A SIMD Vectorizing Compiler for Digital Signal Processing Algorithms*

Franz Franchetti
Applied and Numerical Mathematics
Technical University of Vienna, Austria
franz.franchetti@tuwien.ac.at

Markus Püschel
Electrical and Computer Engineering
Carnegie Mellon University
pueschel@ece.cmu.edu

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. We present 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 our 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 our compiler with SPIRAL yields speed-ups of more than a factor of 2 in several important cases including the FFT and the discrete cosine transform (DCT) used in the JPEG compression standard. For the FFT our automatically generated code is competitive with the hand-coded Intel Math Kernel Library.

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 multimedia applications. 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.

*This work 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.

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 implementations.

A very efficient implementation of the DFT is provided by FFTW [2]. A first SIMD version of FFTW has been presented in [1] showing a substantial improvement in performance.

The implementation of arbitrary DSP transforms, including the DFT, is the target of SPIRAL [3]. 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 1) automatically generated from a transform specification [4]; and 2) automatically translated into optimized code in a high-level language like C or Fortran using the SPL compiler [7]. Platform adaptation is achieved by intelligently searching, for a given transform, the large space of possible algorithms, i.e., formulas, for the fastest one [5]. The code produced by SPIRAL is very competitive [7].

In this paper we present a SIMD vectorizing version of the SPL compiler that is portable across different SIMD architectures. We show that certain mathematical constructs used in the formula representation of a DSP algorithm can be naturally mapped to vectorized code. These constructs are generalized versions of one base case and occur in virtually every 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.

We included our compiler into SPIRAL and automatically generated—on a Pentium III with SSE—vectorized DSP code that is highly competitive and substantially speeds up the code generated by SPIRAL. We obtained speed-up factors of more than 2 for DFTs, WHTs, and 2-D DCTs. Experiments indicate that, under the given conditions, this is near the practical limit on this architecture.

The paper is organized as follows. Section 2 briefly introduces short vector SIMD extensions available on current processors and discusses the vectorization problem. Section 3 briefly describes SPIRAL. In Section 4 we present the mathematical foundation of our vectorizing compiler, which then is explained in Section 5. We conclude with experimental results in Section 6.

2. Short Vector Extensions

Important floating-point short-vector SIMD extensions currently available on general purpose microprocessor architectures include Intel SSE (4-way single-precision), Intel SSE2 (2-way double-precision), Motorola AltiVec (4-way single-precision), and AMD and Enhanced 3DNow! (2-way single-precision). 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 usually misleading) estimate for a possible performance gain. These SIMD extensions share the following characteristics:

- They provide n-way floating-point vector arithmetic.
- Memory access is efficient only for properly aligned unit-stride data through vector loads/stores (on some architectures unaligned access and subvector memory access are supported but cause very high computational cost).
- Some types of in-register permutations are supported.
- Proprietary C interfaces (APIs) are available (except for AMD).

Because of hardware restrictions and the specific structure of DSP algorithms, their efficient vectorization and implementation is a difficult problem. Main challenges include:

- Non-unit stride access and complex data types produce data access patterns that prevent a straightforward vectorization.
- A general purpose vectorizing compiler does not have access to the full structure of a DSP algorithm, and thus cannot find a satisfactory vectorization.

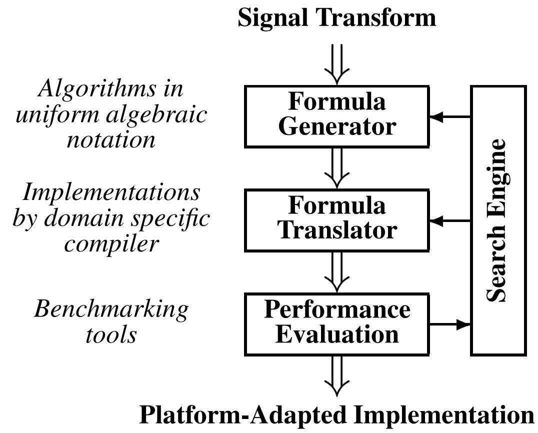

Figure 1. The architecture of SPIRAL.

