Automatic Generation of SIMD DSP Code

Franz Franchetti[*]
Markus Püschel[†]

AURORA TR2001-17

[*]Institute for Applied and Numerical Mathematics
Technical University of Vienna
Wiedner Hauptstrasse 8-10, A-1040 Wien, Austria

E-Mail: franz.franchetti@tuwien.ac.at

[†]Department of Electrical and Computer Engineering
Carnegie Mellon University
5000 Forbes Ave, Pittsburgh, PA, 15213, USA

E-Mail: pueschel@ece.cmu.edu

This work described in this report was supported by the Special Research Program SFB F011 "AURORA" of the Austrian Science Fund FWF and by DARPA through research grant DABT63-98-1-0004 administered by the Army Directorate of Contracting.

Abstract

Short vector SIMD instructions on recent microprocessors, such as SSE on Pentium III and 4, speed up code but are a major challenge to software developers. This report introduces a compiler that automatically generates C code enhanced with short vector instructions for digital signal processing (DSP) transforms, such as the fast Fourier transform (FFT).

The input to the compiler is a concise mathematical description of a DSP algorithm in the language SPL. SPL is used in the SPIRAL system (http://www.ece.cmu.edu/∼spiral) to generate highly optimized architecture adapted implementations of DSP transforms.

Interfacing the newly developed compiler with SPIRAL yields speed-ups of up to a factor of 2 in several important cases including the FFT and the discrete cosine transform (DCT) used, for instance, in the JPEG compression standard. For the FFT the automatically generated code is competitive with the hand-coded Intel Math Kernel Library (MKL).

Contents

Chapter 1

Introduction

Most major vendors of general purpose microprocessors have included short vector SIMD (single instruction multiple data) extensions into their instruction set architecture (ISA) to improve the performance of multi-media applications by exploiting the parallelism inherent to most multi-media kernels. Examples of SIMD extensions supporting both integer operations and floating-point operations include the Intel Streaming SIMD Extensions (SSE and SSE2), AMD 3DNow! (plus extensions) and the Motorola AltiVec extension. Each of these ISA extensions is based on the packing of large registers (64 bits or 128 bits) with smaller data types and providing instructions for the parallel operation on these subwords within one register.

SIMD extensions have the potential to speed up implementations in application areas where performance is crucial and the algorithms used exhibit the fine-grain parallelism necessary for using SIMD instructions. One example of such an application area is digital signal processing (DSP), which is at the heart of modern telecommunication. The computationally most intensive parts in DSP are performed by DSP transforms, such as the discrete Fourier transform (DFT), and require their efficient implementation.

A very efficient implementation of the DFT is provided by the package FFTW (Frigo, Johnson [4]). A first SIMD version of FFTW has been presented in Franchetti, Karner, Ueberhuber [3] showing a substantial improvement in performance.

The implementation of arbitrary DSP transforms, including the DFT, is the target of Spiral (Moura et al. [7]). Spiral is a generator for libraries of DSP transforms. The code produced is highly optimized, and, furthermore, adapted to the given computing platform. Spiral uses a high-level mathematical framework that represents fast algorithms for DSP transforms as *formulas* in a symbolic mathematical language called SPL (signal processing language). These formulas are (i) automatically generated from a transform specification (Püschel et al. [8]), and (ii) automatically translated into optimized code in a high-level language like C or Fortran using the SPL compiler (Xiong et al. [14]). Platform adaptation is achieved by intelligently searching, for a given transform, the large space of

possible algorithms, i. e., formulas, for the fastest one (Singer, Veloso [11]). The code produced by SPIRAL is very competitive (Xiong et al. [14]).

This report introduces a SIMD vectorizing version of the SPL compiler. It is demonstrated that certain mathematical constructs used in the language SPL can be naturally mapped to vectorized code. These constructs are generalized versions of two fundamental cases that occur in virtually every DSP formula, i. e., fast DSP algorithm. Furthermore, in several important cases, including the DFT, the Walsh-Hadamard transform (WHT), and arbitrary two-dimensional transforms, the formulas are built exclusively from these constructs, and thus can be completely vectorized.

The SIMD vectorizing SPL compiler translates the vectorizable constructs into C code enhanced with machine independent SIMD macro code while the non-vectorizable parts are translated into standard floating-point C code. The SIMD macros define a set of basic operations (e. g., load/store and arithmetic operations) that can be expressed on all current short vector SIMD architectures. A machine dependent definition of these macros is used at compile time. Existing vectorizing compilers (e. g., the Intel C++ Compiler) fail to vectorize code generated by SPIRAL due to the structure of the generated code (Sereraman, Govindarajan [10]).

It is demonstrated that the automatically generated vectorized DSP code is highly competitive and substantially speeds up the code generated by SPIRAL. Speed up factors of 1.42 to 2.09 for DFTs, 1.17 to 2.07 for WHTs and 1.43 to 2.31 for two-dimensional DCTs are obtained.

Synopsis

The report is organized as follows. Chapter 2 overviews short vector SIMD extensions available on recent processors. Chapter 3 briefly describes the SPIRAL system with emphasis on the mathematical framework to capture DSP algorithms and the SPL compiler used to translate formulas into C code and Fortran code. The newly developed vectorizing version of the SPL compiler is described in detail in Chapter 4. Chapter 5 shows runtime results for DFTs, WHTs and two-dimensional DCTs on an Intel Pentium III system running at 650 MHz.

Chapter 2

Short Vector SIMD Extensions

This chapter gives a brief overview of several floating-point short vector SIMD extensions of current microprocessor architectures. Table 2.1 summarizes the name and the most important features of the extensions.

Vendor	Extension	n-way	Precision	Register Width	Processor
Intel	SSE	4-way	single	128 bit	Pentium III, Pentium 4
Intel	SSE2	2-way	double	128 bit	Pentium 4
Intel	IA-64	2-way	single	64 bit	Itanium, McKinley
AMD	3DNow!	2-way	single	64 bit	K6
AMD	Enhanced 3DNow!	2-way	single	64 bit	K7, Athlon XP
AMD	SSE	4-way	single	128 bit	Athlon XP
Motorola	AltiVec	4-way	single	128 bit	G4

Table 2.1: Short vector SIMD extensions featuring floating-point arithmetic found in general purpose microprocessors.

Some of these extensions add new SIMD registers and some additionally introduced new execution units to the processor architecture. It is important to note that, because of constraints in the processor architecture, the amount of parallelism (i. e., 2-way or 4-way) can give only a rough (and sometimes misleading) estimate for the possible performance gain.

The Motorola AltiVec extension [6] adds a new vector execution unit to the PowerPC processor core. This vector unit is completely independent from the floating-point unit and vector instructions can be executed in parallel with

floating-point instructions. The vector unit features fused multiply add but no multiplication. Every cycle a vector instruction can be issued.

The AMD 3DNow! extension [1] is able to issue two 2-way SIMD instructions within one cycle leading to a peak performance of four times the core frequency. In contrast, the Intel SSE and SSE2 extension [5] has to wait one cycle between issuing two vector additions and between issuing two vector multiplications. As additions and multiplications can (theoretically) be interleaved, this leads to a peak performance of four times the core frequency for SSE and twice the core frequency for SSE2. However, interleaving requires an algorithm that is locally balanced with respect to the number of additions and multiplications. Thus, for most applications in practice, speed-ups of 4 are not attainable. Although the Pentium 4 is binary compatible to the Pentium III, it features a new core. Thus, code optimized for the Pentium III is not necessarily optimal for the Pentium 4.

A general problem in writing efficient short vector SIMD code is due to the limited memory access capabilities supported by short vector SIMD extensions. Most n-way short vector SIMD architectures support only naturally aligned n-way vector access (e. g., load/store 16-byte from 16-byte aligned memory locations). Unaligned vector access or scalar access requires a computationally more expensive or even several computationally expensive instructions. For example, loading/storing non-unit-stride data from/into a vector register is a very expensive operation.

Chapter 3

SPIRAL: a libraray generator for DSP algorithms

The objective behind Spiral (Moura et al. [7]) is to provide a software system capable of generating highly optimized code for arbitrary DSP (digital signal processing) transforms. Furthermore, the term "optimization" not only includes standard techniques like loop unrolling or common subexpression elimination, but also platform-adaptation by choice of a fast algorithm with optimal dataflow for the given architecture.

The Spiral approach is based on the following facts.

- For every DSP transform there exists a *very large* number of different *fast* algorithms. These algorithms differ in dataflow but are essentially equal in the number of arithmetic operations.
- A fast algorithm for a DSP 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 DSP transform.
- A formula representing a fast DSP algorithm can be mapped *automatically* into a program in a high-level language like C or Fortran.

