otimes \mathrm{I}_2) \\ P_2 &= \mathrm{I}_4 \otimes \mathrm{L}^8_4 \\ Q_2 &= \mathrm{I}_4 \otimes \mathrm{L}^8_2\,. \end{aligned}$$

When applying this factorization, the intermediate permutation

$$(\mathrm{I}_4 \otimes \mathrm{L}^8_2)\,(\mathrm{L}^{16}_4 \otimes \mathrm{I}_2)\,(\mathrm{I}_4 \otimes \mathrm{L}^8_4)$$

is distributed differently—Q_1 and P_2 are chosen differently—and the factorization given in Example 7.2 cannot be applied any more. The permutation Q_1 requires a subvector memory access operation supported by the macro `LOAD_L_8_2_C` and $\mathrm{L}^{16}_4 \otimes \mathrm{I}_2$ becomes readdressing of subvectors.

As explained above, subvector memory access has the potential to slow down codes and thus has to be avoided. Example 7.1 and this example show the degree of freedom in symbolically vectorizing constructs. Depending on the factorization chosen of P_i and Q_i, vector memory access can be achieved or subvector memory access is required.

Table 7.2 summarizes the most important macros of the portable SIMD API featuring subvector memory access.

General Permutations

Any permutation not covered by the two classes from above has to be implemented by loading ν elements into the vector register or storing the ν elements to arbitrary memory locations. As direct data transfer between scalar and vector floating-point registers is not supported directly by most short vector SIMD extensions (see Sections 3.2 to 3.4), the vector has to be assembled or disassembled in memory (usually on the stack) using floating-point loads and stores. The assembled vector has to be transferred between memory and the vector register file using a vector memory access, i. e., data is transferred between memory and the register file *twice*. Thus, this type of permutation degrades performance on virtually all short vector SIMD extensions and has to be avoided at any cost.

Macro	**Permutation** W
`LOAD_C(t, *c0, *c1)`	I_ν
`LOAD_J_C(t, *c0, *c1)`	J_4
`LOAD_L_4_2_C(t, *c0, *c1)`	L^4_2
`LOAD_L_8_2_C(t0, t1, *c0, *c1, *c2, *c3)`	L^8_2
`LOAD_L_8_4_C(t0, t1, *c0, *c1, *c2, *c3)`	L^8_4
`STORE_C(*c0, *c1, t)`	I_ν
`STORE_J_C(*c0, *c1, t)`	J_4
`STORE_L_4_2_C(*c0, *c1, t)`	L^4_2
`STORE_L_8_2_C(*c0, *c1, *c2, *c3, t0, t1)`	L^8_2
`STORE_L_8_4_C(*c0, *c1, *c2, *c3, t0, t1)`	L^8_4

Table 7.2: Subvector memory access operations supported by the portable SIMD API for four-way short vector SIMD extensions.

Example 7.4 (General Permutations) The macro `LOAD_R_4` loads four single-precision floating-point numbers from arbitrary memory locations into a local array of four floating-point variables using standard C statements. This array is aliased to a vector variable and thus the C compiler places this array properly aligned on the stack. After the data is stored into the array, the whole array is loaded into a vector register using one vector memory access. Thus, four FPU loads, four FPU stores and one vector load is required to implement this macro.

Consider a general permutation of 16 single-precision floating-point numbers that requires four such macros. The cost introduced by this permutation can only be amortized by a large number of vector operations. See Appendix C for the actual implementation of this macro.

Formula manipulation is used to produce formulas that—whenever possible—consist only of permutations allowing vector memory access. Thus, it is tried to transform formulas featuring permutations of the second and third type into formulas featuring only permutations of the first type with the additional condition that the global structure of the formula is changed as little as possible. This transformation is the most crucial part in getting performance portable short vector SIMD code. The degree of freedom in normalizing a given formula adds another application for search techniques in SPIRAL's formula generator.

7.1.4 Code Generation for Symbols

Symbols as defined by equation (7.1) are designed to be implemented as follows. Each symbol consists of three *logical* stages, whose implementation can be interleaved by reordering operations, however, for any element in the data vector the order of operations has to be preserved. (*i*) Load phase: $x' = E_i Q_i\, x$. (*ii*) Computation phase: $y' = (A \otimes \mathrm{I}_\nu)\, x'$. (*iII*) Store phase: $y = P_i D_i\, y'$.

Definition 7.2 (Vector Terminal) A *vector terminal* is the

$$\text{subformula} \quad A \quad \text{in} \quad A \otimes \mathrm{I}_\nu\,.$$

Optimized code for a vector terminal $A \otimes \mathrm{I}_\nu$ can be generated by producing optimized scalar code for A and then replacing all scalar arithmetic operation in the generated code by their respective vector operations.

Example 7.5 (Vector Terminal) The constructs

$$\overline{\mathrm{DFT}_m \otimes \mathrm{I}_\nu} \quad \text{and} \quad \overline{\mathrm{DFT}_m \otimes \mathrm{I}_{\frac{n}{\nu}} \otimes \mathrm{I}_\nu}$$

are vector terminals that occur when applying the short Vector Cooley-Tukey rules developed in Section 7.3.

Any construct that can be implemented using *exclusively* (i) vector arithmetic, and (ii) vector memory access macros can be implemented highly efficient. All such constructs required in the scope of this thesis are given by the following definition.

Definition 7.3 (Vector Construct) A vector construct matches one of the following constructs:

$$A \otimes \mathrm{I}_\nu, \quad \bigoplus_{i=0}^{k} \overline{D}'_i, \quad (U \otimes \mathrm{I}_\nu)(\bigoplus_{i=0}^{k-1} W_i)(V \otimes \mathrm{I}_\nu), \quad \text{and} \quad \nu \mid \text{ size of } W_i,$$

with A being an arbitrary *real* matrix, U, V, and W_i permutation matrices, and D_i complex $\nu \times \nu$ matrices.

Example 7.6 (Vector Construct) Any construct $A \otimes \mathrm{I}_\nu$ is a vector construct as it only requires vector arithmetic operations and vector loads and stores.

All permutations covered by vector memory access operations are vector constructs. For instance,

$$\mathrm{L}_2^4 \otimes \mathrm{L}_2^8 = (\mathrm{L}_2^4 \otimes \mathrm{I}_8)(\mathrm{I}_4 \otimes \mathrm{L}_2^8)$$

is a vector construct as it can be implemented using `LOAD_L_8_2`.

The Load Phase and Store Phase

The computation of $x' = E_i Q_i\, x$ is joined with an initial load operation for each data element. $E_i Q_i$ is decomposed into combined load and permutation operations provided by the portable SIMD API and scaling operations that immediately follow these load operations. For each element that is loaded from memory according to the formula, the whole vector register has to be loaded using the appropriate macro of the portable SIMD API. These combined load and permute operations decompose the construct Q_i. E_i is implemented by the scaling operations. The operation $y = P_i D_i\, y'$ is implemented accordingly.

The Computation Phase

$A \otimes \mathrm{I}_\nu$ is the most natural construct that can be mapped onto short vector SIMD hardware. It is obtained by replacing any scalar in $a_{i,j}$ in A by

$$\mathrm{diag}(\underbrace{a_{i,j}, \ldots, a_{i,j}}_{\nu \text{ times}}).$$

Thus vector code is obtained by replacing any scalar arithmetic operation with the respective vector operation. This property is important as it allows to utilize existing code generators for scalar codes like FFTW's `genfft` and SPIRAL's SPL compiler. These compilers are extended to support the generation of short vector SIMD code. Specifically, support for the portable SIMD API and for the different types of permutations is required in addition to infrastructural changes. The extension of `genfft` is described in Section 8.1.1 and the extension of the SPL compiler is described in Section 8.2.1.

Synthesis

The load, store, and computation phase have to be overlapped to gain stack access locality. In this overlapping the order of operations has to be preserved for each data element.

Immediately before the first data element loaded by a specific macro required by $E_i Q_i$ is used within the computation phase, the respective load macro and the scaling operations for this data element are issued. These macros are loading and scaling a set of data elements including the required one. The number of additionally preloaded data elements is depending on the macro's parameters. Thus, for these elements the load phase is finished and the computation phase can be started.

In the computation phase, immediately after the final result is computed for all elements required to issue a specific store macro required by $P_i D_i$, this macro is issued as the next instruction. Thus, for these elements, the application of the symbol has been computed.

7.2 FFTs on Short Vector Hardware

Any Cooley-Tukey FFT algorithm features characteristics that require further special formula manipulation rules to exhibit formulas that consist of vector constructs. In addition, the global structure of the algorithm should not be altered too much to keep the Cooley-Tukey splitting's data access locality. The formal method introduced in this section extends the method from the last section and is based on the standard Cooley-Tukey recursion as given by Theorem 5.1, the vector Cooley-Tukey FFT (the four-step FFT algorithm) as given by Theorem 5.3, and the Cooley-Tukey vector recursion given by Theorem 5.5. It is shown that the short vector Cooley-Tukey rule set leads to formulas for FFTs which completely consist of vector constructs and thus can be mapped to efficient short vector SIMD code. The approach of this section is the foundation of the implementation of performance portable short vector FFTs as discussed in Sections 8.1 and 8.2.

The newly developed short vector Cooley-Tukey recursion rule set and the respective base cases (both developed in Section 7.3) depend merely on the vector length ν of the short vector SIMD extension. Implementing DFTs using short vector SIMD instructions imposes a set of challenges to the programmer and algorithm developer to match. The special properties of short vector SIMD extensions and the goal to provide *one* general method for treating all extensions (regardless of their actual vector length ν and any other machine specifics) are the major challenges to address.

It is important to note that all operations have to be expressed in *real arithmetic*, as (*i*) only real arithmetic is supported by the hardware, and (*ii*) short vector SIMD extensions pose stringent requirements on the location (alignment) and stride (unit stride) of real numbers in memory to be accessed efficiently. Complex arithmetic operations (based on the interleaved complex format) implemented straightforward using real arithmetic violate the unit stride memory access requirement, and thus are leading to an enormous performance breakdown. These restrictions cannot be expressed in the standard Kronecker product notation for FFT algorithms where the entries of matrices are complex numbers.

In both automatic performance tuning systems for discrete linear transforms that are extended in this thesis—Spiral and Fftw—all Cooley-Tukey FFT based algorithms have *recursive* structure (as opposed to *iterative* stage-wise computation found in traditional implementations) to exhibit data locality, and are built from only a few basic constructs. It turns out that the construct imposing the most annoying difficulties is the stride permutation

$$\overline{\mathrm{L}}_m^{mn} = \mathrm{L}_m^{mn} \otimes \mathrm{I}_2, \tag{7.12}$$

especially in conjunction with vector arithmetic, in particular for $\nu \neq 2$. The permutations

$$\mathrm{L}_2^{2k} \text{ and } \mathrm{L}_k^{2k}$$

introduced by the interleaved complex format complicate the situation. Equation (7.12) occurs on every recursion level in all Cooley-Tukey factorizations. The combination of multiple recursion steps may produce digit permutations as the bit-reversal permutation. The source of the problem is that for *arbitrary* ν one cannot find a factorization of $\overline{\mathrm{L}}_m^{mn}$ that (*i*) consists of vector constructs only, and (*ii*) does not require nontrivial transformations involving adjacent constructs. The additional condition that arithmetic operations have to treat ν elements equally as vector arithmetic is used introduces a further difficulty.

In the following, possibilities are discussed how to resolve the problems introduced by the fact that the DFT is a complex-to-complex transform and features stride permutations. Both advantages and disadvantages of the various approaches are summarized. A subset of vector constructs is identified that serves as the building blocks and base cases for short vector Cooley-Tukey FFTs. Finally, the respective short vector Cooley-Tukey rule set is derived.

There are three fundamentally different approaches to resolve the stride permutation issue.

- The permutation $\overline{\mathrm{L}_m^{mn}}$ is performed explicitly in a separate computation stage requiring an iterative approach. One can use high-performance implementations of transpositions (blocked or cache oblivious) and machine specific versions that utilize special instructions.
- The permutation $\overline{\mathrm{L}_m^{mn}}$ is decomposed into subvector memory access operations where pairs of real numbers are accessed. Equation (7.12) shows that this is always possible. To reach satisfactory performance levels, it is required that either (*i*) $\nu = 2$, or (*ii*) the hardware is able to efficiently access two elements at a subvector level (e. g., as provided by SSE).
- The permutation $\overline{\mathrm{L}_m^{mn}}$ is factored into a sequence of permutations within blocks and permutations of blocks (with a block size being a multiple of ν) at the cost of possibly requiring changes of adjacent constructs as well. These permutations can be handled using the portable SIMD API as described in Section 7.1.4.

  This approach only requires a minimum of data reorganization (all done in register). All memory access operations are accomplished using vector operations. This method can be used for any number of ν.

This thesis focusses on the third approach but briefly overviews the other two approaches as well. The remainder of this section discusses the first two approaches while the next section discusses the short vector Cooley-Tukey rule set.

7.2.1 The Cooley-Tukey Recursion

The starting point for the discussion of the standard Cooley-Tukey recursion is

$$\mathrm{DFT}_{mn} = (\mathrm{DFT}_m \otimes \mathrm{I}_n)\,\mathrm{T}_n^{mn}(\mathrm{I}_m \otimes \mathrm{DFT}_n)\,\mathrm{L}_m^{mn}$$

mapped to real arithmetic, i.e.,

$$\overline{\mathrm{DFT}_{mn}} = \overline{(\mathrm{DFT}_m \otimes \mathrm{I}_n)\,\mathrm{T}_n^{mn}(\mathrm{I}_m \otimes \mathrm{DFT}_n)\,\mathrm{L}_m^{mn}}. \tag{7.13}$$

It is assumed that ν divides n and m. The standard way of translating Theorem 5.1 into real code using the complex interleaved format corresponds to a straightforward application of the identities in given Section 4.6.1. Starting with distributing $\overline{(\ )}$ over the factors in equation (7.13) leads to

$$\overline{\mathrm{DFT}}_{mn} = (\overline{\mathrm{DFT}}_m \otimes \mathrm{I}_n)^{(\mathrm{I}_m \otimes \mathrm{L}_2^{2n})} \overline{\mathrm{T}}_n^{mn} (\mathrm{I}_m \otimes \overline{\mathrm{DFT}}_n)(\mathrm{L}_m^{mn} \otimes \mathrm{I}_2). \tag{7.14}$$

This formula features constructs that cannot be directly implemented using only vector memory access.

- The construct $\mathrm{L}_m^{mn} \otimes \mathrm{I}_2$ requires pairwise memory access. Thus, for $\nu \neq 2$ vector memory access cannot be used.
- The construct $\mathrm{I}_m \otimes \overline{\mathrm{DFT}}_n$ is no vector construct and requires pairwise memory access at stride m which can introduce cache problems for large sizes of m, especially for $m = 2^k$, and requires low performance pairwise memory access.
- The construct $\overline{\mathrm{T}}_n^{mn}$ does not match equation (4.6) and thus is not a *vector diagonal*. To allow for vector arithmetic, further formula manipulation is required. Alternatively, pairwise memory access macros can be used which degrade the performance.
- The construct $\overline{\mathrm{DFT}}_m \otimes \mathrm{I}_n$ has to be factored into

  $$\left(\overline{\mathrm{DFT}}_m \otimes \mathrm{I}_{\frac{n}{\nu}}\right) \otimes \mathrm{I}_\nu$$

  to obtain a vector construct.

To obtain vector memory access for the conjugation permutations, further factorization of $\mathrm{I}_m \otimes \mathrm{L}_2^{2n}$ and $\mathrm{I}_m \otimes \mathrm{L}_n^{2n}$ is required. Using Properties 4.21 and 4.22,

$$\begin{aligned}
\mathrm{I}_m \otimes \mathrm{L}_2^{2n} &= \left(\mathrm{L}_2^{2\frac{n}{\nu}} \otimes \mathrm{I}_\nu\right)\left(\mathrm{I}_{\frac{n}{\nu}} \otimes \mathrm{L}_2^{2\nu}\right)\\
\mathrm{I}_m \otimes \mathrm{L}_n^{2n} &= \left(\mathrm{I}_{\frac{n}{\nu}} \otimes \mathrm{L}_\nu^{2\nu}\right)\left(\mathrm{L}_{\frac{n}{\nu}}^{2\frac{n}{\nu}} \otimes \mathrm{I}_\nu\right)
\end{aligned}$$

is obtained which allows to use vector memory access macros. However, the factor $\mathrm{L}_2^{2\frac{n}{\nu}} \otimes \mathrm{I}_\nu$ stores and $\mathrm{L}_{\frac{n}{\nu}}^{2\frac{n}{\nu}} \otimes \mathrm{I}_\nu$ loads logically associated vectors (i. e., the real and imaginary parts of a complex vector) at scalar stride n which can lead to cache conflicts for large numbers of n, especially for $n = 2^k$.

Cooley-Tukey FFTs with Subvector Memory Access

As outline above, equation (7.13) cannot be mapped efficiently to short vector SIMD hardware without further manipulation. But using the identities introduced in Chapter 4 and the ideas outlined above, it is possible to modify equation (7.14) to obtain a better structure leading to the subvector Cooley-Tukey FFT.

Theorem 7.1 (Subvector Cooley-Tukey FFT) For $\nu \mid m$ and $\nu \mid n$

$$\begin{aligned} \overline{\mathrm{DFT}}_{mn} \;=\; & \left(\left(\overline{\mathrm{DFT}_m \otimes \mathrm{I}_{\frac{n}{\nu}}} \otimes \mathrm{I}_\nu\right) \overline{\mathrm{T}}'^{mn}_{n}\right)^{\left(\mathrm{I}_{\frac{mn}{\nu}} \otimes \mathrm{L}^{2\nu}_{\nu}\right)} \\ & \left(\mathrm{I}_{\frac{m}{\nu}} \otimes \left(\overline{\mathrm{DFT}}_n \otimes \mathrm{I}_\nu\right)^{\left(\mathrm{I}_n \otimes \mathrm{L}^{2\nu}_{\nu}\right)\left(\mathrm{L}^{n\nu}_{n} \otimes \mathrm{I}_2\right)}\right) \left(\mathrm{L}^{mn}_{m} \otimes \mathrm{I}_2\right). \end{aligned} \tag{7.15}$$

Theorem 7.1 can be implemented using both subvector memory access and vector memory access and leads to moderate to good performance. The difficulty of $\mathrm{L}^{mn}_{n} \otimes \mathrm{I}_2$ remains and the conjugation with $\mathrm{L}^{n\nu}_{n} \otimes \mathrm{I}_2$ is introduced. Both constructs require sub-vector memory access for $\nu \neq 2$.

Equation (7.15) features nearly the same data access pattern as the original Cooley-Tukey FFT rule and changes the structure of the formula much less than the short vector Cooley-Tukey FFT. Thus, this formula is a trade-off between structural change and vector memory access. Depending on the target machine, this rule can be an interesting alternative, especially for $\nu = 2$.

To support arbitrary vector lengths, formal vectorization developed in Section 7.1 is extended. FFTs for problem sizes which are multiples of ν can be computed using the vector FPU exclusively. Problems of other sizes are computed partly using the scalar FPU. This is achieved by applying Property 4.10 which decomposes any tensor product

$$A \otimes \mathrm{I}_{k\nu+l}$$

into a vector part

$$(A \otimes \mathrm{I}_k) \otimes \mathrm{I}_\nu$$

and a scalar part

$$A \otimes \mathrm{I}_l\,.$$

Applying this idea to (7.15) with $m = m_1\nu + m_2$ and $n = n_1\nu + n_2$ leads to the general subvector Cooley-Tukey FFT.

Theorem 7.2 (General Subvector Cooley-Tukey FFT) For $m = m_1\nu + m_2$ and $n = n_1\nu + n_2$

$$\begin{aligned} \overline{\mathrm{DFT}}_{mn} \;=\; & \left(\left(\overline{\mathrm{DFT}_m \otimes \mathrm{I}_{n_1}} \otimes \mathrm{I}_\nu\right) \oplus \left(\overline{\mathrm{DFT}}_m \otimes \mathrm{I}_{n_2}\right) \left(\overline{\mathrm{T}}^{mn}_{n}\right)^{P_1}\right)^{P_2} \\ & \left(\left(\mathrm{I}_{m_1} \otimes \overline{\mathrm{DFT}}_n \otimes \mathrm{I}_\nu\right) \oplus \left(\mathrm{I}_{m_2} \otimes \overline{\mathrm{DFT}}_n\right)\right)^{P_3} \left(\mathrm{L}^{mn}_{m} \otimes \mathrm{I}_2\right) \end{aligned}$$

with non-digit permutations P_i.

This method leads to high SIMD utilization while supporting arbitrary problem sizes.

The method summarized by Theorems 7.1 and 7.2 is called *Cooley-Tukey FFT for subvectors* as subvector memory access is required. It can be applied to for *arbitrary* problem sizes. Experimental results of a short vector SIMD extension of FFTW based on Theorems 7.1 and 7.2 are summarized in Section 8.1

Internal Vectorization of $\overline{\mathbf{DFT}_\mathbf{N}}$

By restricting the machines to 2-way short vector SIMD architectures ($\nu = 2$), the real arithmetic stride permutation appears as perfect vector operation $\overline{L_n^{mn} \otimes I_2}$. By computing the sub-problems $\overline{\mathrm{DFT}_m}$ and $\overline{\mathrm{DFT}_n}$ using vector instructions (i. e., not vectorizing the tensor product), one can get high performance implementations for the special case of $\nu = 2$. This approach depends crucially on the fact that a complex number can be expressed as a pair of real numbers and thus cannot be extended to arbitrary ν and is beyond the scope of this thesis. FFTW-GEL, a proprietary FFTW extensions for SSE 2 and 3DNow! using this idea can be found in Kral [64].

7.2.2 The Vector Cooley-Tukey Rule

Targeting classical vector processors (as described in Section 3.5) with typically $\nu \geq 64$ and the ability to load vectors at non-unit stride but featuring a rather high startup cost for vector operations, the Stockham autosort FFT algorithm (see Section 5.2.3) was designed. But as short vector SIMD extensions requires unit-stride memory access, this algorithm cannot be used for short vector SIMD architectures. In addition, the Stockham autosort algorithm is an iterative algorithm. Alternatively, the vector Cooley-Tukey FFT (also called four-step algorithm)

$$\mathrm{DFT}_{mn} = \underbrace{(\mathrm{DFT}_m \otimes \mathrm{I}_n)}_{4}\,\underbrace{\mathrm{T}_n^{mn}}_{3}\,\underbrace{\mathrm{L}_m^{mn}}_{2}\,\underbrace{(\mathrm{DFT}_n \otimes \mathrm{I}_m)}_{1}$$

as described in Section 5.2.1 also was designed for classical vector processors and $m \approx n$ is the usual choice. The vector Cooley-Tukey algorithm features unit-stride memory access (for complex numbers) and computes the stride permutation explicitly as one of its four stages. The four stages are (*i*) A vector FFT to compute $\mathrm{DFT}_n \otimes \mathrm{I}_m$, (*ii*) interpretation of the data vector as a $m \times n$ complex matrix and computing a transposition of it, (*iii*) a pointwise (vector) multiplication of the whole data vector with T_n^{mn}, and (*iv*) a vector FFT to compute $\mathrm{DFT}_m \otimes \mathrm{I}_n$.

This corresponds to an iterative interpretation of the four constructs in Theorem 5.3.

In case of short vector SIMD extensions, again the real arithmetic view is required. Straightforward application of the bar operator $\overline{(\ )}$ to Theorem 5.3

leads to

$$\begin{aligned}\overline{\mathrm{DFT}}_{mn} &= \left(\left(\overline{\mathrm{DFT}}_m \otimes \mathrm{I}_n\right) \overline{\mathrm{T}}_n^{mn}\right)^{\left(\mathrm{I}_m \otimes \mathrm{L}_2^{2n}\right)} \left(\mathrm{L}_m^{mn} \otimes \mathrm{I}_2\right) \\ &\quad \left(\overline{\mathrm{DFT}}_n \otimes \mathrm{I}_m\right)^{\left(\mathrm{I}_n \otimes \mathrm{L}_2^{2m}\right)}.\end{aligned} \tag{7.16}$$

Although using

$$\mathrm{DFT}_m \otimes \mathrm{I}_n = (\mathrm{DFT}_m \otimes \mathrm{I}_{\frac{n}{\nu}}) \otimes \mathrm{I}_\nu$$

and

$$\mathrm{DFT}_n \otimes \mathrm{I}_m = (\mathrm{DFT}_n \otimes \mathrm{I}_{\frac{m}{\nu}}) \otimes \mathrm{I}_\nu,$$

the computation phases are vector constructs. However, the twiddle factor matrix $\overline{\mathrm{T}}_n^{mn}$ does not match the equation (4.18) and the complex stride permutation $\mathrm{L}_m^{mn} \otimes \mathrm{I}_2$ is no vector construct for $\nu \neq 2$. In addition, the conjugation by $\mathrm{I}_m \otimes \mathrm{L}_2^{2n}$ and $\mathrm{I}_n \otimes \mathrm{L}_2^{2m}$ introduces large strides when accessing corresponding vectors of real and imaginary parts, which can result in cache problems. Thus, this formula cannot be used for efficient SIMD implementations without further modification.

Using the identities summarized in Chapter 4, the vector Cooley-Tukey FFT can be modified such that it is better suited for short vector SIMD implementation. This results in a short vector SIMD version of the vector Cooley-Tukey algorithm for real arithmetic.

Theorem 7.3 (Short Vector 4-Step FFT) For $\nu \mid m$ and $\nu \mid n$

$$\begin{aligned}\overline{\mathrm{DFT}}_{mn} &= \left(\left(\overline{\mathrm{DFT}_m \otimes \mathrm{I}_{\frac{n}{\nu}}} \otimes \mathrm{I}_\nu\right) \overline{\mathrm{T}}'^{mn}_{n}\right)^{\left(\mathrm{I}_{\frac{mn}{2\nu}} \otimes \mathrm{L}_2^{2\nu}\right)} \\ &\quad \left(\mathrm{I}_{\frac{m}{\nu}} \otimes \mathrm{L}_\nu^n \otimes \mathrm{I}_2 \otimes \mathrm{I}_\nu\right) \left(\mathrm{I}_{\frac{mn}{\nu^2}} \otimes \mathrm{L}_{\frac{\nu}{2}}^\nu \otimes \mathrm{L}_2^\nu \otimes \mathrm{I}_2\right) \left(\mathrm{L}_\nu^{\frac{mn}{\nu}} \otimes \mathrm{I}_2 \otimes \mathrm{I}_\nu\right) \\ &\quad \left(\overline{\mathrm{DFT}_n \otimes \mathrm{I}_{\frac{m}{\nu}}} \otimes \mathrm{I}_\nu\right)^{\left(\mathrm{I}_{\frac{mn}{2\nu}} \otimes \mathrm{L}_2^{2\nu}\right)}.\end{aligned}$$

However, Theorem 7.3 still features three drawbacks: (*i*) The computation requires at least three passes through the data, (*ii*) the stride permutation is done explicitly, and (*iii*) Theorem 7.3 exhibits many register-register permutations.

7.3 A Short Vector Cooley-Tukey Recursion

This section introduces a short vector Cooley-Tukey recursion, which (*i*) breaks down a DFT computation to vector constructs and vector terminals *exclusively*, (*ii*) features additional degrees of freedom to allow for searching the best implementation on a given platform, and (*iii*) allows for a recursive computation and does not introduce artificial stride problems.

The new recursion meets the requirements of the automatic performance tuning systems SPIRAL and FFTW and has been included as short vector SIMD extension into these systems. Run time results are presented in Sections 8.1 and 8.2.

The short vector Cooley-Tukey recursion is based on three classes of vector constructs: (*i*) Vector terminals, (*ii*) permutations, and (*iii*) twiddle factors.

Vector Terminals

The only vector terminals occurring in the new Cooley-Tukey recursion are either of the form

$$\overline{\mathrm{DFT}}_m \otimes \mathrm{I}_\nu$$

or

$$\overline{\mathrm{DFT}_m \otimes \mathrm{I}_{\frac{n}{\nu}}} \otimes \mathrm{I}_\nu \,.$$

In the implementations summarized in Chapter 8 these two constructs are further optimized using the automatic performance tuning facilities provided by SPIRAL and FFTW.

Permutations

Only permutations are required that can be factorized according to equation (7.7). Thus, the permutations W required are

$$\mathrm{I}_\nu \quad \mathrm{L}^{2\nu}_2 \quad \mathrm{L}^{2\nu}_\nu \quad \mathrm{L}^{\nu^2}_\nu$$

which are standard permutations supported by special instructions on most short vector SIMD architectures.

Twiddle Factors

Twiddle factors are diagonal matrices with complex numbers as entries. According to Section 4.6.1 such diagonals can be transformed into special vector constructs called *vector diagonals* by the application of the bar-prime operator$\overline{(\ )}'$. Thus, whenever twiddle factors are used in the context of the new short vector Cooley-Tukey rule set,

$$\overline{\mathrm{T}}'^{\,mn}_{n} = \overline{\mathrm{T}}^{\,mn\left(\mathrm{I}_{\frac{mn}{\nu}} \otimes \mathrm{L}^{2\nu}_{\nu}\right)}_{n}$$

is used.

All these constructs can be implemented efficiently on all current short vector SIMD extensions using the portable SIMD API defined in Chapter 6. As their size is always a multiple of ν and they are composed by the recursive rules in the right way, the required aligned memory access is guaranteed on a formal level.

7.3.1 Short Vector Cooley-Tukey Rules

This section introduces the set of rules needed to recursively decompose DFT_N into vector constructs and vector terminals. These rules are applicable for any $N = \nu^2 N_1$ and thus they are not restricted to powers of two and can be used on all current short vector SIMD extension. The rule set offers two types of degrees of freedom: (i) the way the vector terminals are obtained using the rules, and (ii) the way how the vector terminals are further expanded.

