SHORT VECTOR CODE GENERATION AND ADAPTATION FOR DSP ALGORITHMS

Franz Franchetti[*]

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

Markus Püschel

Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, U.S.A.
pueschel@ece.cmu.edu

ABSTRACT

Most recent general purpose processors feature short vector SIMD instructions, like SSE on Pentium III/4. In this paper we automatically generate platform-adapted short vector code for DSP transform algorithms using SPIRAL. SPIRAL represents and generates fast algorithms as mathematical formulas, and translates them into code. Adaptation is achieved by searching in the space of algorithmic and coding alternatives for the fastest implementation on the given platform. We explain the mathematical foundation that relates formula constructs to vector code, and overview the vector code generator within SPIRAL. Experimental results show excellent speed-ups compared to ordinary C code for a variety of transforms and computing platforms. For the DFT on Pentium 4, our automatically generated code compares favorably with the hand-tuned Intel MKL vendor library.

1. INTRODUCTION

Short Vector Extensions. Recent generations of general purpose processors feature short vector SIMD (single instruction multiple data) extensions of their instruction set to speed up integer and floating-point computation. Examples include SSE on Pentium III/4 and Athlon XP, SSE2 on Pentium 4, and Motorola AltiVec. These instructions provide parallel operation on short vectors (currently of length $\nu = 2$ or $\nu = 4$) of floating-point numbers, and have thus the potential to considerably speed up applications. The most important short vector extensions are shown in Table 1.

Unfortunately, taking advantage of these instructions poses a major challenge to developers of high performance software for the following reasons: 1) Automatic vectorization by a compiler is limited to code of very simple structure; thus, hand coding is required to achieve optimal performance. 2) There is no common C programming interface for different vector extensions; thus, written code is not readily portable. 3) Hand writing vector code requires a high level of programming expertise.

In this paper we present our research on *automatically generating* short vector implementations of DSP transforms including the discrete Fourier transform (DFT), the discrete cosine transform (DCT), the Walsh-Hadamard transform (WHT), and many others. Our approach builds on and extends the library generator SPIRAL to overcome the problems mentioned above.

[*]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.

vendor	name	ν	precision	processor
Intel	SSE	4	single	Pentium III/4, Athlon XP
Intel	SSE2	2	double	Pentium 4
AMD	3DNow!	2	single	K6, K6-II, Athlon
Motorola	AltiVec	4	single	G4

Table 1: Short vector extensions.

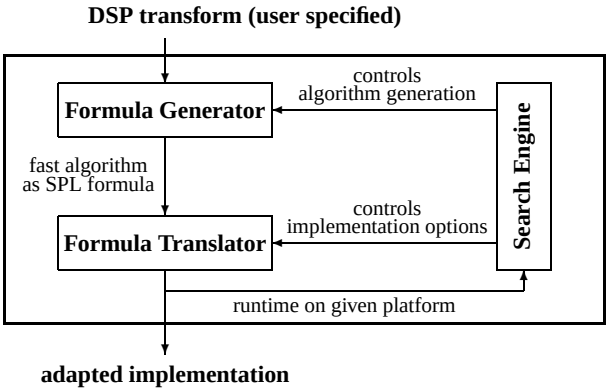

Figure 1: The architecture of SPIRAL.

SPIRAL. The architecture of SPIRAL is displayed in Figure 1. The user specifies a transform she wants to implement and its size, e.g., a DFT of size 256. The rule based **Formula Generator** generates one out of many possible fast algorithms, represented as mathematical formulas in the SPIRAL proprietary language SPL (signal processing language). The SPL program is compiled by the **Formula Translator** (or SPL compiler) into a program in a common language such as C or Fortran. Directives to the Formula Translator control implementation choices such as the degree of unrolling or scheduling. Based on the runtime of the generated program, the **Search Engine** triggers the generation of additional algorithms and their implementations using possibly directives. Iteration of this process produces a program adapted to the given computing platform. In summary, SPIRAL searches in the space of structurally different algorithms and the space of their possible implementations for the best match to the given target architecture. For more information on SPIRAL we refer the reader to [1, 2, 3, 4].

Short Vector Extension of SPIRAL. In this paper we focus on vector code generation with SPIRAL, which is made possi-

ble through two main advances: 1) The mathematical framework that relates mathematical (SPL) formulas to vector code and that provides manipulation rules to modify formulas for vectorization. 2) The implementation of this framework as an extension to the SPL compiler to generate vector code. Intuitively, by approaching vectorization on the "higher" mathematical level, we have access to all structural information necessary to generate very fast code; by using SPIRAL's search mechanism, we find the algorithm that can take best advantage of vector instructions, and is thus fastest on the given platform.