Spiral's architecture displayed in Fig. 3.1. The user specifies the transform to be implemented, e. g., a DFT of size 1024. A formula generator module expands the transform into one (or several) out of many possible fast algorithms, given as a formula in Spiral's proprietary language SPL. The formula is translated by the SPL compiler into a program in a high-level language like C or Fortran. The runtime of this program is fed back to a search module that controls the generation of the next formula. Iteration of this loop leads to an optimized, platform-adapted implementation. In addition to algorithmic choices, the search module also controls implementation choices, as, e. g., the degree of loop unrolling.

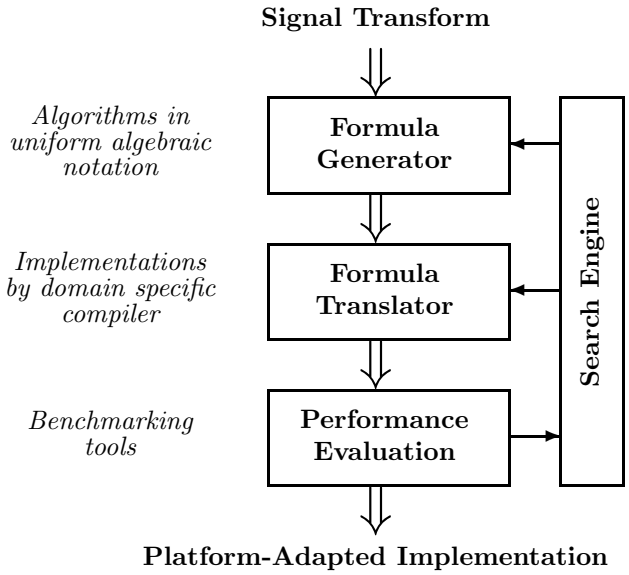


Figure 3.1: The architecture of SPIRAL.

Since SPIRAL is based on a mathematical description of DSP algorithms, it can be easily extended to include new transforms. The generation of different code types, as, e. g., SIMD vectorized code, requires to enhance or replace the SPL compiler, which is the contribution of this report and will be explained in Chapter 5.

The remainder of this chapter explains some details of SPIRAL's mathematical framework, the language SPL, and the SPL compiler. For further information on SPIRAL the reader is referred to [8, 11, 14].

3.1 DSP Transforms and Fast Algorithms

A (linear) DSP transform is a multiplication of a vector x (the sampled signal) by a certain $n \times n$ matrix M (the transform), $x \mapsto Mx$. A particularly important example is the discrete Fourier transform (DFT), which, for size n, is given by the matrix

$$\mathrm{DFT}_n = [e^{2\pi i k\ell/n} \mid k, \ell = 0, 1, \ldots, n-1], \quad i = \sqrt{-1}.$$

An important property of DSP transforms is the existence of fast algorithms. Typically, these fast algorithms reduce the asymptotic cost of evaluating an $n \times n$

transform from $O(n^2)$ arithmetic operations, as required by direct evaluation via matrix-vector multiplication, to $O(n \log n)$. This ensures their efficient applicability for large n. Mathematically, these fast algorithms can be viewed as a factorization of the transform matrix into a product of sparse matrices. It is a specific property of DSP transforms that these factorizations are highly structured and can be written in a very concise way using a small number of mathematical operators.

As an example consider a factorization, i. e., a fast algorithm, for a DFT of size 4, which is then written using mathematical notation.

$$\begin{aligned}
\mathrm{DFT}_4 &= \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix} \\
&= \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & i \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad (3.1) \\
&= (\mathrm{DFT}_2 \otimes \mathrm{I}_2) \quad \cdot \quad \mathrm{T}_2^4 \quad \cdot \quad (\mathrm{I}_2 \otimes \mathrm{DFT}_2) \quad \cdot \quad \mathrm{L}_2^4 .
\end{aligned}$$

Symbols T_2^4 and L_2^4 are used to represent the diagonal matrix $\mathrm{diag}(1, 1, 1, i)$ and the permutation matrix (right-most matrix), respectively. The symbol I_n denotes an identity matrix of size n. Of particular importance is the tensor or Kronecker product $\otimes$ of matrices, defined as

$$A \otimes B = [a_{k,\ell}\, B]_{k,\ell=1,2,\ldots,n} \qquad (3.2)$$

$$= \begin{bmatrix} a_{1,1}\, B & \ldots & a_{1,n}\, B \\ \vdots & \ddots & \vdots \\ a_{n,1}\, B & \ldots & a_{n,n}\, B \end{bmatrix}, \qquad (3.3)$$

where $A = [a_{k,\ell}]_{k,\ell=1,2,\ldots,n}$, and the direct sum

$$A \oplus B = \begin{bmatrix} A & \\ & B \end{bmatrix}. \qquad (3.4)$$

Equation (3.1) is an instantiation of the celebrated Cooley-Tukey algorithm (Cooley, Tukey [2]), also referred to as the fast Fourier transform (FFT). In its general

form, the Cooley-Tukey FFT is given as

$$\mathrm{DFT}_n = (\mathrm{DFT}_r \otimes \mathrm{I}_s)\ \mathrm{T}^n_s\ (\mathrm{I}_r \otimes \mathrm{DFT}_s)\ \mathrm{L}^n_r, \quad n = rs. \tag{3.5}$$

The *twiddle matrix* T^{rs}_s is diagonal, L^{rs}_s is called *stride permutation* matrix. The exact form is not relevant for the purposes of this report and can be found, e. g., in Tolimieri, An, Lu [12]. An equation like (3.5) is called a *breakdown rule* or simply *rule*. A rule is a sparse factorization of the transform and breaks down the computation of the transform (here DFT_n) to transforms of smaller size (here DFT_r and DFT_s). The smaller transforms (which can be of a different type) can be further expanded using the same or other rules. Eventually a mathematical *formula* is obtained where all transforms are expanded into base cases. This formula represents a fast algorithm for the transform. As an example consider a formula for DFT_{16}, which is used as a case study throughout this report:

$$\mathrm{DFT}_{16} = (\mathrm{DFT}_4 \otimes \mathrm{I}_4)\ \mathrm{T}^{16}_4\ (\mathrm{I}_4 \otimes \mathrm{DFT}_4)\ \mathrm{L}^{16}_4, \tag{3.6}$$

where DFT_4 is expanded as in (3.1).

By selecting different breakdown rules, a given DSP 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 (3.5) 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}. Allowing breakdown rules other than (3.5) further extends the formula space.

The formula space consists of fast algorithms that require essentially the same number of arithmetic operations, but differ in the data flow during computation. This leads to very different runtimes for corresponding implementations and thus to an optimization problem that SPIRAL solves by intelligent search in the formula space.

It is important to note that the presented framework is not restricted to the DFT but applies to all (linear) DSP transforms. This fact is illustrated by two further examples, the Walsh-Hadamard transform (WHT) and the discrete cosine transform (DCT) used, for instance, in the JPEG standard (Rao, Hwang [9]).

$$\mathrm{WHT}_{2^k} = \overbrace{\mathrm{DFT}_2 \otimes \ldots \otimes \mathrm{DFT}_2}^{k \text{ times}},$$

with rule

$$\mathrm{WHT}_{2^k} = \prod_{i=1}^{k} \left(\mathrm{I}_{2^{k_1+\cdots+k_{i-1}}} \otimes \mathrm{WHT}_{2^{k_i}} \otimes \mathrm{I}_{2^{k_{i+1}+\cdots+k_t}}\right), \quad k = k_1 + \cdots + k_t, \tag{3.7}$$

and

$$\mathrm{DCT}_n = [\cos((\ell + 1/2)k\pi/n) \mid k, \ell = 0, 1, \ldots, n-1],$$

with rule

$$\mathrm{DCT}_n = P_n \left(\mathrm{DCT}_{n/2} \oplus S_n \ \mathrm{DCT}_{n/2} \ D_n\right) P'_n \left(\mathrm{I}_{n/2} \otimes \mathrm{DFT}_2\right) P''_n,$$

where P_n, P'_n, P''_n are permutation matrices, S_n is bidiagonal, and D_n is a diagonal matrix (see Wang [13] for details).

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

$$M \otimes M = (M \otimes \mathrm{I}_n)\,(\mathrm{I}_n \otimes M) \tag{3.8}$$

is obtained.