Within the short vector SIMD extension developed for the Spiral system these rules are applied in the extended version of the SPL compiler as described in Section 8.2. In the short vector SIMD extension developed for Fftw these rules are implemented as recursive functions which manage the call sequence and parameter values for the newly developed vector codelets which compute the action of vector terminals. See Section 8.1 for details.

The first rule is given by

$$\overline{\mathrm{DFT}}_{mn} = \underbrace{\overline{(\mathrm{DFT}_m \otimes \mathrm{I}_n)\,\mathrm{T}^{mn}_n}}_{(a)}\ \underbrace{\overline{(\mathrm{I}_m \otimes \mathrm{DFT}_n)\,\mathrm{L}^{mn}_m}}_{(b)}, \quad \nu \mid m, n. \tag{7.17}$$

This is the “entry rule”, as equation (7.17) decomposes an arbitrary DFT_{mn} with $\nu \mid m$ and $\nu \mid n$ *exclusively* into subformulas that are either vector terminals, vector diagonals, or formulas that are further handled by the rule set. $\overline{\mathrm{DFT}}_{mn}$ is decomposed into two constructs, (a) and (b), which both are handled further by the short vector Cooley-Tukey rules. Note, that the DFT is written as real matrix, and Property 4.28 has been applied to split the formula into two constructs.

The twiddle matrix T^{mn}_n can be associated with either construct (a) or (b) which is another degree of freedom. In the following it is associated with (a) to conform Fftw’s definition of codelets.[1] Both constructs are further decomposed using the short vector Cooley-Tukey rule set. Construct (a) is matched by

$$\overline{(\mathrm{DFT}_m \otimes \mathrm{I}_n)\,\mathrm{T}^{mn}_n} = \Big(\big(\underbrace{\overline{\mathrm{DFT}_m \otimes \mathrm{I}_{\frac{n}{\nu}}}}_{(c)} \otimes \mathrm{I}_\nu\big)\overline{\mathrm{T}}'^{mn}_{n}\Big)^{\left(\mathrm{I}_{\frac{mn}{\nu}} \otimes \mathrm{L}^{2\nu}_2\right)}, \quad \nu \mid n. \tag{7.18}$$

This rule consists only of vector terminals and vector diagonals and thus equation (7.18) is not further expanded using short vector Cooley-Tukey rules. The construct (c) in (7.18) is a vector terminal and is further handled by the target systems Spiral and Fftw. The most important degree of freedom is introduced by construct (b) in (7.17) which is matched by two rules in the short vector Cooley-Tukey rule set which are discussed in the remainder of this section.

The terminal rule for construct (b) in (7.17) is discussed first. This rule is built from vector constructs exclusively and thus provides for efficient implementation on all current short vector SIMD extensions. The required decomposition of the stride permutation is given by

$$(\mathrm{I}_k \otimes \mathrm{DFT}_n)\,\mathrm{L}^{kn}_k = \left(\mathrm{I}_{\frac{k}{\nu}} \otimes (\mathrm{L}^n_\nu \otimes \mathrm{I}_\nu)\left(\mathrm{I}_{\frac{n}{\nu}} \otimes \mathrm{L}^{\nu^2}_\nu\right)(\mathrm{DFT}_n \otimes \mathrm{I}_\nu)\right)\left(\mathrm{L}^{\frac{kn}{\nu}}_{\frac{k}{\nu}} \otimes \mathrm{I}_\nu\right). \tag{7.19}$$

Equation (7.19) follows from Property 4.26 and shows the decomposition of the stride permutation L^{kn}_k using complex formulas. This is the first of two step which breaks the stride permutation into two parts and partially turns parallel

---

[1] Within the Spiral short vector SIMD implementation this degree of freedom is used.

Kronecker factors into vector Kronecker factors. This moves two factors of the stride permutation factorization to the other side. The permutations

$$\mathrm{L}^{\frac{kn}{\nu}}_{\frac{k}{\nu}} \otimes \mathrm{I}_\nu \quad \text{and} \quad \mathrm{L}^{n}_{\nu} \otimes \mathrm{I}_\nu$$

permute blocks of size ν while the permutation

$$\mathrm{I}_{\frac{n}{\nu}} \otimes \mathrm{L}^{\nu^2}_{\nu}$$

is a permutation within blocks of size ν^2.

In a second step, the bar operator is applied. The identities from Section 4.6 are used to transform the real arithmetic formulas into the required shape

$$\begin{aligned}\overline{(\mathrm{I}_k \otimes \mathrm{DFT}_n)\,\mathrm{L}^{kn}_{k}} \;=\; & \left(\mathrm{I}_{\frac{kn}{\nu}} \otimes \mathrm{L}^{2\nu}_{\nu}\right) \\ & \left(\mathrm{I}_{\frac{k}{\nu}} \otimes \left(\mathrm{L}^{2n}_{\nu} \otimes \mathrm{I}_\nu\right) \left(\mathrm{I}_{\frac{2n}{\nu}} \otimes \mathrm{L}^{\nu^2}_{\nu}\right) \left(\underbrace{\overline{\mathrm{DFT}}_n}_{(f)} \otimes \mathrm{I}_\nu\right)\right) \\ & \left(\mathrm{L}^{\frac{kn}{\nu}}_{\frac{k}{\nu}} \otimes \mathrm{L}^{2\nu}_{2}\right), \quad \nu \mid k, n. \end{aligned} \tag{7.20}$$

Construct (f) again is a vector terminal which is handled by the target systems SPIRAL and FFTW. Equation (7.20) is one of the key elements to achieve a high-performance short vector FFTs.

When only applying equations (7.18) and (7.20) to equation (7.17), a formula is obtained that completely decomposes a DFT into vector constructs and vector terminals leading to the short vector Cooley-Tukey FFT.

Theorem 7.4 (Short Vector Cooley-Tukey FFT) Any DFT_N with $N = \nu^2 N_1$ can be transformed into a formula built *exclusively* from vector constructs by using the following equation.

$$\begin{aligned}\overline{\mathrm{DFT}}_{\nu^2 mn} \;=\; & \left(\mathrm{I}_{\nu mn} \otimes \mathrm{L}^{2\nu}_{\nu}\right) \left(\overline{\mathrm{DFT}_{\nu m} \otimes \mathrm{I}_n} \otimes \mathrm{I}_\nu\right) \overline{\mathrm{T}}'^{\nu^2 mn}_{\nu n} \\ & \left(\mathrm{I}_m \otimes \left(\mathrm{L}^{2\nu n}_{\nu} \otimes \mathrm{I}_\nu\right) \left(\mathrm{I}_{2n} \otimes \mathrm{L}^{\nu^2}_{\nu}\right) \left(\overline{\mathrm{DFT}}_{\nu n} \otimes \mathrm{I}_\nu\right)\right) \\ & \left(\mathrm{L}^{\nu mn}_{m} \otimes \mathrm{L}^{2\nu}_{2}\right). \end{aligned}$$

The only degree of freedom in Theorem 7.4 is the choice of m and n. However, construct (b) can also be expanded by

$$\begin{aligned}\overline{(\mathrm{I}_k \otimes \mathrm{DFT}_{mn})\,\mathrm{L}^{kmn}_{k}} \;=\; & \left(\mathrm{I}_k \otimes \underbrace{\overline{(\mathrm{DFT}_m \otimes \mathrm{I}_n)\,\mathrm{T}^{mn}_{n}}}_{(d)}\right) \\ & \left(\mathrm{L}^{km}_{k} \otimes \mathrm{I}_{2n}\right) \left(\mathrm{I}_m \otimes \underbrace{\overline{(\mathrm{I}_k \otimes \mathrm{DFT}_n)\,\mathrm{L}^{kn}_{k}}}_{(e)}\right) \left(\mathrm{L}^{mn}_{m} \otimes \mathrm{I}_{2k}\right). \end{aligned} \tag{7.21}$$

(7.21) is a complex version of Theorem 5.5 and consists of two constructs. Construct (d) again is matched by equation (7.18). Construct (e) has the same structure as construct (*b*) and thus again offers the freedom to choose either equation (7.20) or (7.21) for further expansion. Equation (7.21) has a strong impact on data locality. The important degree of freedom is the choice of the recursion level where (7.20) is used instead of (7.21) and accordingly switching to vector terminals takes place.

In addition all intermediate permutations $\mathrm{L}_\nu^{2\nu}$ and $\mathrm{L}_2^{2\nu}$ introduced by the conjugations cancel according to Properties 4.1 and 4.1 which is very important to avoid additional data reorganization steps. Applying these properties to equation (7.20) and to equation (7.21) leads to the following rule set.

Theorem 7.5 (Short Vector Cooley-Tukey Rule Set) Any DFT_N with $N = \nu^2 N_1$ can be transformed into a formula built *exclusively* from vector constructs by using the following equations.

$$\begin{aligned}\overline{\mathrm{DFT}}_{\nu^2 mn} \;=\;& \overline{(\mathrm{DFT}_{\nu m} \otimes \mathrm{I}_{\nu n})\, \mathrm{T}_{\nu n}^{\nu^2 mn}} \left(\mathrm{I}_{\nu mn} \otimes \mathrm{L}_\nu^{2\nu}\right) \\ & \left(\mathrm{I}_{\nu mn} \otimes \mathrm{L}_2^{2\nu}\right) \overline{(\mathrm{I}_{\nu m} \otimes \mathrm{DFT}_{\nu n})\, \mathrm{L}_{\nu m}^{\nu^2 mn}}\end{aligned} \tag{7.22}$$

$$\begin{aligned}\overline{(\mathrm{DFT}_m \otimes \mathrm{I}_{\nu n})\, \mathrm{T}_{\nu n}^{\nu mn}}& \\ \left(\mathrm{I}_{mn} \otimes \mathrm{L}_\nu^{2\nu}\right) \;=\;& \left(\mathrm{I}_{mn} \otimes \mathrm{L}_\nu^{2\nu}\right) \\ & \left(\left(\overline{\mathrm{DFT}_m \otimes \mathrm{I}_n} \otimes \mathrm{I}_\nu\right) \overline{\mathrm{T}}'^{\nu mn}_{\nu n}\right)\end{aligned} \tag{7.23}$$

$$\overline{(\mathrm{DFT}_m \otimes \mathrm{I}_{\nu n})\, \mathrm{T}_{\nu n}^{\nu mn}}^{\mathrm{I}_{mn} \otimes \mathrm{L}_\nu^{2\nu}} = \left(\left(\overline{\mathrm{DFT}_m \otimes \mathrm{I}_n} \otimes \mathrm{I}_\nu\right) \overline{\mathrm{T}}'^{\nu mn}_{\nu n}\right) \tag{7.24}$$

$$\begin{aligned}\left(\mathrm{I}_{\nu mn} \otimes \mathrm{L}_2^{2\nu}\right)& \\ \overline{(\mathrm{I}_{\nu m} \otimes \mathrm{DFT}_{\nu n})\, \mathrm{L}_{\nu m}^{\nu^2 mn}} \;=\;& \left(\mathrm{I}_m \otimes \left(\mathrm{L}_\nu^{2\nu n} \otimes \mathrm{I}_\nu\right) \left(\mathrm{I}_{2n} \otimes \mathrm{L}_\nu^{\nu^2}\right) \left(\overline{\mathrm{DFT}}_{\nu n} \otimes \mathrm{I}_\nu\right)\right) \\ & \left(\mathrm{L}_m^{\nu mn} \otimes \mathrm{L}_2^{2\nu}\right)\end{aligned} \tag{7.25}$$

$$\begin{aligned}\left(\mathrm{I}_{\nu m k_1 k_2 n} \otimes \mathrm{L}_2^{2\nu}\right)& \\ \overline{(\mathrm{I}_{\nu m} \otimes \mathrm{DFT}_{\nu k_1 k_2 n})\, \mathrm{L}_{\nu m}^{\nu^2 m k_1 k_2 n}} \;=\;& \left(\mathrm{I}_{\nu m} \otimes \overline{(\mathrm{DFT}_{k_1} \otimes \mathrm{I}_{\nu k_2 n})\, \mathrm{T}_{\nu k_2 n}^{\nu k_1 k_2 n}}^{\mathrm{I}_{k_1 k_2 n} \otimes \mathrm{L}_\nu^{2\nu}}\right) \\ & \left(\mathrm{L}_{\nu m}^{\nu k_1 m} \otimes \mathrm{I}_{2\nu k_2 n}\right) \\ & \left(\mathrm{I}_{k_1} \otimes \left(\mathrm{I}_{\nu m k_2 n} \otimes \mathrm{L}_2^{2\nu}\right) \overline{(\mathrm{I}_{\nu m} \otimes \mathrm{DFT}_{\nu k_2 n})\, \mathrm{L}_{\nu m}^{\nu^2 m k_2 n}}\right) \\ & \left(\mathrm{L}_{k_1}^{\nu k_1 k_2 n} \otimes \mathrm{I}_{2\nu m}\right)\end{aligned} \tag{7.26}$$

Note, that this set of rules also exists as *transposed* rules, starting from the transposed version of Theorem 5.1, i. e., Theorem 5.2.

Example 7.7 (Short Vector Cooley-Tukey Rule Set for $\mathbf{N} = \boldsymbol{\nu}^{\mathbf{2}}\mathbf{rstu}$) This example shows the application of the short vector Cooley-Tukey rule for four factors. The problem size has to be a multiple of ν^2. In this example, the four factors are r, s, t, and u. This is the smallest example which enables to study *all* equations of the rule set. The example is analogous to Example 5.10, but the formula manipulation is done in the real arithmetic formulation and applying the short vector Cooley-Tukey rules instead of the Cooley-Tukey vector recursion.

In a first step equation (7.22) is applied to the initial transform $\mathrm{DFT}_{\nu^2 rstu}$ with $m = r$ and $n = stu$ leading to

$$\begin{aligned} \overline{\mathrm{DFT}}_{\nu^2 rstu} \quad = \quad & \underbrace{\overline{(\mathrm{DFT}_{r\nu} \otimes \mathrm{I}_{\nu stu})\, \mathrm{T}^{\nu^2 rstu}_{\nu stu}}\, (\mathrm{I}_{\nu rstu} \otimes \mathrm{L}^{2\nu}_{\nu})}_{(a)} \\ & \underbrace{(\mathrm{I}_{\nu rstu} \otimes \mathrm{L}^{2\nu}_{2})\, \overline{(\mathrm{I}_{r\nu} \otimes \mathrm{DFT}_{\nu stu})\, \mathrm{L}^{\nu^2 rstu}_{r\nu}}}_{(b)} . \end{aligned} \tag{7.27}$$

Construct (a) in (7.27) is further expanded using rule (7.23) leading to

$$(\mathrm{I}_{\nu rstu} \otimes \mathrm{L}^{2\nu}_{\nu}) \left((\overline{\mathrm{DFT}_{r\nu} \otimes \mathrm{I}_{stu}} \otimes \mathrm{I}_{\nu})\, \overline{\mathrm{T}}'^{\,\nu^2 rstu}_{\nu stu} \right). \tag{7.28}$$

Construct (b) in (7.27) is further expanded using rule (7.26) with $m = n$, $k_1 = s$, $k_2 = t$, and $n = u$ leading to

$$\begin{aligned} & \left(\mathrm{I}_{\nu r} \otimes \underbrace{\overline{(\mathrm{DFT}_{s} \otimes \mathrm{I}_{\nu tu})\, \mathrm{T}^{\nu stu}_{\nu tu}}^{\mathrm{I}_{stu} \otimes \mathrm{L}^{2\nu}_{\nu}}}_{(c)} \right) \\ & (\mathrm{L}^{\nu sr}_{\nu r} \otimes \mathrm{I}_{2\nu tu}) \left(\mathrm{I}_{s} \otimes \underbrace{(\mathrm{I}_{\nu rtu} \otimes \mathrm{L}^{2\nu}_{2})\, \overline{(\mathrm{I}_{\nu r} \otimes \mathrm{DFT}_{\nu tu})\, \mathrm{L}^{\nu^2 rtu}_{\nu r}}}_{(d)} \right) (\mathrm{L}^{\nu stu}_{s} \otimes \mathrm{I}_{2\nu r}) . \end{aligned} \tag{7.29}$$

Construct (c) in (7.29) is expanded using rule (7.24) with $m = s$ and $n = tu$ leading to

$$(\overline{\mathrm{DFT}_{s} \otimes \mathrm{I}_{tu}} \otimes \mathrm{I}_{\nu})\, \overline{\mathrm{T}}'^{\,\nu stu}_{\nu tu} . \tag{7.30}$$

Construct (d) in (7.29) is again expanded using rule (7.26) with $m = n$, $k_1 = t$, $k_2 = 1$, and $n = u$ leading to

$$\begin{aligned} & \left(\mathrm{I}_{\nu r} \otimes \underbrace{\overline{(\mathrm{DFT}_{t} \otimes \mathrm{I}_{\nu u})\, \mathrm{T}^{\nu tu}_{\nu u}}^{\mathrm{I}_{tu} \otimes \mathrm{L}^{2\nu}_{\nu}}}_{(e)} \right) \\ & (\mathrm{L}^{\nu rt}_{\nu r} \otimes \mathrm{I}_{2\nu u}) \left(\mathrm{I}_{t} \otimes \underbrace{(\mathrm{I}_{\nu ru} \otimes \mathrm{L}^{2\nu}_{2})\, \overline{(\mathrm{I}_{\nu r} \otimes \mathrm{DFT}_{\nu u})\, \mathrm{L}^{\nu^2 ru}_{\nu r}}}_{(f)} \right) (\mathrm{L}^{\nu tu}_{t} \otimes \mathrm{I}_{2\nu r}) . \end{aligned} \tag{7.31}$$

Construct (e) in (7.31) is expanded using rule (7.24) with $m = t$ and $n = u$ leading to

$$(\overline{\mathrm{DFT}_{t} \otimes \mathrm{I}_{u}} \otimes \mathrm{I}_{\nu})\, \overline{\mathrm{T}}'^{\,\nu tu}_{\nu u} . \tag{7.32}$$

Construct (f) in (7.31) is expanded using rule (7.25) with $m = r$ and $n = u$ leading to

$$\left(\mathrm{I}_{r} \otimes (\mathrm{L}^{2\nu u}_{\nu} \otimes \mathrm{I}_{\nu}) \left(\mathrm{I}_{2u} \otimes \mathrm{L}^{\nu^2}_{\nu} \right) (\overline{\mathrm{DFT}}_{\nu u} \otimes \mathrm{I}_{\nu}) \right) (\mathrm{L}^{\nu ru}_{r} \otimes \mathrm{L}^{2\nu}_{2}) . \tag{7.33}$$

In the final step the equations are substituted back. Thus equations (7.32) and (7.33) are substituted into (7.31), (7.30) and (7.31) are substituted into (7.29), and (7.28) and (7.29) are substituted into (7.27). Finally, the fully expanded equation is obtained:

$$\begin{aligned}
\overline{\mathrm{DFT}}_{\nu^2 rstu} \quad = \quad & \underbrace{\Big(\mathrm{I}_{\nu rstu}\otimes\mathrm{L}^{2\nu}_{\nu}\Big)\Big(\big(\overline{\mathrm{DFT}_{r\nu}\otimes\mathrm{I}_{stu}}\otimes\mathrm{I}_{\nu}\big)\overline{\mathrm{T}}'^{\,\nu^2 rstu}_{\nu stu}\Big)}_{(a)} \\
& \Big(\mathrm{I}_{\nu r}\otimes\underbrace{\big(\overline{\mathrm{DFT}_{s}\otimes\mathrm{I}_{tu}}\otimes\mathrm{I}_{\nu}\big)\overline{\mathrm{T}}'^{\,\nu stu}_{\nu tu}}_{(c)}\Big) \\
& \Big(\mathrm{L}^{\nu sr}_{\nu r}\otimes\mathrm{I}_{2\nu tu}\Big) \\
& \Big(\mathrm{I}_{s}\otimes\Big(\mathrm{I}_{\nu r}\otimes\underbrace{\big(\overline{\mathrm{DFT}_{t}\otimes\mathrm{I}_{u}}\otimes\mathrm{I}_{\nu}\big)\overline{\mathrm{T}}'^{\,\nu tu}_{\nu u}}_{(e)} \\
& \qquad \Big(\mathrm{L}^{\nu rt}_{\nu r}\otimes\mathrm{I}_{2\nu u}\Big) \\
& \qquad \Big(\mathrm{I}_{t}\otimes\Big(\underbrace{\mathrm{I}_{r}\otimes\big(\mathrm{L}^{2\nu u}_{\nu}\otimes\mathrm{I}_{\nu}\big)\big(\mathrm{I}_{2u}\otimes\mathrm{L}^{\nu^2}_{\nu}\big)\big(\overline{\mathrm{DFT}}_{\nu u}\otimes\mathrm{I}_{\nu}\big)\Big)\big(\mathrm{L}^{\nu ru}_{r}\otimes\mathrm{L}^{2\nu}_{2}\big)}_{(f)}\Big) \\
& \qquad \Big(\mathrm{L}^{\nu tu}_{t}\otimes\mathrm{I}_{2\nu r}\Big)\Big) \\
& \Big(\mathrm{L}^{\nu stu}_{s}\otimes\mathrm{I}_{2\nu r}\Big)
\end{aligned} \tag{7.34}$$

In the resulting equation (7.34) all basic constructs are vector constructs matching

$$A\otimes\mathrm{I}_{\nu},\quad \mathrm{L}^{2\nu}_{2},\quad \mathrm{L}^{2\nu}_{\nu},\quad \mathrm{L}^{\nu^2}_{\nu},\ \text{or}\quad \overline{\mathrm{T}}'^{\,mn}_{n}.$$

Thus, equation (7.34) can be implemented utilizing vector memory access and in-register permutations exclusively.

Example 7.7 shows, how the short vector Cooley-Tukey rules are applied for a small example. In the real application, the utilization of a computer algebra system like GAP (which is used by SPIRAL) or direct coding of the recursion rules as done by FFTW's executor is required to exploit the following degrees of freedom: (*i*) how many factors are used, (*ii*) how are the factors chosen, (*iii*) which rule is applied, and (*iv*) how are the vector terminals further expanded. This provides a rich search space for the application of automatic empirical performance tuning.

The most important property of the short-vector Cooley-Tukey rule set is that the formal approach *guarantees* that the resulting formula can be implemented with vector arithmetic, vector memory access operations, and efficient in-register permutations *exclusively*. In Chapter 8 both an implementation of Theorem 7.4 and Theorem 7.5 is described.

7.4 Short Vector Specific Search

In order to achieve high performance on short vector SIMD architectures, it is necessary to apply search methods to empirically find a good implementation.

This section discusses, how the search methods provided by SPIRAL have to be adopted to support the short vector Cooley-Tukey rule set.

SPIRAL provides different search methods to find the best algorithm for a given computing platform, including dynamic programming (Cormen et al. [14]), STEER (an evolutionary algorithm by Singer and Veloso [84]), a hill climbing search, and exhaustive search.

For scalar DFT implementations, it turns out that in general dynamic programming is a good choice since it terminates fast (using only Theorem 7.22, dynamic programming does at the order of $O(n^2)$ run time experiments, where n is the transform size) and finds close to the best implementations (Püschel et al. [80]).

In the case of short vector SIMD implementations, it turns out that dynamic programming fails to find (nearly) best algorithms, as the run time of subproblems becomes very context sensitive and dynamic programming assumes context independence (the *optimal subproblem asumption* is no longer valid). This is explained in the following and two variations of dynamic programming are presented that are included in the SPIRAL search engine to overcome this problem. In Section 8.2.2 the various dynamic programming variants are evaluated experimentally.

7.4.1 Standard Dynamic Programming

Dynamic programming searches for the best implementation for a given transform recursively. First of all, the list of all possible child transforms (with respect to all applicable rules) is generated and the best implementation for each possible child transform is determined by using dynamic programming recursively. Then the best implementation for the transform is determined by applying all possible rules and plugging in the already known solutions for the children. Since the child transforms are smaller than the original construct, this process terminates. For a DFT_{2^n}, using Theorem 7.22, dynamic programming can be used to find the best implementations for DFT_{2^k}, $k = 1, 2, \ldots, n$ in *one* search run for DFT_{2^n}.

The method works well for scalar code, but for vector code the method is flawed. At some stage, rule (7.25) has to be applied to obtain vector terminals. Using dynamic programming recursively on the obtained children would inevitably apply this rule again, even though the children are vector terminals, i.e., should rather be expanded using rule (7.22). As a result, wrong breakdown strategies are found.

7.4.2 Vector Dynamic Programming

The first obvious change is to disable the vector rule (7.25) for vector terminals. This already leads to reasonably structured formulas. But there is a second problem: dynamic programming optimizes all vector terminals like $\overline{\mathrm{DFT}_k}$ as scalar constructs thus not taking into account the vector tensor product, i. e., ignoring the context $\overline{\mathrm{DFT}}_k \otimes \mathrm{I}_\nu$ of $\overline{\mathrm{DFT}}_k$. Thus, a second modification is made by expanding $\overline{\mathrm{DFT}}_k$ by using scalar rules but always measuring the run time of the vector code generated from $\overline{\mathrm{DFT}}_k \otimes \mathrm{I}_\nu$. For the other construct containing a vector terminal in (7.25), i. e., $\overline{\mathrm{DFT}_m \otimes \mathrm{I}_{\frac{n}{\nu}}} \otimes \mathrm{I}_\nu$, also $\overline{\mathrm{DFT}}_m \otimes \mathrm{I}_\nu$ is measured, independently of n/ν.

7.4.3 Stride Sensitive Vector Dynamic Programming

This variant is directly matched to rule (7.25). For a given DFT_N, this search variant first creates all possible pairs (m, n) with $N = mn$. For any pair (m, n), it searches the best implementation of the vector terminals required by rule (7.25) using vector dynamic programming. But when searching for the best DFT_m by using vector dynamic programming a variant is used that finally measures $\overline{\mathrm{DFT}_m \otimes \mathrm{I}_{\frac{n}{\nu}}} \otimes \mathrm{I}_\nu$ instead of $\overline{\mathrm{DFT}}_m \otimes \mathrm{I}_\nu$, which makes the search sensitive to the stride n/ν. A fast formula for DFT_n is found by using to standard vector dynamic programming. This exactly optimizes the required vector terminals, including the stride. This dynamic programming variant requires much more run time measurements compared to the other two dynamic programming variants and thus saving the results for earlier measured pairs (m, n) speeds up the search crucially.

A second variant of the stride sensitive vector dynamic programming was developed, where the search operation is done without reusing intermediate results across different runs of dynamic programming. This leads to a context and stride sensitive version which is subsequently called “nohash” variant.

Chapter 8

Experimental Results

This chapter presents the experimental evaluation of the methods developed in Chapter 7. To obtain relevant performance results, the newly developed methods have been included into the two state-of-the-art hardware adaptive systems for computing discrete linear transforms, i. e., Spiral and Fftw.

Fftw has been extended to utilize short vector SIMD instructions when computing complex-to-complex DFTs supported by the original version of Fftw, independently of the data stride and the problem size, i. e., the problem size has not been restricted to powers of two. Section 8.1 describes how the Cooley-Tukey FFT for subvectors (developed in Section 7.2) and the short vector Cooley-Tukey FFT (developed in Section 7.3) have been included into Fftw and evaluated across various short vector SIMD extensions and platforms.

The Spiral system has been extended to produce short vector SIMD implementations for any discrete linear transform supported by the original system. The general formula based vectorization method developed in Section 7.1 has been tested with the WHT and two-dimensional DCTs of type II, while the short vector Cooley-Tukey FFT developed in Section 7.3 is used for DFT implementations for problem sizes being powers of two.

The experiments presented in Section 8.2 include a detailed run time analysis, an analysis of the short vector SIMD specific dynamic programming method introduced in Section 7.4, as well as an analysis of the best performing codes with respect to their recursive structure, and a survey of the applicability of vectorizing compilers for short vector SIMD extensions.

All performance values were obtained using the original timing routines provided by Spiral and Fftw. Third-party codes were assessed using these timing routines as well. Details on performance assessment for scientific software can be found in Appendix A and in Gansterer and Ueberhuber [36]. All codes for complex transforms utilize the *interleaved complex* format (described in Section 4.6) unless noted differently.

8.1 A Short Vector Extension for FFTW

In the scope of this thesis, Fftw is extended to portably support short vector SIMD extensions based on two methods developed in Chapter 7, (*i*) the Cooley-Tukey FFT for subvectors, and (*ii*) the short vector Cooley-Tukey FFT.

The necessary modifications are hidden within the Fftw framework such that the utilization of the short vector SIMD extensions is transparent for the user. The system automatically chooses the problem-dependent optimum method to utilize the available hardware. Specifically, the following support was included into Fftw.

- Applying the Cooley-Tukey rule for subvectors, support for (*i*) problem sizes which are multiples of ν^2 based on Theorem 7.1, and (*ii*) for *arbitrary* problem sizes based on Theorem 7.2 is provided. This method supports all features of Fftw for complex-to-complex DFT computations, including arbitrary problem sizes and arbitrary strides.
- Support for problem sizes which are multiples of ν^2 based on the short vector Cooley-Tukey rule set (Theorem 7.5) is provided. This method achieves higher performance but can only be used with a subset of problem sizes and with unit stride only (in the case of four-way short vector SIMD extensions).

Fftw does not directly utilize the formal methods presented in Chapter 4. In case of the short vector SIMD extension an implementation based on Theorems 7.1 and 7.2 as well as the short vector Cooley-Tukey rule set (Theorem 7.5) was included into Fftw which itself is based on the Cooley-Tukey recursion as summarized in Section 5.2.5.

The formal methods developed in this thesis are a valuable tool to study and express the computations carried out by the Fftw short vector SIMD extension developed in this thesis.

