DIPLOMARBEIT

Utilization of SIMD Extensions for Numerical Straight Line Code

ausgeführt am Institut für
Angewandte und Numerische Mathematik
der Technischen Universität Wien

unter der Anleitung von
Prof. Dr. Christoph Überhuber

durch

Peter Wurzinger
Matrikelnummer: 9825672
Muhrengasse 11/10
1100 Wien.

Wien, 11. Dezember 2003 ________________

Vorwort

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

Die immer weiter fortschreitende Zunahme der Bedeutung von Multimedia-Anwendungen hat zur Entwicklung von SIMD-Befehlssatzerweiterungen auf allen gängigen Mikroprozessoren geführt. Es gibt auch entsprechende vektorisierende Compiler, welche die neu hinzugekommene Funktionalität ausnutzen sollen. Deren konventionelle Vektorisierungsmethoden sind jedoch nicht in der Lage, Parallelismus in einem längeren numerischen Straight-Line-Code zu erkennen.

In dieser Diplomarbeit werden Methoden zur automatischen Unterstützung von SIMD-Befehlssatzerweiterungen in Verbindung mit numerischer Hochleistungs-Software vorgestellt. Die beschriebene Compiler-Technologie erreicht die Leistung handoptimierter Libraryroutinen, unterstützt eine große Klasse numerischer Algorithmen der linearen Algebra und der digitalen Signalverarbeitung und schlägt handelsübliche vektorisierende Compiler bei weitem.

Die in dieser Diplomarbeit präsentierten Konzepte stellen die Grundlage des *Special Purpose Compilers MAP* dar, der speziell für die effiziente Vektorisierung großer numerischer Straight-Line-Codes entwickelt wurde. MAP unterstützt die führenden automatischen Performance-Tuning-Softwareprodukte Fftw, Spiral und Atlas. Damit werden die wichtigsten numerischen Algorithmen der linearen Algebra und der digitalen Signalverarbeitung abgedeckt.

Preface

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

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

This thesis provides support for utilizing SIMD extensions in connection with self-tuning numerical software to combine the power of modern hardware design with modern software technology. The described compiler technology (*i*) achieves the superior performance of hand tuned vendor libraries utilizing SIMD extensions, (*ii*) supports a large class of numerical kernels including the most important algorithms from numerical linear algebra and digital signal processing, and (*iii*) produces code that is much more efficient than code obtained with common vectorizing compilers. Such compilers are not able to utilize SIMD extensions well, neither in conjunction with automatically tuning numerical software, nor when compiling standard code (not produced by adaptive code generators).

The presented compiler techniques form the basis of the special purpose compiler MAP, which was designed especially for the efficient vectorization of large basic blocks of numerical straight line code. MAP currently supports the leading edge self-tuning numerical software products FFTW, SPIRAL, and ATLAS, thus, covering a broad range of the most relevant numerical algorithms from DSP and linear algebra.

Acknowledgements

First of all, I would like to express my gratitude to my advisor Christoph Ueberhuber, who gave me the opportunity of working in this interesting field of research and supported my efforts with great dedication, not only during the work on my thesis, but throughout several years of my studies.

Franz Franchetti deserves great appreciation for his support and guidance throughout my work. His ideas and comments were invaluable.

Very special thanks go to my colleague and friend Jürgen Lorenz, with whom I have spent countless rewarding programming afternoons.

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

Above all, I want to thank my family for having provided a background that allowed me to focus on my education. Their support can not be appreciated enough.

Contents

1 Introduction 1
1.1 The Fast Fourier Transform 2
1.2 Automatic Performance Tuning Software 4

2 Short Vector SIMD Extensions 10

3 Abstraction of SIMD Instructions 18

4 Automatic Vectorization 22
4.1 Vectorization of Straight Line Code 22
4.2 The Vectorization Approach 22
4.3 Virtual Machine Models 23
4.4 The Vectorization Engine 28
4.5 Pairing Rules 35

5 Rewriting and Optimization 46
5.1 Optimization of Vectorized Code 46
5.2 Peephole Optimization 47
5.3 Transformation Rules 48
5.4 The Scheduler 58

6 Performance Assessment of MAP 60
6.1 The Vienna MAP Vectorizer and Backend 60
6.2 Experimental Setup 61
6.3 BlueGene/L Experiments 61
6.4 Experiments on IA-32 Architectures 63

A The BlueGene/L Architecture 70

B The Kronecker Product Formalism 71
B.1 Notation 72
B.2 Extended Subvector Operations 75
B.3 Kronecker Products 77
B.4 Stride Permutations 81
B.5 Twiddle Factors and Diagonal Matrices 85
B.6 Kronecker Product Code Generation 86
B.7 Fast Algorithms for Discrete Linear Transforms 94

Chapter 1

Introduction

Computer architects have long recognized that SIMD (single instruction, multiple data) computation can be an effective way to achieve high performance. The fundamental idea behind SIMD architectures is that a single instruction operates on many data elements at once using many functional units, possibly located on many different processors. Recently microprocessor vendors have applied the idea of SIMD computation to instruction set extensions and processor organizations specifically designed to improve the performance of multimedia applications on a *single processor*. This leads to short vector SIMD operations. Examples of such multimedia extensions are Intel's SSE, AMD's 3Dnow!, and Motorola's Altivec.

A significant obstacle to the application of short vector SIMD instructions in digital signal processing or scientific computing has been that compilers do not routinely generate code that delivers performance comparable to that of carefully tuned, hand-coded parallel implementations. On the other hand, not using short vector SIMD extensions wastes 50 % to 75 % of a processor's capabilities.

Extensive experience shows that in numerical linear algebra and in digital signal processing high performance is achieved by either automatic performance tuning or extremely costly hand optimization. Apart from automatic performance tuning (currently not fully supporting short vector SIMD extensions), no performance portable numerical software exists. Moreover, all high performance numerical vendor libraries like MKL, ESSL, or vDSP are hand-coded to utilize SIMD extensions well and must be adapted to each new hardware generation by hand, thus not being performance portable. This shows the need of short vector SIMD support in self-tuning numerical software.

This thesis presents compiler technology for automatically vectorizing numerical computation blocks of straight line code as generated by the automatic performance tuning systems Fftw, Spiral, and Atlas. The newly introduced concepts are realized in the *Vienna MAP* compiler, which is capable of generating SIMD vectorized code for digital signal processing algorithms like FFTs, DCTs and DSTs, as well as matrix multiplication. MAP features a symbolic vectorization engine, which is the main topic of this thesis, as well as a backend that directly outputs assembler code (see [47] and [48]).

MAP achieves the same level of performance as hand-tuned vendor libraries while providing performance portability. Combined with leading-edge self-tuning numerical software—Fftw, Spiral, and Atlas—it produces

- the fastest FFTs running on x86 architecture machines (Intel and AMD processors) for both real and complex FFTs and for arbitrary vector size;
- the only FFT routines supporting IBM's Power PC 440 FP2 double FPU used in BlueGene/L machines (see [16] and Appendix A);
- the only automatically tuned vectorized implementations of DSP transforms (including discrete sine and cosine transforms, Walsh-Hadamard transforms, and multidimensional transforms);
- the only fully automatically vectorized ATLAS kernels.

1.1 The Fast Fourier Transform

The discrete Fourier transform (DFT) is one of the most important tools in modern engineering and therefore one of the main targets of the MAP compiler. Its algorithm was continuously optimized over the years. The key improvement, which reduced computation time and costs dramatically, was to exploit the intrinsic symmetry of the DFT matrix. The breakthrough algorithm of Cooley-Tukey enabled a reduction of the number of operations to be carried out from $2N^2$ to const$\times N \log N$—the constant varying between 3 and 5 depending on the specific variant of the algorithm.

N	N^2	$N \log_2 N$	DFT (sec)	FFT (sec)	Speed-up
4	1.60×10^{1}	8.00×10^{0}	1.60×10^{-8}	8.00×10^{-9}	2
16	2.56×10^{2}	6.40×10^{1}	2.56×10^{-7}	6.40×10^{-8}	4
64	4.10×10^{3}	3.84×10^{2}	4.07×10^{-6}	3.84×10^{-7}	11
256	6.55×10^{4}	2.05×10^{3}	6.55×10^{-5}	2.05×10^{-6}	32
1,024	1.05×10^{6}	1.02×10^{4}	1.05×10^{-3}	1.02×10^{-5}	102
4,096	1.68×10^{7}	4.92×10^{4}	1.68×10^{-2}	4.92×10^{-5}	341
16,384	2.68×10^{8}	2.29×10^{5}	2.68×10^{-1}	2.29×10^{-4}	1,170
65,536	4.29×10^{9}	1.05×10^{6}	4.30×10^{0}	1.05×10^{-3}	4,096
262,144	6.87×10^{10}	4.72×10^{6}	6.87×10^{1}	4.72×10^{-3}	14,564
1,048,576	1.10×10^{12}	2.10×10^{7}	1.10×10^{3}	2.10×10^{-2}	52,429
4,194,304	1.76×10^{13}	9.23×10^{7}	1.76×10^{4}	9.23×10^{-2}	190,650

Table 1.1: Theoretical execution times of the "classical" DFT and the FFT on a 1 GHz processor running at peak performance.

In applications where the DFT has to be performed many times (for example, in meteorology) and in applications where large transformation lengths are needed (especially in seismic applications) the speed-up achieved by using an FFT algorithm is invaluable (see Tab. 1.1).

The key idea behind the Cooley-Tukey algorithm is to use the divide and conquer paradigm. This idea can be explained by means of the 8×8 DFT matrix

$$F_8 = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \omega^4 & \omega^5 & \omega^6 & \omega^7 \\ 1 & \omega^2 & \omega^4 & \omega^6 & 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^3 & \omega^6 & \omega & \omega^4 & \omega^7 & \omega^2 & \omega^5 \\ 1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 \\ 1 & \omega^5 & \omega^2 & \omega^7 & \omega^4 & \omega & \omega^6 & \omega^3 \\ 1 & \omega^6 & \omega^4 & \omega^2 & 1 & \omega^6 & \omega^4 & \omega^2 \\ 1 & \omega^7 & \omega^6 & \omega^5 & \omega^4 & \omega^3 & \omega^2 & \omega \end{pmatrix},$$

where $\omega := \omega_8 = e^{-2\pi i/8}$.

Any matrix F_N, N even, can be rearranged by the perfect shuffle permutation Π_N which groups the even-indexed columns first and then the odd-indexed columns second:

$$F_N \Pi_N = (F_N(:, 0 : N-2 : 2) | F_N(:, 1 : N-1 : 2)).$$

Rearranging the columns of F_8 in this manner

$$F_8\Pi_8 = \left(\begin{array}{cccc|cccc} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega^2 & \omega^4 & \omega^6 & \omega & \omega^3 & \omega^5 & \omega^7 \\ 1 & \omega^4 & 1 & \omega^4 & \omega^2 & \omega^6 & \omega^2 & \omega^6 \\ 1 & \omega^6 & \omega^4 & \omega^2 & \omega^3 & \omega & \omega^7 & \omega^5 \\ \hline 1 & 1 & 1 & 1 & \omega^4 & \omega^4 & \omega^4 & \omega^4 \\ 1 & \omega^2 & \omega^4 & \omega^6 & \omega^5 & \omega^7 & \omega & \omega^3 \\ 1 & \omega^4 & 1 & \omega^4 & \omega^6 & \omega^2 & \omega^6 & \omega^2 \\ 1 & \omega^6 & \omega^4 & \omega^2 & \omega^7 & \omega^5 & \omega^3 & \omega \end{array}\right)$$

establishes a connection between F_8 and F_4. Using the fact that $\omega_8^2 = \omega_4$ is a 4th root of unity, and therefore

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega_8^2 & \omega_8^4 & \omega_8^6 \\ 1 & \omega_8^4 & 1 & \omega_8^4 \\ 1 & \omega_8^6 & \omega_8^4 & \omega_8^2 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega_4 & \omega_4^2 & \omega_4^3 \\ 1 & \omega_4^2 & 1 & \omega_4^2 \\ 1 & \omega_4^3 & \omega_4^2 & \omega_4 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix} = F_4.$$

This leads to the partitioning

$$F_8\Pi_8 = \begin{pmatrix} F_4 & \Omega_4 F_4 \\ F_4 & \omega_8^4 \Omega_4 F_4 \end{pmatrix},$$

where $\Omega_4 = \mathrm{diag}(1, \omega_8, \omega_8^2, \omega_8^3)$. Given the fact that $\omega_8^4 = -1$, the factorization

$$F_8\Pi_8 = \begin{pmatrix} I_4 & I_4 \\ I_4 & -I_4 \end{pmatrix} \begin{pmatrix} I_4 & \\ & \Omega_4 \end{pmatrix} \begin{pmatrix} F_4 & \\ & F_4 \end{pmatrix}$$

is obtained. Thus,

$$F_8\Pi_8(I_2 \otimes \Pi_4) = \begin{pmatrix} I_4 & I_4 \\ I_4 & -I_4 \end{pmatrix} \begin{pmatrix} I_4 & \\ & \Omega_4 \end{pmatrix} \begin{pmatrix} F_4\Pi_4 & \\ & F_4\Pi_4 \end{pmatrix}.$$

Now, both of the smaller transforms can be split up again into two transforms of half the original size, namely 4. This kind of recursive splitting can be performed $\log_2 N$ times. Each of these steps involves the whole input vector, on which $O(N)$ operations are performed. Therefore the total arithmetic complexity is $O(N \log_2 N)$.

$$\begin{pmatrix} F_4\Pi_4 & \\ & F_4\Pi_4 \end{pmatrix} =$$

$$= \begin{pmatrix} I_2 & I_2 & & \\ I_2 & -I_2 & & \\ & & I_2 & I_2 \\ & & I_2 & -I_2 \end{pmatrix} \begin{pmatrix} I_2 & & & \\ & \Omega_2 & & \\ & & I_2 & \\ & & & \Omega_2 \end{pmatrix} \begin{pmatrix} F_2 & & & \\ & F_2 & & \\ & & F_2 & \\ & & & F_2 \end{pmatrix},$$

where $\Omega_2 = \mathrm{diag}(1, \omega_4)$. Thus, a complete factorization of F_8 is obtained.

This matrix representation points out, that the Cooley-Tukey idea can be programmed easily using a recursive approach, which in pseudo-code, would have the following form (if the permutation of the input vector is already done):

Algorithm 1.1 (Recursive FFT)

if *length of input vector equals* 1 **then return**
u := **fft** *of upper part of input*
l := **fft** *of lower part of input*
l := $l\times$weight_vector
overwrite upper part of output with $u + l$
overwrite lower part of output with $u - l$
return output

Unfortunately, such recursive algorithms have some unfavorable characteristics. For example, each recursive call needs additional memory. Therefore recursive FFT algorithms have to be implemented carefully for real applications. Anyhow, they are important tools to understand the Cooley-Tukey concept.

1.2 Automatic Performance Tuning Software

Automatic performance tuning is a step beyond standard compiler optimization. It 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 had 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.

These three numerical software packages exhibit superior performance over standard compilers by applying specialized empirical techniques and exploiting structural information generally not available, but automatically and without any architecture specific additional effort. The subject of this thesis is to provide SIMD extension support for these packages in a similar way.

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 making the compiler's task of deducing 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.

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.

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. [68]).

1.2.1 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. [68]). 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.

Atlas is an empirical optimizer that is composed of a search engine that performs empirical search of certain optimization parameter values and a code generator that generates C code given these values. The generated C code is

compiled, executed, and its performance measured. The system keeps track of the values that produced the best performance which will be used to generate highly tuned routines.

1.2.2 FFTW

FFTW (*fastest Fourier transform in the west*) was the first effort to automatically generate FFT code using a special purpose compiler and using the actual run time as optimization criterion (Frigo [19], Frigo and Johnson [20]). Typically, FFTW performs faster than publicly available FFT codes and faster to equal with hand optimized vendor-supplied libraries across different machines. 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 (see Section 1.1). The actual computation is done by automatically generated routines called *codelets* which form the base case of the Cooley-Tukey recursion. 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 [21].

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 [65]).

The Codelet Generator genfft. At install time, all codelets are generated by the *codelet generator* `genfft`. As an alternative the 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 according codelets can be generated.

1.2.3 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. [52]. 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 formulas are automatically generated and automatically translated into code, thus enabling an automated search. More specifically, SPIRAL is based on the following observations.

- For every discrete linear transform there is 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.
- 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.

The architecture of SPIRAL is shown in Fig. 1.1. 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. [57], Singer and Veloso [60], Xiong et al. [69]. For details about the mathematical framework as well as the SPL compiler, see Appendix B.

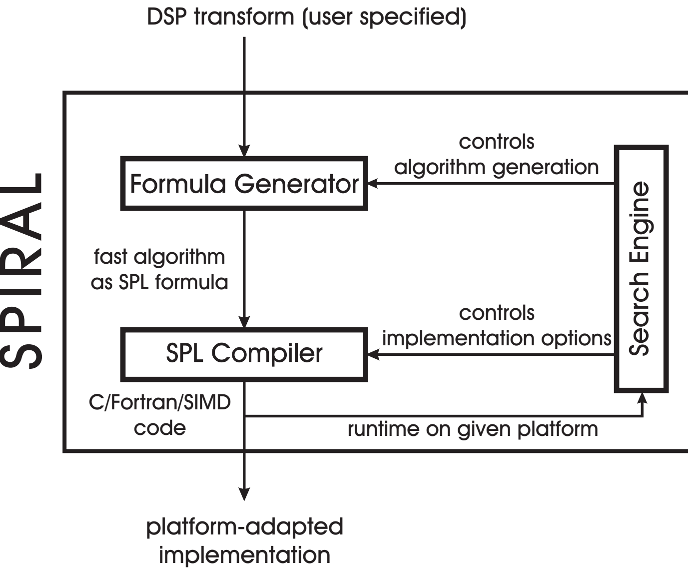


Figure 1.1: SPIRAL's architecture.

Synopsis

Chapter 2 gives an overview over the currently available short vector SIMD instruction extensions and ways to utilize them.

Chapter 3 introduces a portable SIMD API, which serves as an interface to the short vector hardware throughout this thesis.