- No standard API exists for different extensions across architectures.

Our approach solves these problems by using a mathematical description (a *formula*) of the DSP algorithm as input to our compiler. This way, we have access to all structural information and can use formal manipulations to exhibit vectorizable parts. Furthermore, we have explicit access to all data access patterns, which allows us to solve expensive load and store operations efficiently. Finally, we achieve portability, by using our own API (consisting of C macros) that is built only on commonly available vector instructions.

3. SPIRAL

The objective behind SPIRAL [3] is to provide a code generator that is capable of generating highly optimized libraries 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 approach of SPIRAL uses the following facts.

- For every DSP 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 DSP transform can be represented as a *formula* in a natural 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 *automatically* translated into a program in a high-level language like C or Fortran.

SPIRAL's architecture is based on these facts and displayed in Figure 1. The user specifies a transform he wants to implement, 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 the SPIRAL 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 engine 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 details, as, e.g., the degree of loop unrolling. Further information on SPIRAL can be found in [4, 7, 5].

Since SPIRAL is based on a mathematical description of DSP algorithms, it can be easily extended to include new transforms. In this paper we extend SPIRAL to generate SIMD vectorized code by replacing the SPL compiler (i.e., formula translator in Figure 1) by our extended version that generates SIMD code. This way we benefit from SPIRAL's infrastructure, and, in particular, the search engine, to automatically find very fast implementations.

4. Mathematical Framework

In this section we describe SPIRAL's mathematical framework, which is the foundation of our approach. Crucial is the concept of *formulas* to represent fast DSP transform algorithms. Formulas are a natural representation from a mathematical point of view but can also be interpreted as a very high level programming language, which can be compiled into a standard language like C (as it is done in SPIRAL), but also into efficient SIMD vector code, which is our contribution.

DSP Transforms and Algorithms. A (linear) DSP transform is a multiplication of the sampled signal $x \in \mathbb{C}^n$ by a transform matrix M of size $n \times n$, $x \mapsto M \cdot x$. A particularly important example is the discrete Fourier transform (DFT), which, for size n, is given by the matrix

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

DSP transforms have fast algorithms that reduce the arithmetic cost to $O(n \log(n))$ (compared to $O(n^2)$ by direct evaluation) and make them efficient for applications. An algorithm can be viewed as a factorization of the transform matrix into a product of sparse matrices. It is a specific property of DSP transforms—and key to our approach—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 the sparse factorization, i.e., fast algorithm, of DFT_4, which is then written using mathematical notation.

$$\begin{bmatrix} 1&0&1&0\\ 0&1&0&1\\ 1&0&-1&0\\ 0&1&0&-1 \end{bmatrix} \cdot \begin{bmatrix} 1&0&0&0\\ 0&1&0&0\\ 0&0&1&0\\ 0&0&0&i \end{bmatrix} \cdot \begin{bmatrix} 1&1&0&0\\ 1&-1&0&0\\ 0&0&1&1\\ 0&0&1&-1 \end{bmatrix} \cdot \begin{bmatrix} 1&0&0&0\\ 0&0&1&0\\ 0&1&0&0\\ 0&0&0&1 \end{bmatrix}$$

$$= (\mathrm{DFT}_2 \otimes \mathrm{I}_2) \cdot \quad \mathrm{T}^4_2 \quad \cdot (\mathrm{I}_2 \otimes \mathrm{DFT}_2) \cdot \quad \mathrm{L}^4_2 . \tag{1}$$

Symbols T^4_2 and L^4_2 are used to represent the diagonal matrix $\mathrm{diag}(1, 1, 1, i)$ and the permutation matrix (rightmost matrix), respectively, and I_n denotes an identity matrix of size n. Of particular importance is the tensor or Kronecker product $\otimes$ and the direct sum $\oplus$ of matrices, defined as ($A = [a_{k,\ell}]_{k,\ell=1,\dots,n}$)

$$A \otimes B = \begin{bmatrix} a_{1,1} \cdot B & \dots & a_{1,n} \cdot B \\ \vdots & \ddots & \vdots \\ a_{n,1} \cdot B & \dots & a_{n,n} \cdot B \end{bmatrix}, \tag{2}$$

$$A \oplus B = \begin{bmatrix} A & \\ & B \end{bmatrix}. \tag{3}$$

Two special cases are of particular importance, and we give an intuitive interpretation in terms of a multiplication to a vector x: $\mathrm{I}_n \otimes B$ means "apply n times B to consecutive segments of x"; and $A \otimes \mathrm{I}_n$ means "apply n times A at stride n to consecutive segments of x.