8.1.1 Extending FFTW

In order to include the newly developed methods into the Fftw framework, several changes to the original system were required. Due to technical considerations, the experimental version `nfftw2` that is not publicly available was chosen as the version to be extended. Implementation of the short vector Cooley-Tukey rule set would not have been possible without access to `nfftw2`.

This section only gives a high-level overview of the implementation of the Fftw short vector SIMD extension. The reader may refer to the documentation of the latest official release, i. e., Fftw 2.1.3 which provides sufficient information. Throughout this section, the nomenclature of Fftw 2.1.3 is used.

The Vector Codelets

Fftw features one twiddle codelet and one no-twiddle codelet per codelet size. For the short vector SIMD extension new types of scalar and vector codelets are required which feature additional support for permutations. Vector codelets are specialized symbols and thus follow the approach described in Section 7.1.4.

Vector twiddle codelets have the following general structure:

Load phase: $x' = E_i Q_i \, x$,

Computation phase: $y' = (\overline{\mathrm{DFT}_N \otimes \mathrm{I}_{\frac{k}{\nu}}} \otimes \mathrm{I}_\nu) \, x'$,

Store phase: $y = P_i \, y'$.

Vector no-twiddle codelets have the following structure:

Load phase: $x' = Q_i \, x$,

Computation phase: $y' = (\overline{\mathrm{DFT}}_N \otimes \mathrm{I}_\nu) \, x'$,

Store phase: $y = P_i \, y'$.

The codelet generator `genfft` is extended following the ideas from Section 7.1.4, thus it (i) utilizes `genfft`'s core routines to generate code for the computational cores, and (ii) uses a modified unparsing routine to enable the support for P_i and Q_i. P_i and Q_i originate from either $\mathrm{L}_k^{2\nu}$ or $\mathrm{L}_\nu^{\nu^2}$ and the complex diagonal E_i provides the required support for the twiddle factors. These constructs and the arithmetic vector operations required within the computational core of a codelet are implemented using the portable SIMD API.

Although the computational core is the same for all twiddle vector codelets of a specific size on the one hand as well as for all no-twiddle vector codelets of a specific size on the other hand, different versions of vector codelets are required to cope with data alignment and the permutations required by the short vector Cooley-Tukey rule set. The differences between the short vector Cooley-Tukey rule set and the Cooley-Tukey rules for subvectors are hidden in the permutation macros that support P_i and Q_i. Appendix E.2 shows a scalar no-twiddle codelet of size four and Appendix E.3 shows one of the respective vector codelets.

The Vector Framework

The standard FFTW framework had to be extended to support the multitude of codelets. The applicability of a certain codelet had to be managed as well as the various methods to support the short vector SIMD extensions. This is the major reason that the experimental version `nfftw2` was used as the required support was gained more easily compared to extending FFTW 2.1.3. Both the *executor* and the *planner* required changes, thus all major parts of FFTW had to be adopted.

8.1.2 Run Time Experiments

The short vector SIMD extension of FFTW, called FFTW-SIMD, was compared to the standard scalar version of FFTW which served as baseline. As `nfftw2` was extended, this version was used as scalar version. However, for the problem specifications that were analyzed, `nfftw2` delivers about the same performance as the last public release FFTW 2.1.3.

The Test Machines

FFTW-SIMD has been investigated on three IA-32 compatible machines and using one Motorola Power PC machine: (*i*) Intel Pentium 4 running at 1.8 GHz, (*ii*) Intel Pentium III running at 650 MHz, (*iii*) AMD Athlon XP 1800+ running at 1533 MHz, and (*iv*) Motorola MPC7400 G4 running at 400 MHz. Details about the machines can be found in Table 8.1.

The SSE version was tested on the Pentium III, the Pentium 4, and the AMD Athlon XP. The SSE 2 version was tested on the Pentium 4. The AltiVec version was tested on the MPC7400 G4.

For all IA-32 compatible machines the Intel C++ 6.0 compiler with experimentally determined compiler options, optimized for the specific machine, was used. All experiments on IA-32 compatible machines were conducted using the operating system Microsoft WINDOWS 2000 and were validated on the same machines running under RedHat Linux 7.2 and using the Linux Intel C++ 6.0 compiler.

The experiments on the Motorola MPC7400 G4 were conducted under Yellowdog Linux 1.2 and the AltiVec version of the GNU C 2.9. compiler.

For power of two problem sizes on IA-32 compatible machines the performance of the scalar and short vector SIMD version of FFTW was compared to the hand-optimized vendor library Intel MKL 5.1 [58]. Non-powers of two cannot be compared to the Intel MKL due to the missing support by Intel's library.

The performance is displayed in pseudo Gflop/s, i. e., $(5N \log N)/T$, which is a scaled inverse of the run time T and thus preserves run time relations and additionally gives an indication of the absolute performance. Details on the performance unit *pseudo flop/s* can be found in Appendix A and in Frigo and Johnson [33].

Note that SSE has hardware support for loading subvectors of size two while that is a costly operation on the AltiVec extension. On the other hand, AltiVec internally operates on vectors of size four while on all machines featuring SSE the four-way operations are broken internally into two two-way operations thus limiting the vectorization speed-up. However, due to other architectural implications the speed-up achievable by vectorization (ignoring effects like smaller program size due to fewer instructions) is limited by a *factor of four* for SSE on the Pentium III, Pentium 4 and for AltiVec on the MPC7400 G4. For SSE on the Athlon XP and SSE 2 on the Pentium 4 the upper speed-up limit is a *factor of two*.

The Short Vector Cooley-Tukey Rule Set, Powers of Two

This section describes the experimental results which were obtained by investigating the FFTW implementation of the short vector Cooley-Tukey rule set

CPU	**Intel Pentium 4** **1.8 GHz** 8 kB L1 Data Cache 256 kB on-die L2 Cache 7.2 Gflop/s single-precision 3.6 Gflop/s double-precision SSE and SSE 2
RAM	256 MB RDRAM
Operating System	Microsoft Windows 2000
Compiler	Intel C++ Compiler 6.0

CPU	**Intel Pentium III Coppermine** **650 MHz** 16 kB L1 Data Cache 256 kB on-die L2 Cache 7.2 Gflop/s single-precision SSE
RAM	128 MB SDRAM
Operating System	Microsoft Windows 2000
Compiler	Intel C++ Compiler 6.0

CPU	**AMD Athlon XP 1800+** **1533 MHz** 64 kB L1 Data Cache 256 kB on-die L2 Cache 6.1 Gflop/s single-precision 3DNow! Professional (SSE compatible)
RAM	128 MB DDR-SDRAM
Operating System	Microsoft Windows 2000
Compiler	Intel C++ Compiler 6.0

CPU	**Motorola 7400 G4** **400 MHz** 32 kB L1 Data Cache 1024 kB unified L2 Cache 3.2 Gflop/s single-precision AltiVec
RAM	128 MB SDRAM
Operating System	Yellodog Linux 1.2
Compiler	Gnu C Compiler 2.9.5 for AltiVec

Table 8.1: The computer systems used for assessing Fftw's short vector SIMD extension.

developed in Section 7.3. Figures 8.1 to 8.4 show the floating-point performance of the short vector Cooley-Tukey rule set applied to power of two problem sizes tested across all IA-32 compatible machines within the test pool. Both SSE and SSE 2 was tested. Concerning four-way short vector SIMD extensions, these experiments show that that the short vector Cooley-Tukey rule set is superior whenever applicable. The data format used for these experiments is—in contrast to most other experiments—the *split complex format* summarized in Section 4.6.

On the Pentium 4 the highest performance is achieved for problem sizes that fit completely into L1 cache and are not too small, i. e., $N = 64$ or 128. The performance then decreases for problems that fit into L2 cache but do not fit into L1 cache. Speed-ups of more than three are achieved for SSE and more than 1.8 for SSE 2 for data sets that fit into L1 data cache. For data sets that fit into L2 cache but not into L1 cache, speed-ups of 2.5 for SSE and 1.7 for SSE 2 were achieved. This is due to the memory subsystem and the high computational power of the Pentium 4 in combination with the memory access patterns required by the vector recursion and FFTW's restriction to right-expanded trees. For data sets that fit into L1 cache, the Intel MKL is slower than FFTW-SIMD. Intel's MKL is faster than FFTW-SIMD for some problem sizes where the data set does not fit into L1 cache. This is due to (*i*) the usage of prefetching, and (*ii*) due to the fact that in-place computation is used in contrast to FFTW which features out-of-place computation resulting in higher memory requirements. See Figures 8.1 and 8.2 for details.

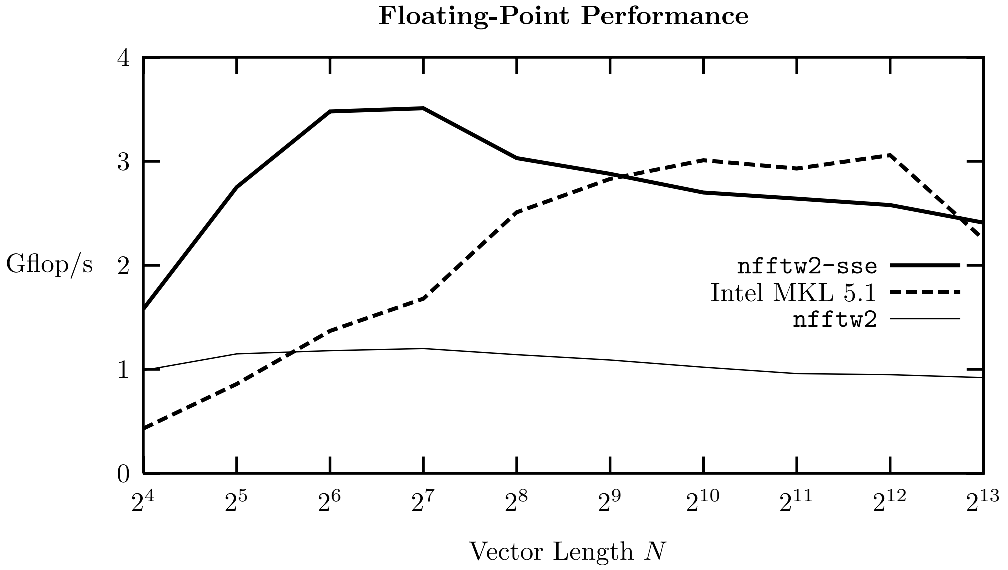


Figure 8.1: FFTW, Short Vector Cooley-Tukey rule set: Performance results for DFT_N with $N = 2^4, 2^5, \dots, 2^{13}$, for single-precision and SSE on an Intel Pentium 4 running at 1.8 GHz.

On the Pentium III the shape of the performance graph looks different. The maximum performance is obtained for $N = 256$ and $N = 512$. The performance slowly decreases for both smaller and larger problem sizes. This is due to the differences in the cache architecture between the Pentium III and the Pentium 4. Speed-up factors of up to 2.8 have been achieved for SSE on the Pentium III. The performance of the Intel MKL is at least 20 % below the performance of FFTW-SIMD. See Figure 8.3 for details.

The performance graph for the Athlon XP looks quite similar to the graph obtained for the Pentium III. The notable differences are (*i*) speed-up factors

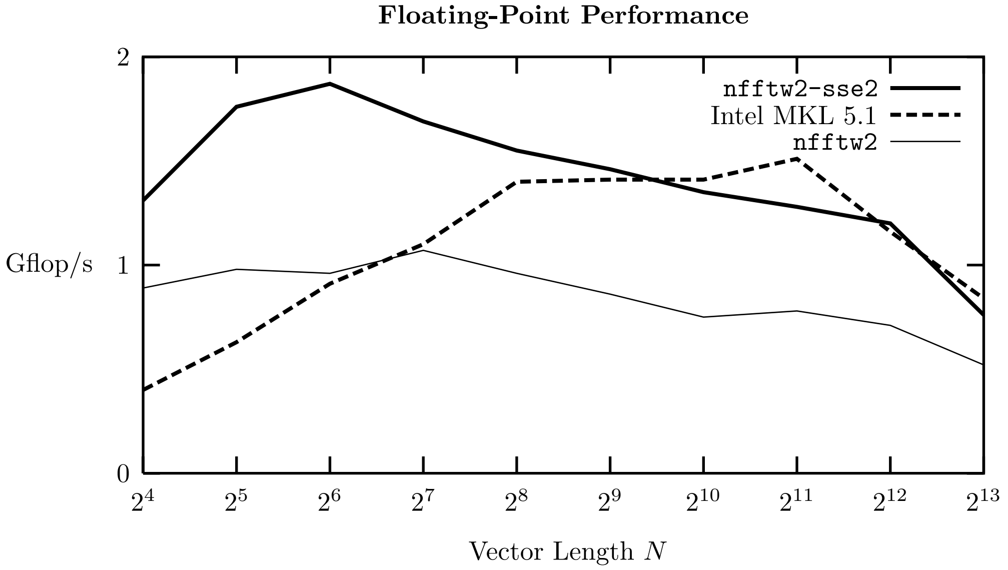


Figure 8.2: FFTW, Short Vector Cooley-Tukey rule set: Performance results for DFT_N with $N = 2^4, 2^5, \ldots, 2^{13}$, for double-precision and SSE 2 on an Intel Pentium 4 running at 1.8 GHz.

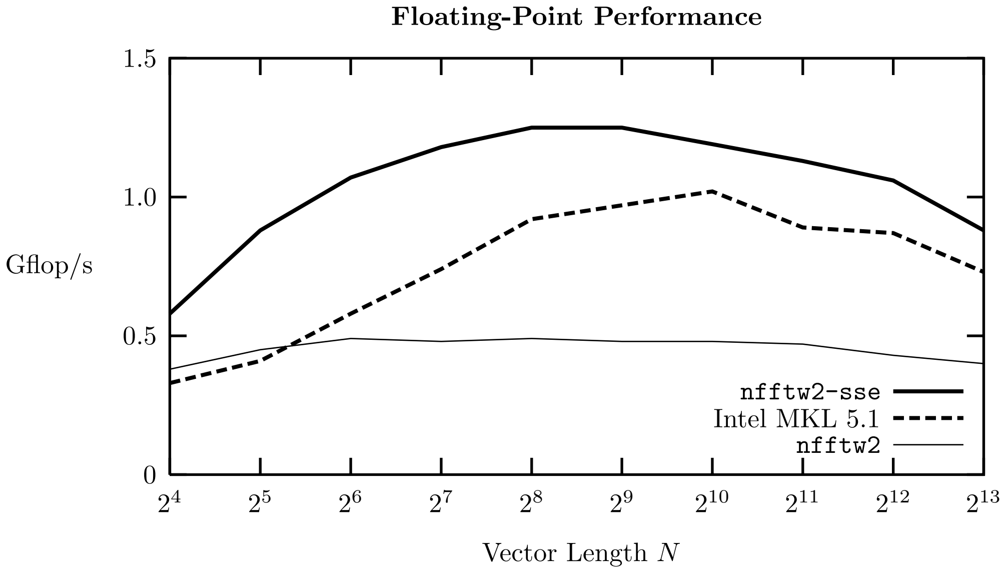


Figure 8.3: FFTW, Short Vector Cooley-Tukey rule set: Performance results for DFT_N with $N = 2^4, 2^5, \ldots, 2^{13}$, for single-precision and SSE on an Intel Pentium III running at 650 MHz.

are lower due to the different microarchitecture as explained at the beginning of this section, and (*ii*) Intel's MKL library performs relatively better than on the Pentium III, although it reaches the performance of FFTW-SIMD only for $N = 1024$ and $N = 2048$. FFTW-SIMD achieves speed-ups of up to 1.8 on the Athlon XP. See Figure 8.4 for details.

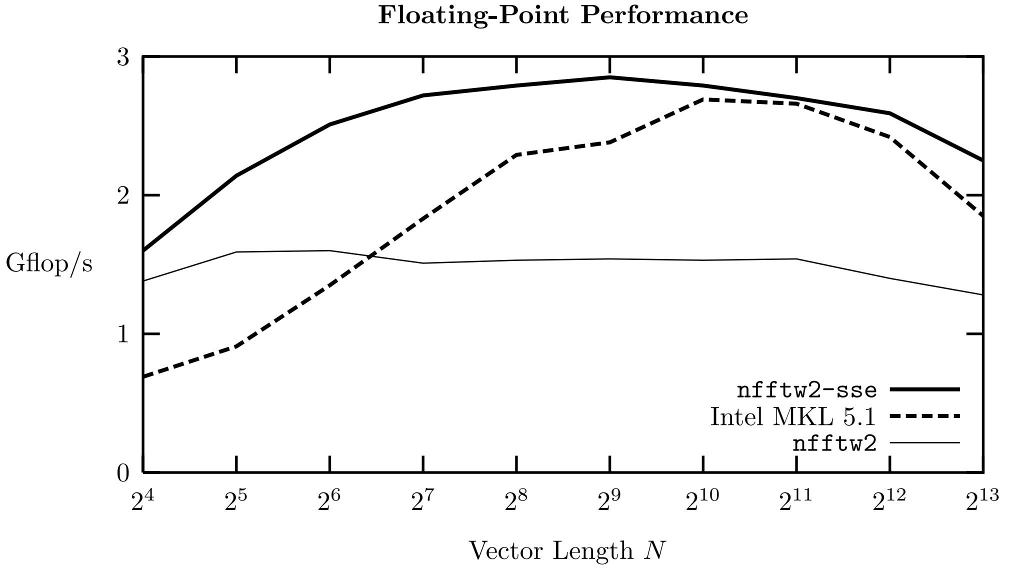


Figure 8.4: FFTW, Short Vector Cooley-Tukey rule set: Performance results for DFT_N with $N = 2^4, 2^5, \ldots, 2^{13}$, for single-precision and SSE on an AMD Athlon XP 1800+ running at 1533 MHz.

The Short Vector Cooley-Tukey Rule Set, Non-powers of Two

This section describes the experimental results which were obtained by investigating the FFTW implementation of the short vector Cooley-Tukey rule set developed in Section 7.3. Figures 8.5 to 8.8 show the vector recursion applied to non-power of two problem sizes tested across all IA-32 compatible machines within the test pool. Both SSE and SSE 2 were tested. Concerning four-way short vector SIMD extensions, these experiments again show that that the short vector Cooley-Tukey rule set is superior whenever applicable. The problem sizes tested are multiples of 16 and can be found in Table 8.2. The data format used for these experiments is—in contrast to most other experiments—the *split complex format* summarized in Section 4.6.

On the Pentium 4 the highest performance is achieved for problem sizes that fit completely into L1 cache and feature small prime factors. For problem sizes with large prime factors the performance gain breaks down significantly. As the problem sizes grow, the performance graph becomes smoother as large prime factors do not occur so often in the tested problem sizes.

16	32	48	64	80
96	112	128	144	
160	176	192	208	224
240	256	320	480	640
800	960	1,120	1,280	1,440
1,600	3,200	4,800	6,400	8,000

Table 8.2: The problem sizes tested with the short vector Cooley-Tukey rule set extension for FFTW in the non-power of two case.

For problem sizes that fit into L2 cache but do not fit into L1 data cache, the performance graph features about the same characteristics as for the power-of-two case. Speed-up factors of up to 3.2 for SSE and more than 1.7 for SSE 2 are achieved. See Figures 8.5 and 8.6 for details.

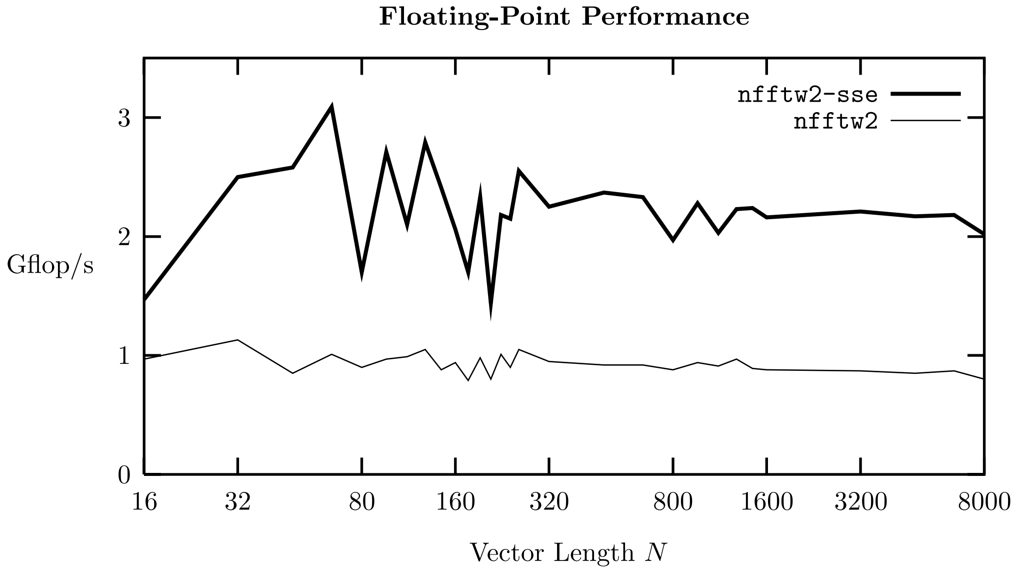


Figure 8.5: FFTW, Short Vector Cooley-Tukey rule set: Performance results for DFT_N with $N = 16, \ldots, 8000$ (see Table 8.2) for single-precision and SSE on an Intel Pentium 4 running at 1.8 GHz.

On the Pentium III again the shape of the performance graph looks different. The maximum performance is obtained between $N = 160$ and $N = 320$ and the performance slowly decreases for both smaller and larger problem sizes. But close to the highest performance two large breakdowns in performance gain can be seen due to large prime factors and relatively slow codelets. Again the differences in the cache architecture between the Pentium III and the Pentium 4 can be seen in the performance graph. Speed-up factors between 2.5 and three are achieved for SSE on the Pentium III. See Figure 8.7 for details.

Again the performance graph for the Athlon XP looks quite similar to the graph obtained for the Pentium III. The notable difference again is the lower speed-up factors compared to the Pentium III. As outlined above this is due to the AMD microarchitecture. Fftw-Simd achieves speed-ups of up to 1.8 on the Athlon XP. See Figure 8.8 for details.

The Cooley-Tukey Rule Set for Subvectors

This section describes the experimental results obtained by investigating the Fftw-Simd implementation based the Cooley-Tukey rules for subvectors given by Theorems 7.1 and 7.2. Figures 8.9 and 8.10 show the application of Theorem 7.1 for powers of two while Figures 8.11 and 8.12 show the application of Theorem 7.2 for non-powers of two. Both four-way extensions were analyzed: (*i*) SSE on the Pentium III, and (*ii*) AltiVec on the Motorola MPC7400 G4. The Pentium III features relatively cheap subvector memory access (64 bit quantities can be loaded into the 128 bit vector register) while this operation is very expensive on the MPC7400 G4. Comparing the performance shows similar speed-up factors.

For power of two problem sizes, speed-up factors of up to two are achieved on the Pentium III and speed-up factors of up to 2.5 on the MPC7400 G4. The performance achieved on the Pentium III is about 30 % less than by using the short-vector Cooley-Tukey rule set. On the MPC7400 G4 the cache influence is more severe than on the Pentium III. See Figures 8.9 and 8.10 for details.

Figures 8.11 and 8.12 focus on non-power of two sizes which are *not necessarily* multiples of 16. Thus the method based on Theorem 7.2 has to be used. On both the Pentium III and the MPC 7400 G4 the shape of the performance graph in the non-power of two case is similar to the power of two case. However, analogously to the non-power of two case for the short vector Cooley-Tukey rule set the performance graph is less smooth due to changing prime factors. The problem sizes tested are *not necessarily* multiples of 16. They can be found in Table 8.3.

10	20	30	40	50	60	70	90
100	200	300		500	600	700	900
1,000	2,000	3,000		5,000	6,000	7,000	9,000
10,000	20,000						

Table 8.3: The problem sizes tested with the short vector SIMD extension for Fftw based on Theorem 7.2 in the non-power of two case.

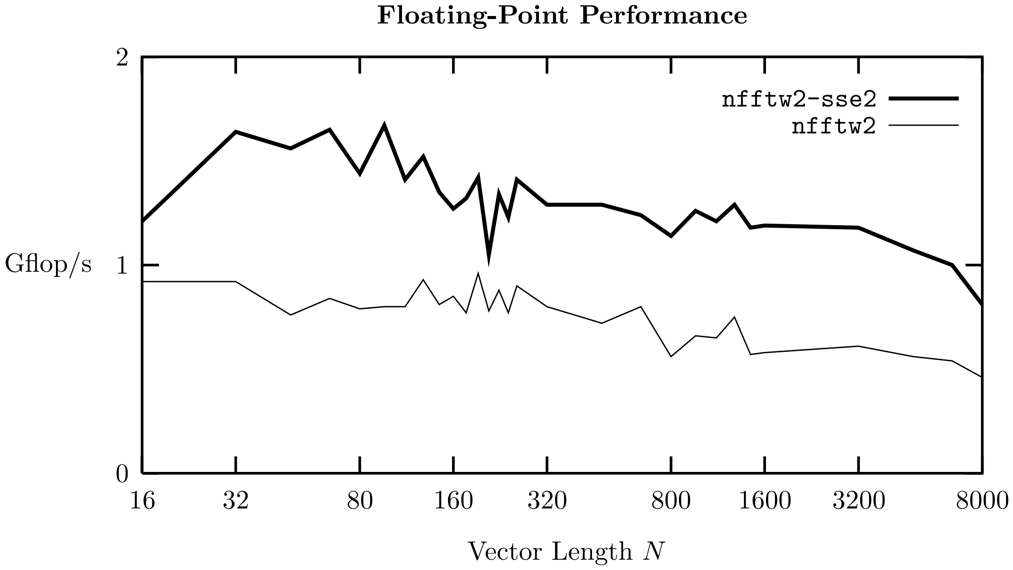


Figure 8.6: FFTW, Short Vector Cooley-Tukey rule set: Performance results for DFT_N with $N = 16, \ldots, 8000$ (see Table 8.2) for double-precision and SSE 2 on an Intel Pentium 4 running at 1.8 GHz.

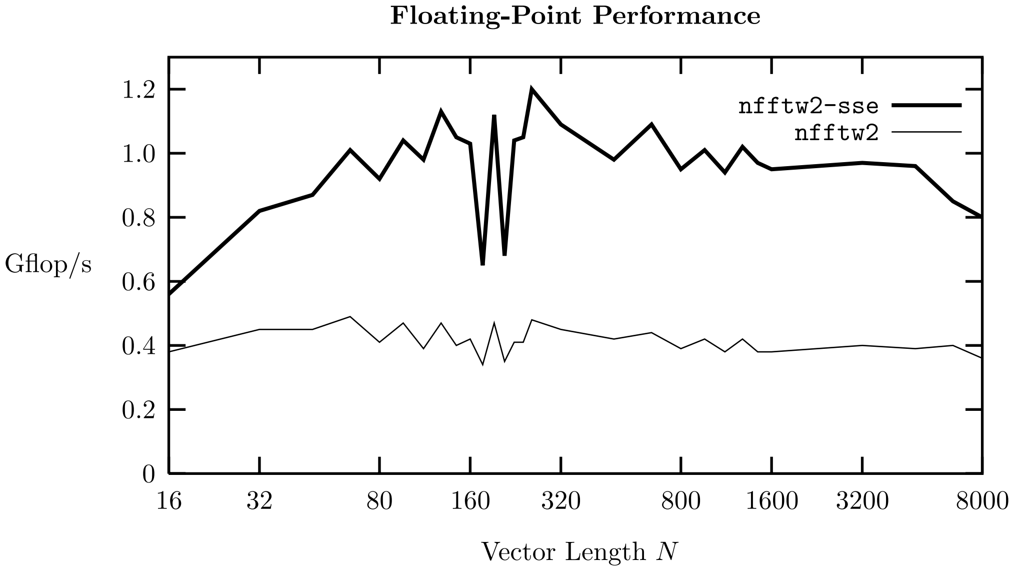


Figure 8.7: FFTW, Short Vector Cooley-Tukey rule set: Performance results for DFT_N with $N = 16, \ldots, 8000$ (see Table 8.2) for single-precision and SSE on an Intel Pentium III running at 650 MHz.

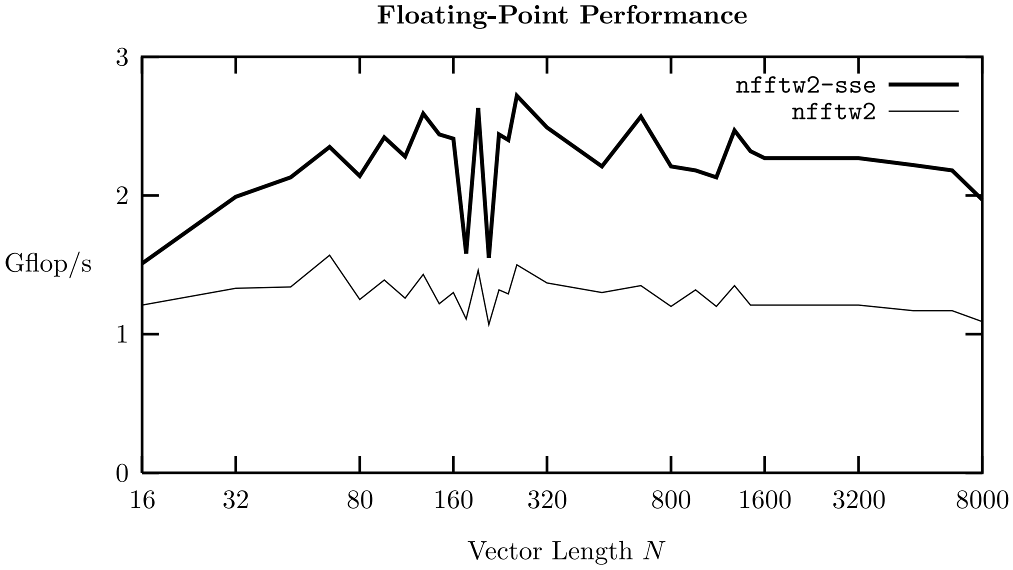


Figure 8.8: FFTW, Short Vector Cooley-Tukey rule set: Performance results for DFT_N with $N = 16, \ldots, 8000$ (see Table 8.2) for single-precision and SSE on an AMD Athlon XP 1800+ running at 1533 MHz.