Chapter 4 describes in detail the ideas behind the implementation of the MAP vectorizer. Necessary concepts are defined, the general algorithm is given, and rules exactly specifying the circumstances under which vectorization is to be performed are displayed.

Chapter 5 introduces concepts for postprocessing the output of the vectorization engine, performing machine specific optimization.

Chapter 6 gives experimental results of the MAP compiler on standard Intel and AMD processors as well as on a prototype of IBM's BlueGene/L supercomputer.

Chapter 2

Short Vector SIMD Extensions

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

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

The double FPU extension which is part of IBM's BlueGene initiative and is implemented in the PowerPC 440 FP2 processors is described in Appendix A.

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.

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.

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.

Integer SIMD Extensions

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

VIS. Next, Sun introduced VIS, a large set of multi-media extensions for 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 a 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.

Floating-Point SIMD Extensions

AltiVec. Motorola's AltiVec SIMD architecture extends the recent MPC74xx G4 generation of the Motorola PowerPC 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 [50, 51].

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 [33, 34, 35] and the C++ compiler manual [36].

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 that 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 [33, 34, 35] and the C++ compiler manual [36].

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 [38, 39, 40] and the C++ compiler manual [36].

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 [3] and in the x86-64 manuals [7, 8].

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

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

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

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

Overview. Tab. 2.1 gives an overview over the SIMD floating-point capabilities found in current microprocessors. In the context of this thesis, especially Intel’s SSE2 and AMD’s 3DNow! extensions as well as IBM’s double FPU are of importance. Thus, only 2-way vectorization is of concern.

Data Streaming

One of the key features needed in multi-media applications such as video decompression is the efficient streaming of data into and out of the processor. They 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	Double FPU	2-way	double	PowerPC 440 FP2	IBM XL Century

Table 2.1: Short vector SIMD extensions providing floating-point arithmetic found in general purpose microprocessors. In the context of this thesis, especially Intel's SSE2 and AMD's 3DNow! extensions as well as IBM's double FPU are of importance.

Software Support

Currently, application developers have three common methods for accessing multi-media hardware on 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 is that existing applications can immediately benefit from the new hardware without recompilation. However, 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.

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.

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 [24], Dongarra et al. [12]). 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 (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. [13]).

Chapter 3

Abstraction of SIMD Instructions

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.

Recently, a sort of common programming model was established. 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. This way, a portable SIMD API consisting of a set of C macros was defined that can be implemented efficiently on all current architectures and features all necessary operations. Using this programming model, a programmer does not have to deal with assembly language. Register allocation and instruction selection is done by the compiler.

The machine models introduced in Section 4.3 are based on this portable SIMD API, which serves two main purposes: (*i*) to abstract from hardware peculiarities, and (*ii*) to abstract from special compiler features.

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. 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 desired algorithms to be built on top of these building blocks. This set is called the *portable SIMD API.*

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. Tab. 2.1 shows that for any current short vector SIMD extension compilers with short vector SIMD language extensions exist.

The portable SIMD API includes macros of five types: (*i*) data types, (*ii*) constant handling, (*iii*) arithmetic operations, (*iv*) reorder operations, and (*v*) memory access operations. An overview of the provided macros is given below. The portable SIMD API can be extended to arbitrary vector length. Thus, optimization techniques like *loop interleaving* (Gatlin and Carter [23]) can be implemented on top of the portable SIMD API.

In this chapter, the SIMD API is defined for an arbitrary vector length ν. As for the remaining part of this thesis only Intel's SSE2 and AMD's 3DNow! extensions as well as IBM's double FPU are featured, this universal representation can be disregarded for the sake of simplicity. Only 2-way SIMD extensions will be of further concern.

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`. Tab. 3.1 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 3.1: 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 may have a different value).

There are three types of constant load operations: (*i*) load a constant vector (both homogeneous and inhomogeneous) that is known at compile time, (*ii*) load a constant vector (both homogeneous and inhomogeneous) that is precomputed at run time (but not known at compile time), and (*iii*) load a precomputed constant real number and build a homogeneous vector constant with that value. Tab. 3.2 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 3.2: Constant handling operations provided by the portable SIMD API. r, $r0$, $r1$, $r2$ and $r3$ denote variables of type `simd_real`. $*r$ denotes a memory location holding a value of type `simd_real`. $*v$ denotes a memory location holding a value of type `simd_vector`.

Arithmetic Operations

The portable SIMD API provides 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 Tab. 3.3 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_ACC_P(v, v1, v2,..., vν)`	$v = (v_1.1 + \cdots + v_1.\nu, \ldots, v_\nu.1 + \cdots + v_\nu.\nu)$
`VEC_UMINUS_P(v, v0)`	$v = -v_0$
`VEC_CHS_P(v, v0, pos)`	$v = (v_0.1, \ldots, -v_0.pos, \ldots, v_0.\nu)$
`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_ACC(v1, v2,..., vν)`	return $(v_1.1 + \cdots + v_1.\nu, \ldots, v_\nu.1 + \cdots + v_\nu.\nu)$
`VEC_UMINUS(v0)`	return $-v_0$
`VEC_CHS(v, v0, pos)`	return $(v_0.1, \ldots, -v_0.pos, \ldots, v_0.\nu)$
`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 3.3: Arithmetic operations provided by the portable SIMD API. v, $v0$, $v1$, $v2$, $v3$, $v4$, $v5$ and $v\nu$ denote variables of type `simd_vector`. *pos* denotes the position of the desired vector element.

Reorder Operations

The SIMD API supports the vector reorder operations `VEC_COPY`, `VEC_UNPACK` and `VEC_SWAP2`. They are summarized in Tab. 3.4. The `VEC_SWAP2` operation is defined for 2-way SIMD architectures only.

Macro	Operation
`VEC_COPY(v, v0)`	$v = v_0$
`VEC_UNPACK(v, v1,..., v`ν`, pos)`	$v = (v_1.pos, \ldots, v_\nu.pos)$
`VEC_SWAP2(v, v0)`	$v.1 = v_0.2$, $v.2 = v_0.1$

Table 3.4: Vector reorder operations provided by the SIMD API. v, $v0$, $v1$ and $v\nu$ denote variables of type `simd_vector`. *pos* denotes the desired position inside the vector of length ν. The `VEC_SWAP2` instruction is defined for 2-way SIMD extensions exclusively.

Memory Access Operations

The portable SIMD API provides load and store instructions both for whole vectors of type `simd_vector` (`VEC_LOAD`, `VEC_STORE`) as well as for single data elements of type `simd_real` (`SCA_LOAD`, `SCA_STORE`). In the latter case, the locations of the `simd_real` value that is to be loaded/stored inside the source and destination vectors must be specified. Tab. 3.5 gives the syntactic and semantic details of the memory access operations supported by the SIMD API.

Macro	Operation
`VEC_LOAD(v, *v0)`	$v = *v0$
`SCA_LOAD(v.pos1, *v0.pos2)`	$v.pos1 = *v0.pos2$
`VEC_STORE(*v, v0)`	$*v = v0$
`SCA_STORE(*v.pos1, v0.pos2)`	$*v.pos1 = v0.pos2$

Table 3.5: Memory access operations provided by the SIMD API. v and $v0$ denote variables of type `simd_vector`. $*v$ and $*v0$ denote memory loacations holding values of type `simd_vector`. $pos1$ and $pos2$ denote the desired position inside the vector of length ν.

Chapter 4

Automatic Vectorization

4.1 Vectorization of Straight Line Code

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

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

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

The available SIMD extensions mirror the underlying hardware features by means of data types and intrinsic or built-in functions that can be utilized by hand-coding. However, the scalar code produced by automatic performance tuning software packages (see Chapter 1.2) often has thousands of lines which makes hand-coding using short vector language extensions unfeasible.

The usefulness of vectorizing compilers is rather limited because they only deal with loop-level parallelism and the underlying scalar code has to be of a special structure to allow for this. Thus, vectorizing compilers are not useful in the vectorization of scalar code emitted by Fftw, Spiral, or Atlas because the code's structure does not allow for loop vectorization. MAP's code generator uses a completely different approach to automatically vectorize numerical straight line code.

4.2 The Vectorization Approach

The goal of the vectorizer is to transform a scalar computation into short vector code while achieving the best possible utilization of SIMD resources. It automatically extracts *2-way SIMD parallelism* out of scalar code blocks (*i*) by fusing pairs of scalar temporary variables into SIMD cells, and (*ii*) by replacing the corresponding scalar instructions by vector instructions as illustrated by Fig. 4.1.

Fusions, i. e., tuples of scalar variables, are declared such that every scalar variable appears in exactly one fusion. Each fusion is assigned a SIMD variable.

On the set of vector variables, a sequence of SIMD instructions has to perform exactly the same computation as the given scalar code. This goal is achieved by pairing scalar instructions and replacing each pair by a sequence of semantically identical SIMD instructions operating on the corresponding SIMD cells. Thus, the vectorizer exhibits the following benefits:

Halving the Instruction Count. Optimally, every pair of scalar floating-point instructions is joined into one equivalent SIMD floating-point instruction, thus cutting the instruction count into half.

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

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

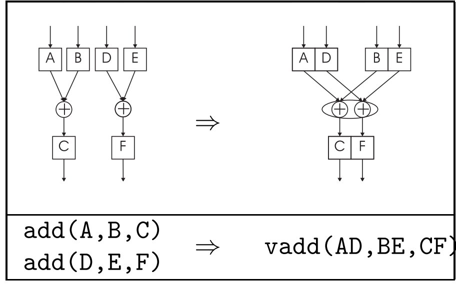


Figure 4.1: 2-way Vectorization. Two scalar `add` instructions are transformed into a vector `vadd` instruction. The result of the vector addition of the fusions `AD` and `BE` is stored in `CF`.

4.3 Virtual Machine Models

Recent hardware development has resulted in a wide range of computer systems having SIMD style instruction set extensions included. These have a lot in common but there are vendor characteristic differences. Even a single vendor enhances and widens the number of SIMD instructions from one hardware generation to the next. For instance, some SIMD architectures (e. g., AMD's K6, K7) include intraoperand style (accumulate) instructions while others (e. g., Intel Pentium 4) don't (Intel has announced that their new SSE3 SIMD instruction set will feature some of them).

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

The usage of such models enables portability and extensibility. The vectorizer can be easily adapted to produce code for any processor architecture supporting

2-way SIMD floating-point instructions. Also, future 2-way SIMD instructions, supported by the successors of existing processors, can be easily included into a model's instruction set allowing the vectorizer to extract them.

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

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

Principally, all instructions of the vectorizer's virtual machine model are available to the vectorization engine with respect to any addressed target architecture. Even those, for which no equivalent instruction is actually implemented there. Nevertheless, those having an equivalent in a target processor's machine model are favored. Vector instructions not supported by the target machine model are only used if otherwise no vectorization is possible at all, because they would have to be rewritten into supported instructions in the optimization step directly following vectorization (see Chapter 5).

The Vectorizer's Virtual Machine Model. The instructions supported and therefore extracted by the vectorizer are: (*i*) load of a SIMD or scalar variable, (*ii*) store of a SIMD or scalar variable, (*iii*) swap, unpack, and copy, (*iv*) multiplication by floating-point constants, (*v*) unary change sign instructions, (*vi*) binary parallel instructions, and (*vii*) binary intraoperand (accumulate) instructions. Tab. 4.2 illustrates the syntactic and semantic details of the instructions.

Virtual Machine Model of Intel's P4. The virtual machine model of Intel's Pentium 4 supports all instructions of the vectorizer's machine model except the intraoperand (i. e., `accPP2`, `accNN2`, `accNP2`, `accPN2`) instructions, the change sign instructions, and the `swap2` instruction. Additionally, it supports a `shuffleXY2` instruction. For details, see Tab. 4.2.

Virtual Machine Model of AMD's K6. The virtual machine model of AMD's K6 processor supports all instructions of the vectorizer's machine model except some intraoperand (i. e., `accNN2`, `accNP2`, `accPN2`) instructions, and the `swap2` instruction. For details, see Tab. 4.2.

Virtual Machine Model of AMD's K7. The virtual machine model of AMD's K7 processor comprises the same virtual instructions as the vectorizer's machine model except the `accPN2` instruction. For details, see Tab. 4.2.

4.3.1 The Virtual Scalar Machine Model

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

Assumptions Underlying the Scalar Model

Any virtual model for scalar machines has to include the following basic operations: (*i*) `load` from, as well as (*ii*) `store` to memory instructions, (*iii*) the unary negation instruction `negate`, instructions for multiplication by a constant, i. e., `mulc`, (*iv*) binary addition, i. e., `add`, subtraction, i. e., `sub`, and multiplication, i. e., `mult` instructions. Tab. 4.1 gives examples of such virtual scalar instructions.

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

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

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

Virtual Scalar Instruction	*Effect*
`load(M[i],A)`	`A := M[i]`
`store(A,M[i])`	`M[i] := A`
`mulc(A,(K),B)`	`B := A * (K)`
`negate(A,B)`	`B := - A`

`add(A,B,C)`
`sub(A,B,C)`
`mul(A,B,C)`
	`C := A + B`
	`C := A - B`
	`C := A * B`

Table 4.1: Virtual Scalar Instructions. The first column contains examples of scalar instructions operating on scalar variables `A,B,...` as well as on a floating-point constant `K`. The second column illustrates the effect of executing these instructions.

4.3.2 The Virtual Vector Machine Model

The virtual vector machine model is based on the portable SIMD API introduced in Chapter 3. It includes virtual instructions needed because of their semantics but

not necessarily included in the actually available target architecture instruction set. Thus, these instructions have to be emulated by a sequence of the cheapest available target architecture vector instructions. This is done in a local rewriting step that directly follows the vectorization step in order to keep the vectorization process as simple as possible (see Chapter 5). Moreover, it makes sense to perform the rewriting step not until a vectorization result is available.

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

Assumptions Underlying the Vector Model

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

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

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

The instructions supported by the vector machine model are grouped into seven categories: (*i*) load of a SIMD or float variable, (*ii*) store of a SIMD or float variable, (*iii*) unary swap and binary unpack, (*iv*) multiplication by floating-point constants, (*v*) unary change of sign operations, (*vi*) binary parallel operations, and (*vii*) binary intraoperand (accumulate) operations. Tab. 4.2 gives an overview of these instructions.

4.3.3 Scalar and Vector Main Memory Layout

The main memory is physically identical in both, scalar and vector computation. Nevertheless, as already described in Sections 4.3.1 and 4.3.2, the indexing of memory locations is different. While indexing to scalar memory locations always addresses one single data value, indexing a vector memory location has a value tuple (i. e., 2-way memory cell) as its target. Each of the cell components, i. e., the lower and the higher part, can be individually addressed.

Taking these assumptions under consideration, locations used by memory access instructions from the scalar machine model have to be aliased to their corresponding memory locations usable by vector machine model instructions. The

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

Table 4.2: Currently Supported Vector Instructions. The first column contains the type of the vector instruction, the second column contains all available vector instructions (containing temporary SIMD variables `A`,`B`,...). The third column illustrates the effect of executing these instructions. The fourth column shows which machine models support each instruction.

following definitions unambiguously specify these mappings and explicitly ascertain whether memory is accessed in scalar or vector layout. The introduced memory layout will be used throughout this chapter whenever memory access is an issue.

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

typedef struct {
    scalar l, h;
} vector;

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

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

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

4.4 The Vectorization Engine

The vectorization engine expects a scalar dag (directed acyclic graph) represented by straight line code consisting of virtual scalar instructions in static single assignment (*SSA*) form as input. The goal of vectorization is to replace as many virtual scalar instructions as possible by virtual vector instructions. To achieve this goal, the vectorization engine has to find pairs of scalar floating-point instructions (and fusions of their respective operands), each yielding — in the optimal case — one SIMD floating-point instruction. In some cases, additional SIMD instructions may be required to obtain a SIMD construct that is semantically equivalent to the original pair of scalar instructions. The vectorization algorithm described hereby is the basis of MAP's vectorization engine. First, some definitions:

Pairing

Pairing rules specify ways of transforming pairs of scalar instructions into a single SIMD instruction or a sequence of semantically equivalent SIMD instructions. A pairing rule often provides several alternatives to do so. The rules used in the MAP vectorizer are classified according to the type of the scalar instruction pair:

Scalar	*ACC*	*PAR1*	*PAR2*

Figure 4.2: SIMD Vectorization of scalar ADD/SUB. Two binary scalar instructions, i. e., addition and subtraction, are transformed into semantically equivalent SIMD instructions. There are three different ways to do so.

unary (i. e., multiplication by a constant), binary (i. e., addition and subtraction), and memory access type (i. e., load and store instructions). Pairings of the following instruction combinations are supported: (*i*) load/load, (*ii*) store/store, (*iii*) unary/unary, (*iv*) binary/binary, (*v*) unary/binary, (*vi*) unary/load, and (*vii*) load/binary. A *pairing rule set* comprises various pairing rules.

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

Fusion

Two scalar operands `S` and `T` are assigned together, i. e., *fused*, to form a SIMD cell of layout `ST = (S,T)` or `TS = (T,S)` if and only if they are the corresponding operands of instructions considered for pairing and neither of them is already involved in another fusion. The position of the scalar variables `S` and `T` inside a SIMD cell (either as its lower or its higher part) strictly defines the fusion, i. e., `ST` $\neq$ `TS`. A special fusion type containing the same scalar variable twice, i. e., `TT = (T,T)`, is needed for some types of code partly containing non-parallel program flow.

An instruction pairing rule forces the fusion layout for the corresponding scalar operands. For two scalar binary instructions, three substantially different layouts for assigning the four operands to two SIMD cells exist, namely (*i*) accumulate (*ii*) parallel 1, and (*iii*) parallel 2. Fig. 4.2 illustrates their semantics in more detail.

A fusion `X12 = (X1,X2)` is *compatible* to another fusion `Y12 = (Y1,Y2)` if and only if `X12 = Y12` or `X1 = Y2` and `X2 = Y1`. In the second case, a swap operation is required to use `Y12` whenever `X12` is needed.