Equation (1) is an instantiation of the celebrated Cooley-Tukey algorithm (e.g., [6]), also referred to as the fast Fourier transform (FFT). In its general form, the Cooley-Tukey FFT is given, for $n = r \cdot s$, as

$$\mathrm{DFT}_n = (\mathrm{DFT}_r \otimes \mathrm{I}_s) \cdot \mathrm{T}^n_s \cdot (\mathrm{I}_r \otimes \mathrm{DFT}_s) \cdot \mathrm{L}^n_r . \tag{4}$$

The *twiddle matrix* T^{rs}_s is diagonal and the *stride permutation* matrix L^{rs}_r maps $k \mapsto kr \bmod rs - 1$ for $k = 0, \dots, rs - 2$ and $rs - 1 \mapsto rs - 1$ [6]. Intuitively, L^{rs}_r reads an input vector at stride r and stores it a stride 1.

We call an equation like (4) 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 we obtain a mathematical *formula* where all transforms are expanded into base cases. This formula represents a fast algorithm for the transform. As an example we give a formula for DFT_{16}, which is used as a case study throughout this paper:

$$\mathrm{DFT}_{16} = (\mathrm{DFT}_4 \otimes \mathrm{I}_4) \cdot \mathrm{T}^{16}_4 \cdot (\mathrm{I}_4 \otimes \mathrm{DFT}_4) \cdot \mathrm{L}^{16}_4, \tag{5}$$

with DFT_4 being expanded as in (1). For a computer representations of formulas, SPIRAL uses the language SPL. The SPL compiler translates the SPL description into C or Fortran code [7].

By selecting different rules in the expansion process, SPIRAL can generate, for one transform, a very large number of different formulas, corresponding to different fast algorithms. These algorithms have essentially the same arithmetic cost, but different data flow, which leads to very different runtime performances. SPIRAL solves the resulting optimization problem by intelligent search in the formula space [5].

It is important to note that the presented framework is not restricted to the DFT but applies to all (linear) DSP transforms. We give two further examples, the Walsh-Hadamard transform (WHT) and arbitrary two-dimensional transforms.

$$\begin{aligned} \mathrm{WHT}_{2^k} &= \overbrace{\mathrm{DFT}_2 \otimes \ldots \otimes \mathrm{DFT}_2}^{k \text{ times}}, \quad \text{with rule} \\ \mathrm{WHT}_{2^k} &= \prod_{j=1}^{t} \left(\mathrm{I}_{2^{\ell_{j-1}}} \otimes \mathrm{WHT}_{2^{k_j}} \otimes \mathrm{I}_{2^{k-\ell_j}} \right), \end{aligned} \tag{6}$$

where $k = k_1 + \cdots + k_t$ is a chosen partition, and we use the short notation $\ell_0 = 0$, and $\ell_j = k_1 + \cdots + k_j$, $j \geq 1$.

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 we obtain the equation, i.e., rule,

$$M \otimes M = (M \otimes \mathrm{I}_n) \cdot (\mathrm{I}_n \otimes M). \tag{7}$$

Formula Manipulation. A given formula for a fast DSP transform algorithm can be manipulated using mathematical identities. Formula manipulation is the first main step in our vectorizing compiler (see Section 5). The goal is to normalize formulas and to exhibit subexpressions that can be vectorized. We use the following identities; A and B are of size $n \times n$ and $m \times m$, respectively, P is a permutation matrix, D, D' are diagonal.