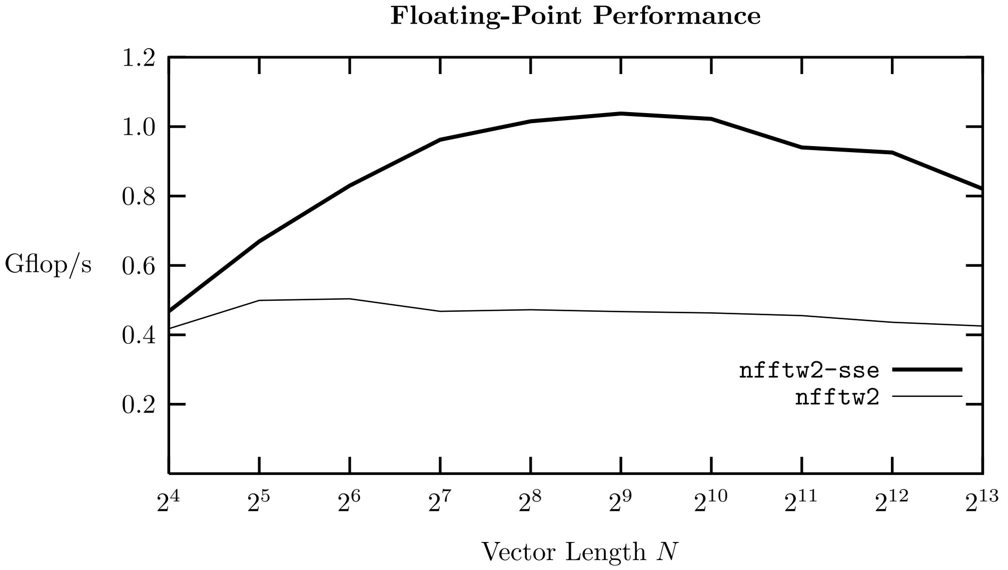


Figure 8.9: FFTW, Subvector Memory Access: Performance results for DFT_N with $N = 2^4, 2^5, \ldots, 2^{13}$, for single-precision and SSE on an Intel Pentium III running at 650 MHz.

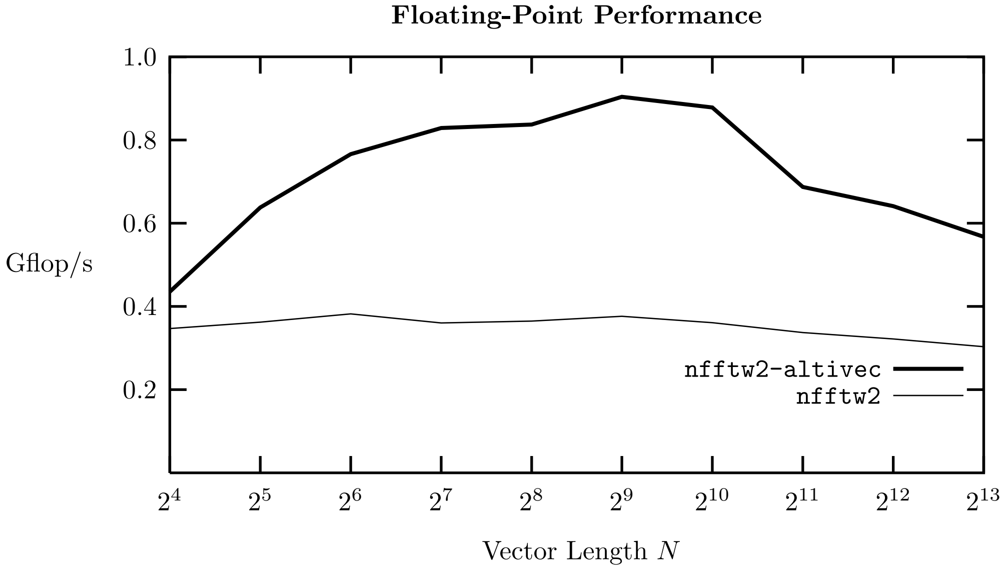


Figure 8.10: FFTW, Subvector Memory Access: Performance results for DFT_N with $N = 2^4, 2^5, \ldots, 2^{13}$, for single-precision and SSE on a Motorola MPC 7400 G4 running at 400 MHz.

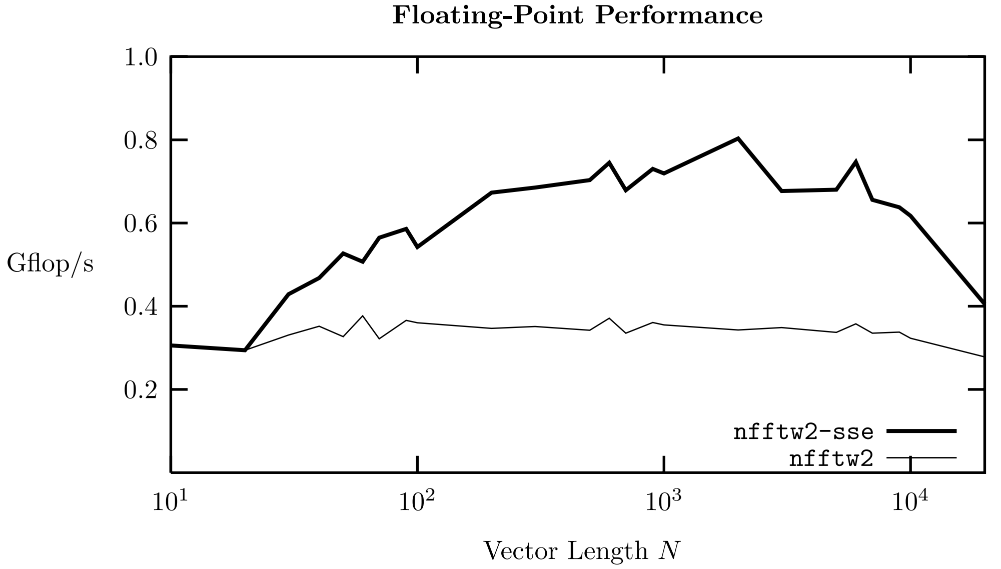


Figure 8.11: FFTW, Subvector Memory Access: Performance results for DFT_N with $N = 10, \ldots, 20\,000$ (see Table 8.3) for single-precision and SSE on an Intel Pentium III running at 650 MHz.

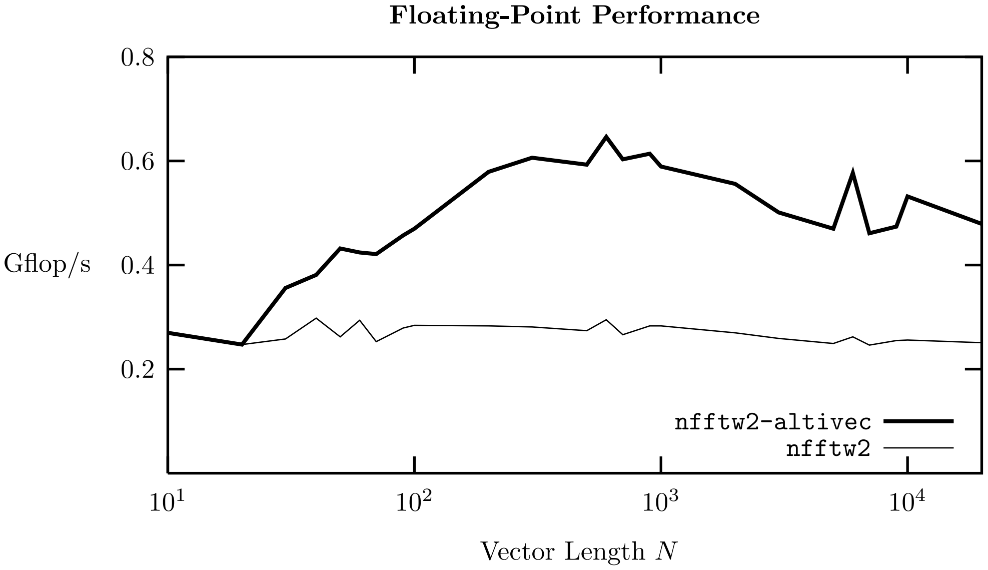


Figure 8.12: FFTW, Subvector Memory Access: Performance results for DFT_N with $N = 10, \ldots, 20\,000$ (see Table 8.3) for single-precision and SSE on a Motorola MPC 7400 G4 running at 400 MHz.

8.2 A Short Vector Extension for SPIRAL

In the scope of this thesis, Spiral has been extended to portably support short vector SIMD extensions based on two methods developed in Chapter 7 and the portable SIMD API. The SPL compiler is extended to support short vector SIMD, thus preserving the interfaces within the Spiral system. The newly developed extension of the SPL compiler formally vectorizes a given SPL program and generates a program that is based on the portable SIMD API.

- General support for discrete linear transforms utilizing the method developed in Section 7.1 is included.
- DFT specific support utilizing the short vector Cooley-Tukey FFT developed in Section 7.3 is provided. In addition, the search module is extended according to Section 7.4.

To enable analysis of compiler vectorization for discrete linear transforms, support for the Intel C++ compiler's vectorization was included into Spiral. The structure of the best found formulas are analyzed and related to relevant hardware parameters. It is shown that adapting the codes to a specific combination of target machine, short vector SIMD extension, and set of compiler options is required to obtain top performance.

8.2.1 Extending the SPL Compiler for Vector Code

This section presents the extended version of the SPL compiler that generates C code enhanced with macros provided by the portable SIMD API using an SPL formula and the SIMD vector length ν (i. e., number of single-precision floating-point numbers contained) as its sole input. The modified SPL compiler supports real and complex transforms and input vectors. Moreover, the compiler produces portable code, which is achieved by restricting the generated code to instructions available on all SIMD architectures and using the portable SIMD API as hardware abstraction layer.

The newly developed vectorization technique is based on the concepts developed in Chapter 7. For example, DFTs, WHTs arising from rules (5.2), and two-dimensional transforms arising from rule (5.4), can be completely vectorized. Implementing the method developed in Section 7.1, a general formula is normalized and vectorized code is generated for symbols matching equation (7.1), while the rest is translated to standard (scalar) C code. In addition, DFTs are handled more efficiently by the short vector Cooley-Tukey FFT provided by Theorem 7.4.

The symbolic vectorization is implemented as manipulation of the abstract syntax tree representation (see Section 4.7.1) of the SPL program. The newly included vectorization rules are based on the identities introduced in Chapters 4 and 7. Special support for complex transforms is provided according to Section 4.6. Table 8.4 shows some examples of the required rules.

Basic Vectorization Rules			Extended Vectorization Rules		
$A \otimes B$	$\mapsto$	$(A \otimes \mathrm{I}_m)(\mathrm{I}_n \otimes B)$	$A \otimes I_\nu$	$\mapsto$	S
$A \otimes \mathrm{I}_{\nu k}$	$\mapsto$	$(A \otimes \mathrm{I}_k) \otimes \mathrm{I}_\nu = A' \otimes I_\nu$	$D\,S$	$\mapsto$	S'
$\mathrm{I}_{\nu k} \otimes A$	$\mapsto$	$\mathrm{I}_\nu \otimes (\mathrm{I}_k \otimes A)$	$S\,D$	$\mapsto$	S'
	$=$	$P^{-1}\left((\mathrm{I}_k \otimes A) \otimes \mathrm{I}_\nu\right) P = P^{-1}(A' \otimes I_\nu)P$	$P\,S$	$\mapsto$	S'
$A \otimes \mathrm{I}_{\nu k+l}$	$\mapsto$	$A \otimes (\mathrm{I}_{\nu k} \oplus \mathrm{I}_l)$	$S\,P$	$\mapsto$	S'
	$=$	$Q^{-1}((\mathrm{I}_{\nu k} \otimes A) \oplus (\mathrm{I}_l \otimes A))Q$			
$\mathrm{I}_{\nu k+l} \otimes A$	$\mapsto$	$(\mathrm{I}_{\nu k} \oplus \mathrm{I}_l) \otimes A = (\mathrm{I}_{\nu k} \otimes A) \oplus (\mathrm{I}_l \otimes A)$			

Table 8.4: Recursion and transformation rules for SIMD vectorization. P and Q are permutation matrices. D is a diagonal matrix, and A and B are arbitrary formulas. S and S' are SIMD symbols.

In addition, the SPL compiler's unparsing stage had to be extended to support the portable SIMD API.

The remainder of this section explains the five stages of translating an SPL program into a C program that is built on top of the portable SIMD API.

Stage 1: Basic Vectorization

In the first stage the abstract syntax tree generated from the SPL program is searched for constructs of the form

$$\mathrm{I}_r \otimes A \quad \text{and} \quad A \otimes \mathrm{I}_s .$$

Then, using the basic vectorization rules from Table 8.4, these constructs are transformed into

$$Q_i \, (A_i' \otimes \mathrm{I}_\nu) \, P_i$$

with suitable permutation matrices P_i and Q_i. Next, in the main abstract syntax tree each of these constructs is replaced by a symbol S_i which thus becomes leaves in this tree. In addition, an abstract syntax tree is generated for each symbol. Within the abstract syntax tree of a symbol, no further steps are needed in this stage. The result of this stage is a collection of abstract syntax trees for the following constructs: (*i*) the remaining part of the formula (subsequently called *main* part), and (*ii*) for each symbol.

Stage 2: Joining Diagonals and Permutations

In this stage real and complex diagonals as well as permutations are vectorized. After this stage all constructs covered by equation (7.2) are transformed into a product of symbols. If any symbol S_i is composed with complex or real diagonals D_i or E_i or any permutation P_i or Q_i within the abstract syntax tree, the symbol S_i is transformed into the symbol S_i' by joining the permutations and/or diagonals using the extended vectorization rules summarized in Table 8.4. The diagonals and permutations will be handled in the load and store phase of the implementation of the corresponding symbol as outlined in Section 7.1.4. The joined permutations and diagonals are subsequently removed from the remaining main abstract syntax tree.

Stage 3: Generating Code

In this stage an internal representation of real arithmetic (i-code) for the main abstract syntax tree and the abstract syntax trees for all symbols is generated by calling the standard SPL compiler's code generator (see Section 4.7.1). For each symbol and the remaining main part of the formula i-code is generated and optimized using the respective stages of the standard SPL compiler.

Stage 4: Memory Access Optimization

In this stage the abstract syntax trees are transformed into the corresponding real arithmetic abstract syntax trees according to Section 4.6.1 and the required permutations $\mathrm{L}_k^{2\nu}$ are deduced. The i-code generated in Stage 3 is extended with load and store operations implementing these permutations which will finally be handled in the unparsing stage.

Stage 5: Unparsing

The i-code for symbols and for the main program generated in Stage 3 and extended in Stage 4 are unparsed as C program enhanced with macros provided by the portable SIMD API. The i-code generated for symbols is unparsed as vector arithmetic and the permutations produced by Stages 1, 2, and 4 are unparsed using memory access macros. The i-code generated for the main program is unparsed as scalar C program and calls to the symbols are inserted.

The short vector Cooley-Tukey rules developed in Section 7.3 have been implemented currently in Stages 1, 2, and 4, as their main purpose is to handle complex arithmetic and generate suitable symbols for parts of the DFT computation. The long-term goal is to move the short vector Cooley-Tukey rules into Spiral's formula generator and enable rule (7.26).

8.2.2 Run Time Experiments

This section describes experimental results for the automatically generated short vector SIMD code for the DFT, WHT and the two-dimensional DCT of type II. All transforms are of size $N = 2^n$ or $N \times N = 2^n \times 2^n$, respectively. The code generated and optimized by Spiral-Simd is compared to the best available DFT implementations across different architectures. In addition, different search methods and the structure of the best algorithms found are analyzed.

To validate the approach introduced in this thesis, both the SSE and SSE 2 extensions on three binary compatible, yet architectural different platforms were chosen: (*i*) Intel Pentium III with SDRAM running at 1 GHz, (*ii*) Intel Pentium 4 with RDRAM running at 2.53 GHz, and (*iii*) AMD Athlon XP 2100+ with DDR-SDRAM running at 1733 MHz.

These processors are based on different cores and have different cache architectures. The machines feature different chip sets, system busses, and memory technology. Details about these machine can be found in Table 8.5.

As outlined in Section 8.1.2, the theoretical speed-up limit achievable due to vectorization (thus ignoring effects like smaller program size due to a smaller number of vector instructions) is a *factor of four* for SSE on the Pentium III and the Pentium 4. For SSE on the Athlon XP and SSE 2 on the Pentium 4 the limit is a *factor of two*.

By finding differently structured algorithms on different machine, high performance across architectures and across short vector SIMD extensions is achieved, which demonstrates the success of the presented approach to provide portable performance independent of the vector length ν and other architectural details.

CPU	**Intel Pentium 4** **2.53 GHz** 8 kB L1 Data Cache 256 kB on-die L2 Cache 10.1 Gflop/s single-precision 5.06 Gflop/s double-precision SSE and SSE 2
RAM	512 MB RDRAM
Operating System	Microsoft WINDOWS 2000
Compiler	Intel C++ Compiler 6.0
CPU	**Intel Pentium III Coppermine** **1 GHz** 16 kB L1 Data Cache 256 kB on-die L2 Cache 7.2 Gflop/s single-precision SSE
RAM	128 MB SDRAM
Operating System	Microsoft WINDOWS 2000
Compiler	Intel C++ Compiler 6.0
CPU	**AMD Athlon XP 2100+** **1733 MHz** 64 kB L1 Data Cache 256 kB on-die L2 Cache 6.9 Gflop/s single-precision 3DNow! Professional (SSE compatible)
RAM	512 MB DDR-SDRAM
Operating System	Microsoft WINDOWS 2000
Compiler	Intel C++ Compiler 6.0

Table 8.5: The computer systems used for benchmarking the short vector SIMD extension for SPIRAL.

Note that using automatic *compiler* vectorization in tandem with SPIRAL code generation provides a fair evaluation of the limits of this technique. By running a dynamic programming search, SPIRAL can find algorithms that are best structured for compiler vectorization. Furthermore, the code generated by SPIRAL is of simple structure (e. g., contains no pointers or variable loop limits). Even though the compiler can improve on SPIRAL's scalar code, the performance is far from being optimal.

In case of the DFT, the vector code generated using the new approach developed in this thesis is compared to the state-of-the-art C code by FFTW 2.1.3 (Frigo and Johnson [33]) and code generated by SPIRAL, compiler vectorized C code generated by SPIRAL, and short vector SIMD code (SSE and SSE 2) provided by the Intel Math Kernel Library MKL 5.1 [58]. The MKL features separate versions optimized for Pentium III and 4. Note that the MKL uses in-place computation and memory prefetching instructions, which gives it an advantage over the code generated by Spiral-Simd, which computes the DFT out-of-place.

In all cases the Intel C++ 6.0 compiler was used. In case of DFTs, the performance is displayed in pseudo Gflop/s, i. e., $(5N \log N)/T$, which is a scaled inverse of the run time T and thus preserves run time relations and additionally gives an indication of the absolute performance. Details on the performance unit *pseudo flop/s* can be found in Appendix A and in Frigo and Johnson [33]. For the other transforms, the actual number of floating-point operations was counted and used for the computation of real Gflop/s performance values. These short vector SIMD implementations were compared to the scalar implementations generated and optimized by standard Spiral because of the lack of standard libraries.

Spiral generated scalar code was found using a dynamic programming search, Spiral generated vector code (using the newly developed SIMD extensions) using the best result of the two vector dynamic programming variants in Section 7.4. In both cases the global limit for unrolling (the size of subblocks to be unrolled) was included into the search.

Pentium 4

DFTs. On this processor the best ratio of flops per cycle among all investigated processors and the highest speed-ups of Spiral generated vector code compared to scalar Spiral generated code was achieved. For DFTs, using SSE, up to 6.25 pseudo Gflop/s (and a speed-up of up to 3.1), and using SSE 2 up to 3.3 pseudo Gflop/s (and a speed-up of up to 1.83) were measured on a 2.53 GHz machine as can be seen in Figures 8.13 and 8.14. Exhaustive search can further increase the performance when using SSE 2 for small problem sizes.

The performance of the code generated by Spiral-Simd is best within L1 cache and only slightly decreases outside L1. Analysis of the generated program shows that the best found code features very small loop bodies (as opposed to medium and large unrolled blocks that typically lead to high performance) and very regular code structure. This is due to Pentium 4's new features, namely (*i*) its new core with a very long pipeline, (*ii*) its new instruction cache that caches instructions *after* they are decoded (trace cache), and (*iii*) its small, but very fast data caches.

The results reported by Rodriguez [82] obtained on a similar machine running at 1.4 GHz are included. As the source code cannot be obtained, the reported results are scaled up to the frequency of the test machine used in this thesis. These performance numbers are only a very rough but instructive estimate. Only half of the performance obtained using the approach presented in this thesis is achieved by Rodriguez' method.

The standard C and Fortran codes generated by scalar Spiral and Fftw 2.1.3 show about the same performance and serve as a base line. Using compiler vectorization, the code is sped up but the result is much slower than the code obtained using Spiral-Simd for SSE. For SSE 2 compiler vectorization does not significantly speed up the code.

For SSE, apart from some problem sizes around 2^{11}, Intel's MKL is much slower than the code obtained using Spiral-Simd. For SSE 2 Intel's MKL is significantly slower across all evaluated problem sizes.

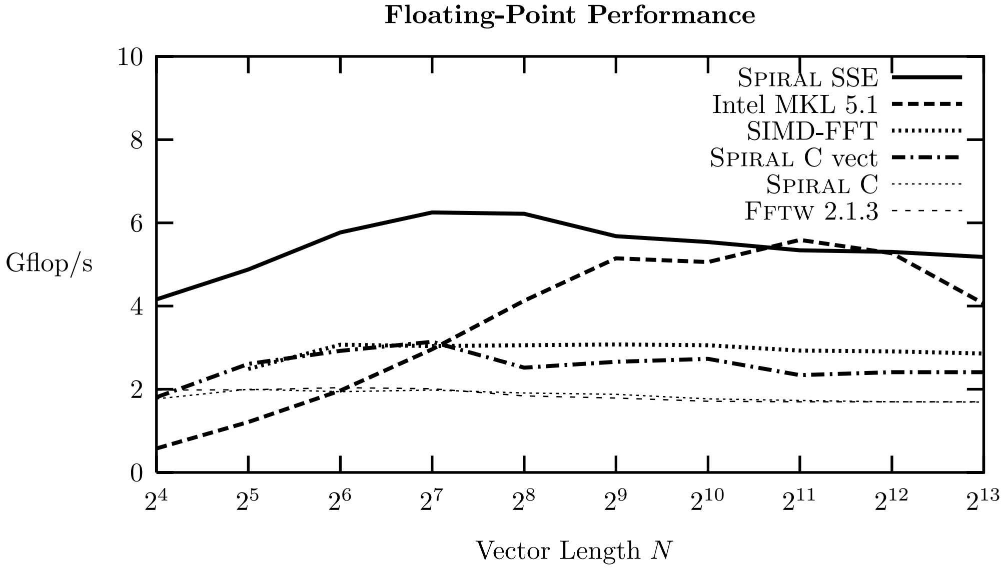


Figure 8.13: SPIRAL: Performance results for DFT_N with $N = 2^4, 2^5, \ldots, 2^{13}$, for single-precision arithmetic and SSE on an Intel Pentium 4 running at 2.53 GHz.

WHTs. Speed-up factors of more than 2.6 are achieved for WHTs when using Spiral-Simd with SSE. The WHT is a very simple transform and thus compiler vectorization can speed up the computation significantly. However, it cannot reach the performance level of the code generated using Spiral-Simd. See Figure 8.15 for details.

For SSE 2, speed-up factors of up to 1.5 are achieved. In this experiment, compiler vectorization is able to reach the same performance level as code generated by Spiral-Simd for larger problem sizes. See Figure 8.16 for details.

2D-DCTs, Type II. Speed-up factors of more than 3.1 are achieved for the 2D-DCT type II using Spiral-Simd with SSE. 2D-DCTs of type II feature a very simple macro structure as they are higher-dimensional transforms and thus compiler vectorization should be able to speed up the computation significantly. However, the predicted speed-up has not been observed in the SSE experiments. See Figure 8.17 for details.

For SSE 2, speed-up factors of up to 1.88 are achieved. In this experiment, compiler vectorization can speed up the code across all problem sizes, yet the performance level of code generated using Spiral-Simd is not achieved. See Figure 8.18 for details.

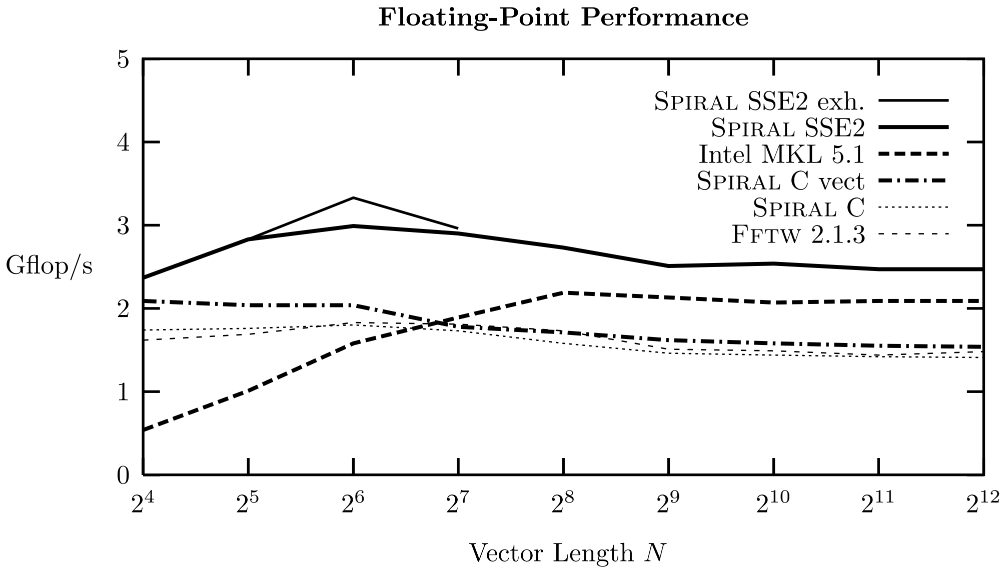


Figure 8.14: SPIRAL: Performance results for DFT_N with $N = 2^4, 2^5, \ldots, 2^{12}$, for double-precision arithmetic and SSE 2 on an Intel Pentium 4 running at 2.53 GHz.

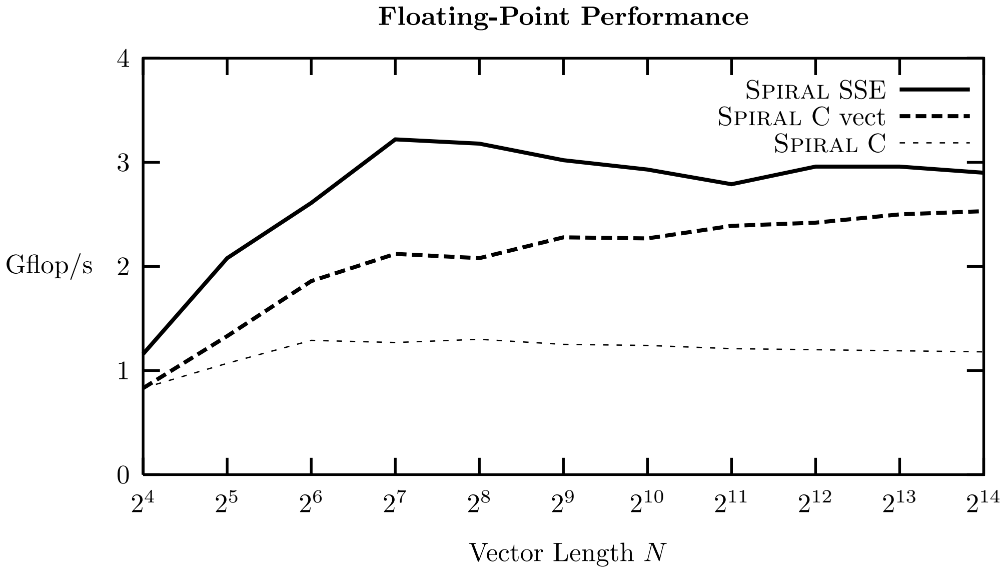


Figure 8.15: SPIRAL: Performance results for WHT_N with $N = 2^4, 2^5, \ldots, 2^{14}$, for single-precision arithmetic and SSE on an Intel Pentium 4 running at 2.53 GHz.