Vectorization Level

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

The vectorization engine utilizes three levels of vectorization in its search process as follows: First, a vectorization is searched that utilizes the most restrictive level, i. e., *full vectorization*. This vectorization level only provides pairing rules for instructions of the same type and allows quadword memory access operations exclusively. If full vectorization is not obtainable, a fallback to a less restrictive vectorization level, i. e., *semi vectorization*, is made. This level provides versatile pairing rules for instructions of mixed type and allows doubleword memory access operations. In the worst case, if neither semi vectorization nor full vectorization is feasible, a fallback is made to a vector implementation of scalar code, i. e., *null vectorization* is applied. Null vectorization rules allow to vectorize any code by leaving half of each SIMD instruction's capacity unused. Even null vectorization results in better performance than using the legacy x87.

Tab. 4.3 illustrates which rules are available for each vectorization level.

Operations	*Vectorization Level*		
	semi	*full*	*null*
Store/Store	×	×	–
Load/Load	×	×	–
Load/Binary	×	–	–
Binary/Load	×	–	–
Unary/Unary	×	×	–
Unary/Any	×	–	–
Any/Unary	×	–	–
Binary/Binary	×	×	–
Null Vectorization	–	–	×

Table 4.3: Rulesets for Three Vectorization Levels. *Semi vectorization* uses all transformation rule groups except null vectorization rules. *Full vectorization* only applies rules which refer to pairs of the same instruction type, i. e., binary/binary, unary/unary, . . . In *null vectorization* there is no vectorization applied at all. Each scalar instruction is straightforwardly transformed into a semantically equivalent SIMD instruction.

4.4.1 The Vectorization Algorithm

MAP's vectorization algorithm implements a depth first search with chronological backtracking. The search space is given by applying the rules allowed in the cur-

rent vectorization level in arbitrary order. Depending on how versatile/restrictive the utilized pairing rule set is, there can be many, one or no possible solution at all.

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

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

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

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

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

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

Example. Figures 4.3 to 4.5 contain an example for *one* possible solution of the vectorization algorithm. Fig. 4.3 depicts the scalar code, an FFT kernel of size 3, which is the input of MAP's vectorizer. Fig. 4.4 lists all transformations (together with the rule group number from Section 4.5) that yield the vectorization solution shown in Fig. 4.5. Due to the fact that different vectorizations are possible, this is just one example out of all possible solutions.

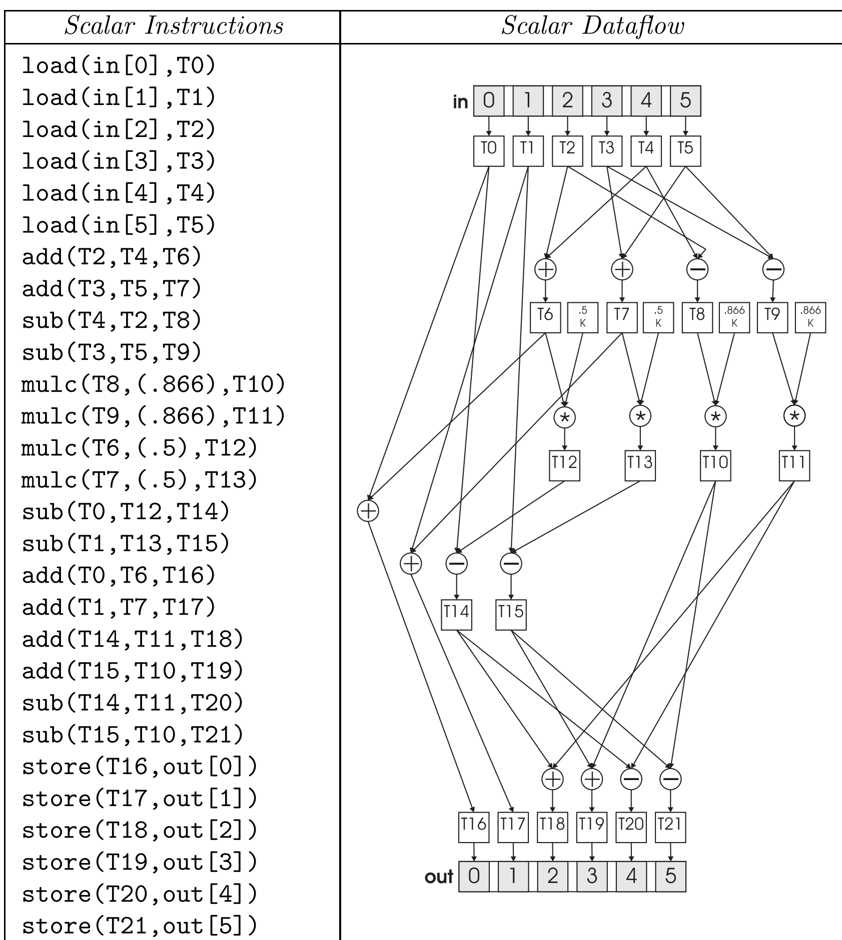


Figure 4.3: (Example) Scalar FFT of Size N = 3. The scalar instruction sequence in the left column corresponds to the scalar data flow layout depicted in the right column.

4.4.2 Vectorization Heuristics

Limiting Pairings for Memory Accesses

The vectorization search space can be pruned by utilizing heuristics to restrict the possible pairings for load/load and store/store vectorization rules (see Sections 4.5.1 and 4.5.2). The details are given in Tables 4.4 and 4.5. The heuristics immediately reject apparently suboptimal vectorization paths and enforce obvious parallelism onto the vectorization process. Full vectorization uses heuristics H0, H1 and H2 to vectorize memory accesses, semi vectorization just relies on heuristic H1.

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

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

Figure 4.4: (Example) Scalar To Vector Transformation of an FFT of Size N = 3. The scalar instructions in the second left column are transformed into the vector instruction sequence in the right column. It will be explained in Section 4.5 that it is not always possible to transform two scalar instructions into one SIMD instruction. The leftmost column shows to which transformation rule group introduced in Section 4.5 the scalar to vector transformation belongs.

Order of Application of Pairing Rules

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

For example, MAP's AMD K7 specific ruleset provides the rules in an order to favor intraoperand and swap instructions because they are inherent in this

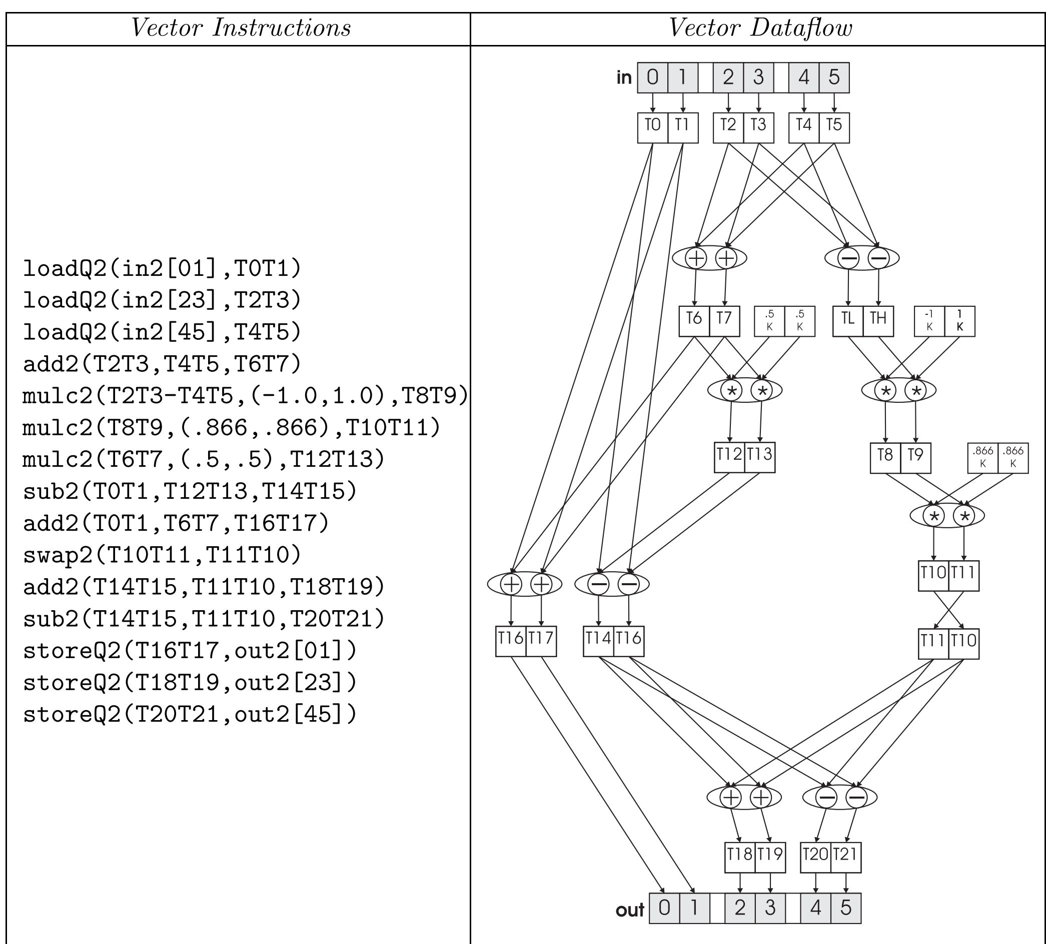


Figure 4.5: Example: 2-way Vector FFT of Size N = 3. The vector instruction sequence in the left column corresponds to the data flow layout depicted in the right column. This code is one possible result of the vectorizer when it is fed the scalar instruction sequence from Fig. 4.3.

processor's target instruction set. On the other hand, the application order in Intel's Pentium 4 ruleset "tries" to avoid these instructions by only extracting them if no vectorization can be achieved otherwise, i. e., the preferred rules do not work. They are rewritten into supported instructions in a separate step after the vectorization process (see Chapter 5).

4.4.3 Handling Different Results of Vectorization

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

The *pick first strategy* immediately commits to the first solution. The *pick best*

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

Table 4.4: Heuristics for Store/Store Vectorization. In the leftmost column the possible types of store memory access in MAP are listed. For further details refer to [21]. The following columns show which constraints the indices i_1 and i_2 of the two stores have to satisfy for a pairing of the corresponding instructions to be allowed. N is the size of memory when addressed with scalar memory layout.

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

Table 4.5: Heuristics for Load/Load Vectorization. In the leftmost column the possible types of load memory access in MAP are listed. For further details refer to [21]. The following columns show which constraints the indices i_1 and i_2 of the two loads have to satisfy for a pairing of the corresponding instructions to be allowed. N is the size of memory when addressed with scalar memory layout.

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

4.5 Pairing Rules

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

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

4.5.1 Group 1: Load/Load Pairing Rules

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

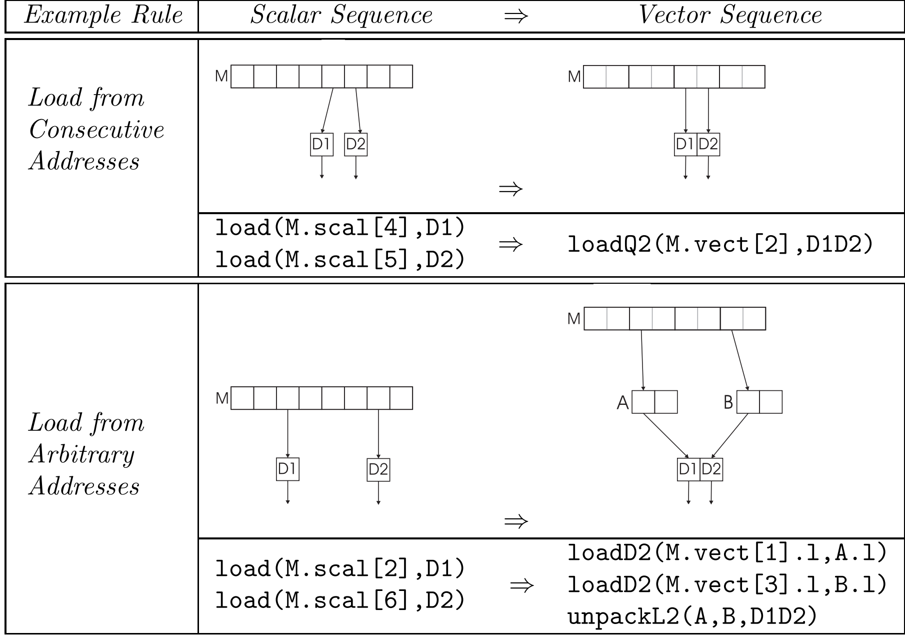


Figure 4.6: (Example) Various Load Pairings. While scalar loads on consecutive addresses can be implemented in SIMD straightforwardly, the scalar loads on arbitrary addresses need to be rearranged. The load on consecutive addresses corresponds to a complex load while the load on arbitrary addresses corresponds to a split load on MAP input arrays.

Load from Consecutive Addresses

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

If `addr1 = addr0 + 1` with `addr0 = 2*i`, `i = 0,1,...,N/2-1`, the scalar loads are transformed into one SIMD load as demonstrated by the upper example in Fig. 4.6. In MAP's input arrays, a load on such consecutive memory cells corresponds to a *complex load*, where the first element is assumed to be the real part and the second element is the imaginary part of a complex number.

Load from Arbitrary Addresses

This type of SIMD load works on any two addresses. Therefore, `addr0` and `addr1` can address any valid position of the input array. This has the drawback that the

two scalar loads from `addr0` and `addr1` cannot be realized using just one vector load instruction, i. e., `loadQ2`.

Therefore, the data elements are loaded into the lower parts of two temporary vector variables `A.l` and `B.l`, using two `loadD2` instructions. An `unpackL2` operation must be applied on `A.l` and `B.l` before the data can be used for computation. This instruction sequence is illustrated by the lower example in Fig. 4.6. In the case of MAP's input arrays, a load on arbitrary memory cells corresponds to a *split load*, where the first and the second element are both real elements of a real transform array or any combination of real and imaginary parts of a complex array.

4.5.2 Group 2: Store/Store Pairing Rules

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

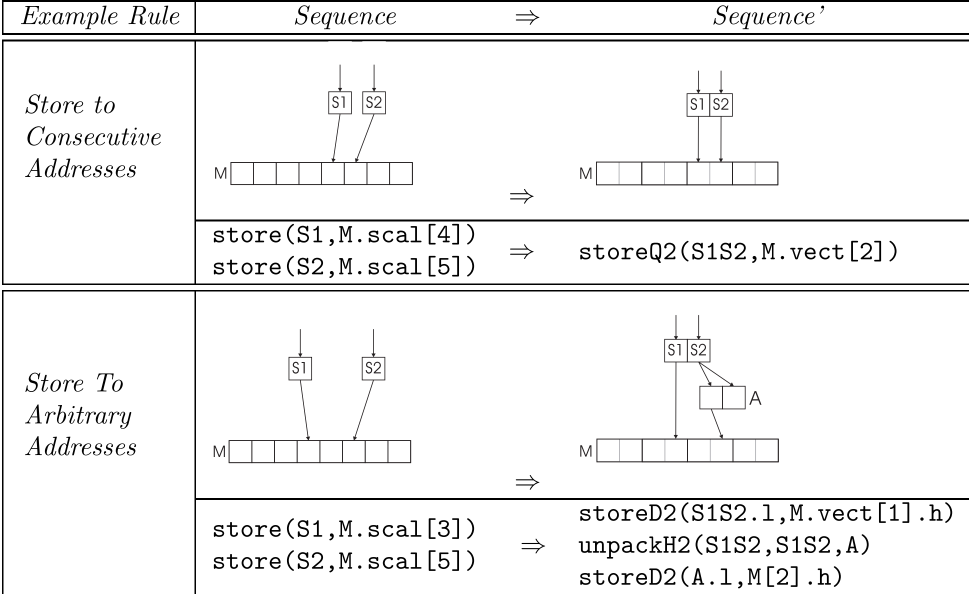


Figure 4.7: (Example) Various Store Pairings. This is the counterpart ruleset to the load rules from Fig. 4.6. The store to consecutive addresses corresponds to a complex store while the store to arbitrary addresses corresponds to a split store to MAP's output arrays.

Store to Consecutive Addresses

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

If `addr1 = addr0 + 1` with `addr0 = 2*i` where `i = 0,1,...,N/2-1` , the scalar stores are transformed into one SIMD store as shown in the upper part of Fig. 4.7. In the case of MAP's output arrays, a store on such consecutive memory cells corresponds to a complex store, where the first element is the real part and the second element the imaginary part of a complex number. This rule is the counterpart to the load to consecutive addresses rule.

Store to Arbitrary Addresses

This type of transforming two scalar stores into one SIMD store works on any two addresses. Therefore, `addr0` and `addr1` can address any valid position of the output array with the same drawback as its counterpart, i. e., load from arbitrary addresses. Two scalar stores to `addr0` and `addr1` cannot be realized with just one vector quad store instruction, i. e., `storeQ2`. The lower part of the SIMD variable can be directly stored to the corresponding memory location by a double store while the higher part has to be rearranged into a temporary vector variable by an unpack operation before it can be stored into memory from there. This approach is shown in the lower example of Fig. 4.7.

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

4.5.3 Group 3: Unary/Unary Pairing Rules

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

These transformations are performed straightforwardly as depicted in Fig. 4.8. Fig. 4.9 defines, according to the scalar machine model, how combinations of scalar operations can be transformed into semantically equivalent SIMD operations. Fig. 4.10 gives an example of an application of a unary/unary transformation rule.

4.5.4 Group 4: Load/Binary Pairing Rules

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

Fig. 4.11 shows the binding of a load instruction with a binary instruction. Not all possible binary instructions supported by the virtual machine model of the vectorizer are supported in this ruleset. Fig. 4.12 illustrates which instructions are allowed to be paired with the unary load and which SIMD operations are utilized to transform these instructions into a semantically equivalent vector instruction sequence. Fig. 4.13 gives an example showing the application of a load/binary transformation.

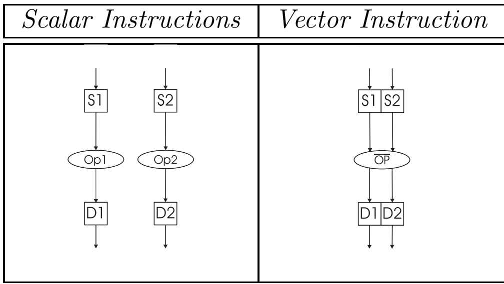


Figure 4.8: Pairing of Two Unary Instructions. Two scalar unary operations `Op1` and `Op2` are transformed into one SIMD operation $\overline{\texttt{Op}}$. The operation transformation is performed according to the unary operation algebra introduced in Fig. 4.9.