Chapter 4

The Signal Processing Language SPL

For a computer representation of DSP algorithms, given as formulas, SPIRAL uses the language SPL (signal processing language). The following are the most important constructs available in SPL and mirror the notation introduced in Section 3.1.

```
(1) General matrices, for example:
    (matrix ((a11 ... a1n) ... (am1 ... amn)) ; generic matrix
    (diagonal (a11 ... ann))                  ; generic diagonal matrix
    (permutation (k1 ... kn))                 ; generic permutation matrix

(2) Parameterized matrices:
    (I n)     ; identity matrix
    (F n)     ; Fourier transform by definition
    (L mn n)  ; stride permutation matrix
    (T mn n)  ; twiddle factors

(3) Matrix operations, for example:
    (compose A1 ... An)    ; matrix product
    (tensor A1 ... An)     ; tensor product
    (direct_sum A1 ... An) ; direct sum
```

In addition to the constructs above, SPL provides control tags to control the SPL compilation process. Examples include tags that control the unrolling strategy or the datatype (real versus complex) used (see Xiong et al. [14] for details). For example, the DFT_4 algorithm given in (3.1) can be written in SPL as represented in Fig. 4.1.

Note that all SPL constructs shown above can be interpreted as a program for computing the corresponding matrix-vector product $x \mapsto M\,x$. As an example take the interpretation for the constructs listed under `(3)` above.

$M = A\,B$: apply B, then A;

$M = A \oplus B$: apply A and B to suitable subvectors of x (see (3.4));

$M = \mathrm{I}_n \otimes A$: apply n times A to suitable subvectors of x (see (3.2));

$M = A \otimes \mathrm{I}_n$: apply n times A at stride n to suitable subvectors of x (see (3.2)).

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

Figure 4.1: The DFT_4 algorithm given in (3.1) written as an SPL program.

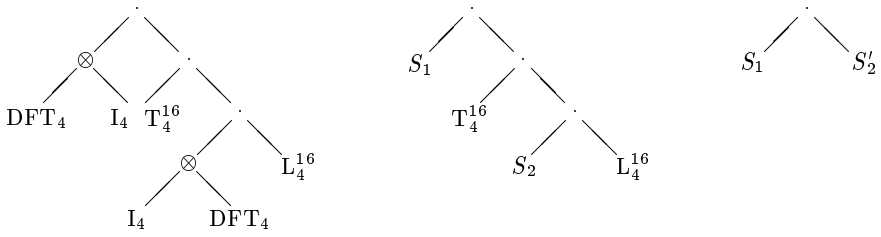
The code generation from an SPL program is performed by the SPL compiler, which is briefly described in the next section.

4.1 The SPL Compiler

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 Fig. 3.1) to try different implementations of the same formula.

In the first stage, the SPL code is parsed and translated into a binary abstract syntax tree. The internal nodes represent operators (e. g., `tensor`, `compose` and `direct_sum`) while 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 is 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. Fig. 4.2 shows the abstract syntax tree generated from (3.6) and symbols replacing subtrees in the abstract syntax tree.

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



(1) Abstract Syntax Tree (2) Basic Vectorization (3) Full Vectorization

Figure 4.2: (1) The abstract syntax tree generated from (3.6); the expansion of DFT_4 is not shown. (2) The abstract syntax tree after basic vectorization (all constructs of type $A \otimes \mathrm{I}_n$ and $\mathrm{I}_n \otimes A$ are replaced by symbols $S_1, S_2, \ldots, S_n$). (3) The abstract syntax tree after joining diagonals and permutations (full vectorization).

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 introduce important optimizations for common 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.

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

Chapter 5

Vectorization of SPL Formulas

This chapter presents an extended version of the SPL compiler that generates C code enhanced with short vector instructions using an SPL formula and the SIMD vector length ν (i. e., number of single precision floating-point numbers contained) as its sole input. The compiler supports real and complex transforms and input vectors. Moreover, the compiler produces portable code, which is achieved by restricting the generated code to instructions available on all SIMD architectures and using C macros to abstract from the architecture's intrinsics (note that no common API exists for SIMD extensions).

The key problem to solve is to identify the SPL constructs that can be vectorized. The new approach is based on the vectorization of the following two basic constructs

$$A \otimes \mathrm{I}_\nu \quad \text{and} \quad \mathrm{I}_\nu \otimes A,$$

where ν is the SIMD vector length and A an arbitrary formula. The construct $A \otimes \mathrm{I}_\nu$ can be naturally implemented using short vector instructions (see Fig. 5.1 for $\nu = 4$), while $\mathrm{I}_\nu \otimes A$ is transformed into the former using the mathematical identity

$$\mathrm{I}_\nu \otimes A = \mathrm{L}^n_\nu \, (A \otimes \mathrm{I}_\nu) \, \mathrm{L}^n_{n/\nu},$$

which involves the additional implementation of stride permutations. Extending from these basic cases, the most general construct to be vectorized is

$$\prod_{i=0}^{n} P_i \, D_i \, (\mathrm{I}_{l_i} \otimes A_i \otimes \mathrm{I}_{m_i}) \, E_i \, Q_i, \quad l_i \geq \nu \text{ or } m_i \geq \nu \quad \text{for all } i, \tag{5.1}$$

with diagonal matrices D_i and E_i, permutation matrices P_i and Q_i, and arbitrary formulas A_i. Note that identities like

$$A \otimes B = (A \otimes \mathrm{I}_m) \, (\mathrm{I}_m \otimes B),$$

$$P \, D \, Q \, E \cdots = P' D' = E' Q',$$

or

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

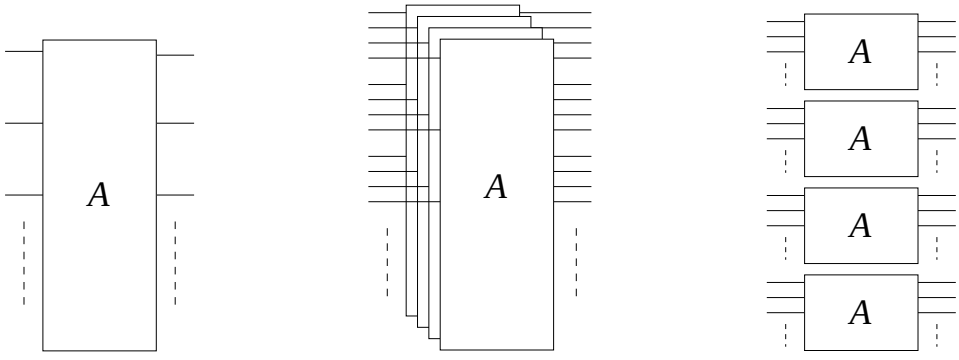


Figure 5.1: Data flow representations of the linear transform $y = A\,x$, its 4-way vectorized version $y = (A \otimes \mathrm{I}_4)\,x$, and the 4-way parallel version $y = (\mathrm{I}_4 \otimes A)\,x$.

can be used to transform formulas to match (5.1). For example, DFT and WHT algorithms arising from rules (3.5) and (3.7), respectively, and two-dimensional transforms (3.8), are special cases of (5.1) and can thus be completely vectorized. For a general formula, vectorized code will be generated for subformulas matching (5.1), while the rest is translated to standard (scalar) C code.

The vectorization of a given formula is performed in the following five stages. All constructs matching formula (5.1) are transformed into a product of symbols (see Section 4.1 for a definition of a symbol), and implemented as vectorized code, while the remaining part of the formula will be implemented as scalar code. Formula (3.6) is used to illustrate the algorithm.

Stage 1: Basic Vectorization

In the first stage the abstract syntax tree (see Section 4.1) generated from the formula is searched for constructs of the form

$$(I_{l_i} \otimes A \otimes I_{m_i}).$$

Then, using the basic vectorization rules from Table 5.1, these constructs are transformed into

$$P_i^{-1}\,(V_i \otimes \mathrm{I}_\nu)\,P_i$$

with suitable permutation matrices P_i. Next, each of these constructs is replaced by a SIMD symbol. These symbols are leaves in the transformed abstract syntax tree. Within a SIMD symbol, no further steps are done in this stage.

Basic Vectorization Rules			Extended Vectorization Rules		
$A \otimes B$	$\mapsto$	$(A \otimes \mathrm{I}_m)(\mathrm{I}_n \otimes B)$	$A \otimes I_\nu$	$\mapsto$	S
$A \otimes \mathrm{I}_{\nu k}$	$\mapsto$	$(A \otimes \mathrm{I}_k) \otimes \mathrm{I}_\nu = A' \otimes I_\nu$	$D\,S$	$\mapsto$	S'
$\mathrm{I}_{\nu k} \otimes A$	$\mapsto$	$\mathrm{I}_\nu \otimes (\mathrm{I}_k \otimes A)$	$S\,D$	$\mapsto$	S'
	$=$	$P^{-1}\,((\mathrm{I}_k \otimes A) \otimes \mathrm{I}_\nu)\,P = P^{-1}(A' \otimes I_\nu)P$	$P\,S$	$\mapsto$	S'
$A \otimes \mathrm{I}_{\nu k+l}$	$\mapsto$	$A \otimes (\mathrm{I}_{\nu k} \oplus \mathrm{I}_l)$	$S\,P$	$\mapsto$	S'
	$=$	$Q^{-1}((\mathrm{I}_{\nu k} \otimes A) \oplus (\mathrm{I}_l \otimes A))Q$			
$\mathrm{I}_{\nu k+l} \otimes A$	$\mapsto$	$(\mathrm{I}_{\nu k} \oplus \mathrm{I}_l) \otimes A = (\mathrm{I}_{\nu k} \otimes A) \oplus (\mathrm{I}_l \otimes A)$			

Table 5.1: Recursion and transformation rules for the SIMD vectorization. P and Q are certain permutation matrices. D is a diagonal matrix and A and B are arbitrary formulas. S and S' are SIMD symbols.