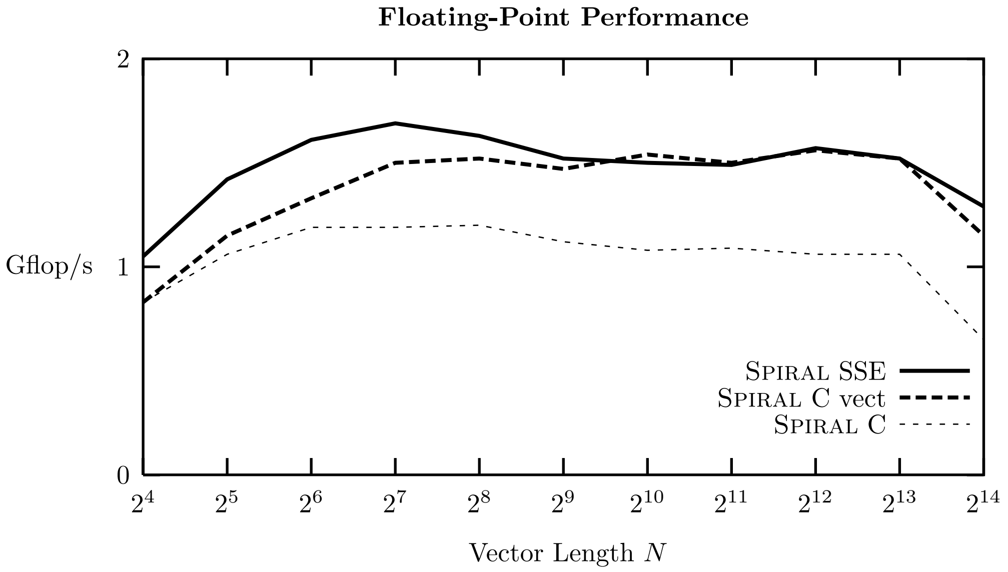


Figure 8.16: SPIRAL: Performance results for WHT_N with $N = 2^4, 2^5, \ldots, 2^{14}$, for double-precision arithmetic and SSE 2 on an Intel Pentium 4 running at 2.53 GHz.

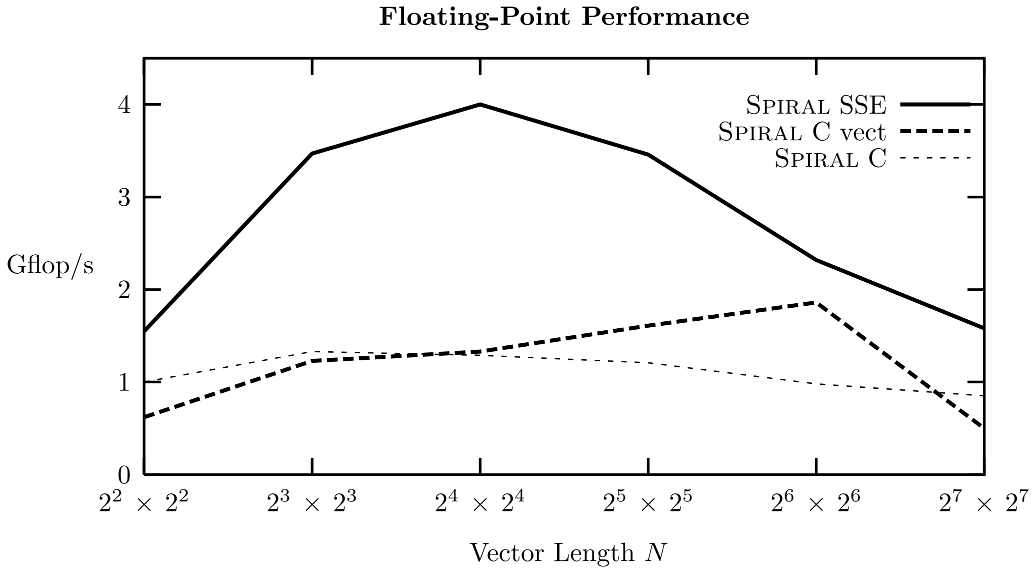


Figure 8.17: SPIRAL: Performance results for 2D $\mathrm{DCT}^{\mathrm{II}}_{N \times N}$ with $N = 2^2, 2^3, \ldots, 2^7$, for single-precision arithmetic and SSE on an Intel Pentium 4 running at 2.53 GHz.

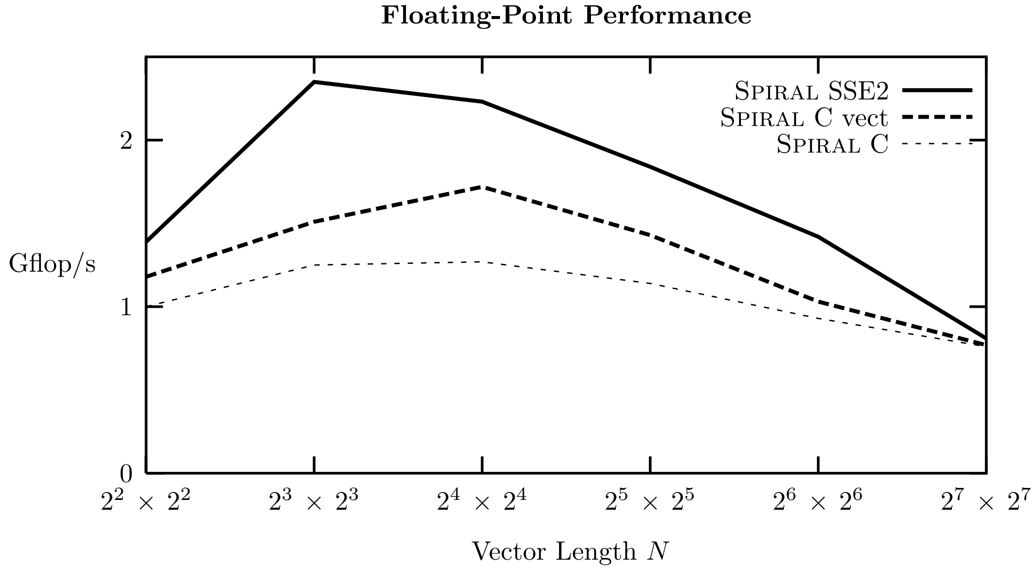


Figure 8.18: SPIRAL: Performance results for 2D $\mathrm{DCT}^{\mathrm{II}}_{N \times N}$ with $N = 2^2, 2^3, \ldots, 2^7$, for double-precision arithmetic and SSE 2 on an Intel Pentium 4 running at 2.53 GHz.

Pentium III

DFTs. Up to 1.7 pseudo Gflop/s (and a speed-up of up to 3.1) on a 1 GHz machine featuring a Coppermine core were achieved. The best DFT implementations obtained by Spiral-Simd featured moderately sized loop bodies. On this machine, codes generated by Spiral-Simd deliver the highest speed-ups with respect to the scalar codes for larger problem sizes. On the Pentium III, Intel's MKL shows lower performance compared to codes generated by Spiral-Simd as when comparing the respective run times on the the Pentium 4. This reflects Intel's additional tuning effort for the Pentium 4 version of the MKL. Again Fftw 2.1.3 (without support for SSE) and scalar code generated by Spiral perform at about the same performance level and serve as a baseline. Compiler vectorization can speed up the code significantly, but again does not reach the performance level obtained with code generated by Spiral-Simd. See Figure 8.19 for details.

WHTs. Speed-up factors of more than 2.6 are achieved when computing WHTs using Spiral-Simd with SSE. Compiler vectorization can speed up the computation significantly, but again does not reach the same performance level as code generated using Spiral-Simd. See Figure 8.20 for details.

2D-DCTs, Type II. Speed-up factors of more than 3.1 are achieved for the 2D-DCT of type II using Spiral-Simd with SSE. For smaller problem sizes compiler vectorization is able to speed up the computation significantly and finally reaches the same performance level as code generated using Spiral-Simd. However, for large problem sizes the performance breaks down to half of the scalar performance. See Figure 8.21 for details.

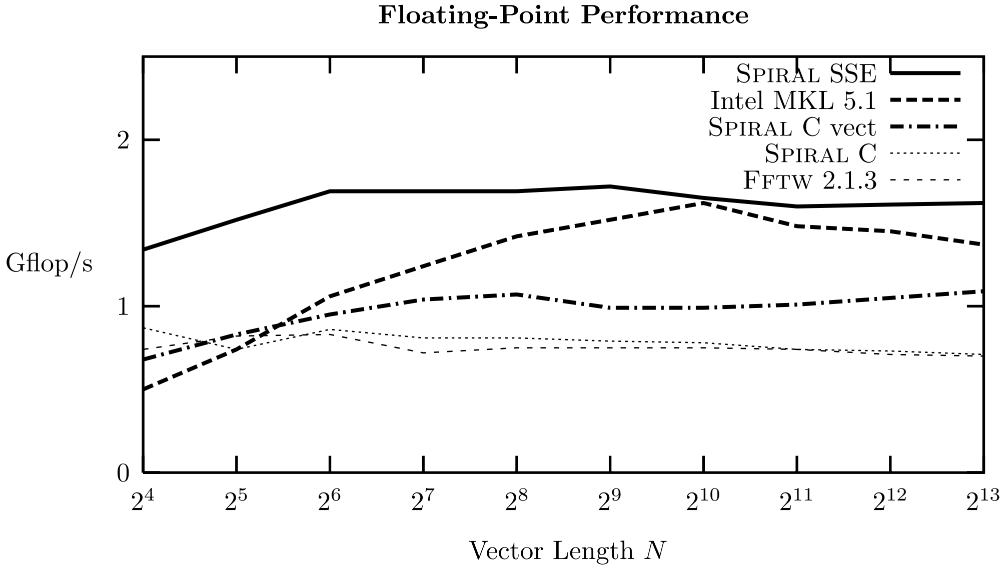


Figure 8.19: SPIRAL: Performance results for DFT_N with $N = 2^4, 2^5, \ldots, 2^{13}$, for single-precision arithmetic and SSE on an Intel Pentium III running at 1 GHz.

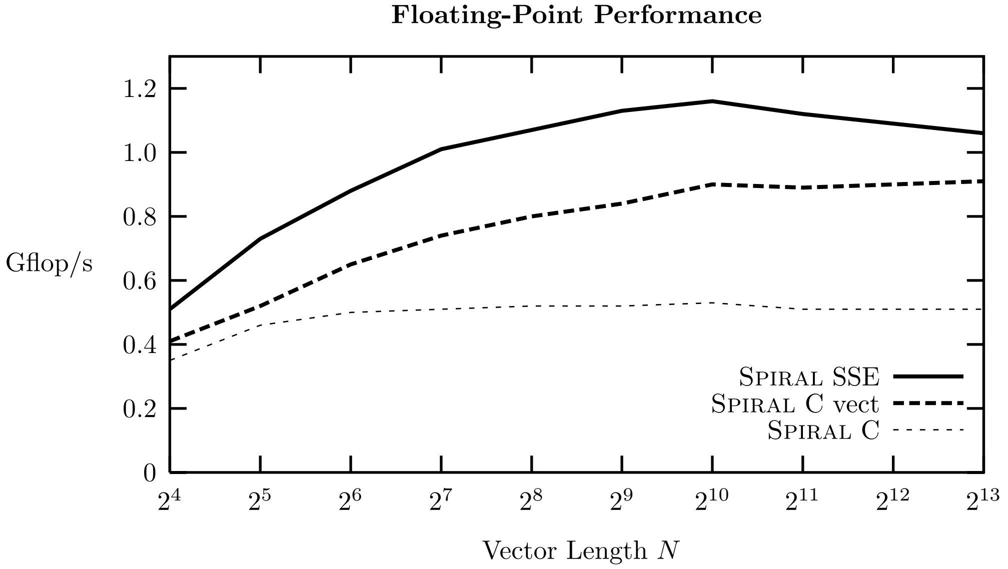


Figure 8.20: SPIRAL: Performance results for WHT_N with $N = 2^4, 2^5, \ldots, 2^{13}$, for single-precision arithmetic and SSE on an Intel Pentium III running at 1 GHz.

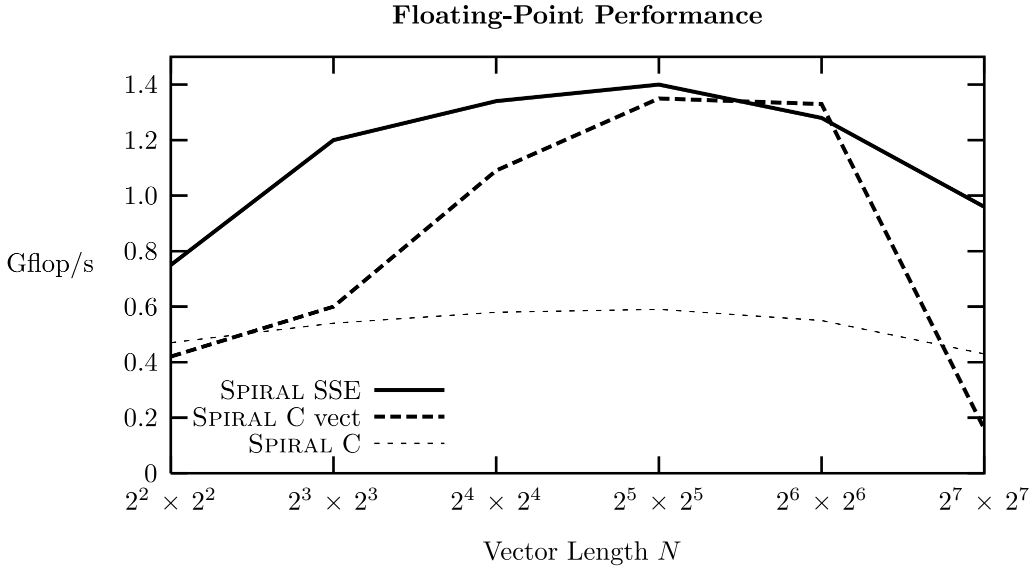


Figure 8.21: SPIRAL: Performance results for 2D $\mathrm{DCT}^{\mathrm{II}}_{N \times N}$ with $N = 2^2, 2^3, \ldots, 2^7$, for single-precision arithmetic and SSE on an Intel Pentium III running at 1 GHz.

Athlon XP

DFTs. Up to 2.8 pseudo Gflop/s (and speed-ups of up to 1.7) on a 1733 MHz machine were achieved. The best DFT implementations obtained by Spiral-Simd featured large loop bodies. On this machine, the performance of the code generated by Spiral-Simd decreases at the L1 boundary while Intel's MKL keeps the performance level. Careful analysis shows, however, that the performance level of codes generated by Spiral-Simd for 2^{n-1} is the same as Intel's MKL achieves for 2^n. This is partly due to the in-place computation featured by Intel's MKL resulting in substantially lower memory requirements. Although the 3DNow! professional extension which is binary compatible to 3DNow! and SSE features 4-way SIMD extensions, the maximum obtainable speed-up is a factor of two, as the Athlon XP's two floating-point units then both operate as two-way SIMD units. See Figure 8.22 for details.

WHTs. Speed-up factors of more than 1.65 are achieved for WHTs using Spiral-Simd with SSE. Compiler vectorization achieves the same performance level as code generated by Spiral-Simd. See Figure 8.23 for details.

2D-DCTs, Type II. Speed-up factors of more than 1.7 are achieved for 2D-DCTs of type II using Spiral-Simd with SSE. For small problem sizes, compiler vectorization is even inferior to scalar code. For medium problem sizes, compiler vectorization can speed up the computation significantly but is not able to reach the performance level of code generated by Spiral-Simd. For large problems, compiler vectorization produces *incorrect* programs! See Figure 8.24 for details.

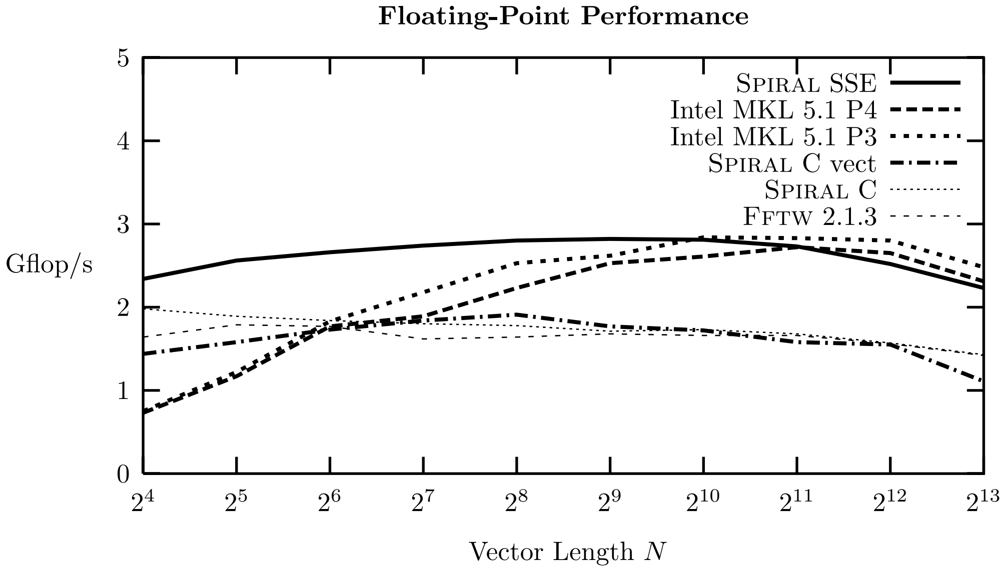


Figure 8.22: SPIRAL: Performance results for DFT_N with $N = 2^4, 2^5, \ldots, 2^{13}$, for single-precision arithmetic and SSE on an AMD Athlon XP 2100+ running at 1733 MHz.

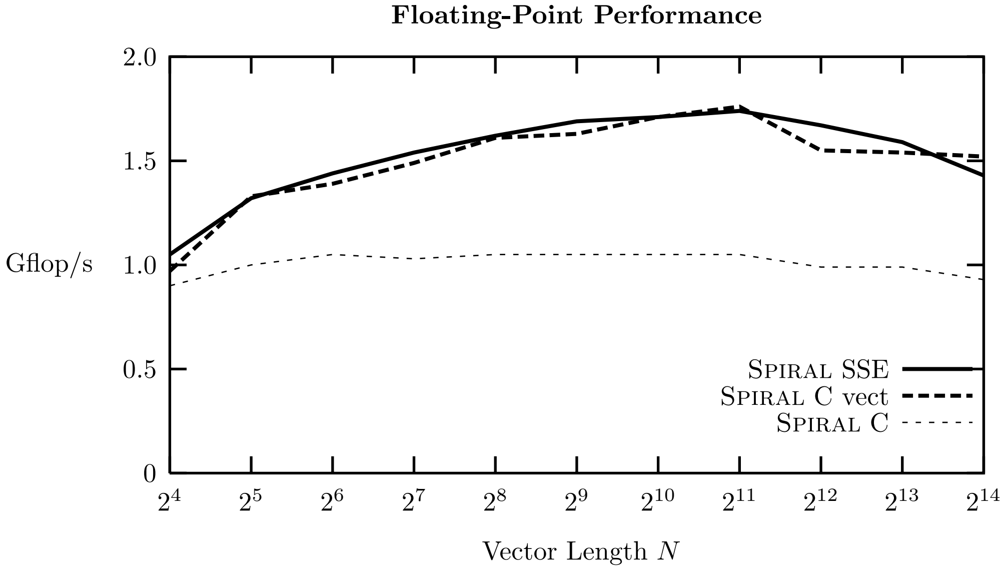


Figure 8.23: SPIRAL: Performance results for WHT_N with $N = 2^4, 2^5, \ldots, 2^{14}$, for single-precision arithmetic and SSE on an AMD Athlon XP 2100+ running at 1733 MHz.

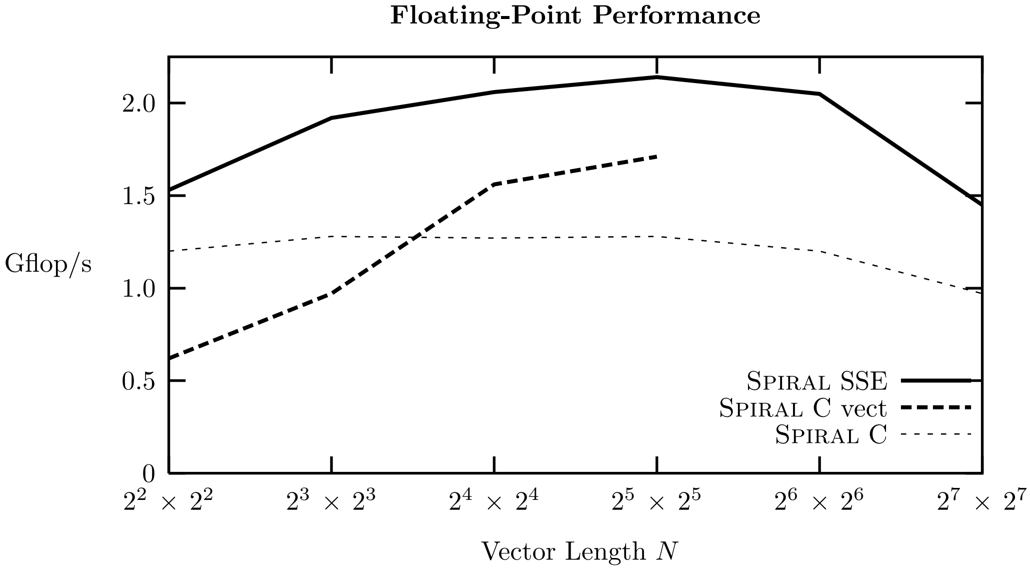


Figure 8.24: SPIRAL: Performance results for 2D $\mathrm{DCT}^{\mathrm{II}}_{N \times N}$ with $N = 2^2, 2^3 \ldots, 2^7$, for single-precision arithmetic and SSE on an AMD Athlon XP 2100+ running at 1733 MHz.

8.2.3 Assessment of Search Methods

The three variants of dynamic programming introduced in Section 7.4 were analyzed: (*i*) Standard dynamic programming, (*ii*) vector dynamic programming, and (*iii*) stride sensitive vector dynamic programming. The major observations made in the experiments were that dynamic programming works on some machines while on others it misses the best implementation considerably, thus requiring the modified versions introduced in Section 7.4. Specifically, on the Pentium III and the Athlon XP, standard dynamic programming finds implementations very close to the optimum. However, on the Pentium 4 the vector-aware dynamic programming variants are required to get the best performance, as standard dynamic programming only reaches 75 % of the best result.

Figure 8.25 shows that on the Pentium 4 using SSE for $N \leq 2^7$, exhaustive search leads to the best result. But for $N > 2^7$ vector dynamic programming is very close to the best result which is obtained by stride sensitive vector dynamic programming. Thus, the additional search time required by stride sensitive vector dynamic programming does not pay off. A combination of exhaustive search (where possible) and vector dynamic programming (for larger sizes) is the most economical search method for short vector SIMD codes.

Figure 8.26 shows for the Pentium 4 using SSE 2 that dynamic programming finds implementations more than 18 % slower than implementations found using vector-aware dynamic programming versions.

On the Pentium III, dynamic programming finds implementations that are not more than 7 % slower than implementations found with the vector-aware dynamic programming versions. See Figure 8.27 for details.

On the Athlon XP, implementations found with dynamic programming are not more than 6 % slower than the best implementations for $N < 2^{13}$. For $N = 2^{13}$ dynamic programming finds implementations that are more than 10 % slower than the best implementations. See Figure 8.28 for details.

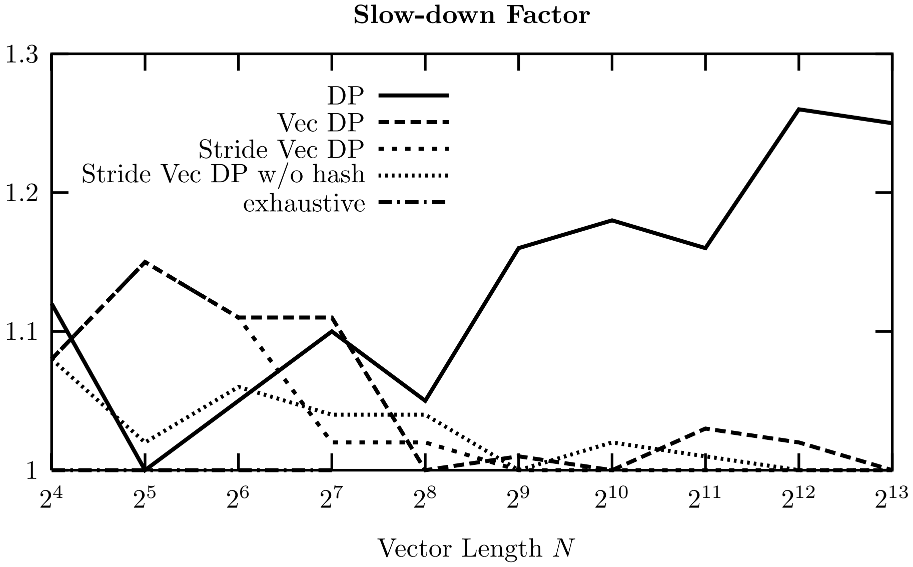


Figure 8.25: SPIRAL: Comparison of the best algorithms found for DFT_N with $N = 2^4, 2^5, \ldots, 2^{13}$ and SSE (single-precision), by different search methods on an Intel Pentium 4 running at 2.53 GHz. The slowdown factor gives the performance relation to the best algorithms found.

8.2.4 Crosstiming

The experiments in this section show how much the best formula for a machine depends on the target architecture and the SIMD extension. To make this explicit a suite of experiments were conducted.

Pentium 4, SSE. Figure 8.29 shows the slow-down of the best found DFT formulas for SSE on the Pentium III and the Athlon XP and the best formulas for scalar and SSE 2 code found on the Pentium 4, all implemented using SSE vector code and run on the Pentium 4 (using the best compiler optimization).

As expected, the code generated from the formula found on the Pentium 4 using SSE performs best and is the baseline. Both the the code generated from the best formula found on the Pentium 4 using the FPU and using SSE 2 perform very badly. But interestingly, the code generated from the formulas found on the Pentium 3 and the Athlon XP using SSE are up to 60 % slower than the respective Pentium 4 SSE version. This reflects the need of searching on the actual target machine. It shows that it is not sufficient, to conduct the search on a binary compatible machine featuring the same vector extension.

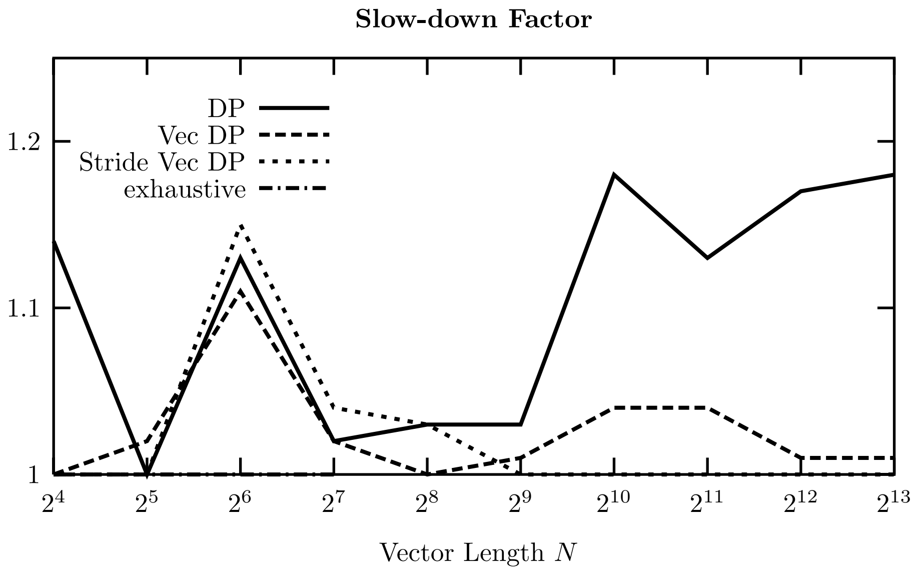


Figure 8.26: SPIRAL: Comparison of the best algorithms found for DFT_N with $N = 2^4, 2^5, \ldots, 2^{13}$ and SSE 2 (double-precision), by different search methods on an Intel Pentium 4 running at 2.53 GHz. The slowdown factor gives the performance relation to the best algorithms found.

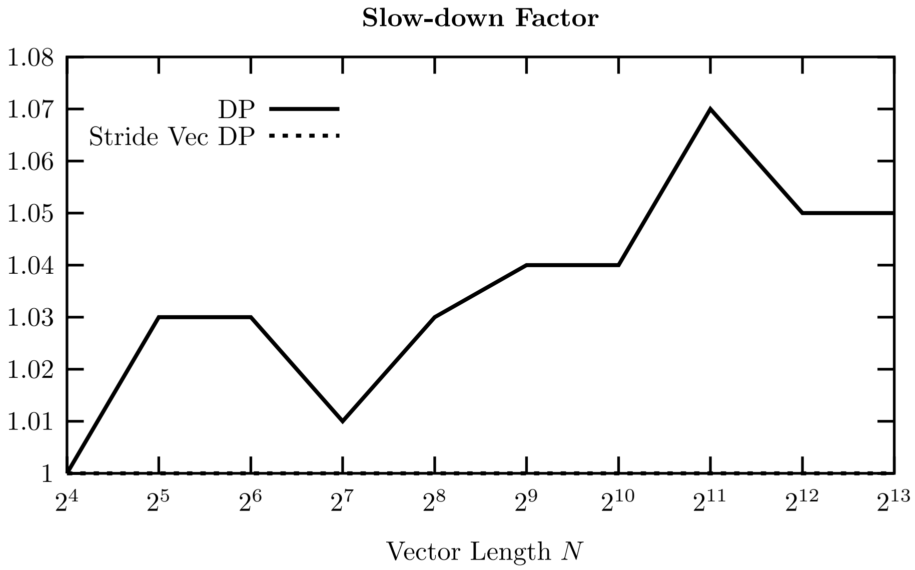


Figure 8.27: SPIRAL: Comparison of the best algorithms found for DFT_N with $N = 2^4, 2^5, \ldots, 2^{13}$ and SSE (single-precision), by different search methods on an Intel Pentium 3 running at 1 GHz. The slowdown factor gives the performance relation to the best algorithms found.