Operation 1 Operation 2	⇒	*Operation'*
`Negate Negate`	⇒	`chsLH2`
`Negate mulc(K)`	⇒	`mulc2(-1,K)`
`mulc(K) Negate`	⇒	`mulc2(K,-1)`
`mulc(K1) mulc(K2)`	⇒	`mulc2(K1,K2)`

Figure 4.9: Unary/Unary Operation Algebra. Two scalar operations are transformed into one semantically equivalent SIMD operation.

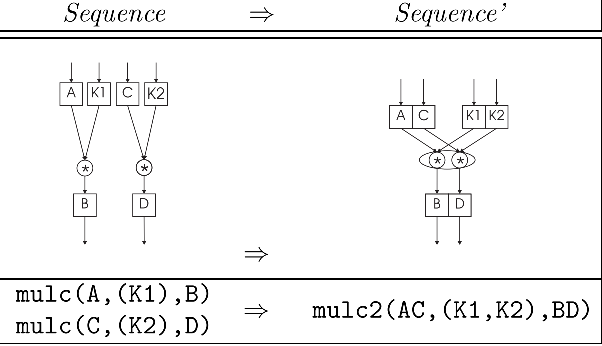


Figure 4.10: (Example) Application of a Unary/Unary Rule. Two scalar `mulc` instructions are transformed into a vector `mulc2` instruction. This is the transformation out of Group 3 which most frequently occurs in FFT kernel vectorization.

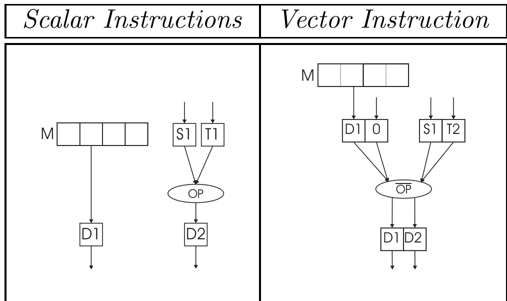


Figure 4.11: Pairing of Load and Binary Instructions. The scalar load and the binary operation `Op` are transformed into two SIMD instructions of which one is a SIMD double load. The operation $\overline{\texttt{Op}}$ of the intraoperand vector instruction is set according to the algebra defined in Fig. 4.12. Not only the depicted fusion of load/binary but also the mirrored binary/load is allowed.

Operation	⇒	*Operation'*
`add`	⇒	`accPP2`
`sub`	⇒	`accNN2`

Figure 4.12: Load/Binary Operation Algebra. Scalar binary operations are transformed into semantically equivalent SIMD operations.

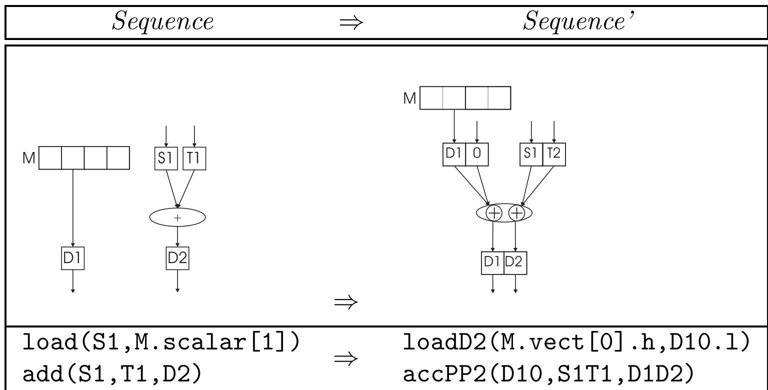


Figure 4.13: (Example) Application of a Load/Binary Rule. Scalar load and add instructions are transformed into one double load into the lower part of a SIMD variable whose higher part is initialized with 0. The accumulate instruction leaves the loaded value unchanged and performs the addition.

Scalar	*ACC*	*PAR1*	*PAR2*
S1 T1 S2 T2 OP 1 OP 2 D1 D2	S1 T1 S2 T2 $\overline{OP}$ D1 D2	S1 S2 T1 T2 $\overline{OP}$ D1 D2	S1 T2 T1 S2 $\overline{OP}$ D1 D2
`add, sub, mul`	`accPP2, accPN2` `accNP2, accNN2`	`add2, sub2, mul2`	`add2, sub2, mul2`

Figure 4.14: Binary Binding Types. The operands `S1, T1 and S2, T2` of the scalar binary operations in the leftmost column can be fused in three different ways shown in columns two, three and four. Accumulate style operand fusion allows the scalar operations `Op1 and Op2` to be transformed into an intraoperand vector operation $\overline{\texttt{Op}}$. Parallel 1 and Parallel 2 operand fusions allow scalar operations to be transformed into a parallel style operation. The last row shows feasible operations that `Op` can be transformed into.

4.5.5 Group 5: Binop/Binop Pairing Rules

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

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

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

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

Fig. 4.14 illustrates the three binding types in more detail. Fig. 4.15 gives an overview of all transformations depending on the binding type. All possible binary instruction pairs are presented together with their transformation into SIMD instruction sequences. Fig. 4.16 depicts an example for computationally equivalent SIMD transformations of a scalar addition and subtraction instruction.

Binding	*ADD/ADD*	*ADD/SUB*	*SUB/ADD*	*SUB/SUB*	*MUL/MUL*
ACC	`accPP2`	`accNP2` `swap2`	`accNP2`	`accNN2`	—
PAR1	`add2`	`chsL2` `sub2`	`chsL2` `add2`	`sub2`	`mul2`
PAR2	`add2`	`chsH2` `add2`	`chsL2` `add2`	`sub2` `chsH2`	`mul2`

Figure 4.15: Binding Types for Two Binary Operations. The columns specify the scalar operations and their transformations into semantically equivalent SIMD instructions. The rows are indexed by the different binding types and therefore indicate which transformations are allowed together with which of the binding types.

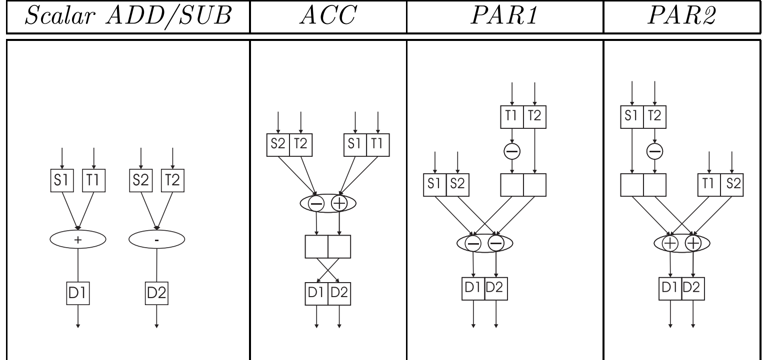


Figure 4.16: (Example) Binary Bindings for ADD/SUB. The ADD/SUB case has been chosen because of its nontrivial transformation from scalar add and sub instructions to a sequence of more than one semantically identical SIMD instructions.

4.5.6 Group 6: Unary/Any Pairing Rules

This set of rules is used to pair a unary instruction with any load or binary instruction. It is characteristic for this set of rules, that the source operand of the unary instruction is always fused with the destination operand of the other instruction. The vector operation is chosen such that it performs a dummy operation on the vector register part which holds the destination of the preceeding "any" instruction. Not only a rule for the combination unary/any but also for any/unary is provided. Figures 4.17 and 4.18 illustrate this approach in more detail.

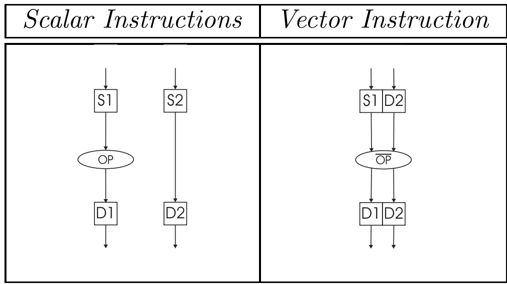


Figure 4.17: Pairing of a Unary Operation with Any Other Operation. It is specific for this pairing type that not the sources `S1` and `S2` of the two scalar operations are fused, but the source of the unary operation `S1` with the destination `D2` of the "any" operation which remains unchanged by the vector operation $\overline{\mathtt{Op}}$. The source operands may also be fused the other way round, i. e., `D2S1`.

Operation	⇒	*Operation'*
`negate`	⇒	`mulc2(-1,1)`
`mulc(K)`	⇒	`mulc2(K,1)`

Figure 4.18: Unary/Any Operation Algebra. The scalar unary operation is transformed into a semantically equivalent SIMD Operation which performs a dummy operation on the corresponding part of the fused SIMD register by simply multiplying one.

4.5.7 Group 7: Reordering of Compatible Fusions

As stated in Section 4.4, every scalar variable is allowed to be a member of not more than one fusion. Therefore, if two scalar operands `A` and `B` already form a SIMD cell fusion of layout `AB = (A,B)`, but a fusion of layout `BA = (B,A)` is needed, an additional vector swap instruction with `AB` as its input is generated. This preserves the existing fusion `AB`, but also provides the desired SIMD cell with switched operands. Two examples are given in Fig. 4.19.

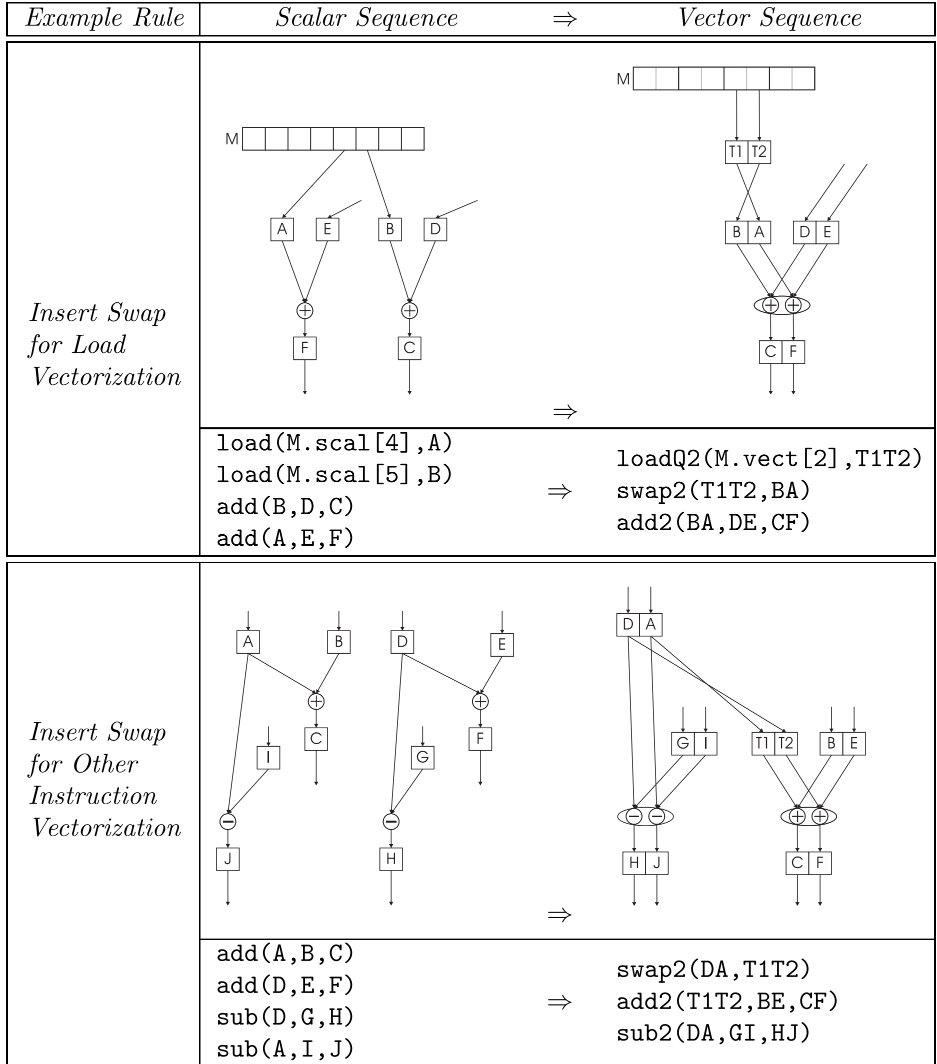


Figure 4.19: (Example) Generation of Swaps Needed for Vectorization. All scalar instruction sequences' dataflow is downwards while the vectorization is performed bottom up. In the upper example, the two scalar loads cannot be vectorized into `loadQ2(M.vect[2],AB)`. This happens because `A` and `B` are already involved in pairing `BA`. Therefore, a temporary SIMD variable `T1T2` is needed along with `swap2(T1T2,BA)` to preserve the dataflow semantics of the scalar dag. In the lower example, the scalar add instructions are not vectorized to `add2(AD,BE,CF)` because the pairing `DA` already exists. The temporary SIMD variable `T1T2` is used along with `swap2(DA,T1T2)` to enable the vectorization of the scalar add instructions.

Group 8: Null Vectorization Pairing Rules

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

Fig. 4.20 depicts all rules for transforming single scalar operations into vectorial ones.

Operation	⇒	*Operation'*
`load(M[i],A)`	⇒	`loadD2(M[i],A2.l)`
`store(A,M[i])`	⇒	`storeD2(A2.l,M[i])`
`mulc(A,K,B)`	⇒	`mulc2(A2,K,1,B2)`
`negate(A,B)`	⇒	`chsL2(A2,B2)`
`add(A,B,C)`	⇒	`add2(A2,B2,C2)`
`sub(A,B,C)`	⇒	`sub2(A2,B2,C2)`
`mul(A,B,C)`	⇒	`mul2(A2,B2,C2)`

Figure 4.20: Null Vectorization Rules. In the case of null vectorization, a general agreement is to rearrange content of the scalar registers `A,B,...` into the lower parts `A.l,B.l,...` of the vector cells `A2,B2,...` Under this assumption it is possible to transform the scalar unary, binary and load/store instructions into equivalent SIMD instructions.

Chapter 5

Rewriting and Optimization

5.1 Optimization of Vectorized Code

5.1.1 Necessity of Optimization

The result of automatic vectorization of scalar straight line code, as described in Chapter 4, is vectorized code with a computationally equivalent instruction sequence, solely consisting of virtual SIMD instructions.

Automatically generated short vector code is not necessarily optimal in terms of (*i*) total number of instructions, (*ii*) copy propagation, (*iii*) redundant calculations, and (*iv*) use of machine specific idioms. Moreover, the vectorization engine uses an abstract machine model, whose instructions are not necessarily supported on a specific target architecture (see Section 4.3).

Therefore, it is necessary to optimize with respect to target-architecture independent as well as target-architecture dependent criterions. Transformations dealing with both of them are applied to the vectorizer output using a local code rewriting technique, commonly referred to as *peephole optimization*. In principle, this optimization step could be combined with the vectorization process. However, a separate step is preferred for the benefit of *modularization*, *maintainability*, and *reduced vectorization complexity*.

5.1.2 Optimization Goals

Optimization aims at enhancing the quality of the input code. This is realized by pursuing the following goals:

Reduced Number of Instructions. Many of the utilized rewriting rules try to reduce the overall number of instructions. This may result in dead code (i. e., code without any dependencies) eliminated throughout the rewriting process.

Shortened Critical Path. The length of the critical path ([61]) of a computational task defines the minimum execution time the computation must inevitably take, regardless of the maximum amount of computation allowed to be carried out simultaneously. Some transformation rules shorten the critical path length of the data dependency graph by removing redundancies in the code.

Reduced Number of Source Operands. Some transformation rules reduce the number of operands needed to perform an operation. This effectively reduces register pressure.

Inclusion of Hardware Specific Features. Many of the transformation rules exploit processor specific features, e. g., the shuffle instruction for Intel P4 code. Also, combinations of instructions defined by the vectorizer's virtual hardware architecture are rewritten into combinations of instructions actually supported by the target hardware.

5.2 Peephole Optimization

Peephole optimization is a local code rewriting technique. A window (*peephole*) uncovers a (small) set of instructions that is considered for optimization. The instructions dealt with inside a peephole are connected by dependencies [1]. Then, if possible, optimizing transformations are applied to the actually considered set of instructions. More specifically, a combination of instructions that matches a corresponding pattern is improved according to the associated transformation rule. The newly obtained sequence is either shorter or comprises faster but semantically equivalent instructions.

It is a typical characteristic of peephole optimization that new rewriting possibilities arise after carrying out a first set of transformations, i. e., several iterations are needed to yield a satisfactory optimization result.

5.2.1 The Local Rewriting Process

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

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

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

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

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

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

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

5.2.2 The Rewriting Engine

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

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

5.2.3 The Order of Rule Application

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

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

5.3 Transformation Rules

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

5.3.1 Target Architecture Independent Rules

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

The Substitution Rules

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

In most cases, copy propagation can be performed straightforwardly as the first three examples in Fig. 5.1 illustrate. The last group of examples in Fig. 5.1 deals with nonstandard register substitutions utilizing special properties of vector operations. These are exclusively algebraic sign identities and are shown in Fig. 5.2. In the following, the substitution rules are described in more detail.

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

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

Nonstandard Unary Substitution. For two read-after-write dependent unary instructions, replace the second instruction according to the rules given in Fig. 5.2.