By applying the basic vectorization rules from Table 5.1 to the example, formula (3.6) is transformed into

$$W = S_1\ \mathrm{T}^{16}_4\ S_2\ \mathrm{L}^{16}_4$$

(see Fig. 4.2, sub-figures (1) and (2)). In the example,

$$V_1 = \mathrm{DFT}_4,\ P_1 = L^{16}_4,\ V_2 = \mathrm{DFT}_4, \quad \text{and} \quad P_2 = \mathrm{I}_{16}\,.$$

Stage 2: Joining Diagonals and Permutations

In this stage diagonals and permutations are vectorized. After this stage all constructs covered by (5.1) are transformed into a product of symbols. If any symbol S_i is composed with diagonals D_i or E_i or any permutation P_i or Q_i within the formula W (i. e., a term covered by $P_iD_iS_iE_iQ_i$ exists in W), create a symbol S'_i by joining the permutations and/or diagonals with the corresponding symbol S_i using the extended vectorization rules summarized in Table 5.1. The diagonals and permutations will be handled in the load and store phase of the implementation of the corresponding symbol. The non-vectorizable part W of the formula is transformed into W' by removing these diagonals and/or permutations.

In the example the two constructs T^{16}_4 and L^{16}_4 will be joined with the SIMD symbol S_2 to create the SIMD symbol S'_2. By joining the permutation and diagonal with symbol S_2, (3.6) is transformed into $W' = S_1\,S'_2$. See Fig. 4.2, sub-figures (2) and (3) for details.

Stage 3: Generating Code

In this stage an internal representation (i-code) for the abstract syntax tree is generated by calling the standard SPL compiler (see Section 4.1). For each SIMD symbol S_i or S_i' and the remaining non-vectorizable part W or W' of the formula i-code is generated and optimized using the respective stages of the standard SPL compiler. Each symbol S_i or S_i' contains a basic vectorized part V_i. I-code is generated for all V_i and for W or W'. The code generated for W or W' contains calls to the symbols S_i or S_i'. In the example, i-code for S_1, S_2' and $W' = S_1\, S_2'$ is generated.

Stage 4: Memory Access Optimizations

In this stage, the position of the symbols within the code is analyzed to avoid unnecessary data conversion. Under certain conditions, the store and load phase of two consecutive symbols is fused. In the example, symbol S_2 uses a combined transpose-store operation leading to vector stores to enable vector loads in symbol S_1. This transpose-store operation conducts the stride permutation required in the load phase of symbol S_1 as symbol S_1 is of type $\mathrm{I}_n \otimes A$. This can be seen in Appendices A.2 and A.3.

Stage 5: Unparsing

The internal representation for each SIMD symbol and for the main program generated in Stage 3 is unparsed as C program including machine independent C macros. These macros expressing basic generic operations (e. g., arithmetic or memory access operations) are mapped onto the SIMD hardware. The code generated for V_i is unparsed using vector arithmetic macros instead of scalar arithmetic. Thus, for each symbol S_i code for $V_i \otimes \mathrm{I}_\nu$ is unparsed, while code for W or W' is unparsed using scalar arithmetic. In the unparsing step permutations introduced by the basic vectorization of terms of type $\mathrm{I}_n \otimes A$ and permutations joined with symbols are handled by issuing specialized memory access macros (which do the required address translation). Diagonals joined with symbols are handled by issuing multiplications between the load/store phase and the arithmetic phase w.r.t. V_i for each symbol S_i. This leads to code computing the whole formula based on the scalar code generated in Stage 3.

Stages 1, 2, 4 and 5 will be explained in detail in the remainder of this chapter.

5.1 Basic Vectorization

The abstract syntax tree representation of the formula is searched recursively for constructs of the form $A \otimes \mathrm{I}_n$, $\mathrm{I}_n \otimes A$ and $A \otimes B$ (A and B are arbitrary formulas). As a top-down search is being carried out, the outermost constructs containing a tensor product will be found first. Constructs of type $A \otimes B$ are transformed into

$$(A \otimes \mathrm{I}_m)\,(\mathrm{I}_n \otimes B).$$

If $n = k\nu + l$ and $l \neq 0$, I_n is split into $\mathrm{I}_{k\nu} \otimes \mathrm{I}_l$ and the respective subtree in the abstract syntax tree is replaced by an equivalent subtree built from a subtree for $\mathrm{I}_{k\nu}$ and a subtree for I_l. Then the recursion rules for tensor products with identities are applied.

According to the recursion and transformation rules summarized in Table 5.1, the basically vectorizable constructs $A \otimes \mathrm{I}_n$ and $\mathrm{I}_n \otimes A$ are transformed into symbols S_1,S_2,...,S_j with

$$S_i = P_i^{-1}\,(V_i \otimes \mathrm{I}_\nu)\,P_i \qquad \text{with} \quad P_i \in \{\mathrm{I}_{n_i}, \mathrm{L}^{n_i}_{k_i}\}.$$

These symbols replace the respective subtree in the abstract syntax tree generated from the formula. If the term is of form $\mathrm{I}_n \otimes A$ additional stride permutations are introduced ($\mathrm{I}_n \otimes A$ is conjugated) in the transformation process and will be handled in the unparsing step. The abstract syntax tree is transformed into an equivalent abstract syntax tree where all basically vectorizable subtrees are replaced by symbols.

5.2 Joining Diagonals and Permutations

The computation of diagonal scalings and permutations adjacent to SIMD symbols (`compose` of a SIMD symbol and a diagonal or permutation) can be joined with the symbols. As an arbitrary diagonal cannot be expressed as $D = D_1 \otimes \mathrm{I}_\nu$, another method opposed to the basic vectorization has to be used to vectorize diagonal scalings. Such an operation can be carried out using multiplications by SIMD constants, where each element of the SIMD constant may be different from all others, while within a basically vectorized block all SIMD constants are ν times the same number.

To solve this problem, the multiplications by the constants w. r. t. an adjacent diagonal are inserted between the load operation and the first arithmetic operation for a data element or between the last arithmetic operation and the store

operation for a data element depending on the position of the diagonal relative to the SIMD symbol.

The permutations are carried out transparently for the vectorized code by different load/store macros which implement the needed address translation. Some types of permutations can be implemented better than others, depending on the underlying hardware. For this permutations, special macros are inserted into the code instead of the generic load/store macros with permutation support. On machines with hardware support for that permutations, the special instructions are used while on other machines the special macros may be mapped onto the generic macros if no hardware support is available. See Table 5.1 for the recursion rules used for diagonal and permutation handling and Appendix A.2 for the utilization of special load/store macros and the handling of diagonal scalings.

5.3 Memory Access Optimization

Code generated by the scalar SPL compiler uses the interleaved complex format (the real and imaginary parts of a complex vector are interleaved) or the real format depending on the data type (complex numbers or real numbers) of the transform to be compiled. As the SIMD symbols are called within the main routine, the input and output format of a routine has to be the interleaved complex format or the real format for compatibility reasons. For SIMD computation, the block interleaved complex format (a complex vector broken into a vector of ν real parts followed by ν imaginary parts) is much better suited as it can be accessed by vector loads/stores. This format is used internally by the SIMD symbols in a natural way.

For subformulas of type

$$S_1\, S_2 = (A \otimes \mathrm{I}_\nu)\,(A \otimes \mathrm{I}_\nu)$$

and

$$S_1\, S_2 = (\mathrm{I}_\nu \otimes A)\,(\mathrm{I}_\nu \otimes A)$$

(S_1, S_2 are complex SIMD symbols) the data can be stored in the block interleaved format by the first SIMD symbol and loaded in that format by the second SIMD symbol. So the expensive transformation from the interleaved complex format to the block interleaved complex format can be removed between two SIMD symbols.