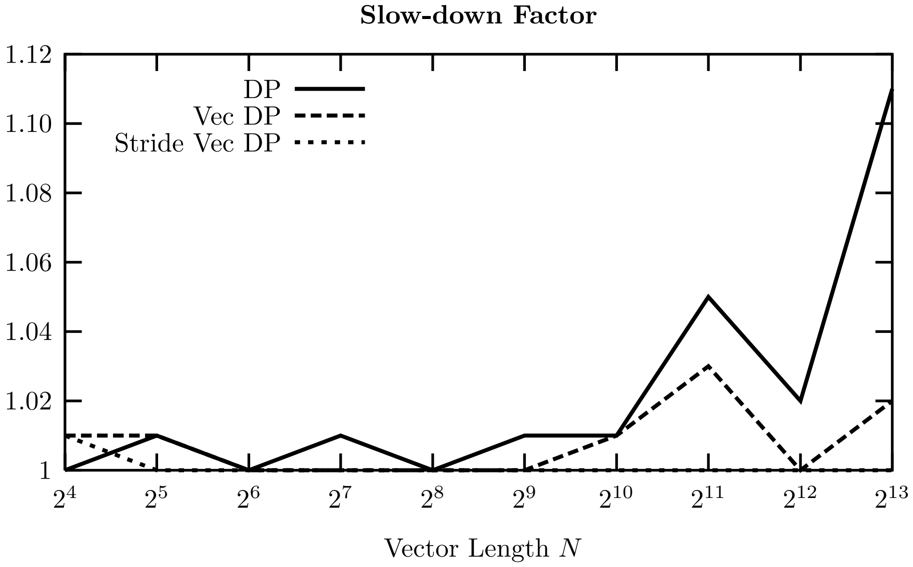


Figure 8.28: SPIRAL: Comparison of the best algorithms found for DFT_N with $N = 2^4, 2^5, \ldots, 2^{13}$ and SSE (single-precision), by different search methods on an AMD Athlon XP 2100+ running at 1733 MHz. The slowdown factor gives the performance relation to the best algorithms found.

Pentium 4, SSE 2. Figure 8.30 shows the respective experiment for the Pentium 4 using double precision and SSE 2. It shows the slow-down of the code generated from the best DFT formula found on the Pentium III, the Athlon XP and the Pentium 4 using SSE and the respective slow-down for the best formulas found on the Pentium 4 using the FPU and SSE, all implemented using SSE 2 vector code and run on the Pentium 4 (using the best compiler optimization).

As expected, the Pentium 4 SSE 2 version performs best and is the baseline. All other formulas lead to codes that run at least 50 % slower and the worst implementation is obtained from the formula found with search for the best scalar implementation which is then implemented using SSE 2. It is up to a factor of 1.8 slower than the formula found for SSE 2.

Pentium III, SSE. Figure 8.31 shows the crosstiming experiment for the Pentium III using single precision arithmetic and SSE. It shows the slow-down of the codes generated from the the best found DFT formulas for SSE and SSE 2 on the Pentium 4, SSE on the Athlon XP and for the best found formulas for scalar code found on the Pentium III, all implemented using SSE vector code and run on the Pentium III (using the best compiler optimization). As one can see, the codes generated from the best found formulas on the Athlon XP perform very good on the Pentium III while the codes generated from the best found formulas for SSE on the Pentium 4 are 30 % slower. As expected, both the scalar Pentium III and the SSE 2 Pentium 4 version are significantly slower than the Pentium III SSE version. However, for intermediate problem sizes, the best found formulas on the Pentium 4 using SSE are *slower* than the formulas found for scalar implementations on the Pentium III. This coincides with the fact that the best found formula for a given problem size is rather different for the Pentium III and Pentium 4 even for SSE.

Athlon XP, SSE. Figure 8.32 shows the crosstiming experiment for the Athlon XP using single precision arithmetic and SSE. It shows the slow-down of the codes generated from the best found DFT formulas for SSE and SSE 2 on the Pentium 4, SSE on the Pentium III and from the best formulas found on the Athlon XP for scalar code, all implemented using SSE vector code and run on the Athlon XP (using the best compiler optimization).

Interestingly, the codes generated from the best found formulas for SSE on the Pentium III and Pentium 4 run within 5 % of the codes generated from the best found formulas for the Athlon XP. Even the best found formulas for SSE 2 on the Pentium 4 runs within 20 %. Thus, the Athlon XP can handle codes optimized for the Pentium III very well and codes optimized for the Pentium 4 well. Exchanging codes between Pentium III and Pentium 4 results in much higher slow-downs relative to the native best-found formula as comparing these formulas on the Athlon XP with the respective native best found formulas. Thus, the Athlon XP is able to handle codes optimized for other processors very well. But in general, these experiments show that for optimal performance the codes have to be adapted to the actual target machine.

8.2.5 The Best Algorithms Found

The approach presented in this thesis delivers high performance on all tested systems. The structure of the best implementations, however, depends heavily on the target machine. Figure 8.6 shows the structure of the best found formulas, displayed as trees representing the breakdown strategy.

Generally speaking, two completely different types of behavior were observed: (*i*) formulas with rather balanced trees, and (*ii*) formulas with unbalanced trees tending to be right-expanded. The first type occurs when the working set fits into L1 cache and for codes generated using compiler vectorization. For the second type, the structure depends on whether scalar or vector code is generated. For all rule trees, parameters and structural details on deeper levels depend on the actual target machine.

Scalar Code

Right-expanded trees with machine-dependent sizes of the left leaves are found to be the most efficient formulas when using only the scalar FPU. These trees provide the best data locality. Due to the larger caches of the Pentium III and Athlon XP, the problem size fits into L1 cache and balanced trees are found.

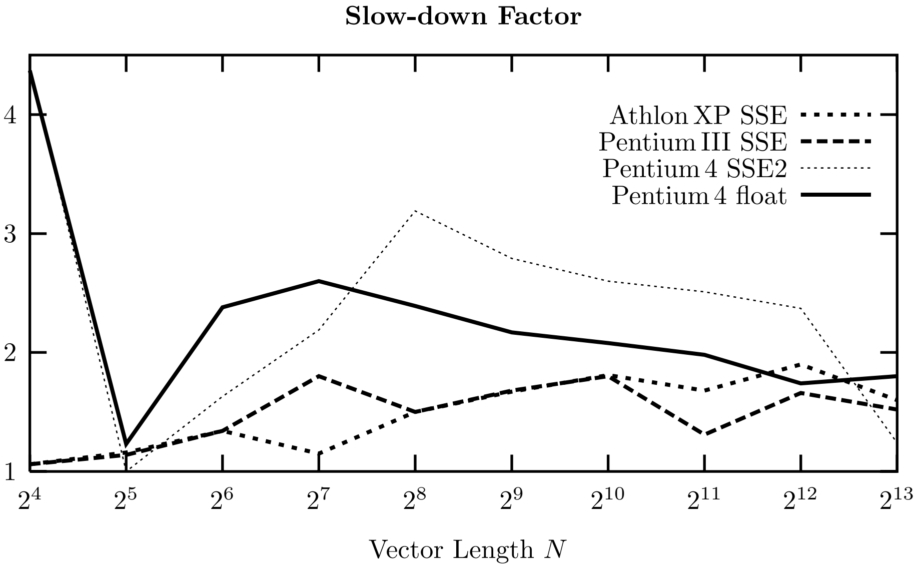


Figure 8.29: SPIRAL: Crosstiming of the best algorithms for DFT_N with $N = 2^4, 2^5, \dots, 2^{13}$, found for different architectures, all measured on the Pentium 4, implemented using SSE. The slowdown factor gives the performance relation to the best algorithms found for Pentium 4, SSE.

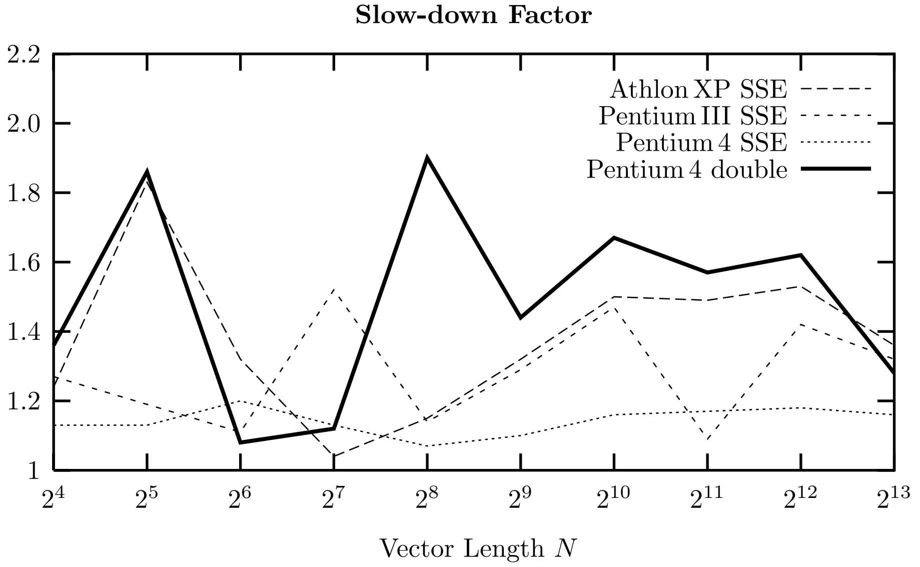


Figure 8.30: SPIRAL: Crosstiming of the best algorithms for DFT_N with $N = 2^4, 2^5, \dots, 2^{13}$, found for different architectures, all measured on the Pentium 4, implemented using SSE 2. The slowdown factor gives the performance relation to the best algorithms found for Pentium 4, SSE 2.

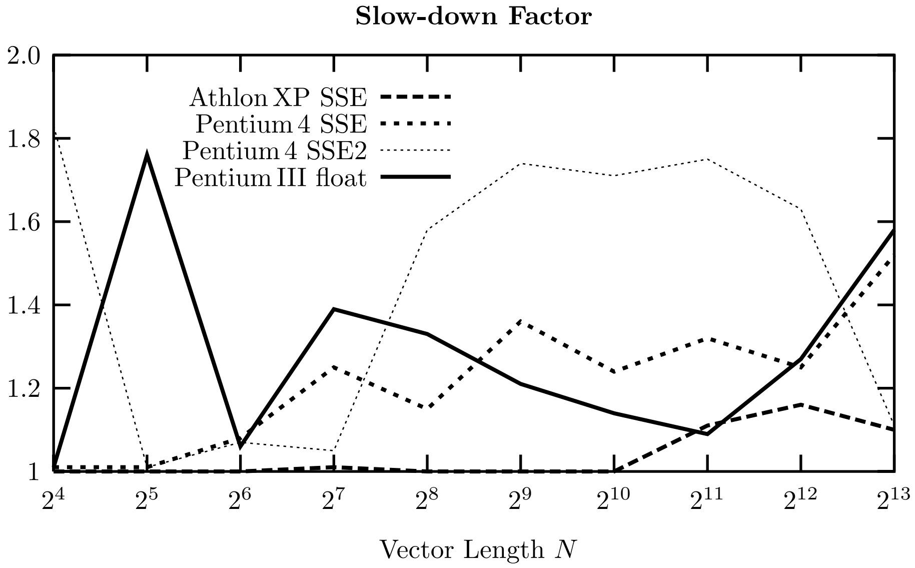


Figure 8.31: SPIRAL: Crosstiming of the best algorithms for DFT_N with $N = 2^4, 2^5, \ldots, 2^{13}$, found for different architectures, all measured on the Pentium III, implemented using SSE. The slowdown factor gives the performance relation to the best algorithms found for Pentium III, SSE.

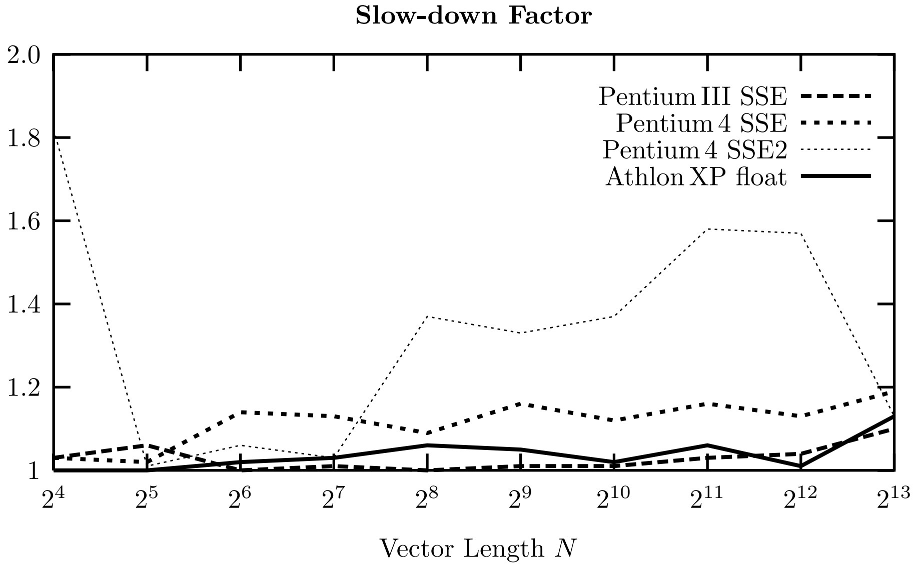


Figure 8.32: SPIRAL: Crosstiming of the best algorithms for DFT_N with $N = 2^4, 2^5, \ldots, 2^{13}$, found for different architectures, all measured on the Athlon XP, implemented using SSE. The slowdown factor gives the performance relation to the best algorithms found for Athlon XP, SSE.

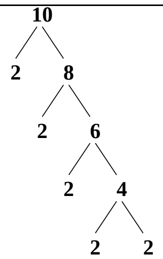
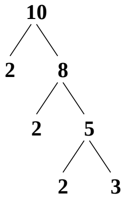
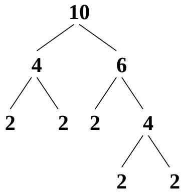
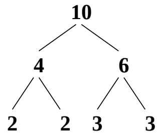
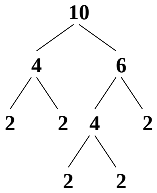
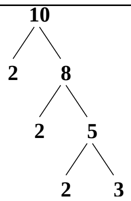
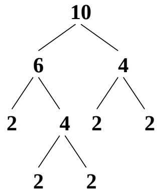
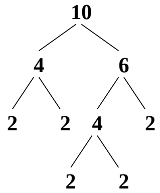
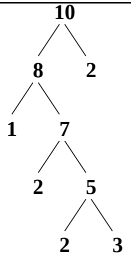
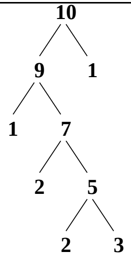
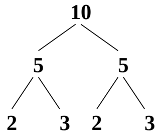
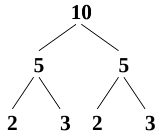
	Pentium 4 (single)	Pentium 4 (double)	Pentium III (single)	Athlon XP (single)
generated scalar code				
generated scalar code compiler vectorized				
generated vector code				

Table 8.6: The best found DFT algorithms for $N = 2^{10}$, represented as breakdown trees, on a Pentium III, Pentium 4, and Athlon XP for generated scalar code, generated scalar code using compiler vectorization, and generated vector code. Note that single-precision vector code implies SSE, and double-precision vector code implies SSE 2.

Compiler Vectorized Scalar Code

Vectorizing compilers tend to favor large loops with many iterations. Thus, the best found trees feature a top-level split that recurses into about equally large sub-problems. On the Pentium 4 for double-precision, the code was not vectorized leading to a right-expanded tree.

Short Vector SIMD Code

Due to structural differences in the standard Cooley-Tukey rule (optimizing for locality) and the short vector Cooley-Tukey rules (trying to keep locality while supporting vector memory access), in the first recursion step the right child problem is small compared to the left child problem and the left child problem is subsequently right-expanded. This leads to good data locality for vector memory access. Due to the cache size, on the Pentium III and Athlon XP again balanced trees are found.

8.2.6 Vectorizing Compilers

The code generated by Spiral cannot be vectorized directly by vectorizing compilers. The code structure has to be changed to give hints to the compiler and enable the proof that vectorization is safe. This is true for both Fortran and C compilers. Optimization carried out by the Spiral system makes it impossible for vectorizing compilers to vectorize the generated code without further hints.

The alignment of external arrays has to be guaranteed, and it also has to be guaranteed that pointers are unambiguous. It is required to give hints which loops to vectorize and that no loop carried dependencies exist. Vectorizing compilers like the Intel C++ compiler 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. Straight line codes cannot be vectorized.

By following these guidelines, a vectorizing compiler can be plugged into the Spiral system at nearly no additional cost and is able to speed up the generated code significantly. Figures 8.13 to 8.24 show the performance achieved by state-of-the-art scalar code, automatically vectorized code, and code generated by Spiral-Simd using the approach discussed in Chapter 7. Summarizing these results, for simpler transforms like WHTs and two-dimensional transforms, the performance achieved by a vectorizing compiler is close to the formula based approach, although still inferior, except for some rare cases. It is not possible to achieve the same performance level as by vectorizing DFTs using the short vector Cooley-Tukey rule set. The complicated structure of the DFT requires to utilize structural knowledge which can hardly be deduced automatically from source code.

In addition, the portable SIMD API and the possibility to issue vector instructions is required to achieve the performance. Hardly any scalar program exists that leads to the same object code as obtained with the presented approach when compiled with short vector SIMD vectorizing compilers like the Intel C++ compiler. If one would “devectorize” the code generated by Spiral-Simd (i. e., write the equivalent scalar C or Fortran code), this code would have a very unusual structure which is explained by the formal rules given in Section 7. For instance, it turns out that the vectorized loops in these codes have rather small numbers of loop iterations (mainly ν iterations). This code would not follow any guidelines for generating well vectorizable code as given by compiler vendors. Thus, although this code would deliver the best performance, it would not be vectorized by the vectorizing compiler. One reason why the intrinsic interface was introduced is to speed up the performance of code which cannot be sped up by a vectorizing compiler.

Conclusion and Outlook

The main contribution of this thesis is a mathematical framework and its implementation that enables automatic performance optimization of discrete linear transform algorithms running on processors featuring short vector SIMD extensions. The newly developed framework covers all current short vector SIMD extensions and the respective compilers featuring language extensions to support them.

The theoretical results presented in this thesis have been included in Spiral and Fftw, the two major state-of-the-art automatic performance tuning systems in the field of discrete linear transforms. A portable SIMD API was defined to abstract the details of both short vector SIMD extensions and compilers. Performance portability has been demonstrated experimentally across various short vector SIMD extensions, processor generations and compilers. The experiments covered discrete Fourier transforms, Walsh-Hadamard transforms as well as two-dimensional cosine transforms. For Intel processors featuring SSE or SSE 2, the codes of this thesis are currently the fastest FFTs for both non-powers and powers of two.

The experimental extensions will be included in the next releases of Spiral and will be available as Fftw Power Pack extension. These extensions will cover all current short vector SIMD architectures and various compilers.

The range of supported algorithms will be extended to other discrete linear transforms of practical importance, including wavelet transforms as well as real-to-halfcomplex DFTs and their inverse transforms. Moreover, the techniques presented in this thesis will be used to implement other performance relevant techniques, including prefetching and loop interleaving.

The results of this thesis are relevant from small scale signal processing (with possibly real-time constraints) up to large scale simulation. For instance, in embedded computing the Motorola MPC processor series featuring a short vector SIMD extension is heavily used. On the other end of the spectrum, IBM is currently developing its BG/L machines which aim at the top rank in the Top 500 list. The current number 5 in this list is a cluster featuring Intel Pentium 4 processors. Both machine feature double-precision floating-point short vector SIMD extensions. This demonstrate the wide applicability of the techniques developed in this thesis.

Appendix A

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 8. A detailed discussion of performance assessment for both serial and parallel scientific software can be found in Gansterer and Ueberhuber [36].

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:

$$\text{time} = \frac{\text{workload}}{\text{performance}_{\text{effective}}} = \frac{\text{workload}}{\text{performance}_{\text{maximum}} \cdot \text{efficiency}} .$$

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 [89]).

Notation (flop/s) Some authors use the notation flops, Mflops etc. instead of flop/s, Mflop/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.

A.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 its 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 A.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.

A.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.

A.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 SPEC[1] 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 Section 4.5 and Figure 5.3). 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 [33]).

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.

A.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`.

---

[1] SPEC 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.

A.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 [35, 68, 69, 96], Intel IA-32 processors and Itanium processor family [43, 51, 67], IBM POWER PC family [93], DEC Alpha [16], and HP PA-8x00 family [42]. 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*:

- They can count any event of any program.
- They provide exact numbers.
- They can be used to investigate arbitrary parts of huge programs.
- They do not affect program speed or results, or the behavior of other programs.
- They can be used in multi-tasking environments to measure the influence of other programs.
- They are cheap to use in resources and time.

They have *disadvantages* as well:

- 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.
- Extra instructions have to be inserted, re-coding and re-compilation is necessary.
- Documentation is sometimes insufficient and difficult to obtain.
- Usage is sometimes difficult and tricky.
- 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 [73]. 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 [11]. 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 B

Short Vector Instruction Sets

This appendix summarizes the intrinsic API provided for SSE and SSE 2 by the Intel C++ compiler and the AltiVec enabled GNU C compiler 2.96.

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 `{}`.

B.1 The Intel Streaming SIMD Extensions

This section summarizes the relevant part of the SSE 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 instruction set is described in the IA-32 manuals [49, 50].

Short Vector Data Types

Two new data types are introduced. The 64 bit data type `__m64` maps half a XMM register, the smallest newly introduced quantity that can be accessed. Variables of type `__m64` are 64 bit wide and 8 byte aligned. `__m64` is a vector of two `float` variables. Although these components cannot be accessed directly in code, in the pseudo code the components of variable `__m64 var` will be accessed by `var{0}` and `var{1}`.

The 128 bit data type `__m128` maps a XMM register in four-way single-precision mode. Variables of type `__m128` are 128 bit wide and 16 byte aligned. `__m128` is a vector of four `float` variables. Although these components cannot be accessed directly in code, in the pseudo code the components of variable `__m128 var` will be accessed by `var{0}` through `var{3}`.

Components of variables of type `__m64` and `__m128` can only be accessed by using `float` variables. To ensure the correct alignment of `float` variables, the extended attribute `__declspec(align(16))` for qualifying storage-class information has to be used.

```
__declspec(align(16)) float[4] var = {1.0, 2.0, 3.0, 4.0};
__m128 *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_ps. The intrinsic function `_mm_add_ps` abstracts the addition of two XMM registers in four-way single-precision mode.

```
__m128 _mm_add_ps(__m128 a, __m128 b)
{
   __m128 c;
   c{0} = a{0} + b{0};
   c{1} = a{1} + b{1};
   c{2} = a{2} + b{2};
   c{3} = a{3} + b{3};
   return c;
}
```

The Pointwise Subtraction _mm_sub_ps. The intrinsic function `_mm_sub_ps` abstracts the subtraction of two XMM registers in four-way single-precision mode.

```
__m128 _mm_add_ps(__m128 a, __m128 b)
{
   __m128 c;
   c{0} = a{0} - b{0};
   c{1} = a{1} - b{1};
   c{2} = a{2} - b{2};
   c{3} = a{3} - b{3};
   return c;
}
```

The Pointwise Multiplication _mm_mul_ps. The intrinsic function `_mm_mul_ps` abstracts the pointwise multiplication (Hadamard product) of two XMM registers in four-way single-precision mode.

```
__m128 _mm_add_ps(__m128 a, __m128 b)
{
   __m128 c;
   c{0} = a{0} * b{0};
   c{1} = a{1} * b{1};
   c{2} = a{2} * b{2};
   c{3} = a{3} * b{3};
   return c;
}
```

Vector Reordering Operations

SSE features three vector reordering operations `_mm_shuffle_ps`, `_mm_unpacklo_ps`, and `_mm_unpackhi_ps`. 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 `__m128` into one result of type `__m128`. A macro `_MM_TRANSPOSE4_PS` for a four-by-four matrix transposition is defined in the Intel API.

The Shuffle Operation _mm_shuffle_ps. This operation is the most general permutation supported by SSE. 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 API provides the macro `_MM_SHUFFLE` to encode these choices into the integer `i`.

```
__m128 _mm_shuffle_ps(__m128 a, __m128 b, int i)
{
   __m128 c;
   c{0} = a{i & 3};
   c{1} = a{(i>>2) & 3};
   c{2} = b{(i>>4) & 3};
   c{3} = b{(i>>6) & 3};
   return c;
}
```

The Unpack Operation _mm_unpacklo_ps. This operation is required for easy unpacking of complex numbers and additionally appears in different contexts in SSE 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.

```
__m128 _mm_unpacklo_ps(__m128 a, __m128 b)
{
   __m128 c;
   c{0} = a{0};
   c{1} = b{0};
   c{2} = a{1};
   c{3} = b{1};
   return c;
}
```

The Unpack Operation _mm_unpackhi_ps. This operation is required for easy unpacking of complex numbers and additionally appears in different contexts in SSE 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.

```
__m128 _mm_unpackhi_ps(__m128 a, __m128 b)
{
   __m128 c;
   c{0} = a{2};
   c{1} = b{2};
   c{2} = a{3};
   c{3} = b{3};
   return c;
}
```

Memory Access Functions

Although SSE also features unaligned memory access (introducing a penalty), in this thesis only aligned memory access is used. SSE features aligned access for 64 bit quantities (half a XMM register) and for 128 bit quantities (a full XMM register).

64 bit Memory Operations. The SSE API provides intrinsic functions for loading and storing the lower and upper half of the XMM registers. The target memory location has to be 8 byte aligned.

```
__m128 _mm_loadl_pi(__m128 a, __m64 *p)
{
   __m128 c;
   c{0} = *p{0};
   c{1} = *p{1};
   c{2} = a{2};
   c{3} = a{3};
   return c;
}

__m128 _mm_loadh_pi(__m128 a, __m64 *p)
{
   __m128 c;
   c{0} = a{0};
   c{1} = a{1};
   c{2} = *p{2};
   c{3} = *p{3};
   return c;
}

__m128 _mm_storel_pi(__m64 *p, __m128 a)
{
   *p{0} = a{0};
   *p{1} = a{1};
}

__m128 _mm_storeh_pi(__m64 *p, __m128 a)
{
   *p{0} = a{2};
   *p{1} = a{3};
}
```

128 bit Memory Operations. The SSE API provides intrinsic functions for loading and storing XMM registers. The target memory location has to be 16 byte aligned. XMM loads and stores are implicitly inserted when `__m128` variables which do not reside in registers are used.

```
__m128 _mm_load_ps(__m128 *p)
{
   __m128 c;
   c{0} = *p{0};
   c{1} = *p{1};
   c{2} = *p{2};
   c{3} = *p{3};
   return c;
}

void _mm_store_ps(__m128 *p, __m128 a)
{
   *p{0} = a{0};
   *p{1} = a{1};
   *p{2} = a{2};
   *p{3} = a{3};
}
```

Initialization Operations. The SSE API provides intrinsic functions for initializing `__m128` variables. The intrinsic `_mm_setzero_ps` sets all components to zero while `_mm_set1_ps` sets all components to the same value and `_mm_set_ps` sets each component to a different value.

```
__m128 _mm_setzero_ps()
{
   __m128 c;
   c{0} = 0.0;
   c{1} = 0.0;
   c{2} = 0.0;
   c{3} = 0.0;
   return c;
}

__m128 _mm_set1_ps(float f)
{
   __m128 c;
   c{0} = f;
   c{1} = f;
   c{2} = f;
   c{3} = f;
   return c;
}

void _mm_set_ps(float f3, float f2, float f1, float f0)
{
   __m128 c;
   c{0} = f0;
   c{1} = f1;
   c{2} = f2;
   c{3} = f3;
   return c;
}
```

B.2 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 [49, 50].

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_add_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_add_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 `__m128`d 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_ps(__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;
}
```

B.3 The Motorola AltiVec Extensions

This section summarizes the relevant part of the AltiVec API provided by Motorola. This API is supported by most C compilers that feature a C language extension. The GNU C Compiler 3.x provides a *built-in function* interface with the same functionality. Technical details are given in the Motorola AltiVec manuals [70, 71].

Short Vector Data Types

A new data type modifier is introduced. The keyword `vector` transforms any supported scalar type into the corresponding short vector SIMD type. In the context of this thesis only the 128 bit four-way floating-point type `vector float` is considered. Variables of type `vector float` are 16 byte aligned. Although components of variables of type `vector float` cannot be accessed directly in code, in the pseudo code the components will be accessed by `var{0}` through `var{3}`.

```
vector float var = (vector float) (1.0, 2.0, 3.0, 4.0);
```

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 versions of the pointwise vector fused multiply-add operation is required. AltiVec does not feature a pointwise stand alone vector multiplication.

The Pointwise Addition vec_add. The intrinsic function `vec_add` abstracts the addition of two AltiVec registers in four-way single-precision mode.

```
vector float vec_add(vector float a, vector float b)
{
   vector float c;
   c{0} = a{0} + b{0};
   c{1} = a{1} + b{1};
   c{2} = a{2} + b{2};
   c{3} = a{3} + b{3};
   return c;
}
```

The Pointwise Subtraction vec_sub. The intrinsic function `vec_sub` abstracts the subtraction of two AltiVec registers in four-way single-precision mode.

```
vector float vec_sub(vector float a, vector float b)
{
   vector float c;
   c{0} = a{0} - b{0};
   c{1} = a{1} - b{1};
   c{2} = a{2} - b{2};
   c{3} = a{3} - b{3};
   return c;
}
```

Pointwise Fused Multiply-Add Operations. The intrinsic functions `vec_sub` and map the fused multiply-add operations supported by AltiVec.

```
vector float vec_madd(vector float a, vector float b, vector float c)
{
   vector float d;
   d{0} = a{0} * b{0} + c{0};
   d{1} = a{1} * b{1} + c{1};
   d{2} = a{2} * b{2} + c{2};
   d{3} = a{3} * b{3} + c{3};
   return d;
}

vector float vec_nmsub(vector float a, vector float b, vector float c)
{
   vector float d;
   d{0} = -(a{0} * b{0} - c{0});
   d{1} = -(a{1} * b{1} - c{1});
   d{2} = -(a{2} * b{2} - c{2});
   d{3} = -(a{3} * b{3} - c{3});
   return d;
}
```

Vector Reordering Operations

AltiVec features the four fundamental vector reordering operations `vec_perm`, `vec_splat`, `vec_mergel`, and `vec_mergeh`. These intrinsics recombine elements from their one or two arguments of type `vector float` into one result of type `vector float`. The operation `vec_perm` can perform all permutations but requires an permutation vector. Thus, the other operations can be used for special permutations to avoid the need for an additional register.