Rule Type	*Sequence*	⇒	*Sequence'*
Register	`copy2(A,B)` `chsL2(B,C)`	⇒	`copy2(A,B)` `chsL2(A,C)`
	`copy2(A,B)` `add2(B,C,D)`	⇒	`copy2(A,B)` `add2(A,C,D)`
Store	`copy2(A,B)` `storeD2(hi,B,M,3)`	⇒	`copy2(A,B)` `storeD2(hi,A,M,3)`
	`copy2(A,B)` `storeQ2(B,M,2)`	⇒	`copy2(A,B)` `storeQ2(A,M,2)`
Unary	`chsH2(A,B)` `chsH2(A,C)`	⇒	`chsH2(A,B)` `copy2(B,C)`
Binary	`sub2(A,B,C)` `sub2(A,B,D)`	⇒	`sub2(A,B,C)` `copy2(C,D)`
Nonstandard Unary	`chsL2(A,B)` `chsH2(B,C)`	⇒	`chsL2(A,B)` `chsLH2(A,C)`
	`chsH2(A,B)` `mulc2(B,(N1,N2),C)`	⇒	`chsH2(A,B)` `mulc2(A,(N1,-N2),C)`
	`mulc2(A,(N1,N2),B)` `mulc2(B,(M1,M2),C)`	⇒	`mulc2(A,(N1,N2),B)` `mulc2(A,(N1*M1,N2*M2),C)`

Figure 5.1: (Example) Subgroups of Substitution Rules. The register, binary and unary substitution transformations are transformed straightforwardly into *Sequence'*. The nontrivial register operations cannot be transformed that easy as they involve vector algebra specific transformations as described in Fig. 5.2.

Operation 1 Operation 2	⇒	*Operation'*
`Swap2 Swap2`	⇒	`Identity`
`chsL2 chsL2`	⇒	`Identity`
`chsL2 chsH2`	⇒	`chsLH2`
`chsL2 chsLH2`	⇒	`chsH2`
`chsH2 chsL2`	⇒	`chsLH2`
`chsH2 chsH2`	⇒	`Identity`
`chsH2 chsLH2`	⇒	`chsL2`
`chsLH2 chsL2`	⇒	`chsH2`
`chsLH2 chsH2`	⇒	`chsL2`
`chsLH2 chsLH2`	⇒	`Identity`
`chsL2 mulc2(N,M)`	⇒	`mulc2(-N,M)`
`chsH2 mulc2(N,M)`	⇒	`mulc2(N,-M)`
`chsLH2 mulc2(N,M)`	⇒	`mulc2(-N,-M)`
`mulc2(N,M) chsL2`	⇒	`mulc2(-N,M)`
`mulc2(N,M) chsH2`	⇒	`mulc2(N,-M)`
`mulc2(N,M) chsLH2`	⇒	`mulc2(-N,-M)`
`mulc2(N1,M1) mulc2(N2,M2)`	⇒	`mulc2(N1*N2,M1*M2)`

Figure 5.2: Nonstandard Operand Transformations for Register Substitution. *Operation 2* is transformed into *Operation'* to eliminate the read-after-write dependency between *Operation 1* and *Operation 2*. These transformations are supposed to remove as many register dependencies between operations as possible to allow for later optimization.

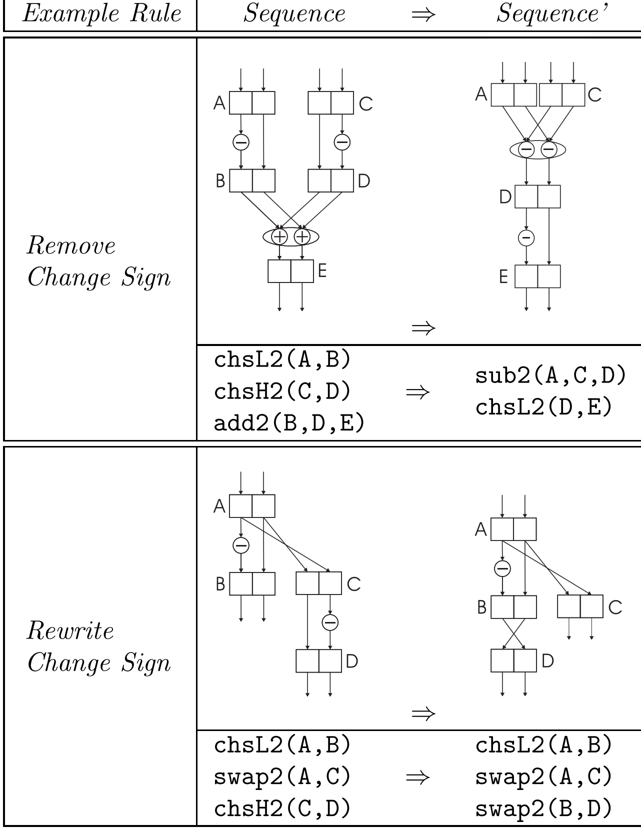
The Sign Change Specific Rules

Whenever two scalar binary instructions, one of them implementing a subtraction, the other one an addition operation, are paired, an additional sign change instruction is unavoidable for implementing a parallel minus/plus or plus/minus vector operation. Sign change specific transformations remove sign change instructions as well as rewrite sign changes in combinations with other instructions into more optimal code sequences.

Some sign change specific rules are used to find code sequences including sign change instructions where optimization allows to minimize the overall instruction count. Other rules, are used to rewrite sign change instructions into swap instructions which are generally known to be cheaper or at least of the same quality in terms of latency and throughput. The only constraint is that the overall instruction count is not to be increased.
Fig. 5.3 exemplarily shows one characteristic example for each of these rule targets. In Fig. 5.4 all utilizable rules are given showing the basic transform idea without further details about operand registers and their constraints.

The Code Motion Rules

This set of rules is used to topologically shift swap instructions up and down in the considered instruction sequence. It allows for the interchange of swap instruc-



Example Rule	*Sequence*	⇒	*Sequence'*

Remove Change Sign	`chsL2(A,B)`
	`chsH2(C,D)`
	`add2(B,D,E)`
		⇒	`sub2(A,C,D)`
			`chsL2(D,E)`

Rewrite Change Sign	`chsL2(A,B)`
	`swap2(A,C)`
	`chsH2(C,D)`
		⇒	`chsL2(A,B)`
			`swap2(A,C)`
			`swap2(B,D)`

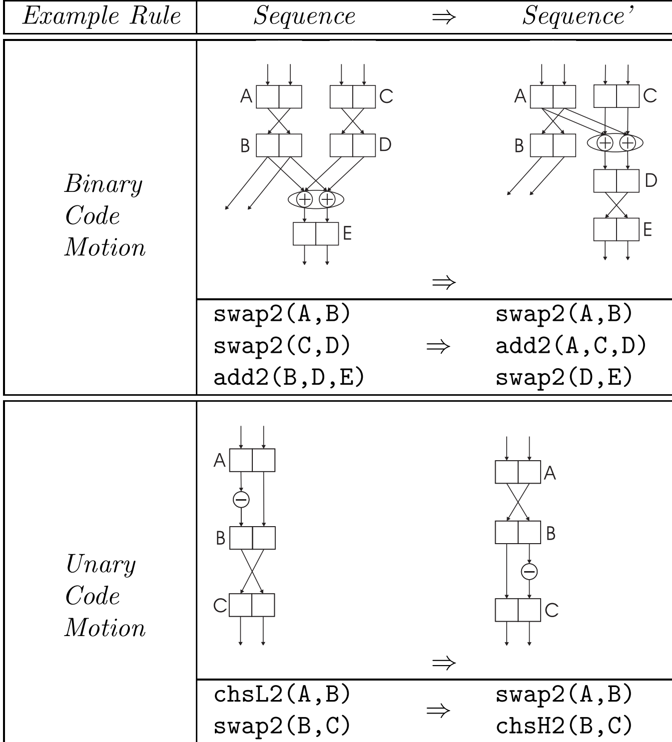
Figure 5.3: (Example) Change Sign Specific Transformations. One example rule for each subgroup of these transformation rules is given.

Sequence	⇒	*Sequence'*	*Goal*
`chsL2, chsH2, add2, add2`	⇒	`chsL2, add2, sub2`	instr
`chsH2, chsL2, add2`	⇒	`sub2, chsH2`	instr
`chsL2, chsH2, add2`	⇒	`sub2, chsL2`	instr
`chsL2, swap2, chsH2`	⇒	`chsL2, swap2, swap2`	swap
`chsH2, swap2, chsL2`	⇒	`chsH2, swap2, swap2`	swap

Figure 5.4: Rewriting Ruleset for Change Sign Specific Transformations. All combinations of remove and rewrite change sign rule transformations are given together with the goal they aim at. The goal *instr* aims at minimizing the instruction count. *swap* occurs when the associated transformation generates this operation.

tions with unary or binary instructions, which opens appliance possibilities for other rules at first not firing. The code motion rules which interchange a swap instruction with an arbitrary sign change instruction are used in the rulesets for optimization steps one and two as described in Section 5.2.2.

Fig. 5.5 gives an example for the functionality of these code motion transformations. Fig. 5.6 briefly illustrates all available code motion transformations.



Example Rule	*Sequence*	⇒	*Sequence'*
Binary Code Motion	swap2(A,B) swap2(C,D) add2(B,D,E)	⇒	swap2(A,B) add2(A,C,D) swap2(D,E)
Unary Code Motion	chsL2(A,B) swap2(B,C)	⇒	swap2(A,B) chsH2(B,C)

Figure 5.5: (Example) Code Motion Transformations. Code motion transformations are intended to change the topological position of swap operations inside the examined instruction sequence.

Sequence	⇒	*Sequence'*
swap2, swap2, binop2	⇒	swap2, binop2, swap2
swap2, binop2, swap2	⇒	swap2, swap2, binop2
mulc2, swap2	⇒	swap2, mulc2
swap2, mulc2, mulc2	⇒	swap2, mulc2, mulc2
chsL2, swap2	⇒	swap2, chsH2
swap2, chsL2	⇒	chsH2, swap2

Figure 5.6: Rule Set for Code Motion Transformations. The shift transformations allow for topological position interchange of swap operations with unary *mulconst* operations and all types of binary operations.

5.3.2 Target Architecture Specific Rules

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

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

Specific Rules for AMD's K7

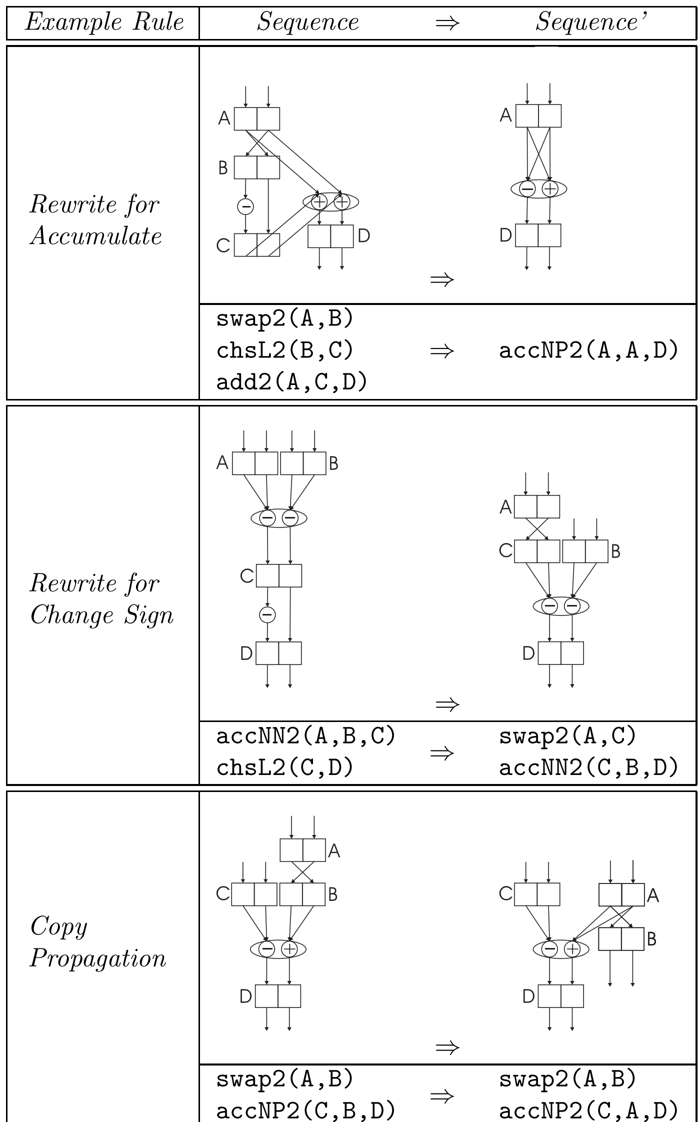
Fig. 5.7 presents examples for transforming code to the *AMD K7 Virtual Machine Model*. Accumulate instructions are generated and optimizations concerning these highly efficient instructions are performed. Fig. 5.8 shows the basic AMD K7 transformation ideas without further operand and constraint details.

Specific Rules for AMD's K6

Fig. 5.9 presents examples for transforming instructions unsupported by the *AMD K6 Virtual Machine Model* into supported ones. Fig. 5.10 shows the basic AMD K6 transformation ideas without further operand and constraint details.

Specific Rules for Intel's Pentium 4

Fig. 5.11 shows example transformations for the *Intel P4 Virtual Machine Model*. Fig. 5.12 shows the basic Intel Pentium 4 transformation ideas without further operand and constraint details.

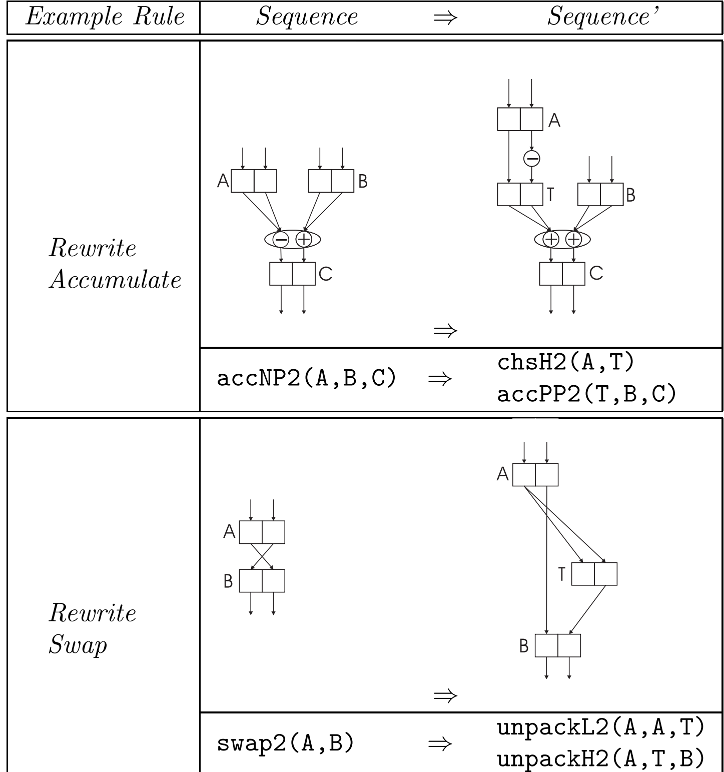


Example Rule	*Sequence*	⇒	*Sequence'*
Rewrite for Accumulate	swap2(A,B) chsL2(B,C) add2(A,C,D)	⇒	accNP2(A,A,D)
Rewrite for Change Sign	accNN2(A,B,C) chsL2(C,D)	⇒	swap2(A,C) accNN2(C,B,D)
Copy Propagation	swap2(A,B) accNP2(C,B,D)	⇒	swap2(A,B) accNP2(C,A,D)

Figure 5.7: (Example) Transformations for AMD's K7. One example rule for each subgroup of transformation rules is given.

Sequence	⇒	*Sequence'*	*Goal*
`swap2, accXp2`	⇒	`swap2, accXp2`	reg
`swap2, accpX2`	⇒	`swap2, accpX2`	reg
`swap2, accNP2`	⇒	`swap2, accNP2`	reg
`swap2,swap2,nnacc2,mulc2`	⇒	`swap2,swap2,nnacc2,mulc2`	reg
`accXX2, swap2`	⇒	`accXX2`	instrcnt
`swap2, chsL2, sub2`	⇒	`accNP2, swap2`	acc
`swap2, chsL2, add2`	⇒	`accNP2`	acc
`swap2, chsH2, add2`	⇒	`accPN2, swap2`	acc
`accNN2, chsL2`	⇒	`swap2, accNN2`	swap
`accNN2, chsH2`	⇒	`swap2, accNN2`	swap

Figure 5.8: Rule Set for AMD K7 Transformations. All AMD K7 specific rules are given together with the goals they are objected at. The goals *reg* and *instrcnt* are targeted at minimizing the number of operand registers and the instruction count. *acc* and *swap* are quoted when the associated transformation generates these operations.

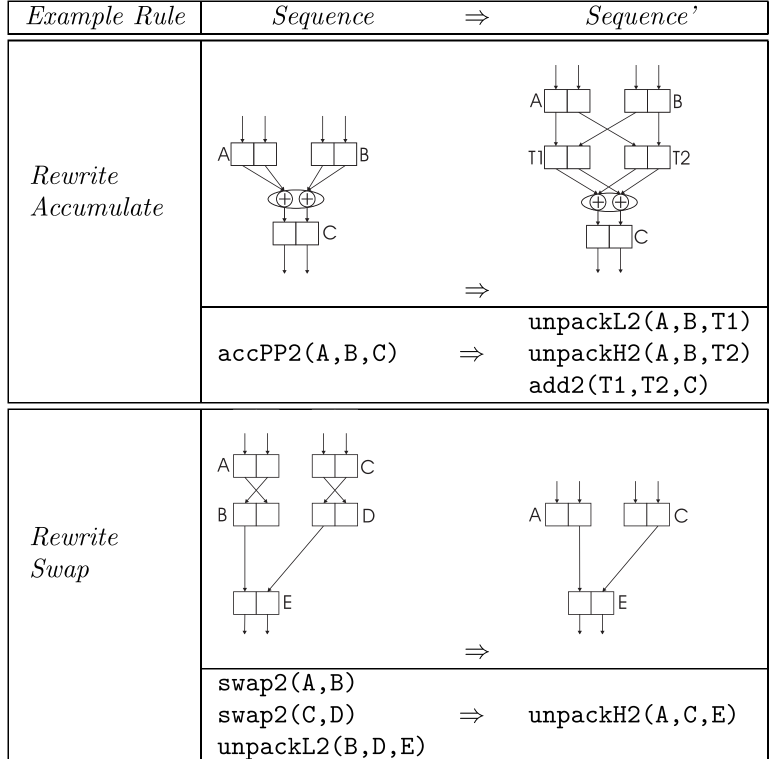


Example Rule	*Sequence*	⇒	*Sequence'*
Rewrite Accumulate	`accNP2(A,B,C)`	⇒	`chsH2(A,T)` `accPP2(T,B,C)`
Rewrite Swap	`swap2(A,B)`	⇒	`unpackL2(A,A,T)` `unpackH2(A,T,B)`

Figure 5.9: (Example) Transformations for AMD's K6. As the AMD K6 does not support swap operations and not all combinations of accumulate operations, they have to be rewritten.