SIMD symbols of the type $\mathrm{I}_n \otimes A$ require very expensive load and store operations with permutation support. For subformulas of type

$$S_1\, S_2 = (A \otimes \mathrm{I}_\nu)\,(\mathrm{I}_\nu \otimes A)$$

and

$$S_1\, S_2 = (\mathrm{I}_\nu \otimes A)\,(A \otimes \mathrm{I}_\nu)$$

meeting some criteria (S_1, S_2 are complex SIMD symbols) and for real SIMD symbols this expensive operations can be replaced by an in-register permutation and vector load/stores. This optimizations increase the register pressure because multiple loads/stores are joined to one bigger memory operation plus in-register permutations and may result in more register spills. Performing the optimizations presented in this section is essential to get high performance short vector SIMD code. These optimizations require knowledge of the structure of DSP algorithms and therefore cannot be conducted by general purpose vectorizing compilers.

5.4 Unparsing

The method presented in this report relies on a C compiler that features language extensions (data types and operation primitives) for a short vector SIMD extension ISA. To overcome the problem of platform-specific programming models and software support, the generated code uses a set of C macros as unifying interface to the short vector extensions. These C macros constitute the basic instructions needed to vectorize SPL formulas, and require only a few SIMD primitives that are provided by each vendor. These basic instructions include (i) loading/storing a complex number, (ii) loading/storing a vector, (iii) declaring and using a constant, (iv) extracting the real or imaginary parts into a vector, (v) building complex numbers from a vector of real parts and a vector of imaginary parts, (vi) adding/subtracting/multiplying two vectors. On top of these macros, more complicated operations such as permutations, strided vector access, etc., are built. All operations within the vectorized code parts are done using these macros (see Appendices A.1 and A.3 for details).

As all symbols can be expressed as

$$S_i = P_i^{-1}\,(V_i \otimes \mathrm{I}_\nu)\,P_i \qquad \text{with} \quad P_i \in \{\mathrm{I}_{n_i}, \mathrm{L}_{k_i}^{n_i}\}$$

(see Table 5.1) any SIMD symbol generated by the two vectorization steps can be implemented using only short vector SIMD instructions by unparsing the i-code for the formula V_i as vector code in conjunction with the handling of diagonals and permutations. In the unparsing step, for SIMD symbols all arithmetic operations are unparsed as architecture independent C macros (e. g., `c = a + b` is unparsed as `c = SIMD_ADD(a, b)`). Additionally, explicit load and store macros

are issued in the SIMD code. These macros also implement transparently the needed permutations by conducting the required address translations by loading data from the permuted memory location or storing data at the permuted memory location.

If diagonal scalings are present for a SIMD symbol, the corresponding real or complex multiplications are inserted between the load/store operations and the code for the formula $V_i \otimes \mathrm{I}_\nu$ related to S_i or S_i'. Standard optimization techniques like removing multiplications by 1, additions of 0, etc., are performed. See Fig. 4.2 for implementation details of the architecture independent macros.

For small SIMD symbol sizes and inner loops, the generated code is unrolled. As a consequence all constants and permutations are inlined. Special optimized macros for constant handling and permutation handling are used. For larger SIMD symbol sizes, loop code is generated. In this case, permutation tables and constant tables are generated. That means load/store operations with permutation support result in an indirect memory access. In the general permutation case, the permuted index is looked up in the permutation table and used to load/store the respective element. Some stride permutations can be implemented by transforming the loop indices linearly resulting in much better performance.

Chapter 6

Experimental Results

The new SIMD version of SPIRAL was tested on a 650 MHz Pentium III system operating under Windows 2000 using the Intel C++ Compiler 5.0. The displayed runtimes are the minima over several measurements. Per measurement, the average over k iterations (k sufficiently large and depending on the vector length n) transforming the null vector is taken. Within the SPIRAL system, dynamic programming was used as search method for fast implementations.

In general, smaller unrolled code with higher arithmetic complexity is implemented most efficiently (e. g., smaller FFTs and two-dimensional DCTs). Formulas containing no permutations and diagonals are implemented most efficiently, when partially unrolled code can be used (WHTs for medium sized vector lengths). The smallest and biggest measured formulas usually are less efficient than medium sized formulas.

For the FFT routines, the new SPIRAL SIMD system was compared to the original SPIRAL 3.1 system, Intel MKL 5.1 (the newest vendor library with highly optimized FFT codes) and FFTW 2.1.3 (the actual release of the popular FFT package).

Vector lengths from 2^4 to 2^{12} were tested. For 2^4 to 2^9 the runtimes of the SPIRAL SIMD FFT routine scale nearly like the arithmetic complexity $\mathrm{O}(n \log n)$ while for 2^9 to 2^{12} the runtime grows faster than the complexity. For 2^4 to 2^9 the SPIRAL SIMD FFT routine was the fastest routine measured. Speed-up factors (FFT SPIRAL 3.1 vs. FFT SPIRAL SIMD) of 1.42 to 2.09 were achieved. See Figs. 6.1 and 6.2, and Table 6.1 for details.

To test the WHT routines and the 2D DCT and DST routines, SPIRAL 3.1 was compared to the new SPIRAL SIMD version. The WHT routines were tested with vector lengths from 2^4 to 2^{14}. The new SIMD routine achieved the best performance for 2^7 to 2^{10} while for smaller and bigger vector lengths the performance gradually decreases. For vector lengths smaller than 2^5 and bigger than 2^{13} the performance was dropping. Speed-up factors of 1.17 up to 2.04 were achieved. See Figs. 6.3 and 6.4, and Table 6.2 for details.

The two-dimensional DCT routines were tested from 4×4 up to 128×128 data sets. Apart from 4×4, speed-ups around 2.25 were achieved. For 8×8 to

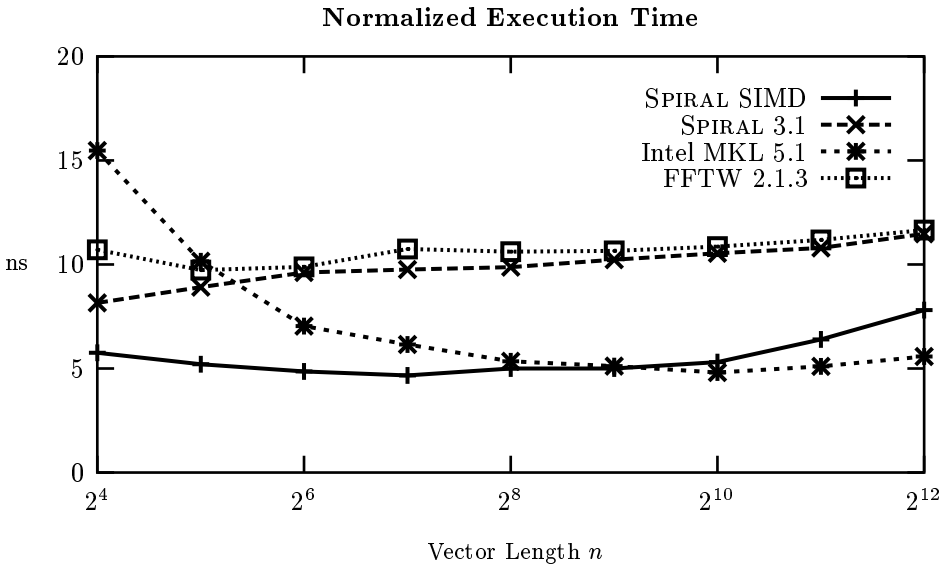


Figure 6.1: Normalized runtimes ($T/n \log n$) of complex single precision FFT routines on an Intel Pentium III running at 650 MHz.