$$PD = D'P \tag{8}$$

$$\mathrm{I}_{n\nu+l} = \mathrm{I}_{n\nu} \oplus \mathrm{I}_l \tag{9}$$

$$\mathrm{I}_{mn} = \mathrm{I}_m \otimes \mathrm{I}_n \tag{10}$$

$$A \otimes B = (A \otimes \mathrm{I}_m)(\mathrm{I}_n \otimes B) \tag{11}$$

$$\mathrm{I}_\nu \otimes A = \mathrm{L}_\nu^{n\nu} (A \otimes \mathrm{I}_\nu)\, \mathrm{L}_n^{n\nu} \tag{12}$$

$$(\mathrm{I}_n \otimes \mathrm{L}_2^{2\nu})(\mathrm{I}_n \otimes \mathrm{L}_\nu^{2\nu}) = \mathrm{I}_{2n\nu} \tag{13}$$

Complex Transforms. In SPIRAL, complex transforms are realized in real arithmetic using the interleaved complex format (alternating real and imaginary part) for the input vector. This can be expressed formally. We use the simple fact that the complex multiplication $(u + iv) \cdot (y + iz)$ is equivalent to the real multiplication $\left[\begin{smallmatrix} u & -v \\ v & u \end{smallmatrix}\right] \cdot \left[\begin{smallmatrix} y \\ z \end{smallmatrix}\right]$. Thus, the complex matrix-vector multiplication $M \cdot x \in \mathbb{C}^n$ corresponds to $\overline{M} \cdot x' \in \mathbb{R}^{2n}$, where $\overline{M}$ arises from M by replacing every entry $u + iv$ by the corresponding (2×2)-matrix above, and x' is in interleaved complex format. The operator $\overline{(\cdot)}$ allows us to formally translate complex transforms and formulas into real ones, which simplifies the code generation. As in the previous paragraph, we need a suitable set of manipulation rules, which is given by

$$\overline{A \cdot B} = \overline{A} \cdot \overline{B} \tag{14}$$

$$\overline{A} = A \otimes \mathrm{I}_2, \quad A \text{ real} \tag{15}$$

$$\overline{D} = (\mathrm{I}_{n/\nu} \otimes \mathrm{L}_\nu^{2\nu}) \overline{D}' (\mathrm{I}_{n/\nu} \otimes \mathrm{L}_2^{2\nu}), \quad \nu \mid n \tag{16}$$

$$\overline{A \otimes \mathrm{I}_\nu} = (\mathrm{I}_n \otimes \mathrm{L}_\nu^{2\nu})(\overline{A} \otimes \mathrm{I}_\nu)(\mathrm{I}_n \otimes \mathrm{L}_2^{2\nu}) \tag{17}$$

Here A and D are of size $n \times n$; the matrix $\overline{D}'$ has a certain block diagonal structure suitable for vectorization.

5. Vectorization of SPL Formulas

In this section we present an extended version of the SPL compiler that generates C code enhanced with machine independent SIMD macros, using a formula, given in SPL, and the SIMD vector length ν (i.e., number of floats contained) as its sole input. This new version is a replacement for the standard SPL compiler within the SPIRAL system and provides the generation of vectorized code. The machine independent SIMD macros can be implemented on all current short vector SIMD architectures using native short vector instructions and constitute an hardware abstraction layer. Compiler support is required to utilize these instructions within source code leaving all lower-level optimizations including register allocation and instruction scheduling to the C compiler.

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

$$A \otimes \mathrm{I}_\nu, \quad \nu = \text{ vector length}, \tag{18}$$

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

Extending from this base case, the normalized most general construct that we vectorize (i. e., that can be implemented using exclusively our architecture independent SIMD macros) is the formula

$$\prod_{i=1}^{k} P_i D_i (A_i \otimes \mathrm{I}_\nu) E_i Q_i, \tag{19}$$

with arbitrary matrices A_i, permutation matrices P_i, Q_i, and matrices D_i, E_i that are either diagonal or have a certain block diagonal structure (arising from complex formulas transformed into corresponding real formulas). For example, all DFT and WHT algorithms arising from Rules (4) and (6), respectively, and two-dimensional transforms (7), can be normalized to formulas matching (19) and can thus be completely vectorized with our approach.