Sequence	⇒	*Sequence'*
swap2	⇒	unpackL2, unpackH2
swap2	⇒	unpackH2, unpackL2
accNN2	⇒	unpackL2, unpackH2, sub2
accNP2	⇒	unpackH2, accPP2

Figure 5.10: Rule Set for AMD K6 Transformations. This figure contains all rewriting combinations of unsupported operations on the AMD K6.



Example Rule	*Sequence*	⇒	*Sequence'*
Rewrite Accumulate	accPP2(A,B,C)	⇒	unpackL2(A,B,T1) unpackH2(A,B,T2) add2(T1,T2,C)
Rewrite Swap	swap2(A,B) swap2(C,D) unpackL2(B,D,E)	⇒	unpackH2(A,C,E)

Figure 5.11: (Example) Transformations for Intel's Pentium 4. As Intel's Pentium 4 has no special operations for accumulate and swap in its instruction set, these operations have to be rewritten. For each type of unsupported instructions, an example is given.

Sequence	⇒	*Sequence'*
swap2, chsH2	⇒	chsL2, swap2
chsH2, swap2	⇒	swap2, chsL2
swap2, swap2, unpackL2	⇒	swap2, swap2, unpackH2
swap2, swap2, unpackH2	⇒	swap2, swap2, unpackL2
unpackL2, swap2	⇒	unpackL2
unpackH2, swap2	⇒	unpackH2
swap2	⇒	shufpd2
accPP2	⇒	unpackL2, unpackH2, add2
accNN2	⇒	unpackL2, unpackH2, sub2
accPN2	⇒	unpackL2, unpackH2, chsL2, sub2
accNP2	⇒	unpackL2, unpackH2, chsL2, add2

Figure 5.12: Rule Set for Pentium 4 Transformations. All rewriting combinations of unsupported operations on Intel's Pentium 4.

5.4 The Scheduler

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

The scheduling phase creates a topological order on the dag, a so called *schedule*. This schedule can be executed by a sequential processor. The goal of scheduling is to minimize the variable lifetime in the resulting code to enhance locality, i. e., reduce register pressure by reducing register spills.

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

The *scheduling phase* transforms the dag into a recursive decomposition of serial and parallel subdags. A serial decomposition specifies that a subdag `D1` has dependencies to another subdag `D2` and therefore `D1` has to be executed before `D2`. In a parallel decomposition the relative execution order of a subdag is not specified.

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

A more detailed description of the scheduler besides an example illustrating its functionality can be found in [22].

Code Motion Transformation

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

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

Instruction Neighbor. An instruction `X` is neighbored to an instruction `Y` when `X` is directly followed by `Y` in a subdag of the schedule or vice versa.

Instruction Sequence. An instruction sequence `<I1,...,In>` consists of `n` neighboring instructions.

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

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

Moving an Instruction Down. Consider an instruction sequence `<I1,I2>` where `I1` and `I2` are neighbors. Instruction `I1` is moved down in the sequence leading to the new sequence `<I2,I1>` if and only if all source operands of the instruction `I2` are neither source nor target operands of instruction `I1`.

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

As the code dealt with is of SSA form, for the reasons described in Appel [9], the code motion process can be simplified. In the up shifts, it is not necessary to check for the destination of `I1` being source or destination of `I2`. In the down shifts, it is not necessary to check for the destination of `I2` being source or destination of `I1`.

Chapter 6

Performance Assessment of MAP

This chapter presents numerical experiments carried out to demonstrate the applicability and the performance boosting effects of the Vienna MAP vectorizer and backend tools [46, 48].

6.1 The Vienna MAP Vectorizer and Backend

The Vienna MAP Vectorizer is the main subject of this thesis and is explained in detail in Chapters 4 and 5. It automatically extracts parallelism out of a sequence of operations in static single assignment (SSA) straight-line code while maintaining data locality and utilizing special features of short vector SIMD extensions.

MAP provides vectorized code either (*i*) via a source-to-source transformation producing macros compliant with a portable SIMD API (see Chapter 3) and additionally providing support for FMA instructions, or (*ii*) via a source-to-assembly transformation utilizing the Vienna MAP backend.

The Vienna MAP Backend [47, 48] generates assembly code optimized for short vector SIMD hardware. It is able to exploit special features of automatically generated straight-line codes. The MAP backend is included in the most current version of FFTW, FFTW-GEL. It currently supports assembly code for x86 with 3DNow! or SSE2.

The MAP vectorizer and backend have been combined with leading-edge self-tuning numerical software—FFTW, SPIRAL, and ATLAS (see Chapter 1.2)—resulting in high-performance SIMD implementation of DSP and linear algebra codes as presented in the following.

In the following sections, experimental evidence is given for the performance gain unleashed by the MAP vectorizer and backend.

High-performance implementations of DSP algorithms have been obtained by extending SPIRAL and FFTW to support short vector SIMD extensions.

It has been demonstrated, that the Vienna MAP vectorizer provides one of the fastest FFTs that are currently available on x86 processors featuring 3DNow! and SSE2. In addition, these methods provide the only FFTs for BlueGene/L's PowerPC 440 FP2 processor taking full advantage of its double FPU (see [2] and Appendix A). Formal vectorization and the Vienna MAP vectorizer leverage the only vectorized implementations for general DSP transforms like DST and 2D-DCT that are automatically tuned. Moreover, the only fully automatically

vectorized ATLAS kernels are obtained utilizing the Vienna MAP vectorizer.

Instruction statistics show that in most cases the Vienna MAP vectorizer reduces the number of arithmetic instructions significantly (up to 50 %). In some cases, however, a considerable amount of reorder instructions is required, thus slightly increasing the overall instruction count. On modern processors like the Pentium 4 this is not a critical issue, as floating-point and reordering instructions can be carried out in parallel by specialized execution units providing an overall speed-up anyway.

Furthermore, the Vienna MAP backend accelerates automatically generated DSP code by up to 25 %, compared to standard C compilers like the GNU C compiler. This allows code vectorized by the Vienna MAP vectorizer to achieve a *superlinear* speed-up value of 2.2 for two-way SIMD extensions. Using the Intel C++ compiler, speed-up values of up to 1.8 are reached using the MAP vectorizer.

6.2 Experimental Setup

Numerical experiments on machines featuring different short vector extensions were conducted: (*i*) a prototype of BlueGene/L's PowerPC 440 FP2 running at 500 MHz featuring a *double FPU*, (*ii*) a Pentium 4 featuring SSE and SSE 2 running at 1.8 GHz and, (*iii*) a 800 MHz Athlon Thunderbird featuring 3DNow!.

3DNow!, SSE2, and the *double FPU* are 2-way vector extensions (for single-precision and double-precision, respectively) providing a theoretical speed-up of two. The *double FPU* additionally provides fused multiply-add (FMA) instructions.

First, the results of the experiments carried out on IBM's BlueGene/L will be presented. Then the two IA-32 machines, i. e., the 1.8 GHz Pentium 4 and the 1.53 GHz Athlon XP 1800+ will be assessed. The latter machines are interesting to experiment with as their processors have different cache architectures and the investigated codes fit into data caches.

FFT performance is displayed in *pseudo Gflop/s*, i. e., $5N \log_2 N/T$ (in nanoseconds) while actual performance is displayed for all other transforms.

6.3 BlueGene/L Experiments

Very recently, experiments were carried out on a prototype of IBM's BlueGene/L (BG/L) top performance supercomputer. In tandem with SPIRAL, the Vienna MAP vectorizer was evaluated on this prototype. Performance data of 1D FFTs with vector lengths $N = 2^2, 2^3, \ldots, 2^{10}$ were obtained on one single PowerPC 440 FP2 running at 500 MHz.

The following FFT implementations were tested: (*i*) The best vectorized code found by SPIRAL utilizing formal vectorization, (*ii*) the best vectorized code found

by SPIRAL utilizing the Vienna MAP vectorizer, (*iii*) the best scalar FFT implementation found by SPIRAL (XL C's vectorizer and FMA extraction turned off), (*iv*) the best vectorized FFT implementation found by SPIRAL using the XL C compiler's vectorizer and FMA extraction, and (*v*) the mixed-radix FFT implementation provided by the GNU scientific library (GSL). Figures 6.1 and 6.2 display the respective performance and speed-up data.

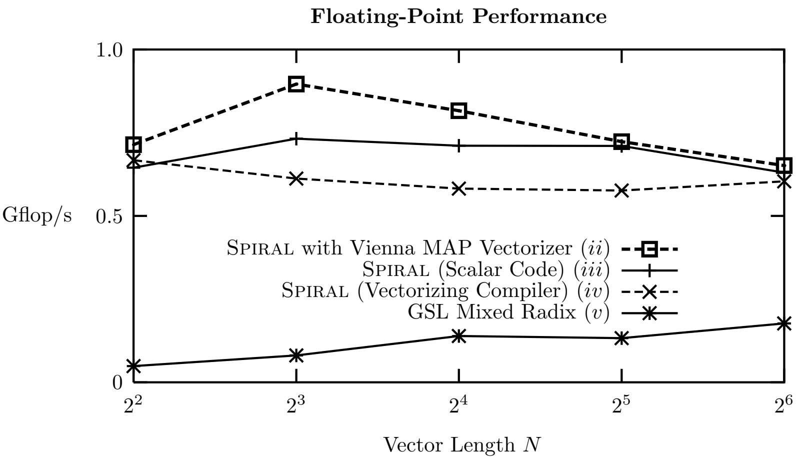


Figure 6.1: Floating-point performance of the Vienna MAP vectorizer in tandem with SPIRAL compared to another vectorized FFT implementation and scalar implementations on BlueGene/L's PowerPC 440 FP2 running at 500 MHz.

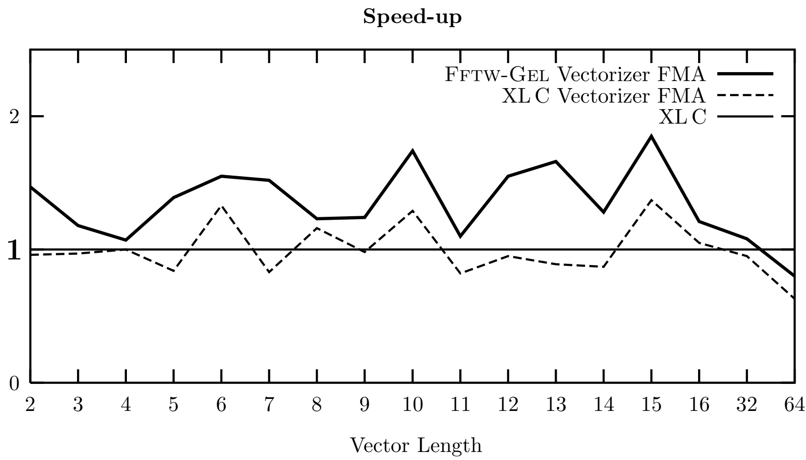


Figure 6.2: Speed-up of (*i*) the newly developed BlueGene/L MAP vectorizer *with* FMA extraction but without backend optimizations and (*ii*) FFTW codelets vectorized by XLC *with* FMA extraction, compared to (*iii*) scalar FFTW codelets using XLC *without* FMAs and *without* vectorization. The experiment has been carried out running no-twiddle codelets.

The best scalar codes found by SPIRAL serve as baseline for the assessment of the various vectorization techniques. The tested codes are very fast scalar implementations that do not utilize FMA instructions. The Vienna MAP vectorizer is restricted to problem sizes that can be fully unrolled fitting into instruction cache and the resulting code is such that it can be handled well by the XL C compiler's register allocator. This is the case for problem sizes $N \leq 32 = 2^5$. The third-party GNU GSL FFT library reaches about 30 % of the performance of the best scalar SPIRAL generated code and thus shows a very disappointing performance. XL C's vectorization and FMA extraction produces code which is 15 % slower than scalar XL C without FMA extraction, i. e., XL C's techniques used to vectorize straight-line code do not handle SPIRAL generated FFT codes well.

Fig. 6.2 shows the relative performance (speed-up) of FFTW 2.1.5 no-twiddle codelets vectorized using the Vienna MAP vectorizer compared to scalar FFTW codelets and codelets vectorized by IBM's XL C compiler.

IBM's XL C compiler for BlueGene/L using code generation *with* SIMD vectorization and *with* FMA extraction (using the compiler techniques introduced in [49]) sometimes accelerates the code slightly but also slows down the code in some cases. MAP's vectorization yields speed-up values up to 1.8 for sizes where the XL C compiler's register allocator generates reasonable code. For codes with more than 1,000 lines (vector lengths 16, 32, 64) the performance degrades because of the lack of efficient register allocation.

6.4 Experiments on IA-32 Architectures

This section assesses the Vienna MAP vectorizer on IA-32 architectures. Experimental evidence is provided that MAP successfully vectorizes a large class of straight-line codes, including SPIRAL generated codes, FFTW codelets, and ATLAS kernels.

MAP has been investigated on two IA-32 compatible machines: (*i*) one with an Intel Pentium 4 processor featuring SSE 2 two-way double-precision SIMD extension and (*ii*) one with an AMD Athlon Thunderbird featuring 3DNow! professional two-way single-precision SIMD extension.

All codes were generated using the GNU C compiler 2.95.2 and the GNU assembler 2.9.5.

Performance data are displayed in pseudo Gflop/s, i. e., $5N \log N/T$ (in nanoseconds) for complex-to-complex and $2.5N \log N/T$ for real-to-halfcomplex FFTs.

6.4.1 MAP Vectorization and Backend Applied to FFTW Codelets

MAP was connected to FFTW leading to the AMD specific K7/FFTW-GEL and the Pentium 4 specific P4/FFTW-GEL. In the newest release of FFTW, MAP for AMD machines has been included.

It turned out in the experiments that the maximum speed-up value achievable by vectorization (ignoring other effects like smaller code size, wider register files, etc.) is two on both test machines. However, additional backend optimization leads to further performance improvement of the generated codes.

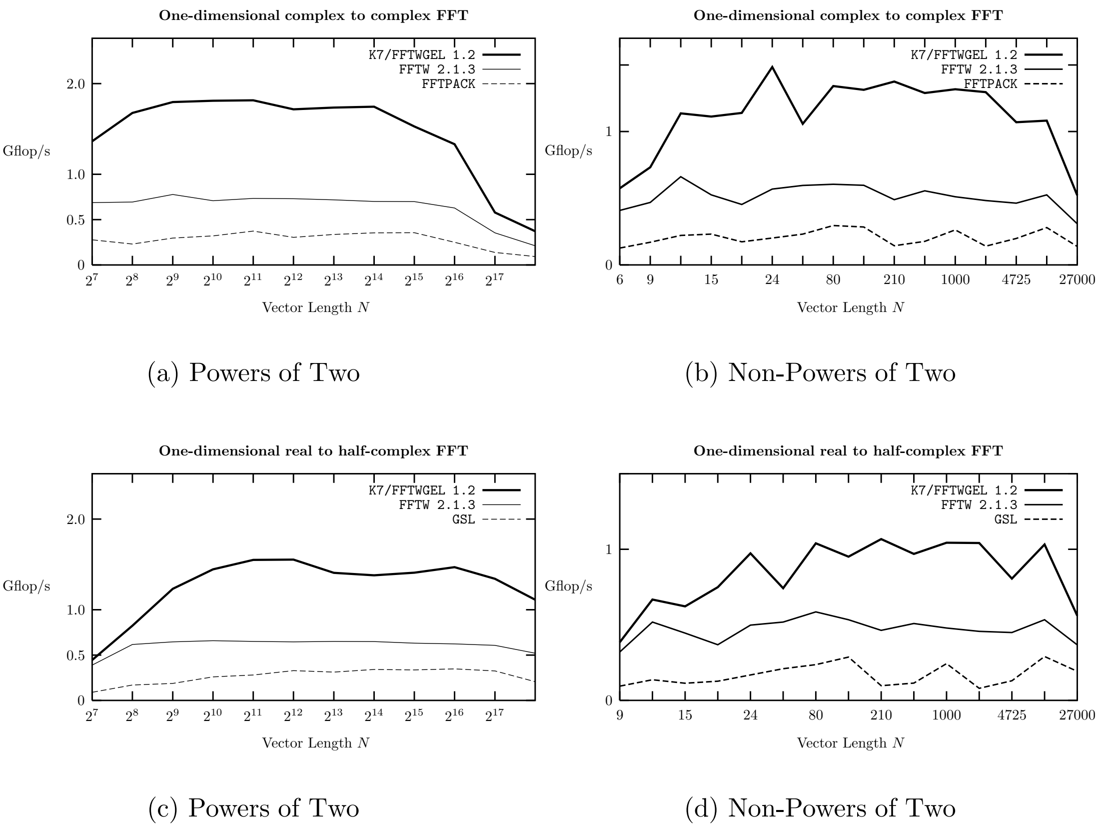
Both complex-to-complex FFTs and real-to-halfcomplex FFTs for power of two and non-powers of two problem sizes were evaluated using BENCHFFT. Real-to-halfcomplex FFTs are notoriously hard to vectorize (especially for vector lengths being non-powers of two) due to their much more complicated algorithmic structure compared to complex-to-complex FFT algorithms.

Performance on the AMD Athlon. Figures 6.3 (a)–(d) illustrate the performance of K7/FFTW-GEL using codelets vectorized and assembled by MAP on an Athlon Thunderbird utilizing 3DNow! which provides two-way single-precision SIMD instructions. Both the vectorizer and the backend were used. Standard C code generated by scalar FFTW, FFTPACK, and GSL demonstrates the performance boosting effect of K7/FFTW-GEL.