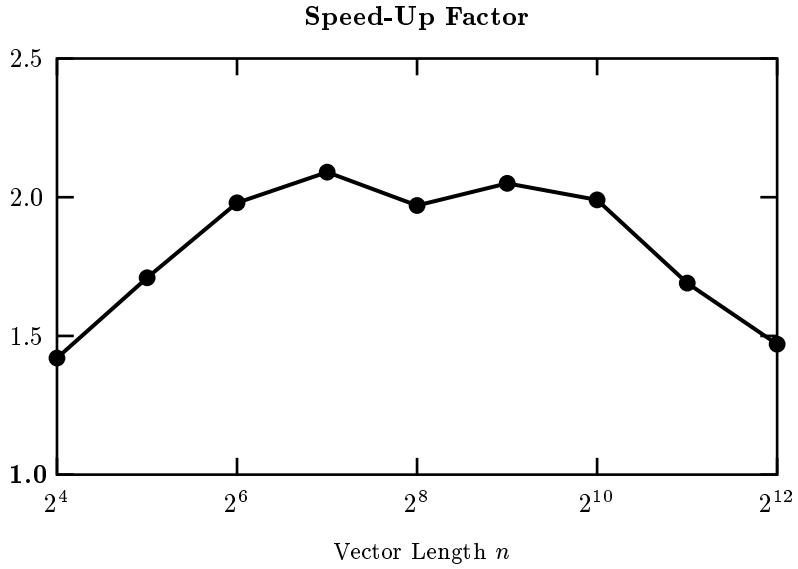


Figure 6.2: Speed-up of the SPIRAL SIMD implementation relative to the original SPIRAL 3.1 implementation on an Intel Pentium III running at 650 MHz.

n	SPIRAL SIMD	SPIRAL orig.	Speed-Up	Intel MKL 5.1	FFTW 2.1.3
2^4	3.68×10^{-7}	5.22×10^{-7}	1.42	9.91×10^{-7}	6.86×10^{-7}
2^5	8.32×10^{-7}	1.43×10^{-6}	1.71	1.62×10^{-6}	1.56×10^{-6}
2^6	1.87×10^{-6}	3.70×10^{-6}	1.98	2.70×10^{-6}	3.79×10^{-6}
2^7	4.18×10^{-6}	8.75×10^{-6}	2.09	5.51×10^{-6}	9.63×10^{-6}
2^8	1.02×10^{-5}	2.02×10^{-5}	1.97	1.09×10^{-5}	2.17×10^{-5}
2^9	2.30×10^{-5}	4.71×10^{-5}	2.05	2.36×10^{-5}	4.91×10^{-5}
2^{10}	5.43×10^{-5}	1.08×10^{-4}	1.99	4.92×10^{-5}	1.11×10^{-4}
2^{11}	1.44×10^{-4}	2.43×10^{-4}	1.69	1.15×10^{-4}	2.52×10^{-4}
2^{12}	3.84×10^{-4}	5.64×10^{-4}	1.47	2.74×10^{-4}	5.73×10^{-4}

Table 6.1: Runtimes in seconds of (interleaved) complex single precision FFTs on an Intel Pentium III 650 MHz running under Windows 2000. SPIRAL SIMD is compared to SPIRAL 3.1, FFTW 2.1.3 and Intel MKL 5.1 using the Intel C++ Compiler 5.0 (used options: `-O3 -G6`).

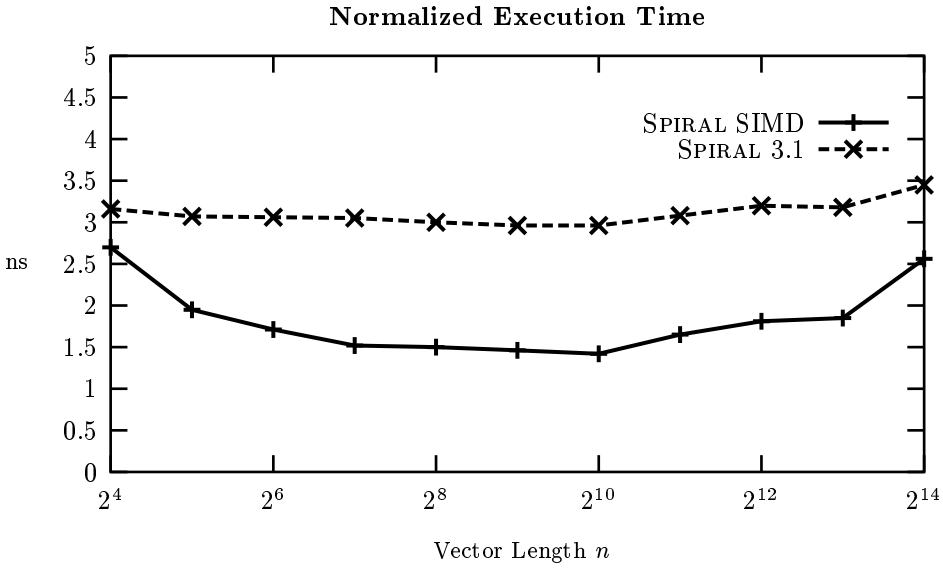


Figure 6.3: Normalized runtimes ($T/n \log n$) of real single precision WHT routines on an Intel Pentium III running at 650 MHz.

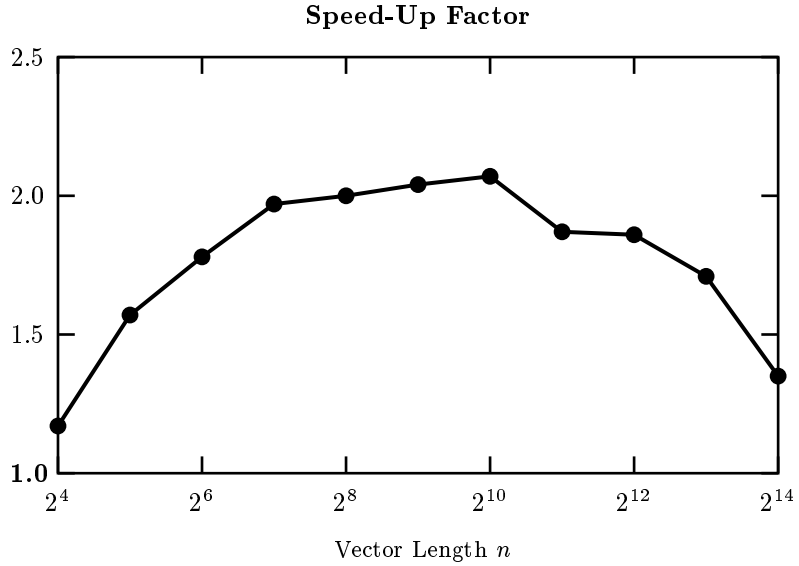


Figure 6.4: Speed-up of the SPIRAL SIMD implementation relative to the original SPIRAL 3.1 implementation on an Intel Pentium III running at 650 MHz..

n	SPIRAL SIMD	SPIRAL orig.	Speed-Up
2^4	1.73×10^{-7}	2.02×10^{-7}	1.17
2^5	3.12×10^{-7}	4.91×10^{-7}	1.57
2^6	6.59×10^{-7}	1.18×10^{-6}	1.78
2^7	1.39×10^{-6}	2.73×10^{-6}	1.97
2^8	3.06×10^{-6}	6.13×10^{-6}	2.00
2^9	6.68×10^{-6}	1.36×10^{-5}	2.04
2^{10}	1.46×10^{-5}	3.03×10^{-5}	2.07
2^{11}	3.72×10^{-5}	6.94×10^{-5}	1.87
2^{12}	8.47×10^{-5}	1.57×10^{-4}	1.86
2^{13}	1.98×10^{-4}	3.39×10^{-4}	1.71
2^{14}	5.88×10^{-4}	7.92×10^{-4}	1.35

Table 6.2: Runtimes in seconds of real single precision WHTs on an Intel Pentium III 650 MHz running under Windows 2000. SPIRAL 3.1 is compared to SPIRAL SIMD using the Intel C++ Compiler 5.0 (used options: `-O3 -G6`).

$n \times n$	SPIRAL SIMD	SPIRAL orig.	Speed-Up
4×4	2.09×10^{-7}	2.99×10^{-7}	1.43
8×8	8.29×10^{-7}	1.84×10^{-6}	2.26
16×16	4.41×10^{-6}	9.82×10^{-6}	2.25
32×32	2.31×10^{-5}	5.04×10^{-5}	2.31
64×64	1.50×10^{-4}	3.05×10^{-4}	2.22

Table 6.3: Runtimes in seconds of real single precision 2D DCTs of type 2 on an Intel Pentium III 650 MHz running under Windows 2000. SPIRAL 3.1 is compared to SPIRAL SIMD using the Intel C++ Compiler 5.0 (used options: `-O3 -G6`).

32×32 the routine is scaling with the arithmetic complexity $n^2 \log n$. See Fig. 6.5, Fig. 6.6 and Table 6.3 for details.

The results presented in this report outperform the results presented in Franchetti, Karner, Ueberhuber [3]. (The performance of the reference systems—FFTW 2.1.3 and SPIRAL 3.1—is comparable for FFTs and $n < 2^{12}$; see Fig. 6.1 for details.) While speed-up factors of 1.4 to 2.1 are achieved in this report, speed-up factors between 1.5 and 1.65 were achieved in Franchetti, Karner, Ueberhuber [3]. One reason is that load/store operations and data alignment problems are solved more efficiently in this report.

For small transforms (with $n \leq 256$) no faster implementation of a single-precision interleaved complex FFT routine than the routines generated with the framework presented in this report is known to the authors. Even the vendors highly optimized Intel Math Kernel library is clearly outperformed on the Intel Pentium III.