Our compiler vectorizes a given formula in two steps:

- The *symbolic vectorization* normalizes a formula, using manipulation rules, to exhibit maximal subformulas that match (19).
- The *code generation* phase translates vectorizable subformulas into vectorized code built from portable SIMD macros; the rest of the formula is implemented in C. In addition, several optimizations are performed.

Efficient utilization of short vector extensions (and straightforward vectorization) requires unit-stride data access, but other access patterns are inherent in the structure of DSP algorithms. An important example are subformulas of the form $\mathrm{I}_n \otimes A$. A similar problem arises from the interleaved data format used by complex transforms. We solve this problem by formula manipulations that formally substitute these expressions to make them match (19) by introducing permutations P_i, Q_i.

Concerning the performance of the generated code, finding an efficient implementation of the permutations P_i and Q_i, using vector instructions, is the most important problem to solve. The efficiency of the implementation depends on the type of permutation and also on the underlying SIMD architecture. A permutation can be negligible concerning runtime, but it may also slow down the whole DSP algorithm dramatically. We identify a class of permutation that can be realized efficiently and includes the permutations occurring in the considered transforms.

We want to emphasize that our approach uses high-level structural information of the DSP algorithm. This information is available in the formula representation, but not in a C code representation of the algorithm. For this reason, a general purpose vectorizing compiler fails, e.g., to vectorize the construct $\mathrm{I}_\nu \otimes A$, even though it is completely vectorized with our methods.

We detail the compilation process in the following, using Formula (5) as illustrative example.

Symbolic Vectorization. In this step we apply to a given formula F the manipulation rules (8)–(13) and (14)–(17) to obtain an expression of the form

$$F = \prod_{i=1}^{n} (R_i \oplus \mathrm{S}_i \oplus T_i) U_i, \quad \nu \mid \text{ size of } R_i \tag{20}$$

where R_i, T_i, U_i are arbitrary formulas and the *symbols* S_i are defined as in (19) by

$$\mathrm{S}_i = P_i D_i (A_i \otimes \mathrm{I}_\nu) E_i Q_i. \tag{21}$$

We reserve the symbols S_i for vectorizable parts of the formula, while R_i, T_i, U_i will be translated into standard C code. For example, a formula that has no vectorizable parts degenerates to $F = U_1$, while in a completely vectorizable formula R_i, T_i vanish, U_i is the identity, and thus F matches (19). Note that the normalization is not unique. Important subformulas that become symbols S_i in this step include $A \otimes B$, $A \otimes \mathrm{I}_k$, $\mathrm{I}_k \otimes A$, $\overline{A \otimes \mathrm{I}_k}$, and $\overline{\mathrm{I}_k \otimes A}$, as can be seen from the manipulation rules.

Furthermore, products $Q_i P_{i+1}$ of adjacent permutations are entirely or partially canceled out. In particular, in the real realization of a complex formula using (17), the simplification (13) can be often applied.

After the normalization, the symbols S_i are extracted and treated as independent formulas for which code is generated.

We illustrate this step with Formula (5) and vector length $\nu = 4$. We obtain the following completely vectorizable expression, which matches (19):

$$\begin{aligned}\overline{\mathrm{DFT}}_{16} = &\Big((\mathrm{I}_4 \otimes \mathrm{L}_4^8)(\overline{\mathrm{DFT}}_4 \otimes \mathrm{I}_4)\overline{\mathrm{T}}_4'^{16} \Big) \\ &\cdot \Big((\mathrm{I}_4 \otimes \mathrm{L}_2^8)(\mathrm{L}_4^{16} \otimes \mathrm{I}_2)(\mathrm{I}_4 \otimes \mathrm{L}_4^8) \\ &\quad (\overline{\mathrm{DFT}}_4 \otimes \mathrm{I}_4)(\mathrm{I}_4 \otimes \mathrm{L}_2^8) \Big)\end{aligned} \tag{22}$$