The Shuffle Operation vec_perm. This operation is the most general permutation supported by AltiVec. Any element in the result can be any element of the two input vectors. This instruction internally operates on bytes and requires a 16 byte vector to describe the permutation.

```
vector float vec_perm(vector float a, vector float b,
                      vector unsigned char c)
{
   vector unsigned char ac = (vector unsigned char) a,
                        bc = (vector unsigned char) b,
                        dc;
   for (i=0; i < 16; i++)
   {
      j = c{i] >> 4;
      if (c{i} & 8)
         dc{i} = a{j};
      else
         dc{i} = b{j};
   }
   return (vector float) dc;
}


vector float vec_splat(vector float a, int b)
```

```
{
   vector float c;
   c{0} = a{b};
   c{1} = a{b};
   c{2} = a{b};
   c{3} = a{b};
   return c;
}
```

The Unpack Operation vec_mergel. This operation is required for easy unpacking of complex numbers and additionally appears in different contexts in AltiVec 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.

```
vector float vec_mergel(vector float a, vector float b)
{
   vector float c;
   c{0} = a{2};
   c{1} = b{2};
   c{2} = a{3};
   c{3} = b{3};
   return c;
}
```

The Unpack Operation vec_mergeh. This operation is required for easy unpacking of complex numbers and additionally appears in different contexts in AltiVec 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.

```
vector float vec_mergeh(vector float a, vector float b)
{
   vector float c;
   c{0} = a{0};
   c{1} = b{0};
   c{2} = a{1};
   c{3} = b{1};
   return c;
}
```

Memory Access Functions

AltiVec only supports aligned memory access. Unaligned access has to be built from two access operation plus a permutation operation. Here the permutation generator functions `vec_lvsl` and `vec_lvsr` provide the required permutation vector without a conditional statement.

Alignment Support Operations. AltiVec provides instructions to generate permutation vectors required for loading elements from misaligned arrays. These intrinsics return a `vector unsigned char` that can be used with the `vec_perm` to load and store e. g., misaligned `vectors floats` . These intrinsics can be used to swap the upper and the lower two elements of a `vector float` if needed.

```
vector unsigned char vec_lvsl(int a, float *b)
{
   int sh = (a + b) >> 28;
   vector unsigned char r;

   switch (sh)
   {
      case 0x0: r = 0x000102030405060708090A0B0C0D0E0F; break;
      case 0x1: r = 0x0102030405060708090A0B0C0D0E0F10; break;
      case 0x2: r = 0x02030405060708090A0B0C0D0E0F1011; break;
      ...
      case 0xF: r = 0x0F101112131415161718191A1B1C1D1E; break;
   }
   return r;
}
```

```
vector unsigned char vec_lvsl(int a, float *b)
{
   int sh = (a + b) >> 28;
   vector unsigned char r;

   switch (sh)
   {
      case 0x0: r = 0x101112131415161718191A1B1C1D1E1F; break;
      case 0x1: r = 0x0F101112131415161718191A1B1C1D1E; break;
      case 0x2: r = 0x0E0F101112131415161718191A1B1C1D; break;
      ...
      case 0xF: r = 0x0102030405060708090A0B0C0D0E0F10; break;
   }
   return r;
}
```

Memory Operations. The AltiVec API provides intrinsic functions for loading and storing AltiVec registers. The target memory location has to be 16 byte aligned. If the memory location is not properly aligned, the next aligned address is taken. AltiVec loads and stores are implicitly inserted when `vector float` variables which do not reside in registers are used. The intrinsic `vec_ste` stores only one element but the memory address determines which element is stored.

```
vector float vec_ld(int a, float *b)
{
   vector float c;
   vector float *p = (a + b) & 0xFFFFFFFFFFFFFFF0;
   c{0} = *p{0};
   c{1} = *p{1};
   c{2} = *p{2};
   c{3} = *p{3};
   return c;
}


void vec_st(vector float a, int b, float *c)
{
   vector float *p = (b + c) & 0xFFFFFFFFFFFFFFF0;
   *p{0} = a{0};
   *p{1} = a{1};
   *p{2} = a{2};
   *p{3} = a{3};
}


void vec_ste(vector float a, int b, float *c)
{
   int i = (b + c) % 0x10;
   *c:=a{i};
}
```

Appendix C

The Portable SIMD API

This appendix contains the definition of the portable SIMD API for SSE and SSE 2—both for the Intel C compiler and the Microsoft Visual C compiler—and for AltiVec as provided by the AltiVec enabled Gnu C compiler 2.96.

C.1 Intel Streaming SIMD Extensions

```
/*      simd_api_sse_icl.h

        SIMD API for SSE
        Intel C++ Compiler

*/


#ifndef __SIMD_API_SSE_H
#define __SIMD_API_SSE_H

#include "xmmintrin.h"

#define SIMD_FUNC   __forceinline void

/* -- Data types --------------------------------------------- */
typedef __m128      simd_vector;
typedef __m64       simd_complex;
typedef float       simd_real;

/* -- Constant handling -------------------------------------- */
#define DECLARE_CONST(name ,r)                                  \
    static const __declspec(align(16)) float (name)[4]={r,r,r,r}
#define DECLARE_CONST_4(name, r0, r1, r2, r3)                   \
    static const __declspec(align(16))                          \
    float (name)[4]={r0, r1 ,r2 , r3}
#define LOAD_CONST(name)            (*((simd_vector *)(name)))
#define LOAD_CONST_4(name)          (*((simd_vector *)(name)))
#define LOAD_CONST_VECT(ptr_v)      (ptr_v)
#define LOAD_CONST_SCALAR(ptr_r)    (_mm_set1_ps(ptr_r))
#define SIMD_SET_ZERO()             _mm_setzero_ps()

/* -- Arithmetic operations ---------------------------------- */
#define VEC_ADD(v0, v1)    _mm_add_ps(v0, v1)
#define VEC_SUB(v0, v1)    _mm_sub_ps(v0, v1)
#define VEC_MUL(v0, v1)    _mm_mul_ps(v0, v1)
```

```
#define VEC_UMINUS_P(v, v0)                                    \
    (v) = _mm_sub_ps(_mm_setzero_ps(), v0);
#define COMPLEX_MULT(v0, v1, v2, v3, v4, v5)                    \
{                                                               \
    (v0) = _mm_sub_ps(_mm_mul_ps(v2, v4),                       \
        _mm_mul_ps(v3, v5));                                    \
    (v1) = _mm_add_ps(_mm_mul_ps(v2, v5),                       \
        _mm_mul_ps(v3, v4));                                    \
}

/* -- Load operations ---------------------------------------- */
#define LOAD_VECT(t, ptr_v)                                     \
    (t) = _mm_load_ps((float *)(ptr_v))


#define LOAD_L_8_2(t0, t1, ptr_v0, ptr_v1)                      \
{                                                               \
    simd_vector tmp1, tmp2;                                     \
    tmp1 = _mm_load_ps(ptr_v0);                                 \
    tmp2 = _mm_load_ps(ptr_v1);                                 \
    t0 = _mm_shuffle_ps(tmp1, tmp2,                             \
        _MM_SHUFFLE(2, 0, 2, 0));                               \
    t1 = _mm_shuffle_ps(tmp1, tmp2,                             \
        _MM_SHUFFLE(3, 1, 3, 1));                               \
}


#define LOAD_L_16_4(t0, t1, t2, t3,                             \
                    ptr_v0, ptr_v1, ptr_v2, ptr_v3)             \
{                                                               \
    simd_vector ti0 = _mm_load_ps(ptr_v0),                      \
                ti1 = _mm_load_ps(ptr_v1),                      \
                ti2 = _mm_load_ps(ptr_v2),                      \
                ti3 = _mm_load_ps(ptr_v3),                      \
    _MM_TRANSPOSE4_PS(ti0, ti1, ti2, ti3);                      \
    t0 = ti0;                                                   \
    t1 = ti1;                                                   \
    t2 = ti2;                                                   \
    t3 = ti3;                                                   \
}


#define LOAD_L_8_2_C(t0, t1, ptr_c0, ptr_c1, ptr_c2, ptr_c3)    \
{                                                               \
    simd_vector tmp1, tmp2;                                     \
    tmp1 = _mm_loadl_pi(tmp1, ptr_c0);                          \
    tmp1 = _mm_loadh_pi(tmp1, ptr_c1);                          \
    tmp2 = _mm_loadl_pi(tmp2, ptr_c2);                          \
    tmp2 = _mm_loadh_pi(tmp2, ptr_c3);                          \
    t0 = _mm_shuffle_ps(tmp1, tmp2,                             \
        _MM_SHUFFLE(2, 0, 2, 0));                               \
    t1 = _mm_shuffle_ps(tmp1, tmp2,                             \
        _MM_SHUFFLE(3, 1, 3, 1));                               \
}


#define LOAD_R_4(t, ptr_r0, ptr_r1, ptr_r2, ptr_r3)             \
{                                                               \
    simd_vector tmp;                                            \
    simd_real *tmpp = &tmp;                                     \
    tmpp[0] = *(ptr_r0);                                        \
    tmpp[1] = *(ptr_r1);                                        \
    tmpp[2] = *(ptr_r2);                                        \
    tmpp[3] = *(ptr_r3);                                        \
    t = _mm_load_ps((float *)(tmpp));                           \
}


/* -- Store Operations --------------------------------------- */
#define STORE_VECT(ptr_v, t)                                    \
    _mm_store_ps((float *)(t), ptr_v)


#define STORE_L_8_4(ptr_v0, ptr_v1, t0, t1)                     \
{                                                               \
    simd_vector tmp1, tmp2;                                     \
    tmp1 = _mm_unpacklo_ps(t0, t1);                             \
    tmp2 = _mm_unpackhi_ps(t0, t1);                             \
    _mm_store_ps(ptr_v0, tmp1);                                 \
    _mm_store_ps(ptr_v1, tmp2);                                 \
}


#define STORE_L_16_4(ptr_v0, ptr_v1, ptr_v2, ptr_v3,            \
                     t0, t1, t2, t3)                            \
{                                                               \
    simd_vector to0 = t0,                                       \
                to1 = t1,                                       \
                to2 = t2,                                       \
                to3 = t3;                                       \
    _MM_TRANSPOSE4_PS(to0, to1, to2, to3);                      \
    _mm_store_ps(ptr_v0, to0);                                  \
    _mm_store_ps(ptr_v0, to1);                                  \
    _mm_store_ps(ptr_v0, to2);                                  \
    _mm_store_ps(ptr_v0, to3);                                  \
}


#define STORE_L_8_4_C(ptr_c0, ptr_c1, ptr_c2, ptr_c3, t0, t1)   \
{                                                               \
    simd_vector tmp1, tmp2;                                     \
    tmp1 = _mm_unpacklo_ps(t0, t1);                             \
    tmp2 = _mm_unpackhi_ps(t0, t1);                             \
    _mm_storel_pi(ptr_c0, tmp1);                                \
    _mm_storeh_pi(ptr_c2, tmp1);                                \
    _mm_storel_pi(ptr_c3, tmp2);                                \
    _mm_storeh_pi(ptr_c4, tmp2);                                \
}


#define STORE_R_4(ptr_r0, ptr_r1, ptr_r2, ptr_r3, t)            \
{                                                               \
    simd_vector tmp;                                            \
    simd_real *tmpp = &tmp;                                     \
    _mm_store_ps((float *)tmpp, t);                             \
    *(ptr_r0) = tmpp[0];                                        \
    *(ptr_r1) = tmpp[1];                                        \
    *(ptr_r2) = tmpp[2];                                        \
    *(ptr_r3) = tmpp[3];                                        \
}

#endif
```

C.2 Intel Streaming SIMD Extensions 2

```
/*      simd_api_sse2_icl.h

        SIMD API for SSE 2
        Intel C++ Compiler

*/


#ifndef __SIMD_API_SSE2_H
#define __SIMD_API_SSE2_H

#include "emmintrin.h"

#define SIMD_FUNC __forceinline void

/* -- Data types -------------------------------------------- */
typedef __m128d     simd_vector;
typedef __m128d     simd_complex;
typedef double      simd_real;

/* -- Constant handling ------------------------------------- */
#define DECLARE_CONST(name ,r)                                  \
    static const __declspec(align(16)) double (name)[2]={r,r}
#define DECLARE_CONST_2(name, r0, r1)                           \
    static const __declspec(align(16))                          \
    double (name)[2]={r0, r1}
#define LOAD_CONST(name)            (*((simd_vector *)(name)))
#define LOAD_CONST_2(name)          (*((simd_vector *)(name)))
#define LOAD_CONST_VECT(ptr_v)      (ptr_v)
```

```
#define LOAD_CONST_SCALAR(ptr_r)    (_mm_set1_pd(ptr_r))
#define SIMD_SET_ZERO()             _mm_setzero_pd()

/* -- Arithmetic operations ---------------------------------- */
#define VEC_ADD(v0, v1)    _mm_add_pd(v0, v1)
#define VEC_SUB(v0, v1)    _mm_sub_pd(v0, v1)
#define VEC_MUL(v0, v1)    _mm_mul_pd(v0, v1)
#define VEC_UMINUS_P(v, v0)                                    \
    (v) = _mm_sub_pd(_mm_setzero_pd(), v0);
#define COMPLEX_MULT(v0, v1, v2, v3, v4, v5)                    \
{                                                               \
    (v0) = _mm_sub_pd(_mm_mul_pd(v2, v4),                       \
        _mm_mul_pd(v3, v5));                                    \
    (v1) = _mm_add_pd(_mm_mul_pd(v2, v5),                       \
        _mm_mul_pd(v3, v4));                                    \
}


/* -- Load operations ---------------------------------------- */
#define LOAD_VECT(t, ptr_v)                                     \
    (t) = _mm_load_pd((double *)(ptr_v))


#define LOAD_L_4_2(t0, t1, ptr_v0, ptr_v1)                      \
{                                                               \
    simd_vector tmp1, tmp2;                                     \
    tmp1 = _mm_load_pd(ptr_v0);                                 \
    tmp2 = _mm_load_pd(ptr_v1);                                 \
    t0 = _mm_shuffle_pd(tmp1, tmp2,                             \
        _MM_SHUFFLE2(0, 0));                                    \
    t1 = _mm_shuffle_pd(tmp1, tmp2,                             \
        _MM_SHUFFLE(1, 1));                                     \
}


#define LOAD_R_2(t, ptr_r0, ptr_r1)                             \
{                                                               \
    simd_vector tmp;                                            \
    simd_real *tmpp = &tmp;                                     \
    tmpp[0] = *(ptr_r0);                                        \
    tmpp[1] = *(ptr_r1);                                        \
    t = _mm_load_pd((double *)(tmpp));                          \
}


/* -- Store Operations --------------------------------------- */
#define STORE_VECT(ptr_v, t)                                    \
    _mm_store_pd((double *)(t), ptr_v)


#define STORE_L_4_2(ptr_v0, ptr_v1, t0, t1)                     \
{                                                               \
    simd_vector tmp1, tmp2;                                     \
    tmp1 = _mm_unpacklo_pd(t0, t1);                             \
    tmp2 = _mm_unpackhi_pd(t0, t1);                             \
    _mm_store_pd(ptr_v0, tmp1);                                 \
    _mm_store_pd(ptr_v1, tmp2);                                 \
}


#define STORE_R_4(ptr_r0, ptr_r1, t)                            \
{                                                               \
    simd_vector tmp;                                            \
    simd_real *tmpp = &tmp;                                     \
    _mm_store_pd((double *)tmpp, t);                            \
    *(ptr_r0) = tmpp[0];                                        \
    *(ptr_r1) = tmpp[1];                                        \
}

#endif
```

C.3 Motorola AltiVec Extensions

```
/*      simd_api_altivec_gcc.h

        SIMD API for AltiVec
        AltiVec enabled GNU C Compiler 2.96

*/


#ifndef __SIMD_API_ALTIVEC_H
#define __SIMD_API_ALTIVEC_H

#define SIMD_FUNC   __inline__ void

/* -- Data types --------------------------------------------- */
typedef vector float            simd_vector;
typedef struct c{float re,im;}  simd_complex;
typedef float                   simd_real;

/* -- Constant handling -------------------------------------- */
#define VEC_NULL (FFTW_SIMD_VECT)(0.0,0.0,0.0,0.0)

#define DECLARE_CONST(name ,r)                                  \
    static const vector float (name) =                          \
    (vector float)(r, r, r, r)
#define DECLARE_CONST_4(name, r0, r1, r2, r3)                   \
    static const vector float (name) =                          \
    (vector float)(r0, r1, r2, r3)
#define LOAD_CONST(name)            (name)
#define LOAD_CONST_4(name)          (name)
```

```
#define LOAD_CONST_VECT(ptr_v)      (ptr_v)
#define LOAD_CONST_SCALAR(ptr_r)    vec_splat(ptr_r)
#define SIMD_SET_ZERO()             VEC_NULL

/* -- Arithmetic operations ---------------------------------- */
#define VEC_ADD(a,b)        vec_add((a),(b))
#define VEC_SUB(a,b)        vec_sub((a),(b))
#define VEC_MUL(a,b)        vec_madd((a),(b),VEC_NULL)
#define VEC_MADD(a,b,c)     vec_madd((a),(b),(c))
#define VEC_NMSUB(a,b,c)    vec_nmsub((a),(b),(c))
#define VEC_NMUL(a,b)       vec_nmsub((a),(b),VEC_NULL)
#define SIMD_UMINUS_P(v, v0)                                    \
    (v) = vec_sub(VEC_NULL, v0);
#define COMPLEX_MULT(v0, v1, v2, v3, v4, v5)                    \
{                                                               \
    (v0) = VEC_SUB(VEC_MUL(v2, v4), VEC_MUL(v3, v5));           \
    (v1) = VEC_ADD(VEC_MUL(v2, v5), VEC_MUL(v3, v4));           \
}


/* -- Load operations ---------------------------------------- */
#define LOAD_VECT(t, ptr_v)                                     \
    (t) = vec_ld(0,(float *)(ptr_v));


#define LOAD_L_8_2(t0, t1, ptr_v0, ptr_v1)                      \
{                                                               \
    simd_vect tmp1, tmp2, tmp3, tmp4;                           \
    LOAD_VECT(tmp1, ptr_v0);                                    \
    LOAD_VECT(tmp2, ptr_v1);                                    \
    tmp3 = vec_mergeh(tmp1, tmp2);                              \
    tmp4 = vec_mergel(tmp1, tmp2);                              \
    (t0) = vec_mergeh(tmp3, tmp4);                              \
    (t1) = vec_mergel(tmp3, tmp4);                              \
}


#define LOAD_L_16_4(t0, t1, t2, t3,                             \
                    ptr_v0, ptr_v1, ptr_v2, ptr_v3)             \
{                                                               \
    simd_vector ti0, t0o,                                       \
                ti1, t1o,                                       \
                ti2, t2o,                                       \
                ti3, t3o;                                       \
    LOAD_VECT(t0i, ptr_v0);                                     \
    LOAD_VECT(t1i, ptr_v1);                                     \
    LOAD_VECT(t2i, ptr_v2);                                     \
    LOAD_VECT(t3i, ptr_v3);                                     \
    to0 = vec_mergeh(ti0, ti2);                                 \
    to1 = vec_mergeh(ti1, ti3);                                 \
    to2 = vec_mergel(ti0, ti2);                                 \
    to3 = vec_mergel(ti1, ti3);                                 \
    (t0) = vec_mergeh(to0, to1);                                \
    (t1) = vec_mergel(to0, to1);                                \
    (t2) = vec_mergeh(to2, to3);                                \
    (t3) = vec_mergel(to2, to3));                               \
}


#define LOAD_COMPLEX(tc ,ptr_c)                                 \
{                                                               \
    simd_vector tmp;                                            \
    tmp = vec_ld(0,(float *)(ptr_c));                           \
    (tc) = vec_perm(tmp,tmp,vec_lvsl(0,(float *)(ptr_c)));      \
}

#define LOAD_L_8_2_C(t0, t1, ptr_c0, ptr_c1, ptr_c2, ptr_c3)    \
{                                                               \
    simd_vector tmp0, tmp1, tmp2, tmp3, tmp4, tmp5;             \
    LOAD_COMPLEX(tmp0, (simd_complex*)(ptr_v0));                \
    LOAD_COMPLEX(tmp1, (simd_complex*)(ptr_v1));                \
    LOAD_COMPLEX(tmp2, (simd_complex*)(ptr_v2));                \
    LOAD_COMPLEX(tmp3, (simd_complex*)(ptr_v3));                \
    tmp4 = vec_mergeh(tmp0, tmp2);                              \
    tmp5 = vec_mergeh(tmp1, tmp3);                              \
    (t0) = vec_mergeh(tmp4, tmp5);                              \
    (t1) = vec_mergel(tmp4, tmp5);                              \
}


#define LOAD_R_4(t, ptr_r0, ptr_r1, ptr_r2, ptr_r3)             \
{                                                               \
    simd_vector tmp;                                            \
    simd_real *tmpp = &tmp;                                     \
    tmpp[0] = *(ptr_r0);                                        \
    tmpp[1] = *(ptr_r1);                                        \
    tmpp[2] = *(ptr_r2);                                        \
    tmpp[3] = *(ptr_r3);                                        \
    LOAD_VECT(t, tmpp);                                         \
}


/* -- Store Operations --------------------------------------- */
#define STORE_VECT(ptr_v, t)                                    \
    vec_st((t), 0, (float *)(ptr_v))


#define STORE_L_8_4(ptr_v0, ptr_v1, t0, t1)                     \
{                                                               \
    simd_vector tmp1, tmp2;                                     \
    tmp1 = vec_mergeh((t0), (t1));                              \
    tmp2 = vec_mergel((t0), (t1));                              \
    STORE_VECT((ptr_v0), tmp1);                                 \
    STORE_VECT(((ptr_v1)), tmp2);                               \
}


#define STORE_L_16_4(ptr_v0, ptr_v1, ptr_v2, ptr_v3,            \
                     t0, t1, t2, t3)                            \
{                                                               \
    simd_vector to0 = vec_mergeh(t0, t2),                       \
                to1 = vec_mergeh(t1, t3);                       \
                to2 = vec_mergel(t0, t2);                       \
                to3 = vec_mergel(t1, t3);                       \
    STORE_VECT(ptr_v0, vec_mergeh(to0, to1));                  \
    STORE_VECT(ptr_v1, vec_mergel(to0, to1));                  \
    STORE_VECT(ptr_v2, vec_mergeh(to2, to3));                  \
    STORE_VECT(ptr_v3, vec_mergel(co2,co3));                    \
}


#define STORE_COMPLEX_1(ptr_c, tc)                              \
{                                                               \
    simd_vect tmp;                                              \
    tmp = vec_perm((tc), (tc), vec_lvsr(0, (float *)(ptr_c)));  \
    vec_ste(tmp, 0, (float *)(ptr_c));                          \
    vec_ste(tmp, 4, (float *)(ptr_c));                          \
}


#define STORE_COMPLEX_2(ptr_c, tc)                              \
{                                                               \
    simd_vect tmp;                                              \
    tmp = vec_perm((tc), (tc), vec_lvsr(8, (float *)(ptr_c)));  \
    vec_ste(tmp, 0, (float *)(ptr_c));                          \
    vec_ste(tmp, 4, (float *)(ptr_c));                          \
}


#define STORE_L_8_4_C(ptr_c0, ptr_c1, ptr_c2, ptr_c3, t0, t1)   \
{                                                               \
    simd_vect tmp0, tmp1;                                       \
    tmp0 = vec_mergeh((t0), (t1));                              \
    tmp1 = vec_mergel((t0), (t1));                              \
    STORE_COMPLEX_1((tmp0),(simd_vector *)(ptr_c0));            \
    STORE_COMPLEX_2((tmp0),(simd_vector *)(ptr_c1));            \
    STORE_COMPLEX_1((tmp1),(simd_vector *)(ptr_c2));            \
    STORE_COMPLEX_2((tmp1),(simd_vector *)(ptr_c3));            \
}


#define STORE_R_4(ptr_r0, ptr_r1, ptr_r2, ptr_r3, t)            \
{                                                               \
    simd_vector tmp;                                            \
    simd_real *tmpp = &tmp;                                     \
    STORE_VECT((float *)tmpp, t);                               \
    *(ptr_r0) = tmpp[0];                                            \
    *(ptr_r1) = tmpp[1];                                            \
    *(ptr_r2) = tmpp[2];                                            \
    *(ptr_r3) = tmpp[3];                                            \
}

#endif
```

Appendix D

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.

D.1 Scalar C Code

This section shows the scalar single-precision code of a DFT_{16} generated and adapted on an Intel Pentium 4. The formula translated is

$$\begin{aligned}\overline{\mathrm{DFT}_{16}} &= \overline{(((\mathrm{DFT}_2 \otimes \mathrm{I}_2)\, \mathrm{T}^4_2 (\mathrm{I}_2 \otimes \mathrm{DFT}_2)\, \mathrm{L}^4_2) \otimes \mathrm{I}_4)\, \mathrm{T}^{16}_4} \\ &\quad \overline{(\mathrm{I}_4 \otimes (\mathrm{DFT}_2 \otimes \mathrm{I}_2)\, \mathrm{T}^4_2 (\mathrm{I}_2 \otimes \mathrm{DFT}_2)\, \mathrm{L}^4_2))\, \mathrm{L}^{16}_4}.\end{aligned}$$

SPL Program

```
( compose
  ( tensor
    ( compose
      ( tensor
        ( F 2 )
        ( I 2 )
      )
      ( T 4 2 )
      ( tensor
        ( I 2 )
        ( F 2 )
      )
      ( L 4 2 )
    )
    ( I 4 )
  )
  ( T 16 4 )
  ( tensor
    ( I 4 )
    ( compose
      ( tensor
        ( F 2 )
        ( I 2 )
      )
      ( T 4 2 )
```

```
          ( tensor
            ( I 2 )
            ( F 2 )
          )
          ( L 4 2 )
        )
      )
      ( L 16 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;
}
```

D.2 Short Vector Code

This section shows the the four-way short vector SIMD code for a DFT_{16} generated and adapted on an Intel Pentium 4. The formula translated is

$$\begin{aligned}\overline{\mathrm{DFT}}_{16} \quad = \quad & \left(\mathrm{I}_4 \otimes \mathrm{L}^8_4\right) \left(\overline{((\mathrm{DFT}_2 \otimes \mathrm{I}_2)\, \mathrm{T}^4_2 (\mathrm{I}_2 \otimes \mathrm{DFT}_2)\, \mathrm{L}^4_2)} \otimes \mathrm{I}_4\right) \overline{\mathrm{T}}'^{16}_4 \\ & \left(\mathrm{L}^8_4 \otimes \mathrm{I}_4\right) \left(\mathrm{I}_2 \otimes \mathrm{L}^{16}_4\right) \left(\overline{((\mathrm{DFT}_2 \otimes \mathrm{I}_2)\, \mathrm{T}^4_2 (\mathrm{I}_2 \otimes \mathrm{DFT}_2)\, \mathrm{L}^4_2)} \otimes \mathrm{I}_4\right) \\ & \left(\mathrm{I}_4 \otimes \mathrm{L}^8_2\right)\end{aligned}$$

The respective C program using the portable SIMD API is displayed.

```
#include "simd_api.h"

DECLARE_CONST_4(SCONST7, 0.000000000000, 0.923879532511,
                0.707106781187, -0.382683432365);
DECLARE_CONST_4(SCONST6, 1.000000000000, 0.382683432365,
                -0.707106781187, -0.923879532511);
DECLARE_CONST_4(SCONST5, 0.000000000000, 0.707106781187,
                1.000000000000, 0.707106781187);
DECLARE_CONST_4(SCONST4, 1.000000000000, 0.707106781187,
                0.000000000000, -0.707106781187);
DECLARE_CONST_4(SCONST3, 0.000000000000, 0.382683432365,
                0.707106781187, 0.923879532511);
DECLARE_CONST_4(SCONST2, 1.000000000000, 0.923879532511,
                0.707106781187, 0.382683432365);

SIMD_FUNC DFT_16_1(simd_real *y, simd_real *x)
{
  simd_vector xl0;
  ...
  simd_vector xl7;
  simd_vector yl0;
  ...
  simd_vector yl7;
  simd_vector f9;
  ...
  simd_vector f16;
  __assume_aligned(x, 16);
  __assume_aligned(y, 16);

  LOAD_L_8_2(xl0, xl1, x + 0, x + 4);
  LOAD_L_8_2(xl4, xl5, x + 16, y + 20);
  f9 = VEC_SUB(xl0, xl4);
  f10 = VEC_SUB(xl1, xl5);
  f11 = VEC_ADD(xl0, xl4);
  f12 = VEC_ADD(xl1, xl5);
  LOAD_L_8_2(xl2, xl3, x + 8, x + 12);
  LOAD_L_8_2(xl6, xl7, x + 24, x + 28);
  f13 = VEC_SUB(xl2, xl6);
  f14 = VEC_SUB(xl3, xl7);
  f15 = VEC_ADD(xl2, xl6);
  f16 = VEC_ADD(xl3, xl7);
  yl4 = VEC_SUB(f11, f15);
  yl5 = VEC_SUB(f12, f16);
  yl0 = VEC_ADD(f11, f15);
  yl1 = VEC_ADD(f12, f16);
  yl6 = VEC_ADD(f9, f14);
  yl7 = VEC_SUB(f10, f13);
  yl2 = VEC_SUB(f9, f14);
  STORE_L_16_4(y + 0, y + 8, y + 16, y + 24, yl0, yl2, yl4, yl6);
  yl3 = VEC_ADD(f10, f13);
  STORE_L_16_4(y + 4, y + 12, y + 20, y + 28, yl1, yl3, yl5, yl7);
}
```

```
SIMD_FUNC DFT_16_0(simd_real *y, simd_real *x)
{
  simd_vector xl0;
  ...
  simd_vector xl7;
  simd_vector yl0;
  ...
  simd_vector yl7;
  simd_vector ldtmp0;
  ...
  simd_vector ldtmp7;
  simd_vector f9;
  ...
  simd_vector f16;
  __assume_aligned(x, 16);
  __assume_aligned(y, 16);

  LOAD_VECT(xl0, x + 0);
  LOAD_VECT(xl1, x + 4);
  LOAD_VECT(ldtmp4, x + 16);
  LOAD_VECT(ldtmp5, x + 20);
  COMPLEX_MULT(xl4, xl5, ldtmp4, ldtmp5,
               LOAD_CONST(SCONST4), LOAD_CONST(SCONST5));
  f9 = VEC_SUB(xl0, xl4);
  f10 = VEC_SUB(xl1, xl5);
  f11 = VEC_ADD(xl0, xl4);
  f12 = VEC_ADD(xl1, xl5);
  LOAD_VECT(ldtmp2, x + 8);
  LOAD_VECT(ldtmp3, x + 12);
  COMPLEX_MULT(xl2, xl3, ldtmp2, ldtmp3,
               LOAD_CONST(SCONST2), LOAD_CONST(SCONST3));
  LOAD_VECT(ldtmp6, x + 24);
  LOAD_VECT(ldtmp7, x + 28);
  COMPLEX_MULT(xl6, xl7, ldtmp6, ldtmp7,
               LOAD_CONST(SCONST6), LOAD_CONST(SCONST7));
  f13 = VEC_SUB(xl2, xl6);
  f14 = VEC_SUB(xl3, xl7);
  f15 = VEC_ADD(xl2, xl6);
  f16 = VEC_ADD(xl3, xl7);
  yl4 = VEC_SUB(f11, f15);
  yl5 = VEC_SUB(f12, f16);
  STORE_L_8_4(yl4, yl5, y + 16, y + 20);
  yl0 = VEC_ADD(f11, f15);
  yl1 = VEC_ADD(f12, f16);
  STORE_L_8_4(yl0, yl1, y + 0, y + 4);
  yl6 = VEC_ADD(f9, f14);
  yl7 = VEC_SUB(f10, f13);
  STORE_L_8_4(yl6, yl7, y + 24, y + 28);
  yl2 = VEC_SUB(f9, f14);
  yl3 = VEC_ADD(f10, f13);
  STORE_L_8_4(yl2, yl3, y + 8, y + 12);
}

void DFT_16(y,x)
simd_scalar_float *restrict y, *restrict x;
{
  static simd_vector t53[8];
  __assume_aligned(x, 16);
  __assume_aligned(y, 16);

  DFT_16_1((simd_vector *)t53 + 0, x + 0);
  DFT_16_0((simd_vector *)y + 0, t53 + 0);
}
```

Appendix E

FFTW Example

This appendix displays the Fftw framework in pseudo code as well as a scalar and the respective short vector SIMD no-twiddle codelet of size four. The short-vector codelet implements a part of rule (7.25) which belongs to the short-vector Cooley-Tukey rule set (Theorem 7.5 on page 124). This is the smallest no-twiddle that can be used for implementing this rule.