More specifically, Figures 6.3 (a)–(b) display the performance in pseudo Gflop/s of complex-to-complex FFTs. (a) refers to power of two and (b) to non-powers of two vector lengths. Figures 6.3 (c)–(d) display the corresponding results for real-to-halfcomplex FFTs. (c) refers to power of two and (d) to nonpowers of two vector lengths. In the experiments underlying Figures 6.3 (a)–(d) the data sets fit into L2 cache.

FFTW is up to two times faster than FFTPACK, the industry standard in non-hardware-adaptive FFT software. Fig. 6.3 (a) shows that for most problem sizes of complex-to-complex power of two FFTs FFTW-GEL is about twice as fast as FFTW with a peak performance of nearly 2 pseudo Gflop/s. In Fig. 6.3 (b), the complex-to-complex non-power of two FFTs of K7/FFTW-GEL are about twice as fast as FFTW with a peak performance of nearly 1.4 pseudo Gflop/s. Fig. 6.3 (c) shows that for most problem sizes of real-to-halfcomplex power of two FFTs FFTW-GEL is about twice as fast as FFTW with a peak performance of nearly 1.5 pseudo Gflop/s. Fig. 6.3 (d) shows that for real-to-halfcomplex non-power of two FFTs FFTW-GEL is about twice as fast as FFTW with a peak performance of nearly 1.2 pseudo Gflop/s.

Performance on the Intel Pentium 4. Figures 6.4 (a)–(d) illustrate the performance of P4/FFTW-GEL on the Pentium 4. P4/FFTW-GEL utilizes the two-way double-precision SIMD operations provided by SSE 2. Standard C code generated by scalar FFTW and FFTPACK enables the assessment of P4/FFTW-GEL.



(a) Powers of Two

(b) Non-Powers of Two

(c) Powers of Two

(d) Non-Powers of Two

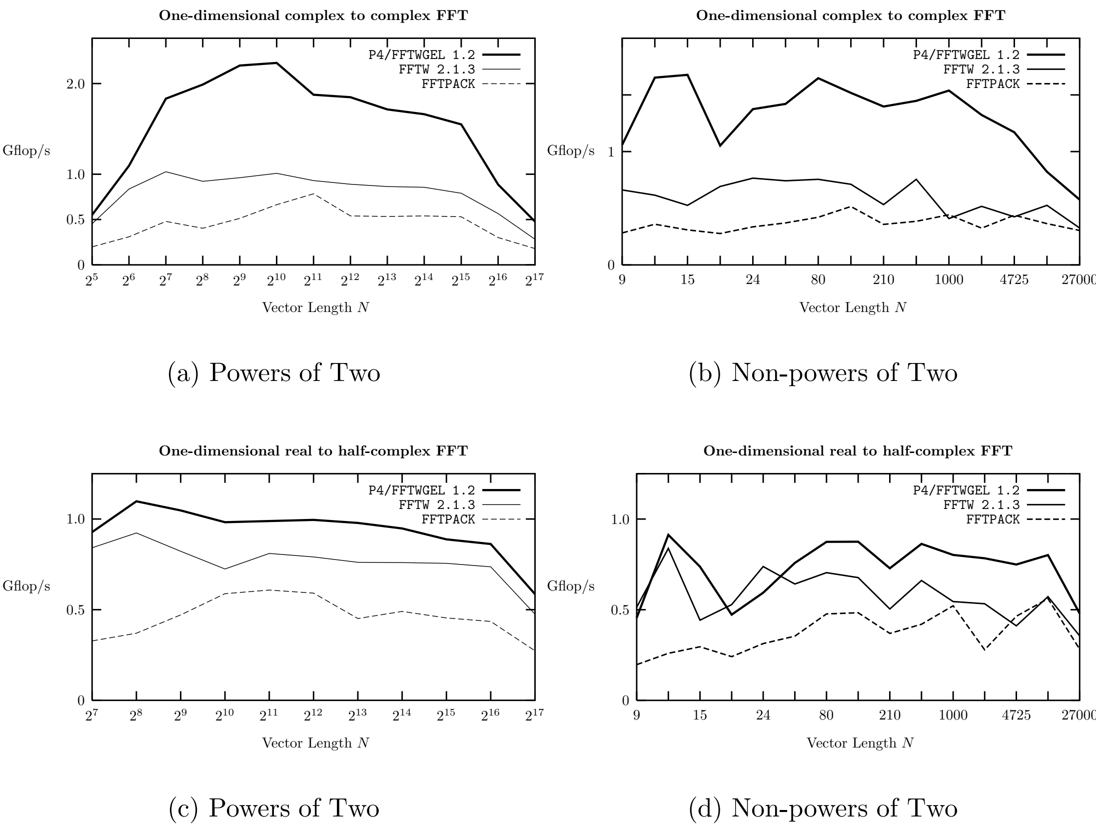
Figure 6.3: Floating-point performance of K7/Fftw-Gel (3DNow!) compared to Fftpack and Fftw 2.1.3 on a 800 MHz AMD Athlon Thunderbird carrying out complex-to-complex and real-to-halfcomplex FFTs in single-precision both of power of two and nonpowers of two problem sizes.

Figures 6.4 (a)–(b) display the performance in pseudo Gflop/s of complex-to-complex FFTs. (a) refers to power of two and (b) to non-powers of two transform sizes. Figures 6.4 (c)–(d) display the runtimes of real-to-halfcomplex FFTs. (c) refers to power of two and (d) to non-powers of two transform sizes. In the experiments underlying Figures 6.4 (a)–(d) the data sets fit into L2 cache.

Fig. 6.4 (a) shows that a speed-up of up to 2.2 has been achieved for complex FFTs of powers of two (including the performance boost originating from the backend). This is leading to FFTs running at 2.2 pseudo Gflop/s on a 1.8 GHz Pentium 4, utilizing two-way SIMD extensions.

The performance of the code is best within L1 cache and decreases outside L1. This happens to be the case only for vector lengths $N = 2^9$ and 2^{10} due to the size of the Pentium 4's very fast data cache.

Fig. 6.4 (b) shows that for complex-to-complex non-power of two FFTs Fftw-Gel is about twice as fast as Fftw 2.1.3 with a peak performance of nearly 1.2 pseudo Gflop/s. Fig. 6.4 (c) shows that the real-to-halfcomplex FFTs of powers



(a) Powers of Two

(b) Non-powers of Two

(c) Powers of Two

(d) Non-powers of Two

Figure 6.4: Floating-point performance of P4/Fftw-Gel (SSE 2) compared to Fftw (double-precision) on an Intel Pentium 4 running at 1.8 GHz carrying out complex-to-complex and real-to-halfcomplex FFTs having problem sizes of both power of two and nonpowers of two.

of two produced by Fftw already have a remarkable peak performance of nearly 1 pseudo Gflop/s. Nevertheless, P4/Fftw-Gel is up to 30 % faster than Fftw 2.1.3. The performance improvement is primarily due to backend optimization as vectorization cannot yield a significant improvement in this case. Fig. 6.4 (*d*) illustrates that for real-to-halfcomplex non-powers of two FFTs, Fftw-Gel is up to 60 % faster than Fftw 2.1.3.

6.4.2 MAP Vectorization and Backend Applied to SPIRAL Generated Codes

The Vienna MAP compiler was investigated on a Pentium 4 running at 1.8 GHz using DFTs, DCTs, DSTs and WHTs generated by Spiral. Fig. 6.5 shows that MAP significantly reduces the number of arithmetic instructions. Vectorization also introduces a significant number of data reordering operations, but these can be executed in parallel with the arithmetic operations on the P4.

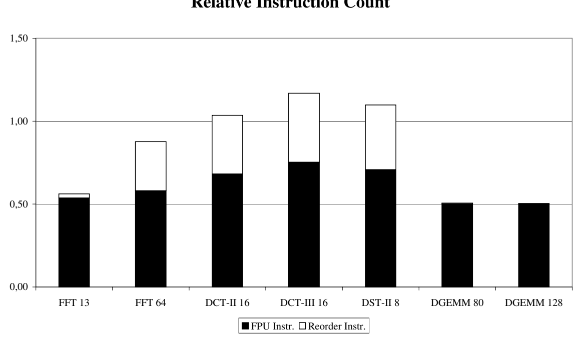


Figure 6.5: Relative instruction count of various DSP transforms and linear algebra routines. The number of arithmetic instructions is significantly reduced by the Vienna MAP vectorizer.

To provide evidence that register allocation in standard compilers cannot handle large straight-line code, the Vienna MAP backend was compared with the Intel C++ compiler's backend using DFTs (unit stride memory access) of size 4, 8,. . . , 64 that require up to 1,300 floating-point instructions. DFT codes were generated and optimized by Spiral using (*i*) the MAP vectorizer and the MAP backend and (*ii*) the MAP vectorizer outputting C code with intrinsics. In the second case, the Intel C++ compiler's backend was utilized. As largest code in this experiment, DFT_{64} features approximately 2000 lines of SSA straight-line code. This experiment only assesses register allocation and instruction scheduling, as unit-stride code versions were used.

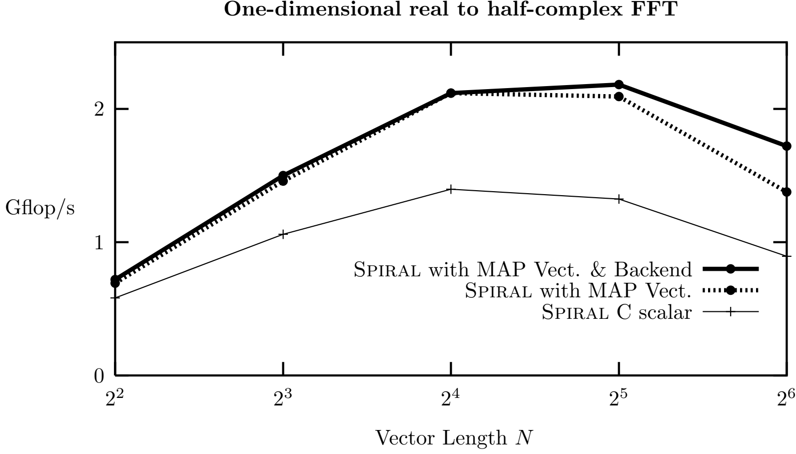


Figure 6.6: Performance of the Intel C++ compiler compared to the Vienna MAP backend for a Spiral generated one dimensional FFT on an Intel Pentium 4 with 1.8 GHz.

These experiments show that code generated using the MAP backend maintains its performance level even for larger problem sizes while the performance of DFT_{64} code compiled by Intel's C++ compiler degrades by 25 % (see Fig. 6.6).

6.4.3 MAP Vectorization of ATLAS Kernels

The Vienna MAP vectorizer is able to deal with Atlas kernels leading to the same instruction count (arithmetic operations, data reorder operations, loads and stores) as the semi-automatically produced (hand-coded) SSE2 and 3DNow! kernels contributed to the Atlas project by various people. However, it can automatically vectorize linear algebra code for kernel sizes that were not yet hand-coded by contributors but would be required for top performance when machine parameters like cache sizes change. Fig. 6.5 shows that the number of arithmetic instructions in `dgemm` kernels is halved by the MAP vectorizer and hardly any reorder instructions are needed.

Conclusion

This thesis presents special purpose compilation techniques targeting the SIMD vectorization of straight-line DSP and linear Algebra code; both in the context of automatic performance tuning.

The leading automatic performance tuning systems Atlas, Fftw, and Spiral hardware independently provide highly performant scalar implementations of numerical algorithms. SIMD short vector hardware support was provided, at the most, by hand written kernels utilizing SIMD instructions. The newly introduced compiler technology helps to achieve the same level of performance as do hand-tuned vendor libraries while accomplishing performance portability.

In conjunction with the software packages mentioned above, SIMD vectorized high performance implementations of FFTs, general DSP transforms, and Blas kernels have been obtained. Some of the techniques described in this thesis have been included in the current release of the industry-standard numerical library Fftw and will become part of IBM's numerical library for BlueGene/L supercomputers.

The Vienna MAP vectorizer, relying on the techniques introduced in this thesis, provides the fastest FFTs currently available on x86 architectures featuring 3DNow! or SSE 2 SIMD extensions. In addition, these methods rendered possible the only currently existing FFT software for BlueGene/L's PowerPC 440 FP2 processors taking full advantage of their double FPU. The Vienna MAP vectorizer leveraged the so far only vectorized implementation of automatically tuned general DSP transforms like DCTs, DSTs, and multidimensional DSP transforms. Moreover, the Vienna MAP vectorizer has produced the only *fully automatically vectorized* Atlas kernels so far.

Appendix A

The BlueGene/L Architecture

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

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

The BlueGene/L machine [2] will be built from 65,536 PowerPC 440 FP2 processors connected by a 3D torus network leading to 360 Tflop/s peak performance. The Earth Simulator, currently leading the Top 500 list, provides 40 Tflop/s peak performance. A small prototype of the BlueGene/L machine was built recently. In contrast to BlueGene/L's 700 MHz target frequency, the current prototype runs at 500 MHz.

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

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

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

Appendix B

The Kronecker Product Formalism

This chapter introduces the formalisms of Kronecker products (tensor products) and stride permutations, which are the foundations of most algorithms for discrete linear transforms. This includes various FFT algorithms, the Walsh-Hadamard transform, different sine and cosine transforms, wavelet transforms as well as all multidimensional linear transform.

Kronecker products allow to derive and modify algorithms on the structural level instead of using properties of index values in the derivation process. The Kronecker product framework provides a rich algebraic structure which captures most known algorithms for discrete linear transforms. Both iterative as well as recursive algorithms are captured. Most proofs in this section are omitted. They can be found in Van Loan [65].

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. [44]). 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 [54] 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 [65] 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 [62] and Johnson et al. [43] for FFT implementations on classic vector computers and Norton and Silberger [53] on parallel computers with MIMD architecture. Gupta et al. [25] and Pitsianis [55] 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. [43],

the SPIRAL project (Moura et al. [52] and Püschel et al. [57]) 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. [43] 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.

B.1 Notation

The notational conventions introduced in the following are used throughout this chapter. Integers denoting problem sizes are referred to by capital letters M, N, etc. Loop indices and counters are denoted by lowercase letters i, j, etc. General integers are denoted by k, m, n, etc. as well as r, s, t, etc.

B.1.1 Vector and Matrix Notation

In this chapter, vectors of real or complex numbers will be referred to by lowercase letters x, y, z, etc., while matrices appear as capital letters A, B, C, etc.

Parameterized matrices (where the size and/or the entries depend on the actual parameters) are denoted by upright capital letters and their parameters.

Example (Parameterized Matrices) L_8^{64} is a stride permutation matrix of size 64×64 with stride 8 (see Section B.4), T_2^8 is a complex diagonal matrix of size 8×8 whose entries are given by the parameter "2" (see Section B.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 (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 B.7).

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 (Complex Matrix) The 2-by-3 complex matrix $A \in \mathbb{C}^{2\times 3}$ is expressed as

$$A = \begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \end{pmatrix}, \quad a_{00}, \dots, a_{12} \in \mathbb{C}.$$

Rows and columns are indexed from zero.

B.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 (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 (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.

B.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 (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

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

B.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

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

In this chapter P is a permutation matrix in most cases.

Example (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 B.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*

$$P A^P = A P.$$

Property B.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*

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

B.1.5 Direct Sum of Matrices

Definition B.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*

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

B.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 B.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

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

B.2 Extended Subvector Operations

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

Definition B.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*

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

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

B.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 B.3 (Distributivity)

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

Definition B.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.*

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

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

Example (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 B.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 (B.1)*.*

Example (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.

B.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 B.4 (Read Multiplicativity)

$$\mathrm{R}_{b_1+b_2 s_1, s_1 s_2}^{N_1 N_2 n, n} = \mathrm{R}_{b_2, s_2}^{N_2 n, n} \quad \mathrm{R}_{b_1, s_1}^{N_1 N_2 n, N_2 n}.$$

The same applies to two consecutive basic write operations.

Property B.5 (Write Multiplicativity)

$$\mathrm{W}_{b_1+b_2 s_1, s_1 s_2}^{N_1 N_2 n, n} = \mathrm{W}_{b_1, s_1}^{N_1 N_2 n, N_2 n} \quad \mathrm{W}_{b_2, s_2}^{N_2 n, n}.$$

Multiplication of read and write matrices yields an identity matrix.

Property B.6 (Read Write Identity)

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

Property B.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 B.8 (Read Write Transposition)

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

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

B.3 Kronecker Products

Definition B.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*

$$A \otimes B := \begin{pmatrix} a_{0,0}B & \dots & a_{0,N_1-1}B \\ \vdots & \ddots & \vdots \\ a_{M_1-1,0}B & \dots & a_{M_1-1,N_1-1}B \end{pmatrix} \in \mathbb{C}^{M_1 M_2 \times N_1 N_2}.$$

Definition B.7 (Tensor Basis) *Set $N = N_1 N_2$ and form the set of tensor products*

$$e_i^{N_1} \otimes e_j^{N_2}, \quad i = 0, 1, \dots, N_1 - 1, \quad j = 0, 1, \dots, N_2 - 1. \tag{B.2}$$

This set is called tensor basis.

Since any element e_k^N of the standard basis (B.1) can be expressed as

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

the tensor basis of Definition B.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 $\mathbb{C}^N$, $N = N_1N_2$.

The following two special cases of Kronecker products involving identity matrices are of high importance.

Definition B.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 (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 B.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 (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:

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \end{pmatrix} := a_{0,0} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} + a_{0,1} \begin{pmatrix} x_3 \\ x_4 \\ x_5 \end{pmatrix}$$
$$\begin{pmatrix} y_3 \\ y_4 \\ y_5 \end{pmatrix} := a_{1,0} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} + a_{1,1} \begin{pmatrix} x_3 \\ x_4 \\ x_5 \end{pmatrix}.$$

B.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 B.7.

Property B.9 (Identity) *If I_m and I_n are identity matrices, then*

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

Property B.10 (Identity) *If* I_m *and* I_n *are identity matrices, then*

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

Property B.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 B.12 (Transposition) *If* A, B *are arbitrary matrices, then*

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

Property B.13 (Inversion) *If* A *and* B *are regular matrices, then*

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

Property B.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 B.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 B.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 B.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 B.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.