The vectorizable subformulas are replaced by symbols leading to $\overline{\mathrm{DFT}}_{16} = \mathrm{S}_1 \cdot \mathrm{S}_2$ with the definition of the symbols S_1 and S_2 and the non-trivial parameters (i. e., all parameters not equal to I_{32}) $P_1 = (\mathrm{I}_4 \otimes \mathrm{L}_4^8)$, $A_1 = \overline{\mathrm{DFT}}_4$, $E_1 = \overline{\mathrm{T}}_4'^{16}$, $P_2 = (\mathrm{I}_4 \otimes \mathrm{L}_2^8)(\mathrm{L}_4^{16} \otimes \mathrm{I}_2)(\mathrm{I}_4 \otimes \mathrm{L}_4^8)$, $A_2 = \overline{\mathrm{DFT}}_4$, and $Q_2 = (\mathrm{I}_4 \otimes \mathrm{L}_2^8)$. This factorization is implemented efficiently using exclusively our machine independent SIMD macros as outlined in the following section.

Code Generation. In this step, the normalized formula F is translated into C code including a set of portable SIMD macros. The method relies on a C compiler that features a language extension (*intrinsic functions* mapped to the short vector instructions and vector data types) for a short vector SIMD extension ISA, i. e., the SIMD hardware can be accessed explicitly within a C program without the usage of inline assembly. E. g., the Microsoft Visual Studio (requiring the installation of the Processor Pack) or the Intel C++ Compiler (for both Windows and Linux) can be used for SSE and SSE2, while some vendors (including GNU) support Motorola's AltiVec interface. To overcome the problem of platform-specific instruction set architectures and APIs, our generated code is built on top of a set of C macros as unifying interface to the short vector extensions. All required operations within the vectorized code are defined as C macros utilizing only basic SIMD instructions including 1) vector memory access, 2) hardware supported in-register permutations, 3) vector arithmetics. The required functionality can be implemented on all current SIMD extensions.[1]

[1]For example, the implementation of the permutation $\mathrm{I}_k \otimes \mathrm{L}_2^8$

For all symbols S_i in (20) vectorized code is generated while for the remaining non-vectorized part of the formula (R_i, T_i, and U_i) standard C code is generated. The remainder of this section describes the implementation of the symbols.

Implementation of Symbols. Each symbol (21) is implemented as a C function consisting of vectorized code and consists of the following logical parts:

- Load phase: $x' = E_i Q_i \cdot x$.
- Computation phase: $y' = (A \otimes \mathrm{I}_\nu) \cdot x'$.
- Store phase: $y = P_i D_i \cdot y'$.

The SIMD code for $y' = (A \otimes \mathrm{I}_\nu) \cdot x'$ is readily generated by replacing all scalar floating-point operations of the code for $v = A \cdot u$ (generated by the original SPL compiler's core routines) with vector operation macros (e.g., `c=a+b` is replaced by `SIMD_ADD(c,a,b)`) leading to vector code for $y' = (A \otimes \mathrm{I}_\nu) \cdot x'$.

The operations of $y = P_i D_i \cdot y'$ and $x' = E_i Q_i \cdot x$ are—in general—handled by memory access macros that carry out these permutations transparently and include the operations imposed by D_i, E_i (avoiding explicit arrays x' and y' whenever possible). For one specific class of permutations, however, the implementation can be done using only vector memory access and in-register permutations (if subvector memory access is available or the code is completely unrolled, this class if even more general). This class is given by permutations of the form

$$(U \otimes \mathrm{I}_\nu)(\mathrm{I}_k \otimes W)(V \otimes \mathrm{I}_\nu), \quad \nu \mid \text{ size of } W, \qquad (23)$$

where $U, V,$ and W are permutation matrices. (If W is a $(k \times k)$-matrix, we refer here to k as the size of W.) We focus on this class, since it includes all permutations needed for a vectorized implementation of the DFTs, WHTs, and two-dimensional transforms. In our example, $P_1 = \mathrm{I}_4 \otimes \mathrm{L}_4^8$, $Q_2 = \mathrm{I}_4 \otimes \mathrm{L}_2^8$, and $P_2 = (\mathrm{I}_4 \otimes \mathrm{L}_2^8)(\mathrm{L}_4^{16} \otimes \mathrm{I}_2)(\mathrm{I}_4 \otimes \mathrm{L}_4^8) = (\mathrm{L}_4^8 \otimes \mathrm{I}_4)(\mathrm{I}_2 \otimes \mathrm{L}_4^{16})(\mathrm{L}_2^8 \otimes \mathrm{I}_4)$.