Experimental results show that our automatically generated vector code achieves excellent speed-ups compared to state-of-the-art C code (FFTW [5], or SPIRAL generated), and, for the DFT, outperforms the results from [6] and compares favorably with the hand-tuned Intel vendor library MKL.

A predecessor of our approach is in [7]; further details on vector code generation with SPIRAL can be found in [8].

2. MATHEMATICAL BACKGROUND

We give an overview on SPIRAL's mathematical foundation. The key concept is the representation of fast transform algorithms as mathematical formulas, which can be efficiently generated and manipulated. We use the discrete Fourier transform (DFT) as illustrative example, noting that the framework captures all linear transforms.

Transforms. The current scope of SPIRAL is the domain of linear DSP transforms, which are represented as matrices. Examples include the DFT, discrete cosine transforms (DCTs), Walsh-Hadamard transform (WHT), filters, and many others.

Algorithms: Rules and Formulas. To capture fast transform algorithms, SPIRAL uses mathematical constructs and primitives. Examples for constructs include the *tensor* or *Kronecker product* of matrices, and the *direct sum*, respectively defined by

$$A \otimes B = [a_{k,\ell} \cdot B], \quad \text{for } A = [a_{k,\ell}], \quad A \oplus B = \begin{bmatrix} A & \\ & B \end{bmatrix},$$

and the *conjugation* $A^P = P^{-1}AP$ by a permutation matrix P. Examples of primitives include diagonal matrices, rotation matrices, and permutation matrices, such as the *stride permutation*

$$\mathrm{L}^{mn}_m : jk + i \mapsto im + j, \quad 0 \le i < k,\ 0 \le j < m.$$

Fast algorithms are recursive, like the Cooley-Tukey FFT, written as

$$\mathrm{DFT}_{mn} = (\mathrm{DFT}_m \otimes \mathrm{I}_n)\, \mathrm{T}^{mn}_n (\mathrm{I}_m \otimes \mathrm{DFT}_n)\, \mathrm{L}^{mn}_m, \quad (1)$$

where T^{mn}_n is diagonal [9]. An equation like (1) is called *rule*; a rule expands a transform into other, usually smaller transforms. Another example for a rule is the row-column computation of an arbitrary 2-dimensional transform $T_{\text{2-dim}}$:

$$T_{\text{2-dim}} = T_{\text{1-dim}} \otimes T_{\text{1-dim}}.$$

Recursive application of rules, until all transforms are expanded, yields a *formula*, which represents a fast algorithm. Using rules, formula generation is efficient and fast [4].

Algorithm Search Space. The degree of freedom in recursively expanding a transform leads to a large number of structurally different formulas. This is the algorithmic search space SPIRAL uses for optimization. Examples for search methods include dynamic programming and an evolutionary algorithm [4, 10].

Complex Arithmetic. Several transforms, e.g., the DFT, included in SPIRAL are complex, whereas short vector extensions provide only real arithmetic. Thus, we translate complex formulas into real ones, by replacing every complex number $a + bj$ by the matrix $\left(\begin{smallmatrix} a & -b \\ b & a \end{smallmatrix}\right)$, which corresponds to the interleaved format. This translation can be done formally using an operator $\overline{(\cdot)}$ using mathematical properties. Examples include

$$\overline{A} = A \otimes \mathrm{I}_2, \quad A \text{ real}, \quad (2)$$

$$\overline{A_m \otimes \mathrm{I}_n} = (\overline{A_m \otimes \mathrm{I}_{\frac{n}{\nu}}} \otimes \mathrm{I}_\nu)^{\left(\mathrm{I}_{\frac{mn}{\nu}} \otimes \mathrm{L}^{2\nu}_2\right)}. \quad (3)$$

Formula Manipulation. Formulas can be automatically manipulated using mathematical identities. Examples include

$$\mathrm{L}^{kmn}_n = (\mathrm{L}^{kn}_n \otimes \mathrm{I}_m)(\mathrm{I}_k \otimes \mathrm{L}^{mn}_n), \quad (4)$$

$$(B_m \otimes A_n) = (A_n \otimes B_m)^{\mathrm{L}^{mn}_n}. \quad (5)$$

Formula manipulation is crucial for generating efficient vector code. It provides the means to change the structure of an algorithm with the goal to improve data access patterns and exhibit maximal subformulas that can be vectorized (see Section 3). Formula manipulation is one key distinction between our approach and vectorizing compilers, which do not have access to the mathematical structure and thus produce suboptimal code (see Section 4).