B.4 Stride Permutations

Definition B.10 (Stride Permutation) *For a vector* $x \in \mathbb{C}^{mn}$ *with*

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

the stride permutation L_n^{mn} *is defined by its action on the tensor basis* (B.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 B.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 B.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 B.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

Combinations of tensor products and stride permutations have both vector and parallel characteristics like stride permutations and additionally feature arithmetic operations like parallel and vector Kronecker products.

The factorization of these constructs leads to the short vector Cooley-Tukey FFT.

Definition B.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 B.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.

B.4.1 Algebraic Properties of Stride Permutations

Property B.17 (Identity)

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

Property B.18 (Inversion/Transposition) *If $N = mn$ the*

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

Property B.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 (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 B.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. [43].

Several special cases are worth noting.

Corollary B.4 *If* $A \in \mathbb{C}^{m\times m}$ *and* $B \in \mathbb{C}^{n\times n}$ *then*

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

Application of this relation leads to

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

Stride permutations interchange parallel and vector Kronecker factors. The readdressing prescribed by L^{mn}_n on input and L^{mn}_m 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

$$\begin{aligned}
A \otimes B &= (A \otimes \mathrm{I}_n)(\mathrm{I}_m \otimes B)\\
&= \mathrm{L}^{mn}_m (\mathrm{I}_n \otimes A)\, \mathrm{L}^{mn}_n (\mathrm{I}_m \otimes B),
\end{aligned} \tag{B.3}$$

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

$$A \otimes B = (A \otimes \mathrm{I}_n)\, \mathrm{L}^{mn}_m (B \otimes \mathrm{I}_m)\, \mathrm{L}^{mn}_n, \tag{B.4}$$

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 B.20 *If* $A \in \mathbb{C}^{m\times m}$ *and* $B \in \mathbb{C}^{n\times n}$ *then*

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

Property B.21 *If* $N = kmn$ *then*

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

Property B.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 B.23 *If $N = kmn$ then*

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

Proof: Using Properties B.18 and B.22 lead to

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

Property B.24 *If $N = kmn$ then*

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

Proof: Using Property B.23 leads to

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

The following two properties show, how the mixed Kronecker product can be decomposed. Property B.25 shows the more general case and Property B.26 shows the full factorization.

Property B.25

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

Property B.26

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

B.4.2 Digit Permutations

The following permutation generalizes the stride permutation.

Definition B.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*

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

Theorem B.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*

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

Proof: Johnson et al. [43].

Digit reversal is a special permutation arising in FFT algorithms.

Definition B.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 B.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. [43].

B.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 B.17 (Twiddle Factor Matrix) *Let* $\omega_N = e^{2\pi i/N}$ *denote the* N*th root of unity. The twiddle factor matrix, denoted by* T_m^{mn}*, is a diagonal matrix defined by*

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

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

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 B.5 (Conjugating Diagonal Matrices) *Let*

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

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 B.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 B.27 (Twiddle Factor $\mathrm{I}_r \otimes \mathrm{T}_m^{mn}$)

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

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

$(\mathrm{I}_r \otimes \mathrm{T}_m^{mn})^P$ 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 (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

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

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

B.6 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. [43] and Moura et al. [52].

The RW Notation

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

$$y := \sum_{i=0}^{n} \mathrm{W}_{b(i),s}^{N,k}\; x, \tag{B.5}$$

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

Property B.28 (Loop Independence) *A sum of type* (B.5) *can be translated into a 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 (B.5) acts as *loop*. The kth component of y is computed by a sum where exactly *one* summand is not zero, which follows from Property B.28. 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 (B.5) 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 B.1 (update($\mathbf{y}, \mathbf{W}_{\mathbf{b,s}}^{\mathbf{N,k}}\, \mathbf{x}$))

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 B.1, equation (B.5) can be implemented as a loop using the following algorithm.

Algorithm B.2 $(\mathbf{y} := \sum_{\mathbf{i=0}}^{\mathbf{n}} \mathrm{W}^{\mathbf{N,k}}_{\mathbf{b(i),s}}\ \mathbf{x})$

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

The implementation of sums as loops by Algorithm B.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 B.3 $(\mathbf{y} := (\mathbf{A_0} \oplus \mathbf{A_1})\, \mathbf{x})$

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

Corollary B.7 (Direct Sum) *A direct sum*

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

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

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

Applying Algorithm B.2 leads to the following algorithm.

Algorithm B.4 $(\mathbf{y} := (\bigoplus_{\mathbf{i=0}}^{\mathbf{k-1}} \mathbf{A_i})\, \mathbf{x})$

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

Kronecker products with identity matrices represent loops in a natural way. According to Corollary B.1 (on page 80) 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 B.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 B.8 (Parallel Kronecker Products) *A parallel Kronecker product*

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

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 B.2 leads to the following algorithm.

Algorithm B.5 $(\mathbf{y} := (\mathbf{I_k} \otimes \mathbf{A})\,\mathbf{x})$

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 B.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 B.9 (Vector Kronecker Product) *A vector Kronecker product*

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

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 B.2 leads to the following algorithm.

Algorithm B.6 $(\mathbf{y} := (\mathbf{A} \otimes \mathbf{I_k})\,\mathbf{x})$

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 B.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 B.2 leads to the following algorithm featuring vector reads and parallel writes.

Algorithm B.7 $(\mathbf{y} := \mathbf{L}_{\mathbf{m}}^{\mathbf{mn}}\, \mathbf{x})$

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 B.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 B.2 leads to the following algorithm featuring parallel reads and vector writes.

Algorithm B.8 $(\mathbf{y} := \mathbf{L}_{\mathbf{m}}^{\mathbf{mn}}\, \mathbf{x})$

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 B.7 and B.8 and Algorithms B.5 and B.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 be 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 B.4 (on page 83) 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 B.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 B.2 leads to the following algorithm.

Algorithm B.9 $(\mathbf{y} := (\mathbf{I_k} \otimes \mathbf{A})\mathbf{L_k^{mk}}\,\mathbf{x})$

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 B.12 has the same access pattern as Algorithm B.7, i. e., it reads with vector characteristics and writes with parallel characteristics.

B.6.1 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 B.1 shows an example SPL program.

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:

```
(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_n^{mn}, and (*iv*) twiddle factor matrices T_n^{mn}. 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. [69]). For example, the DFT_4 algorithm given in equation (B.6) on page 95 can be written in SPL as represented in Figure B.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 1.1 on page 9) to try different implementations of the same formula.

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

Figure B.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.

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.

B.7 Fast Algorithms for Discrete Linear Transforms

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 B.18 (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 B.19 (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 is the discrete Fourier transform (DFT), which, for size N, is given by the following definition.

Definition B.20 (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 (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 B.21 (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 B.

Example ($\mathbf{DFT_4}$) Consider a factorization, i. e., a fast algorithm, for DFT_4. Using the mathematical notation from Chapter B it follows that

$$\mathrm{DFT}_4 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{pmatrix}$$

$$= \left(\begin{array}{cc|cc} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ \hline 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{array}\right) \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & i \end{pmatrix} \cdot \left(\begin{array}{cc|cc} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ \hline 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \end{array}\right) \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{B.6}$$

$$= (\mathrm{DFT}_2 \otimes \mathrm{I}_2) \cdot \mathrm{T}_2^4 \cdot (\mathrm{I}_2 \otimes \mathrm{DFT}_2) \cdot \mathrm{L}_2^4 .$$

T_2^4 denotes a twiddle matrix as defined in Section B.5, i. e.,

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

L_2^4 denotes a stride permutation as defined in Section B.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 an Püschel [15] 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 Püschel [14]), a library for the computer algebra system Gap [63] used in Spiral. Arep is able to factorize transform matrices and to find fast algorithms automatically. In Püschel [56] 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 [59]), as well as the fast Fourier transform (Cooley and Tukey [11]).

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 [66]).

Example (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

$$\mathrm{WHT}_{2^k} = \prod_{i=1}^{k} \left(\mathrm{I}_{2^{k_1+\cdots+k_{i-1}}} \otimes \mathrm{WHT}_{2^{k_i}} \otimes \mathrm{I}_{2^{k_{i+1}+\cdots+k_t}} \right), \quad k = k_1 + \cdots + k_t. \tag{B.7}$$

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

$$\mathrm{DFT}_{mn} = \left(\mathrm{DFT}_m \otimes \mathrm{I}_n \right) \mathrm{T}_n^{mn} \left(\mathrm{I}_m \otimes \mathrm{DFT}_n \right) \mathrm{L}_m^{mn}. \tag{B.8}$$

(B.8) is the Cooley-Tukey FFT written in the notation of Chapter B (Johnson et al. [43]).

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 B.1. Using the respective property of the tensor product, the rule

$$M \otimes M = (M \otimes \mathrm{I}_N)\,(\mathrm{I}_N \otimes M) \tag{B.9}$$

is obtained.

The set of rules used in SPIRAL is constantly growing. A set of important rules can be found in Püschel et al. [57] and Püschel et al. [58].

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 , 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. However, the general framework of having recursive rules and base cases is intrinsic to all approaches.

Example (Fully Expanded Formula for $\mathbf{WHT}_8$) According to rule (B.7), 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 B.2 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.

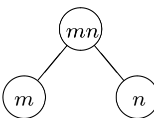


Figure B.2: Tree representation of a discrete linear transform of size $N = mn$ with one recursion step applied.

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 B.3 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.

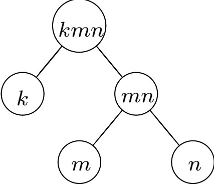


Figure B.3: Right-expanded tree, two recursive steps.

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 (B.8) 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 (B.8) 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.

References

[1] A. V. Aho, R. Sethi, J. D. Ullman: *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., 1986.

[2] G. Almasi, R. Bellofatto, J. Brunheroto, C. Causcaval, J. G. Castanos, L. Ceze, P. Crumley, C. Ch. Erway, J. Gagliano, D. Lieber, X. Martorell, J. E. Moreira, A. Sanomiya, and K. Strauss, “An Overview of the Blue-Gene/L System Software Organization,” in *Proceedings of the Euro-Par ’03 Conference on Parallel and Distributed Computing LNCS 2790*, 2003.

[3] AMD Corporation: *3DNow! Technology Manual*, 2000.

[4] AMD Corporation: *3DNow! Instruction Porting Guide Application Note*, 2002a.

[5] AMD Corporation: *AMD Athlon Processor x86 Code Optimization Guide*, 2002b.

[6] AMD Corporation: *AMD Extensions to the 3DNow! and MMX Instruction Sets Manual*, 2002c.

[7] AMD Corporation: *AMD x86-64 Architecture Programmers Manual, Volume 1: Application Programming*. Order Number 24592, 2002d.

[8] AMD Corporation: *AMD x86-64 Architecture Programmers Manual, Volume 2: System Programming*. Order Number 24593, 2002e.

[9] A. W. Appel: *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[10] 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.

[11] J. W. Cooley, J. W. Tukey: *An Algorithm for the Machine Calculation of Complex Fourier Series*. Math. Comp. 19 (1965), pp. 297–301.

[12] 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.

[13] 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.

[14] S. Egner, M. Püeschel: *The* AREP *WWW Home Page*, 2000. `www.ece.cmu.edu/~smart/arep/arep.html`

[15] S. Egner, M. Püschel: *Symmetry-Based Matrix Factorization*. Journal of Symbolic Computation, *to appear*.

[16] F. Franchetti, S. Kral, J. Lorenz, M. Püschel, C.W. Ueberhuber, P. Wurzinger, *Automatically Optimized FFT Codes for the BlueGene/L Supercomputer*, submitted to *6th International Meeting on High Performance Computing for Computational Science (VecPar '04)*

[17] F. Franchetti, M. Püschel, *Short Vector Code Generation for the Discrete Fourier Transform*, in *Proc. International Parallel and Distributed Processing Symposium (IPDPS'03)*, 2003, pp. 58–67.

[18] F. Franchetti and M. Püschel, *A SIMD Vectorizing Compiler for Digital Signal Processing Algorithms*, in *Proc. International Parallel and Distributed Processing Symposium (IPDPS'02)*, 2002, pp. 20–26.

[19] 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.

[20] 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.

[21] M. Frigo, S. G. Johnson: *FFTW: An Adaptive Software Architecture for the FFT*. In *Proceeding of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, Vol. 3, pp. 1381–1384.

[22] M. Frigo, S. Kral: *The Advanced FFT Program Generator GENFFT*. Technical Report AURORA TR2001-03, Institute for Applied Mathematics and Numerical Analysis, Vienna University of Technology, 2001.

[23] 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.

[24] G. H. Golub, C. F. Van Loan: *Matrix Computations*, 2nd edn. Johns Hopkins University Press, Baltimore, 1989.

[25] 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.

[26] G. Haentjens: *An Investigation of Cooley-Tukey Decompositions for the FFT*. Masters Thesis, Electrical and Computer Engineering Department, Carnegie Mellon University, Pittsburgh, PA, 2000.

[27] Intel Corporation: *Survey of Pentium Processor Performance Monitoring Capabilities & Tools*. App. Note, 1996.

[28] Intel Corporation: *AP-808 Split Radix Fast Fourier Transform Using Streaming SIMD Extensions*, 1999a.

[29] Intel Corporation: *AP-833 Data Alignment and Programming Issues for the Streaming SIMD Extensions with the Intel C/C++ Compiler*, 1999b.

[30] Intel Corporation: *AP-931 Streaming SIMD Extensions—LU Decomposition*, 1999c.

[31] Intel Corporation: *Intel Architecture Optimization—Reference Manual*, 1999d.

[32] Intel Corporation: *Desktop Performance and Optimization for Intel Pentium 4 Processor*, 2002a.

[33] Intel Corporation: *Intel Architecture Software Developer's Manual—Volume 1: Basic Architecture*, 2002b.

[34] Intel Corporation: *Intel Architecture Software Developer's Manual—Volume 2: Instruction Set Reference*, 2002c.

[35] Intel Corporation: *Intel Architecture Software Developer's Manual—Volume 3: System Programming*, 2002d.

[36] Intel Corporation: *Intel C/C++ Compiler User's Guide*, 2002e.

[37] Intel Corporation: *Intel Itanium 2 Processor Reference Manual for Software Development and Optimization*, 2002f.

[38] Intel Corporation: *Intel Itanium Architecture Software Developer's Manual Vol. 1 rev. 2.1: Application Architecture*, 2002g.

[39] Intel Corporation: *Intel Itanium Architecture Software Developer's Manual Vol. 2 rev. 2.1: System Architecture*, 2002h.

[40] Intel Corporation: *Intel Itanium Architecture Software Developer's Manual Vol. 3 rev. 2.1: Instruction Set Reference*, 2002i.

[41] Intel Corporation: *Intel Itanium Processor Reference Manual for Software Optimization*, 2002j.

[42] Intel Corporation: *Math Kernel Library*, 2002k.
`http://www.intel.com/software/products/mkl`

[43] 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.

[44] R. W. Johnson, C. H. Huang, J. R. Johnson: *Multilinear Algebra and Parallel Programming*. J. Supercomputing 5 (1991), pp. 189–217.

[45] S. Kral: FFTW-GEL, 2002.
`www.fftw.org/~skral`

[46] S. Kral, F. Franchetti, J. Lorenz, and C. W. Ueberhuber, "SIMD Vectorization of Straight Line FFT Code," in *Proceedings of the Euro-Par '03 Conference on Parallel and Distributed Computing LNCS 2790*, 2003, pp. 251–260.

[47] S. Kral, F. Franchetti, J. Lorenz, C. W. Ueberhuber: *Backend Optimization for Straight Line Code*. Technical Report AURORA TR2003-11, Institute for Applied Mathematics and Numerical Analysis, Vienna University of Technology, 2003a.

[48] S. Kral, F. Franchetti, J. Lorenz, C. W. Ueberhuber, P. Wurzinger, *FFT Compiler Techniques*, submitted to *13th International Conference on Compiler Construction.*

[49] S. Larsen, S. Amarasinghe: *Exploiting superword level parallelism with multimedia instruction sets*. ACM SIGPLAN Notices 35 (2000)(5), pp. 145–156.

[50] Motorola Corporation: *AltiVec Technology Programming Environments Manual*, 2000a.

[51] Motorola Corporation: *AltiVec Technology Programming Interface Manual*, 2000b.

[52] 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.

[53] 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.

[54] M. C. Pease: *An Adaptation of the Fast Fourier Transform for Parallel Processing*. Journal of the ACM 15 (1968), pp. 252–264.

[55] N. P. Pitsianis: *The Kronecker Product in Optimization and Fast Transform Generation*. Ph. D. Thesis, Department of Computer Science, Cornell University, 1997.

[56] M. Püschel: *Decomposing Monomial Representations of Solvable Groups*. Journal of Symbolic Computation 2002, Vol. 34, No. 6, pp. 561–596.

[57] 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.

[58] 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 (2001b), *submitted*.

[59] K. R. Rao, J. J. Hwang: *Techniques & standards for image, video and audio coding*. Prentice Hall PTR, 1996.

[60] B. Singer, M. Veloso: *Stochastic Search for Signal Processing Algorithm Optimization*. In *Proc. Supercomputing 01*.

[61] Y. N. Srikant, P. Shankar: *The Compiler Design Handbook*. CRC Press LLC, Boca Ration London New York Washington D.C., 2003.

[62] C. Temperton: *Fast Mixed-Radix Real Fourier Transforms*. J. Comput. Phys. 52 (1983), pp. 340–350.

[63] The GAP Group: GAP—*Groups, Algorithms, and Programming, Version 4.2*, 2000.
www-gap.dcs.st-and.ac.uk/~gap

[64] C. W. Ueberhuber: *Numerical Computation*. Springer-Verlag, Berlin Heidelberg New York Tokyo, 1997.

[65] C. F. Van Loan: *Computational Frameworks for the Fast Fourier Transform*. SIAM Press, Philadelphia, 1992.

[66] 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.

[67] R. C. Whaley, J. J. Dongarra: *Automatically Tuned Linear Algebra Software*. LAPACK Working Note 131, 1997.

[68] R. C. Whaley, A. Petitet, J. J. Dongarra: *Automated Empirical Optimizations of Software and the* ATLAS *Project*. Parallel Comput. 27 (2001), pp. 3–35.

[69] 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.