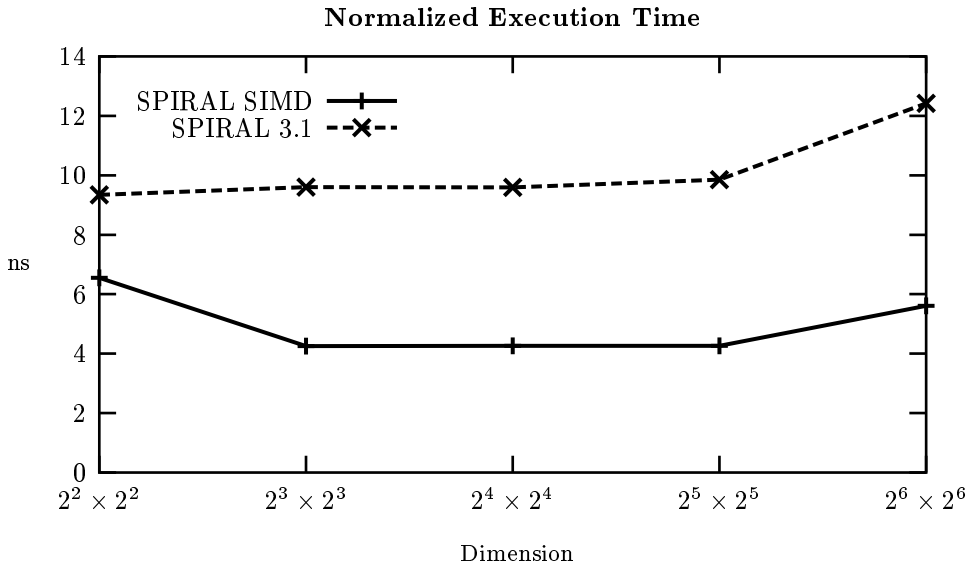


Figure 6.5: Normalized runtimes ($T/n^2 \log n$) of real single precision two-dimensional DCT type 2 routines on an Intel Pentium III running at 650 MHz.

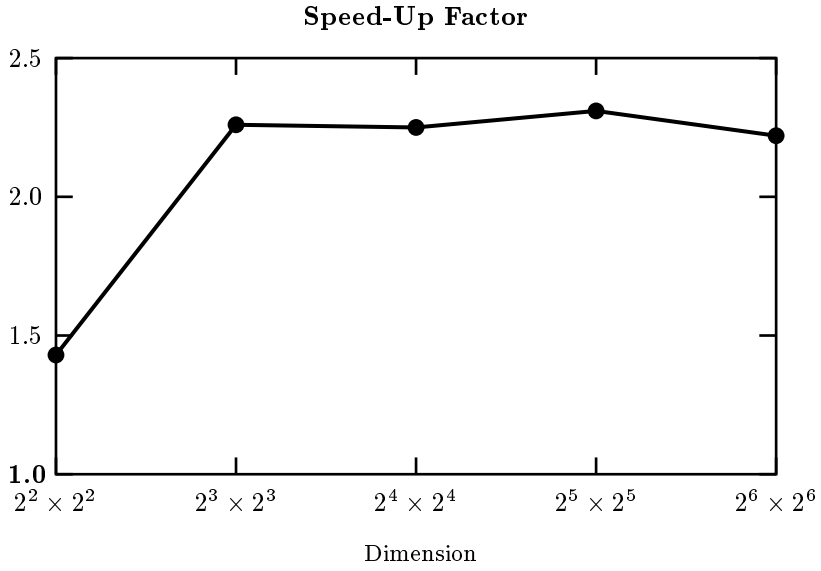


Figure 6.6: Speed-up of the SPIRAL SIMD implementation relative to the original SPIRAL 3.1 implementation on an Intel Pentium III running at 650 MHz.

Chapter 7

Conclusion

This report introduces a compiler that translates symbolic mathematical descriptions (SPL) of fast DSP transform algorithms into SIMD vectorized code. By interfacing this compiler with Spiral it is possible to automatically search the algorithm space of a given transform for a fast implementation. The experimental results show that the automatically produced code is very fast.

There are the following advantages in this approach. (i) The vectorized code is automatically generated and verified; (ii) it is possible to automatically search the algorithm space for an optimal implementation; (iii) by abstracting from a specific implementation task (as, e. g., the DFT) to its mathematical structure, a large class of DSP algorithms can be vectorized at the same time, as long as they can be written in vectorizable constructs.

Further work needs to be done in extending the number of vectorizable constructs in the language SPL to produce vectorized code for a larger class of DSP algorithms.

Acknowledgement

Finally, we thank Prof. Christoph Ueberhuber (Technical University of Vienna) and Prof. José Moura (Carnegie Mellon University) for initiating and supporting the authors' collaboration.

References

[1] Advanced Micro Devices Corporation. *3DNow! Technology Manual*, 2000.

[2] J. W. Cooley and J. W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Math. of Computation*, 19, pp. 297–301, 1965.

[3] F. Franchetti, H. Karner, S. Kral, and C. W. Ueberhuber. Architecture Independent Short Vector FFTs. In *Proc. ICASSP*, volume 2, pp. 1109–1112, 2001.

[4] M. Frigo and S. G. Johnson. FFTW: An Adaptive Software Architecture for the FFT. In *ICASSP 98*, volume 3, pp. 1381–1384, 1998. `http://www.fftw.org`.

[5] Intel Corporation. *Intel C/C++ Compiler User's Guide — With Spport for the Streaming SIMD Extensions*, 1999.

[6] Motorola Corporation. *AltiVec Technology Programming Interface Manual*, 2000.

[7] J. M. F. Moura, J. Johnson, R. W. Johnson, D. Padua, V. Prasanna, M. Püschel, and M. M. Veloso. SPIRAL: Portable library of optimized signal processing algorithms, 1998.
`http://www.ece.cmu.edu/~spiral`.

[8] M. Püschel, B. Singer, M. Veloso, and J. M. F. Moura. Fast Automatic Generation of DSP Algorithms. In *Proc. ICCS 2001*, LNCS 2073, pp. 97–106. Springer-Verlag, 2001.

[9] K. R. Rao and J. J. Hwang. *Techniques and Standards for Image, Video and Audio Coding*. Prentice Hall, 1996.

[10] N. Sereraman and R. Govindarajan. A Vectorizing Compiler for Multimedia Extensions . *International Journal of Parallel Programming, Special Issue on Instruction and Loop Level Parallelism*, 2000. To appear.

[11] B. Singer and M. Veloso. Stochastic Search for Signal Processing Algorithm Optimization. In *Proc. Supercomputing*, 2001. To appear.

[12] R. Tolimieri, M. An, and C. Lu. *Algorithms for discrete Fourier transforms and convolution*. Springer-Verlag, 2nd edition, 1997.

[13] 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(4), pp. 803–816, 1984.

[14] J. Xiong, J. Johnson, R. Johnson, and D. Padua. SPL: A Language and Compiler for DSP Algorithms. In *Proc. PLDI*, pp. 298–308, 2001.

Appendix A

SIMD Code

In this appendix code used for the examples given in this report is listed.

A.1 SIMD Macros

This section shows the subset of the SIMD macros for the Intel Pentium III needed in the example. The abstract and portable SIMD operations described in Section 5.4 are mapped onto the intrinsics supplied by the Intel C++ compiler.

```
/*  define vector type  */
typedef float simd_scalar_float;
typedef __m128 simd_vector_float;

/*  define arithmetic operations    */
#define SIMD_ADD(a,b)   _mm_add_ps((a),(b))
#define SIMD_SUB(a,b)   _mm_sub_ps((a),(b))
#define SIMD_MUL(a,b)   _mm_mul_ps((a),(b))

#define COMPLEX_MULT(rr, ri, ar, ai, br, bi)                    \
{                                                               \
    (rr) = _mm_sub_ps(_mm_mul_ps(ar, br),                       \
        _mm_mul_ps(ai, bi));                                    \
    (ri) = _mm_add_ps(_mm_mul_ps(ar, bi),                       \
        _mm_mul_ps(ai, br)); }

/*  constant handling   */
#define DECLARE_SIMD_CONST_4(c, r, s, t, u)                     \
    static const float c##_0 = r, c##_1 = s, c##_2 = t, c##_3 = u;

#define LOAD_SIMD_CONST_4(c) _mm_set_ps(c##_3, c##_2, c##_1, c##_0)

/*  define load operations          */
#define LOAD_VECT(trg, src)                                     \
    (trg) = _mm_load_ps((float *)(src))

#define LOAD_INTERL_PERM(re, im, in0, in1, in2, in3)            \
{                                                               \
```

```
    simd_vector_float _ldtmp1, _ldtmp2;                          \
    _ldtmp1 = _mm_loadl_pi(_ldtmp1, in0);                        \
    _ldtmp1 = _mm_loadh_pi(_ldtmp1, in1);                        \
    _ldtmp2 = _mm_loadl_pi(_ldtmp2, in2);                        \
    _ldtmp2 = _mm_loadh_pi(_ldtmp2, in3);                        \
    re = _mm_shuffle_ps((_ldtmp1), (_ldtmp2),                    \
        _MM_SHUFFLE(2, 0, 2, 0));                                \
    im = _mm_shuffle_ps((_ldtmp1), (_ldtmp2),                    \
        _MM_SHUFFLE(3, 1, 3, 1));                                \
}

/*  define store operations          */
#define STORE_INTERL_USTRIDE(re, im, out)                        \
{                                                                \
    simd_vector_float _sttmp1,_sttmp2;                           \
    _sttmp1 = _mm_unpacklo_ps(re, im);                           \
    _sttmp2 = _mm_unpackhi_ps(re, im);                           \
    _mm_store_ps(out, _sttmp1);                                  \
    _mm_store_ps((out) + SIMD_VECLEN, _sttmp2);                  \
}

#define STORE_TRANSPOSED(c0, c1, c2, c3, output, stride)         \
{                                                                \
    simd_vector_float co0 = c0, co1 = c1, co2 = c2, co3 = c3;    \
    _MM_TRANSPOSE4_PS(co0, co1, co2, co3);                       \
    _mm_store_ps(output, co0);                                   \
    _mm_store_ps((output) + (stride), co1);                      \
    _mm_store_ps((output) + 2 * (stride), co2);                  \
    _mm_store_ps((output) + 3 * (stride), co3);                  \
}
```