Standard Code Generation. Within SPIRAL, formulas are represented in the language SPL (signal processing language). The SPL compiler translates formulas into code, using the natural correspondence between formula constructs and coding constructs [3]. For example, tensor products with identity matrices are translated into loops. Diagonals and rotations are translated into arithmetic operations while permutation matrices determine the data access. Vector code generation is more challenging and explained in the next section.

3. GENERATING EFFICIENT VECTOR CODE

In this section we present the vector code generation using the SPIRAL system. Three key elements are required:

- A *portable SIMD API* to hide platform specifics of short vector extensions. The API is a set of C macros that provides all vector operations needed to implement DSP transforms, and can itself be implemented on any current short vector architecture. Examples are given in Table 2.
- *Formula vectorization* manipulates a given formula, or algorithm, to obtain a structure suitable for mapping to vector code. This technique is the main distinction to vectorizing compilers, which do not have access to the mathematical structure.
- *Vector code generation* maps the manipulated formula into C code using, to the extent possible, the short vector instructions provided by the SIMD API.

We expand on the last two bullets in the following. Further details are provided in [8].

Formula Vectorization. The goal of this first step in the code generation is to manipulate a given formula to exhibit maximal subformulas that can be mapped into short vector code. Our method is based on the observation that the prototypical short vector construct is given by

$$A \otimes \mathrm{I}_\nu, \quad (6)$$

Macro	Operation
`VEC_ADD(c, a, b)`	vector addition
`LOAD_VECT(r, *m)`	load vector
`DECLARE_CONST(c)`	declare vector constant
`STORE_L_4_2(*m0, *m1, r0, r1)`	load two vectors and permute using L^4_2 ($\nu = 2$)

Table 2: Some operations provided by the portable SIMD API.

where ν is the vector length. Vector code for (6) is obtained by generating scalar code for A and replacing any scalar operation by the respective vector operation. Extending from (6), the following more general construct can be completely vectorized:

$$PD(A \otimes \mathrm{I}_\nu)EQ, \tag{7}$$

with permutation matrices P and Q and scaling matrices D and E that originate from real and complex diagonals or direct sums of rotations. Most formulas, i.e., algorithms for DSP transforms do not directly match (7), and thus have to be manipulated accordingly. We provide a few examples.

The construct $\mathrm{I}_\nu \otimes A$ is transformed into $(A \otimes \mathrm{I}_\nu)^{\mathrm{L}^{n\nu}_n}$ using identity (5). Both complex diagonals and direct sums of rotation matrices D of length ν (which are due to the bar operator essentially the same) are manipulated as

$$\overline{D}' = \overline{D}^{(\mathrm{L}^{2\nu}_\nu)}, \quad D = \mathrm{diag}(c_0, \ldots, c_{\nu-1}),\ c_i \in \mathbb{C}.$$

The result $\overline{D}'$ is structurally equivalent to $A \otimes \mathrm{I}_\nu$ with a 2×2 matrix A (although the actual numbers vary). Larger diagonals are converted into direct sums of diagonals of length ν and transformed blockwise.

Equation (4) provides a way of factoring stride permutations into two permutations: a permutation $\mathrm{L}^{kn}_n \otimes \mathrm{I}_m$ of blocks, and a permutation $\mathrm{I}_k \otimes \mathrm{L}^{mn}_n$ that operates within blocks. When k, m, and n are appropriate, the first permutation can be implemented by renaming vector variables while the second permutation is implemented using primitives provided by the portable SIMD API. Factoring the occurring permutations in the right way is the key to performance. Note that not all permutations can be factored into these two classes of operations while keeping the number of operations low.

Using formula manipulation, DFTs, WHTs, and multidimensional transforms of 2-power size can be built exclusively from constructs that match equation (7) and can thus be completely vectorized [8].

Generating Vector Code. In this final step, manipulated formulas are translated into short vector code. Any formula at this stage consists of two types of constructs: 1) vectorizable constructs that match equation (7), and 2) non-vectorizable constructs. The non-vectorizable constructs are translated into standard C code utilizing the standard FPU. In the remainder of this section we briefly discuss how vectorizable constructs are translated into code.

As explained above, the implementation of $A \otimes \mathrm{I}_\nu$ is straightforward by generating scalar code for the construct A using the scalar SPL compiler [3], and then replacing all scalar operations by their respective vector operations using the SIMD API.

Both permutation matrices and scaling matrices in (7) are handled as pre- or postprocessing operations to the memory access. This uses the facts, that vector loads and stores have to be issued explicitly, and that any permutation and scaling matrix is by construction close to such an operation as provided by the portable SIMD API.