The operation $x' = E_i Q_i \cdot x$, where Q_i matches (23) is implemented using the following facts:

- $V \otimes \mathrm{I}_\nu$ matches (18) and is implemented using vector loads from the input array x.
- $\mathrm{I}_k \otimes W$ is a permutation that operates on blocks of $l\nu$ elements and thus is implemented using in-register permutations using l short vector registers.
- $U \otimes \mathrm{I}_\nu$ matches (18) and is implemented by vector stores to the temporary array x'.
- E_i is implemented as generalized scaling operation utilizing vector memory access and vector arithmetics.

The array x' is substituted by a set of temporary variables whenever the algorithm structure allows to save stack cells and in this case the store operations of $U \otimes \mathrm{I}_\nu$ are realized by variable renaming. For a given symbol S_i, data can only be loaded and scaled at chunks of l vectors because of the size of W. The permutations P_i and Q_i are not carried out in one step, but are broken into pieces of size $l\nu$. Technically, a load macro and a scale macro is issued previous to the first usage of a vector element $x'[i]$ in the computation phase. These macros have 3 sets of parameters: l source vectors $x[i]$, l destination vectors $x'[j]$ and the permutation W. Different macros exist for different types of W to support an efficient implementation. To load l vectors into the destination locations $x'[j]$ according to the permutation Q_i, the first l vector elements $x[i]$ are loaded into l vector registers. This step handles a part of $V \otimes \mathrm{I}_\nu$. A segment of $l\nu$ floating-point numbers is then permuted according to W using in-register permutations. In the next step, the vector registers are stored into the the destination vectors $x'[j]$ according to $U \otimes \mathrm{I}_\nu$.

After this step, a scaling macro carries out the operations to compute the generalized scaling introduced by E_i. In the case that E_i is a diagonal matrix, the generalized scaling is a standard scaling (a pointwise multiplication of the vector $x[i]$ or the respective temporary variable by a vector of ν constants). In the case that E_i originates from a complex transform, its structure is more complicated, but handled analogously by an appropriate macro. We omit the details due to space limitations. The load macro and the scaling macro lead to properly prepared (i.e, in vector format) input variables $x'[i]$ for the computation phase. Different types of constants require a set of complex scaling macros for an efficient implementation. In the store phase the operation $y = P_i D_i \cdot y'$ is carried out analogously.

The use of temporary variables instead of the arrays x' and y' is the most crucial issue despite vector memory access, because it dramatically cuts the number of required stack cells. For small SIMD symbol sizes and inner loops, the generated code is unrolled. As a consequence all constants and permutations are inlined and special optimized macros for constant handling and permutation handling are used. For larger SIMD symbol sizes, loop code is generated. If loops and permutations are not compatible, permutation tables are required leading to indirect memory access. The structure of nested loops is analyzed to avoid permutation tables whenever possible to prevent performance degeneration.

Using our machine independent macros, the factorization of $\overline{\mathrm{DFT}_{16}}$ obtained by symbolic vectorization is implemented utilizing only vector memory access, in-register permutations and vector arithmetics leading to a very efficient library function for $y = \mathrm{DFT}_{16} \cdot x$.

via the macro `LOAD_STRIDE_8_2(src1,src2,dst1,dst2)` requires the intrinsic `_mm_shuffle_ps()` on Intel SSE and the intrinsics `vec_mergel()` and `vec_mergeh()` on Motorola AltiVec.