E.1 The FFTW Framework

```
main(n, int, out)
{
  complex in[n], out[n]
  plan p

  p:=planner(n)

  fftw(in, out, p);
}


fftw(input, output, plan)
{
  execute(plan.n, input, output, plan, 1, 1)
  return output
}


execute(n, input, output, plan, instride, outstride)
{
  if plan.type = NO_TWIDDLE
    plan.notwiddlecodelet(input, output, instride, outstride)
  if plan.type = TWIDDLE
    r:=plan.twiddlesize
    m:=n/r
    for i:=0 to r-1 do
      execute(m, input+i*instride, output+i*m*outstride,
        plan.nextstep, instride*r, outstride)
    plan.twiddlecodelet(output, plan.twiddlefactors,
      m * outstride, m, outstride);
}
```

```
notwiddlecodelet<N>(input, output, instride, outstride)
{
  complex in[N], out[N]

  for i:=0 to N-1 do
    in[i]:=input[i*instride]

  FFT(N, in, out)

  for i:=0 to N-1 do
    output[i*outstride]:=out[i]
  return output
}
```

```
twiddlecodelet<N>(inoutput, twiddlefactors, stride,  m, dist)
{
  complex in[N], out[N]

  for i:=0 to m-1
    for j:=0 to N-1
      in[i]:=inoutput[i*dist+j*stride]

    apply_twiddlefactors(N, out, i, twiddlefactors)
    FFT(N, in, out)

    for j:=0 to N-1
      inoutput[i*dist+j*stride]:=out[j]
}
```

E.2 Scalar C Code

This section shows a standard Fftw no-twiddle codelet of size 4. The computation done by this codelet is

$$\mathbf{update}\left(y', \overline{\mathrm{W}^{N,4}_{\text{output, ostride}}}, \overline{\mathrm{DFT}_4\, \mathrm{R}^{N,4}_{\text{input, istride}} x'}\right).$$

```
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);
  }
```

E.3 Short Vector Code

This section shows a four-way SIMD vectorized FFTW no-twiddle codelet of size 4. The computation done by this short vector SIMD codelet is

$$\mathbf{update}\Bigg(y',\quad \mathrm{W}^{N/4,4}_{\mathrm{output}/8,\,\mathrm{ostride}/8} \otimes \mathrm{I}_8,\\ \left(\mathrm{L}^8_4 \otimes \mathrm{I}_4\right)\left(\mathrm{I}_2 \otimes \mathrm{L}^{16}_4\right)\left(\mathrm{L}^8_2 \otimes \mathrm{I}_4\right)\left(\overline{\mathrm{DFT}_4} \otimes \mathrm{I}_4\right)\\ \left(\mathrm{R}^{N/4,4}_{\mathrm{input}/8,\,\mathrm{istride}/8} \otimes \mathrm{L}^8_2\right) x'\Bigg).$$

```
#include "codelet.h"
#include "simd_api.h"

void fftw_no_twiddle_4_simd (const simd_vector * in,
    const simd_vector * out, int is, int os)
{
  simd_vector tmp3;
  simd_vector tmp11;
  simd_vector tmp9;
  simd_vector tmp15;
  simd_vector tmp6;
  simd_vector tmp10;
  simd_vector tmp14;
  simd_vector tmp16;
  {
    simd_vector tmp1;
    simd_vector tmp2;
    simd_vector tmp7;
    simd_vector tmp8;
    LOAD_L_8_2(tmp1, tmp7, in + 0, in + 4);
    LOAD_L_8_2(tmp2, tmp8, in + 2 * is + 9, in + 2 * is + 4);
    tmp3 = ADD_VEC (tmp1, tmp2);
    tmp11 = SUB_VEC (tmp1, tmp2);
    tmp9 = SUB_VEC (tmp7, tmp8);
    tmp15 = ADD_VEC (tmp7, tmp8);
  }
  {
    simd_vector tmp4;
    simd_vector tmp5;
    simd_vector tmp12;
    simd_vector tmp13;
    LOAD_L_8_2(tmp4, tmp12, in + 1 * is + 0, in + 1 * is + 4):
    LOAD_L_8_2(tmp5, tmp13, in + 3 * is + 0, in + 3 * is + 4):
    tmp6 = ADD_VEC (tmp4, tmp5);
    tmp10 = SUB_VEC (tmp4, tmp5);
    tmp14 = SUB_VEC (tmp12, tmp13);
    tmp16 = ADD_VEC (tmp12, tmp13);
  }
  {
    simd_vector tmp17;
    simd_vector tmp23;
    simd_vector tmp18;
    simd_vector tmp24;
    simd_vector tmp19;
    simd_vector tmp22;
    simd_vector tmp20;
    simd_vector tmp21;
    tmp17 = SUB_VEC (tmp3, tmp6);
    tmp23 = SUB_VEC (tmp15, tmp16);
    tmp18 = ADD_VEC (tmp3, tmp6);
    tmp24 = ADD_VEC (tmp15, tmp16);
    tmp19 = SUB_VEC (tmp9, tmp10);
    tmp22 = ADD_VEC (tmp11, tmp14);
    tmp20 = ADD_VEC (tmp10, tmp9);
    STORE_L_16_4 (out, out + os, out + 2 * os, out + 3 * os,
          tmp24, tmp19, tmp23, tmp20);
    tmp21 = SUB_VECT (tmp11, tmp14);
    STORE_L_16_4 (out + 4, out + 4 + os, out + 4 + 2 * os,
          out + 4 + 3 * os, tmp18, tmp22, tmp17, tmp21);
  }
}
```

Table of Abbreviations

API	Application programming interface
AEOS	Automatical empirical optimization of software
CISC	Complex instruction set computer
CPI	Cycles per instruction
CPU	Central processing unit
DCT	Discrete cosine transform
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
PMC	Performance monitor counter
RAM	Random access memory
ROM	Read-only memory
RISC	Reduced instruction set computer
SIMD	Single instruction, multiple data
SPL	Signal processing language
SRAM	Static random access memory
SSE	Streaming SIMD extension
TLB	Transaction lookaside buffer
VLIW	Very long instruction word
WHT	Walsh-Hadamard transform

Bibliography

[1] A. V. Aho, R. Sethi, J. D. Ullman: *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., 1986.

[2] AMD Corporation: *3DNow! Technology Manual*, 2000.

[3] AMD Corporation: *3DNow! Instruction Porting Guide Application Note*, 2002.

[4] AMD Corporation: *AMD Athlon Processor x86 Code Optimization Guide*, 2002.

[5] AMD Corporation: *AMD Extensions to the 3DNow! and MMX Instruction Sets Manual*, 2002.

[6] AMD Corporation: *AMD x86-64 Architecture Programmers Manual, Volume 1: Application Programming*. Order Number 24592, 2002.

[7] AMD Corporation: *AMD x86-64 Architecture Programmers Manual, Volume 2: System Programming*. Order Number 24593, 2002.

[8] Apple Computer: *vDSP Library*, 2001.
`http://developer.apple.com/`

[9] M. Auer, R. Benedik, F. Franchetti, H. Karner, P. Kristöfel, R. Schachinger, A. Slateff, C. Ueberhuber: *Performance Evaluation of FFT Routines—Machine Independent Serial Programs*. Technical Report AURORA TR1999-05, Institute for Applied Mathematics and Numerical Analysis, Vienna University of Technology, 1999.

[10] D. H. Bailey: *FFTs in External or Hierarchical Memory*. J. Supercomputing 4 (1990), pp. 23–35.

[11] R. Berrendorf, H. Ziegler: PCL, 2002.
`http://www.fz-juelich.de/zam/PCL`

[12] 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.

[13] J. W. Cooley, J. W. Tukey: *An Algorithm for the Machine Calculation of Complex Fourier Series*. Math. Comp. 19 (1965), pp. 297–301.

[14] T. Cormen, C. Leiserson, R. Rivest: *Introduction to Algorithms*. The MIT Press, Cambridge, Massatschusetts, 1990.

[15] R. Crandall, J. Klivington: *Supercomputer-Style FFT Library for the Apple G4*. Advanced Computation Group, Apple Computer, 2002.

[16] Digital Equipment Corporation: PFM—*The 21064 Performance Counter Pseudo-Device*. DEC OSF/1 Manual pages, 1995.

[17] 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.

[18] 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.

[19] P. Duhamel: *Implementation of Split-Radix FFT Algorithms for Complex, Real, and Real-Symmetric Data*. IEEE Trans. Acoust. Speech Signal Processing 34 (1986), pp. 285–295.

[20] P. Duhamel: *Algorithms Meeting the Lower Bounds on the Multipliative Complexity of Length-2^n DFTs and Their Connection with Practical Algorithms*. IEEE Trans. Acoust. Speech Signal Processing 38 (1990), pp. 1504–1511.

[21] P. Duhamel, H. Hollmann: *Split Radix FFT Algorithms*. Electron. Lett. 20 (1984), pp. 14–16.

[22] S. Egner, M. Püeschel: *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: *A Portable Short Vector Version of* FFTW. Proc. of the MATHMOD 2003, *to appear*.

[25] F. Franchetti, H. Karner, S. Kral, C.W. Ueberhuber: *Architecture Independent Short Vector FFTs*. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP'01)*, IEEE Press, New York, Vol. 2, pp. 1109–1112.

[26] F. Franchetti, J. Lorenz, C. Ueberhuber: *Low Communication FFTs*. Technical Report AURORA TR2002-27, Institute for Applied Mathematics and Numerical Analysis, Vienna University of Technology, 2002.

[27] F. Franchetti, M. Püschel: *A SIMD Vectorizing Compiler for Digital Signal Processing Algorithms.* In *Proc. International Parallel and Distributed Processing Symposium (IPDPS'02).*

[28] F. Franchetti, M. Püschel: *Short Vector Code Generation and Adaptation for DSP Algorithms.* Proceedings of the International Conerence on Acoustics, Speech, and Signal Processing. Conference Proceedings (ICASSP'03), *to appear.*

[29] F. Franchetti, M. Püschel: *Short Vector Code Generation for the Discrete Fourier Transform.* Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS'03), *to appear.*

[30] F. Franchetti, M. Püschel, J. Moura, C. Ueberhuber: *Short Vector SIMD Code Generation for DSP Algorithms.* In *Proc. of the 2002 High Performance Embedded Computing Conference*, MIT Lincoln Laboratory.

[31] 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, 1999, pp. 169–180.

[32] M. Frigo, S. G. Johnson: *The Fastest Fourier Transform in the West.* Technical Report MIT-LCS-TR-728, MIT Laboratory for Computer Science, Cambridge, 1997.

[33] M. Frigo, S. G. Johnson: FFTW*: An Adaptive Software Architecture for the FFT.* In *ICASSP 98*, Vol. 3, pp. 1381–1384.

[34] M. Frigo, C. E. Leiserson, H. Prokop, S. Ramachandran: *Cache-Oblivious Algorithms.* In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS'99)*, IEEE Comp. Society Press, Los Alamitos.

[35] M. Galles, E. Williams: *Performance Optimizations, Implementation, and Verification of the SGI Challenge Multiprocessor.* Proceedings of the 27th Annual Hawaii International Conference on System Sciences, 1994.

[36] W. N. Gansterer, C. W. Ueberhuber: *Hochleistungsrechnen mit HPF.* Springer-Verlag, Berlin Heidelberg New York Tokyo, 2001.

[37] 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, pp. 249–260.

[38] G. H. Golub, C. F. Van Loan: *Matrix Computations*, 2nd edn. Johns Hopkins University Press, Baltimore, 1989.

[39] 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.

[40] G. Haentjens: *An Investigation of Cooley-Tukey Decompositions for the FFT*. Masters Thesis, Electrical and Computer Engineering Department, Carnegie Mellon University, Pittsburgh, PA, 2000.

[41] H. Hlavacs, C. W. Ueberhuber: *High-Performance Computers – Hardware, Software and Performance Simulation.* SCS-Europe, Ghent, 2002.

[42] D. Hunt: *Advanced Performance Features of the 64-bit PA8000.* COMPCON'95, 1995.

[43] Intel Corporation: *Survey of Pentium Processor Performance Monitoring Capabilities & Tools.* App. Note, 1996.

[44] Intel Corporation: *AP-808 Split Radix Fast Fourier Transform Using Streaming SIMD Extensions*, 1999.

[45] Intel Corporation: *AP-833 Data Alignment and Programming Issues for the Streaming SIMD Extensions with the Intel C/C++ Compiler*, 1999.

[46] Intel Corporation: *AP-931 Streaming SIMD Extensions—LU Decomposition*, 1999.

[47] Intel Corporation: *Intel Architecture Optimization—Reference Manual*, 1999.

[48] Intel Corporation: *Desktop Performance and Optimization for Intel Pentium 4 Processor*, 2002.

[49] Intel Corporation: *Intel Architecture Software Developer's Manual—Volume 1: Basic Architecture*, 2002.

[50] Intel Corporation: *Intel Architecture Software Developer's Manual—Volume 2: Instruction Set Reference*, 2002.

[51] Intel Corporation: *Intel Architecture Software Developer's Manual—Volume 3: System Programming*, 2002.

[52] Intel Corporation: *Intel C/C++ Compiler User's Guide*, 2002.

[53] Intel Corporation: *Intel Itanium 2 Processor Reference Manual for Software Development and Optimization*, 2002.

[54] Intel Corporation: *Intel Itanium Architecture Software Developer's Manual Vol. 1 rev. 2.1: Application Architecture*, 2002.

[55] Intel Corporation: *Intel Itanium Architecture Software Developer's Manual Vol. 2 rev. 2.1: System Architecture*, 2002.

[56] Intel Corporation: *Intel Itanium Architecture Software Developer's Manual Vol. 3 rev. 2.1: Instruction Set Reference*, 2002.

[57] Intel Corporation: *Intel Itanium Processor Reference Manual for Software Optimization*, 2002.

[58] Intel Corporation: *Math Kernel Library*, 2002.
`http://www.intel.com/software/products/mkl`

[59] 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.*

[60] J. R. Johnson, R. W. Johnson, D. Rodriguez, R. Tolimieri: *A Methodology for Designing, Modifying, and Implementing FFT Algorithms on Various Architectures.* Circuits Systems Signal Process. 9 (1990), pp. 449–500.

[61] R. W. Johnson, C. H. Huang, J. R. Johnson: *Multilinear Algebra and Parallel Programming.* J. Supercomputing 5 (1991), pp. 189–217.

[62] S. Joshi, P. Dubey: *Radix-4 FFT Implementation Using SIMD Multi-Media Instructions.* In *Proc. ICASSP 99*, pp. 2131–2135.

[63] H. Karner, C. Ueberhuber: *Overlapped Four-Step FFT Computation, in "Parallel Computation.* In *Lecture Notes in Computer Science* (P. Zinterhof, M. Vajtersic, A. Uhl, eds.), Springer-Verlag, Berlin Heidelberg New York Tokyo, Vol. 1557, 1999, pp. 590–591.

[64] S. Kral: FFTW-GEL, 2002.
`www.fftw.org/ skral`

[65] S. Lamson: SCIPORT, 1995.
`http://www.netlib.org/scilib/`

[66] X. Leroy: *The Objective Caml System, Release 3.06—Documentation and User Manual*, 2002.

[67] T. Mathisen: *Pentium Secrets.* Byte 7 (1994), pp. 191–192.

[68] MIPS Technologies Inc.: *R10000 Microprocessor Technical Brief*, 1994.

[69] MIPS Technologies Inc.: *Definition of MIPS R10000 Performance Counter*, 1997.

[70] Motorola Corporation: *AltiVec Technology Programming Environments Manual*, 2000.

[71] Motorola Corporation: *AltiVec Technology Programming Interface Manual*, 2000.

[72] 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.

[73] P. Mucci, S. Browne, G. Ho, C. Deane: PAPI, 2002.
`http://icl.cs.utk.edu/projects/papi`

[74] I. Nicholson: LIBSIMD, 2002.
`http://libsimd.sourceforge.net/`

[75] 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.

[76] M. C. Pease: *An Adaptation of the Fast Fourier Transform for Parallel Processing*. Journal of the ACM 15 (1968), pp. 252–264.

[77] N. P. Pitsianis: *The Kronecker Product in Optimization and Fast Transform Generation*. Ph. D. Thesis, Department of Computer Science, Cornell University, 1997.

[78] M. Püschel: *Decomposing Monomial Representations of Solvable Groups*. Journal of Symbolic Computation, *to appear*.

[79] M. Püschel, B. Singer, M. Veloso, J. M. F. Moura: *Fast Automatic Generation of DSP Algorithms*. In *Proc. ICCS 2001*, Springer, LNCS 2073, pp. 97–106.

[80] 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*. Journal of High Performance Computing and Applications (2001), *submitted*.

[81] K. R. Rao, J. J. Hwang: *Techniques & standards for image, video and audio coding*. Prentice Hall PTR, 1996.

[82] P. Rodriguez: *A Radix-2 FFT Algorithm for Modern Single Instruction Multiple Data (SIMD) Architectures*. In *Proc. ICASSP 2002*.

[83] B. Ryan: *PowerPC 604 weighs in.* BYTE 6 (1994), pp. 265–266.

[84] B. Singer, M. Veloso: *Stochastic Search for Signal Processing Algorithm Optimization.* In *Proc. Supercomputing 01.*

[85] H. V. Sorensen, M. T. Heideman, C. S. Burrus: *On Computing the Split-Radix FFT.* IEEE Trans. Acoust. Speech Signal Processing 34 (1986), pp. 152–156.

[86] P. N. Swarztrauber: *FFT Algorithms for Vector Computers.* Parallel Comput. 1 (1984), pp. 45–63.

[87] C. Temperton: *Fast Mixed-Radix Real Fourier Transforms.* J. Comput. Phys. 52 (1983), pp. 340–350.

[88] The GAP Group: GAP—*Groups, Algorithms, and Programming, Version 4.2*, 2000.
`www-gap.dcs.st-and.ac.uk/~gap`

[89] C. W. Ueberhuber: *Numerical Computation.* Springer-Verlag, Berlin Heidelberg New York Tokyo, 1997.

[90] C. F. Van Loan: *Computational Frameworks for the Fast Fourier Transform.* SIAM Press, Philadelphia, 1992.

[91] M. Vetterli, P. Duhamel: *Split-Radix Algorithms for Length-p^m DFT's.* IEEE Trans. Acoust. Speech Signal Processing 37 (1989), pp. 57–64.

[92] 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.

[93] 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.

[94] R. C. Whaley, A. Petitet, J. J. Dongarra: *Automated Empirical Optimizations of Software and the* ATLAS *Project.* Parallel Comput. 27 (2001), pp. 3–35.

[95] J. Xiong, J. R. Johnson, R. W. Johnson, D. Padua: *SPL: A Language and Compiler for DSP Algorithms.* In *Proc. PLDI 01*, pp. 298–308.

[96] M. Zagha, B. Larson, S. Turner, M. Itzkowitz: *Performance Analysis Using the MIPS R10000 Performance Counters.* Proceedings Supercomputing'96, IEEE Computer Society Press, 1996.

CURRICULUM VITAE

Name: Franz R. Franchetti

Title: Dipl.-Ing.

Date and Place of Birth: 18 September 1974, Mödling, Lower Austria

Nationality: Austria

Home Address: Hartiggasse 3/602, A-2700 Wiener Neustadt, Austria

Affiliation

Institute for Applied Mathematics and Numerical Analysis
Vienna University of Technology
Wiedner Hauptstrasse 8-10/1152, A-1040 Vienna
Phone: +43 1 58801 11524
Fax: +43 1 58801 11599
E-mail: franz.franchetti@tuwien.ac.at

Education

1994	High School Diploma (*Matura*) with distinction
1995 – 2000	Studies in Technical Mathematics at the Vienna University of Technology
2000	Dipl.-Ing. (Technical Mathematics) with distinction at the Vienna University of Technology
2000 – 2003	Ph.D. studies

Employment

1994 – 2002	Part time system administrator with Zentraplan HKL GesmbH in Wr. Neustadt
1998, 1999	Summer internships with SIEMENS AG (PSE) in Vienna
1997 –	Part time system administrator at the Vienna University of Technology
2000 –	Research Assistant at the Institute for Applied Math. and Numerical Analysis, funded by the SFB Aurora

Project Experience

1997 – Participation in the SFB Aurora

Societies and Activities

Member, Association for Computing Machinery (ACM)

Member, Society for Industrial and Applied Mathematics (SIAM)

Scientific Cooperations

SFB Aurora, Austria

Carnegie Mellon University, USA

Massachusetts Institute of Technology, USA

Drexel University, USA

IBM T.J. Watson Research Center

Lawrence Livermore National Laboratory

Awards

OCG-Förderpreis 2001

Refereed Publications

1. F. Franchetti: *A Portable Short Vector Version of* Fftw. Proc. of the MATHMOD 2003, *to appear.*
2. F. Franchetti, M. Püschel: *Short Vector Code Generation and Adaptation for DSP Algorithms.* In *Proceedings of the International Conerence on Acoustics, Speech, and Signal Processing. Conference Proceedings (ICASSP '03)*, IEEE Comput. Soc. Press, Los Alamitos, USA, 2003, *to appear.*
3. 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)*, IEEE Comp. Society Press, Los Alamitos, CA, 2003, *to appear.*
4. T. Fahringer, F. Franchetti, M. Geissler, G. Madsen, H. Moritsch, R. Prodan: *On Using* Zenturio *for Performance and Parameter Studies on Clusters and Grids.* In *Proceedings of the 11th Euromicro Conference on Parallel Distributed and Network based Processing*, IEEE Comput. Soc. Press, Los Alamitos, USA, 2003, *to appear.*

5. F. Franchetti, F. Kaltenberger, C. W. Ueberhuber: *FFT Kernels with FMA Utilization.* In *Proceedings of the* APLIMAT *2002 Conference*, 2002, *to appear.*

6. F. Franchetti, M. Püschel, J. Moura, C. W. Ueberhuber: *Short Vector SIMD Code Generation for DSP Algorithms.* In *Proc. of the 2002 High Performance Embedded Computing*, MIT Lincoln Laboratory.

7. F. Franchetti, M. Püschel: *A SIMD Vectorizing Compiler for Digital Signal Processing Algorithms.* In *Proc. International Parallel and Distributed Processing Symposium (IPDPS'02)*, 2002.

8. F. Franchetti, H. Karner, S. Kral, C. W. Ueberhuber: *Architecture Independent Short Vector FFTs.* Proceedings of ICASSP2001—International Conference on Acoustics, Speech, and Signal Processing 2001, vol. 2, pp. 1109–1112.

Others

1. F. Franchetti, C. Ortner, C. W. Ueberhuber: *Reif für die Insel?—Wie weit ist Java?* ZIDline 7, Vienna University of Technology, 2002.

2. F. Franchetti, M. Püschel: *Top Performance SIMD FFTs.* AURORA Technical Report TR2002-31, Institute for Applied Mathematics and Numerical Analysis, Vienna University of Technology, 2002.

3. F. Franchetti, G. Langs, C. W. Ueberhuber, W. Weidinger: *Approximative 1D and 2D FFTs.* AURORA Technical Report TR2002-30, 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 TR2002-29, Institute for Applied Mathematics and Numerical Analysis, Vienna University of Technology, 2002.

5. F. Franchetti, F. Kaltenberger, C. W. Ueberhuber: *FMA Optimized FFT Codelets.* AURORA Technical Report TR2002-28, Institute for Applied Mathematics and Numerical Analysis, Vienna University of Technology, 2002.

6. 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.

7. T. Fahringer, F. Franchetti, M. Geissler, G. Madsen, H. Moritsch, R. Prodan: *On Using* ZENTURIO *for Performance and Parameter Studies on Clusters and Grids.* Tech Report AURORA TR2002-20, Institute for Applied Mathematics and Numerical Analysis, Vienna University of Technology, 2002.

8. F.Franchetti, F. Kaltenberger, C. W. Ueberhuber: *Dimensionless FFT Algorithms.* Tech Report AURORA TR2002-12, Institute for Applied Mathematics and Numerical Analysis, Vienna University of Technology, 2002.

9. F. Franchetti, F. Kaltenberger, C. W. Ueberhuber: *Reduced Transform Algorithms for Multidimensional Fourier Transforms.* Tech Report Aurora TR2002-09, Institute for Applied Mathematics and Numerical Analysis, Vienna University of Technology, 2002.

10. F. Franchetti, C. Ortner, C. W.Ueberhuber: *Performance of Linear Algebra Routines in Java and C.* Tech Report Aurora TR2002-06, Institute for Applied Mathematics and Numerical Analysis, Vienna University of Technology, 2002.

11. C. Fabianek, F. Franchetti, M. Frigo, H. Karner, L. Meirer, C. W. Ueberhuber: *Survey of Self-adapting FFT Software.* Tech Report Aurora TR2002-01, Institute for Applied Mathematics and Numerical Analysis, Vienna University of Technology, 2001.

12. F. Franchetti, C. Ortner, C. W. Ueberhuber: *Java vs. C—A Performance Assessment Based on FFT Algorithms.* Institute for Applied Mathematics and Numerical Analysis, Vienna University of Technology, 2001.

13. F. Franchetti, M. Püschel: *Automatic Generation of SIMD DSP Code.* Tech Report Aurora TR2001-17, Institute for Applied Mathematics and Numerical Analysis, Vienna University of Technology, 2001.

14. M. Auer, R. Benedik, F. Franchetti, H. Karner, P. Kristoefel, R. Schachinger, A. Slateff, C. W. Ueberhuber: *Performance Evaluation of FFT Routines — Machine Independent Serial Programs.* Tech. Report Aurora TR1999-05, Institute for Applied Mathematics and Numerical Analysis, Vienna University of Technology, 1999.

15. M. Auer, F. Franchetti, H. Karner, C. W. Ueberhuber: *Performance Evaluation of FFT Algorithms Using Performance Counters.* Tech. Report Aurora TR1998-12, Institute for Applied Mathematics and Numerical Analysis, Vienna University of Technology, 1998.