Efficient implementation of permutations across different short vector extensions is one of the main challenges in obtaining fast vector code. All current short vector extensions support some (and different) in-vector permutations in their hardware. Some extensions support unaligned memory access and some types of sub-vector memory access with moderate performance penalty while on other extensions unaligned access and sub-vector access has to be built from multiple aligned vector memory access operations leading to extreme performance degradation. The portable SIMD API handles these problems efficiently.

4. EXPERIMENTAL RESULTS

We benchmarked SPIRAL generated vector code on three different IA-32 compatible machines with very different architectures: a 1 GHz Pentium III (Coppermine core), a 2.53 GHz Pentium 4, and a 1.73 GHz Athlon XP 2100+. All three machines feature the four-way single-precision SSE extension; the Pentium 4 additionally features the two-way double-precision SSE2. We ran experiments for 2-power problem sizes for DFTs, WHTs and 2D-DCTs (type II) on all machines for problem sizes that fit into the L2 cache. For all experiments the Intel C++ Compiler 6.0 was used. The SPIRAL generated code was found by dynamic programming searches including a search for the threshold for loop unrolling.

The performance results are given in Figure 2, for the DFT_{2^n} in pseudo Gflop/s ($5n2^n$/runtime), else in Gflop/s; thus, higher is better. For all transforms we compared (legend identifier, line style): SPIRAL generated C code (SPIRAL C, solid, diamond); SPIRAL generated C code with C compiler vectorization (SPIRAL C vect, dotted, bullet); SPIRAL generated SSE or SSE2 code (SPIRAL SSE or SSE2, solid, square); for the DFT, (a)–(d), we included FFTW 2.1.3 C code (FFTW, dash-dotted, x) using the built-in adaptation; and Intel MKL 5.1 SSE/SSE2 code (MKL SSE/SSE2, dashed, triangle); for DFT on Pentium 4, (a), we included the runtimes from [6] (Rodriguez, dash-dotted, star), linearly scaled up from 1.4 GHz to 2.53 GHz, as we could not get the code for benchmarking. We summarize the main observations.

- Using C compiler vectorization in tandem with SPIRAL C code generation (SPIRAL C vect) in general improves performance, but is far from being optimal. (As an aside, compiler vectorization of FFTW does not improve its performance.)
- SPIRAL generated vector code achieves excellent speed-ups over the fastest scalar codes on all platforms for all considered transforms. For real four-way extensions (SSE on Pentium III and Pentium 4) we achieve up to a factor of 3.3, and for two-way extensions (including SSE on the Athlon XP, which is implemented on top of 3DNow!) we achieve up to a factor of 1.8. The best performance was measured for a DFT_{128} on Pentium 4 with 6.25 pseudo Gflop/s (single precision SSE).
- For the DFT, with few exceptions, our generated code outperforms the Intel vendor library across platforms and across the sizes considered. We note that the MKL computes the DFT in-place (SPIRAL code is out-of-place) and also uses prefetching instructions.

In a final experiment we demonstrate the necessity for platform-adaptation. We cross-timed SPIRAL generated code—found for different platforms/datatypes—on a Pentium 4, implemented in SSE. Figure 3 shows the slowdown factors (i.e., the penalty payed by