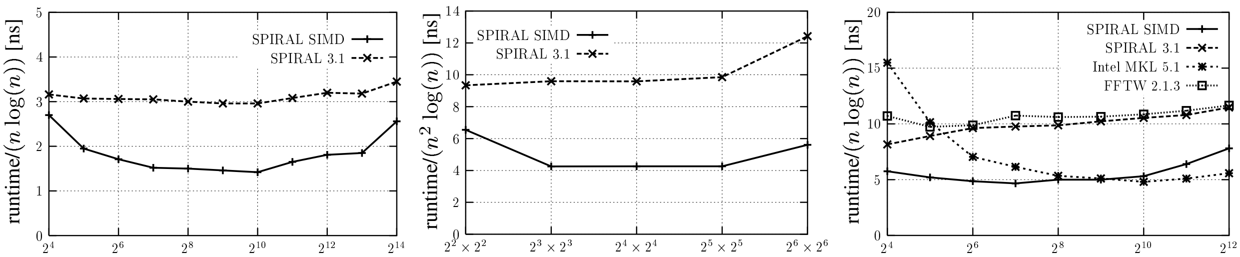

Figure 2. From left to right ($n = 2^k$): normalized runtimes for a real WHT of size $n = 2^4, \ldots, 2^{14}$, a real 2-D DCT of size $n \times n$, $n = 2^2, \ldots, 2^6$, and a complex DFT of size $n = 2^4, \ldots, 2^{12}$.

6. Experimental Results

We tested our vectorizing SPL compiler, included in SPIRAL (later called SPIRAL SIMD), on a 650 MHz Pentium III operating under Windows 2000 using the Intel C++ compiler 5.0. The runtimes are minimums over several measurements. Per measurement a sufficiently large number of iterations was performed, transforming the null vector. The transforms considered are the DFT, the WHT, and the 2-dimensional DCT used in JPEG. We use single precision floats for the real WHT and DCT, and two floats for the complex DFT. Within SPIRAL we used dynamic programming as search method, running separate searches for SPIRAL SIMD and the original SPIRAL 3.1. We only considered transform sizes that fit in level 2 cache. For convenient display, we normalized the runtime by the asymptotic complexity of the transforms ($n\log(n)$ for WHT and DFT, and $n^2\log(n)$ for 2-D DCT).

For the WHT and the 2-D DCT, we achieved speed-up factors of 2.04 and 2.31, respectively, comparing SPIRAL SIMD and SPIRAL 3.1 (Figure 2, left and middle).

For the DFT, we compared SPIRAL SIMD, SPIRAL 3.1, Intel MKL 5.1 (the newest vendor library with highly optimized DFT codes), and FFTW 2.1.3. For 2^4 to 2^9 the SPIRAL SIMD DFT was the fastest routine measured. We obtained speed-up factors (SPIRAL SIMD vs. SPIRAL 3.1 or FFTW) SIMD) of up to 2.09 (Figure 2, right). These results also outperform [1], where speed-up factors of up to 1.65 were achieved. The main reason is the new method of handling complex numbers presented and in this paper. For small size, we are not aware of any faster DFT implementation for the Pentium III; even the vendors hand coded Intel Math Kernel library is clearly outperformed.

Finally, we want to note that the speed-ups we obtained are near the practical limit on this architecture.

Conclusion. We summarize the key points. 1) In contradistinction to other approaches, we are *not* implementing (and vectorizing) a specific transform algorithm (as, e.g., the DFT), but generate code for the mathematical constructs the algorithm is built from. This way, every algorithm using these constructs is automatically included. 2) Our generated code can not be produced by a general purpose vectorizing compiler, since it misses the high level information provided by the formula representation. 3) Using SPIRAL, the code generation (including verification) and algorithmic optimization is entirely automatic from the transform specification.

We are currently finishing an extended version of the compiler that includes Motorola's AltiVec architecture.

Finally, we want to thank Prof. Überhuber (Technical University of Vienna) and Prof. Moura (Carnegie Mellon University) for initiating and supporting the authors collaboration.

References

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

[2] M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *ICASSP 98*, volume 3, pages 1381–1384, 1998. `http://www.fftw.org`.

[3] 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`.

[4] M. Püschel, B. Singer, M. Veloso, and J. M. F. Moura. Fast Automatic Generation of DSP Algorithms. In *Proc. ICCS 2001*, pages 97–106. Springer, 2001.

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

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

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