A.2 SIMD Code for Symbol S_1

This section shows the generated SIMD code for the interleaved complex symbol

$$S_1 = \mathrm{DFT}_4 \otimes \mathrm{I}_4 = \left((\mathrm{DFT}_2 \otimes \mathrm{I}_2)\ \mathrm{T}_2^4\ (\mathrm{I}_2 \otimes \mathrm{DFT}_2)\ \mathrm{L}_2^4\right) \otimes \mathrm{I}_4$$

in (3.6). The previous call to symbol S_2' has transformed the data into the block interleaved complex format, which is the input to the symbol S_1. The output of S_1 is in the interleaved complex format.

```
void symbol_S1(simd_scalar_float *y, simd_scalar_float *x)
{
      simd_vector_float xl0, xl1, xl2, xl3, xl4, xl5, xl6, xl7;
```

```
    simd_vector_float yl0, yl1, yl2, yl3, yl4, yl5, yl6, yl7;
    simd_vector_float f0, f1, f2, f3, f4, f5, f6, f7;

    LOAD_VECT(xl0, x + 0);
    LOAD_VECT(xl4, x + 16);
    f0 = SIMD_SUB(xl0, xl4);
    LOAD_VECT(xl1, x + 4);
    LOAD_VECT(xl5, x + 20);
    f1 = SIMD_SUB(xl1, xl5);
    f2 = SIMD_ADD(xl0, xl4);
    f3 = SIMD_ADD(xl1, xl5);
    LOAD_VECT(xl2, x + 8);
    LOAD_VECT(xl6, x + 24);
    f4 = SIMD_SUB(xl2, xl6);
    LOAD_VECT(xl3, x + 12);
    LOAD_VECT(xl7, x + 28);
    f5 = SIMD_SUB(xl3, xl7);
    f6 = SIMD_ADD(xl2, xl6);
    f7 = SIMD_ADD(xl3, xl7);
    yl4 = SIMD_SUB(f2, f6);
    yl5 = SIMD_SUB(f3, f7);
    STORE_INTERL_USTRIDE(yl4, yl5, y + 16);
    yl0 = SIMD_ADD(f2, f6);
    yl1 = SIMD_ADD(f3, f7);
    STORE_INTERL_USTRIDE(yl0, yl1, y + 0);
    yl6 = SIMD_ADD(f0, f5);
    yl7 = SIMD_SUB(f1, f4);
    STORE_INTERL_USTRIDE(yl6, yl7, y + 24);
    yl2 = SIMD_SUB(f0, f5);
    yl3 = SIMD_ADD(f1, f4);
    STORE_INTERL_USTRIDE(yl2, yl3, y + 8);
}
```

A.3 SIMD Code for Symbol S_2'

This section shows the generated SIMD code for the interleaved complex symbol

$$\begin{aligned} S_2' &= \mathrm{T}_4^{16}\,(\mathrm{DFT}_4 \otimes \mathrm{I}_4)\;\mathrm{L}_4^{16} \\ &= \mathrm{T}_4^{16}\,\left(\mathrm{I}_4 \otimes \left((\mathrm{DFT}_2 \otimes \mathrm{I}_2)\;\mathrm{T}_2^4\;(\mathrm{I}_2 \otimes \mathrm{DFT}_2)\;\mathrm{L}_2^4\right)\right)\;\mathrm{L}_4^{16} \end{aligned}$$

in (3.6). The symbol S_2' has the input in the interleaved complex format and the output in the block interleaved complex format. The permutation L_4^{16} is carried out by the load macros. The diagonal scaling for computing T_4^{16} is conducted by

the complex multiplications inserted between the SIMD arithmetic macros and the store macros.

```
void symbol_S2(simd_scalar_float *y, simd_scalar_float *x)
{
      simd_vector_float xl0, xl1, xl2, xl3, xl4, xl5, xl6, xl7;
      simd_vector_float yl0, yl1, yl2, yl3, yl4, yl5, yl6, yl7;
      simd_vector_float f0, f1, f2, f3, f4, f5, f6, f7;
      simd_vector_float st2, st3, st4, st5, st6, st7;

      LOAD_INTERL_PERM(xl0, xl1, x + 0, x + 2, x + 4, x + 6);
      LOAD_INTERL_PERM(xl4, xl5, x + 16, x + 18, x + 20, x + 22);
      f0 = SIMD_SUB(xl0, xl4);
      f1 = SIMD_SUB(xl1, xl5);
      f2 = SIMD_ADD(xl0, xl4);
      f3 = SIMD_ADD(xl1, xl5);
      LOAD_INTERL_PERM(xl2, xl3, x + 8, x + 10, x + 12, x + 14);
      LOAD_INTERL_PERM(xl6, xl7, x + 24, x + 26, x + 28, x + 30);
      f4 = SIMD_SUB(xl2, xl6);
      f5 = SIMD_SUB(xl3, xl7);
      f6 = SIMD_ADD(xl2, xl6);
      f7 = SIMD_ADD(xl3, xl7);
      yl4 = SIMD_SUB(f2, f6);
      yl5 = SIMD_SUB(f3, f7);
      yl0 = SIMD_ADD(f2, f6);
      yl1 = SIMD_ADD(f3, f7);
      yl6 = SIMD_ADD(f0, f5);
      yl7 = SIMD_SUB(f1, f4);
      yl2 = SIMD_SUB(f0, f5);
      yl3 = SIMD_ADD(f1, f4);
      COMPLEX_MULT(st2, st3, yl2, yl3,
          LOAD_SIMD_CONST_4(SC2), LOAD_SIMD_CONST_4(SC3));
      COMPLEX_MULT(st4, st5, yl4, yl5,
          LOAD_SIMD_CONST_4(SC4), LOAD_SIMD_CONST_4(SC5));
      COMPLEX_MULT(st6, st7, yl6, yl7,
          LOAD_SIMD_CONST_4(SC6), LOAD_SIMD_CONST_4(SC7));
      STORE_TRANSPOSED(yl0, st2, st4, st6, y + 0, 8);
      STORE_TRANSPOSED(yl1, st3, st5, st7, y + 4, 8);
}
```

A.4 Constant Declaration and `main()`

This section shows the constant declaration and SIMD code for the main function for (3.6).

```
DECLARE_SIMD_CONST_4(SC7, 0.000000000, 0.923879532,
                          0.707106781, -0.382683432);
DECLARE_SIMD_CONST_4(SC6, 1.000000000, 0.382683432,
                         -0.707106781, -0.923879532);
DECLARE_SIMD_CONST_4(SC5, 0.000000000, 0.707106781,
                          1.000000000,  0.707106781);
DECLARE_SIMD_CONST_4(SC4, 1.000000000, 0.707106781,
                          0.000000000, -0.707106781);
DECLARE_SIMD_CONST_4(SC3, 0.000000000, 0.382683432,
                          0.707106781,  0.923879532);
DECLARE_SIMD_CONST_4(SC2, 1.000000000, 0.923879532,
                          0.707106781,  0.382683432);

void DFT_16(simd_scalar_float *y, simd_scalar_float *x)
{
   static simd_scalar_float_aligned t0[32];

    symbol_S2(t0 + 0, x + 0);
    symbol_S1(y + 0, t0 + 0);
}
```