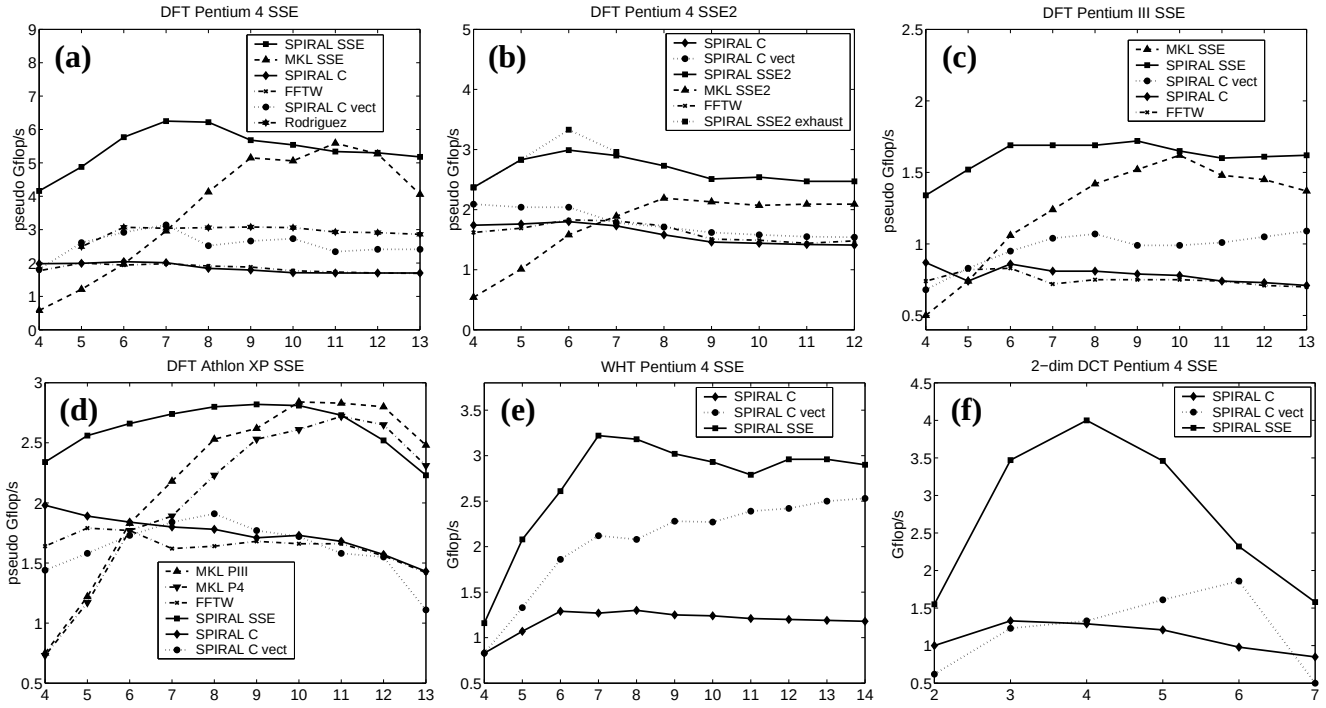

Figure 2: Performance results: (a)–(d): DFT_{2^n}, $4 \leq n \leq 12/13$ (pseudo Gflop/s); (e): WHT_{2^n}, $4 \leq n \leq 14$ (Gflop/s); (f) 2D-$\mathrm{DCT}_{2^n \times 2^n}$, $2 \leq n \leq 7$ (Gflop/s). Platforms: 1 GHz Pentium III, 2.53 Ghz Pentium 4, and 1.73 Ghz Athlon XP 2100+. Higher is better.

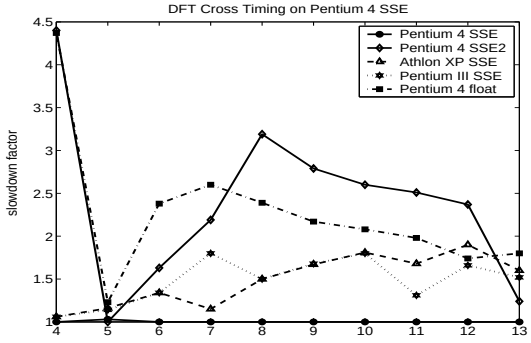

Figure 3: Slowdown factor of the best algorithms for DFT_{2^n}, $4 \leq n \leq 13$, found for different architectures and data types, measured on Pentium 4, implemented using SSE.

adapting to another platform and/or datatype and implementing on Pentium 4, SSE). As expected, the scalar formulas and the SSE2 formulas performed very bad. However, it is intriguing that the SSE optimized formulas, found for Pentium III or Athlon XP, performed up to 1.6 times slower than the Pentium 4 SSE adapted formulas.

5. REFERENCES

[1] J. M. F. Moura, J. Johnson, R. Johnson, D. Padua, V. Prasanna, M. Püschel, and M. M. Veloso, "SPIRAL: Automatic Library Generation and Platform-Adaptation for DSP Algorithms," 1998, http://www.ece.cmu.edu/∼spiral.

[2] M. Püschel, B. Singer, J. Xiong, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, and R. W. Johnson, "SPIRAL: A Generator for Platform-Adapted Libraries of Signal Processing Algorithms," *to appear in* Journal of High Performance Computing and Applications, 2003.

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

[4] 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.

[5] M. Frigo and S. G. Johnson, "FFTW: An adaptive software architecture for the FFT," in *Proc. ICASSP*, 1998, vol. 3, pp. 1381–1384, http://www.fftw.org.

[6] P. Rodriguez, "A Radix-2 FFT Algorithm for Modern Single Instruction Multiple Data (SIMD) Architectures," in *Proc. ICASSP*, 2002.

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

[8] F. Franchetti and M. Püschel, "A SIMD Vectorizing Compiler for Digital Signal Processing Algorithms," in *Proc. IPDPS*, 2002.

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

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