A Rewriting System for the Vectorization of Signal Transforms

Franz Franchetti, Yevgen Voronenko, and Markus Püschel*

Electrical and Computer Engineering,
Carnegie Mellon University,
5000 Forbes Avenue, Pittsburgh, PA 15213
{franzf, yvoronen, pueschel}@ece.cmu.edu
WWW home page: http://www.spiral.net

Abstract. We present a rewriting system that automatically vectorizes signal transform algorithms at a high level of abstraction. The input to the system is a transform algorithm given as a formula in the well-known Kronecker product formalism. The output is a "vectorized" formula, which means it consists exclusively of constructs that can be directly mapped into short vector code. This approach obviates compiler vectorization, which is known to be limited in this domain. We included the formula vectorization into the Spiral program generator for signal transforms, which enables us to generate vectorized code and optimize through search over alternative algorithms. Benchmarks for the discrete Fourier transform (DFT) show that our generated floating-point code is competitive with and that our fixed-point code clearly outperforms the best available libraries.

1 Introduction

Most recent architectures feature short vector SIMD instructions that provide data types and instructions for the parallel execution of scalar operations in short vectors of length ν (called ν-way). For example Intel's SSE family provides $\nu = 2$ for double precision and $\nu = 4$ for single precision floating point arithmetic as well as $\nu = 8$ for 16-bit and $\nu = 16$ for 8-bit integer arithmetic. The potential speed-up offered by these instructions makes them attractive in domains were high performance is crucial, but they come at a price: Compilers often cannot make optimal use of vector instructions, since the necessary program transformations are not well understood; this moves the burden to the programmer, who is required to leave the standard C programming model, for example by using so-called intrinsics interfaces to the instruction set.

In [1] we have argued that for the specific domain of signal transforms such as the discrete Fourier transform (DFT) there is an attractive solution to this

* This work was supported by NSF through awards 0234293, 0325687, and by DARPA through the Department of Interior grant NBCH1050009. Franz Franchetti was supported by the Austrian Science Fund FWF's Erwin Schroedinger Fellowship J2322.

problem, namely to perform the vectorization at a higher level of abstraction by manipulating Kronecker product expressions through mathematical identities. The Kronecker product formalism has been known to be useful for the representation and derivation for DFT algorithms [2] but also in the derivation of parallel algorithms [3].

In this paper we describe an implementation of this formal vectorization in the form of a rewriting system [4], the common tool used in symbolic computation. We then include the rewriting system into the Spiral program generator [5], which uses the Kronecker product formalism as internal algorithm representations. This enables us to automatically generate optimized vectorized code through the search over available alternatives provided by Spiral. We show that our approach works for the DFT (speed-up in parenthesis) for 2-way (1.5 times) and 4-way floating point (3 times), and 8-way (5 times) and 16-way (6 times) integer code. Benchmarks of our generated code against the Intel MKL and IPP libraries and FFTW [6] show that our generated floating-point code is competitive and our generated fixed-point code is at least a factor of 2 faster than the vendor library.

Related work. The Intel C++ compiler includes a vectorizer based on loop vectorization and translation of complex operations into two-way vector code. FFTW 3.0.1 combines loop vectorization and an approach based on the linearity of the DFT to obtain 4-way single-precision SSE code [6]. This combines a hardcoded vectorization approach with FFTW's capability to automatically tune for the memory hierarchy. An approach to designing embedded processors with vector SIMD instructions and for designing software for these processors is presented in [7].

Organization of the paper. In Section 2 we provide background on SIMD vector instructions, signal transforms and their fast algorithms, and the Spiral program generator. The rewriting system is explained in Section 3 including examples of vectorization rules and vectorized formulas. Section 4 shows a number of runtime benchmarks for the DFT. We offer conclusions in Section 5

2 Background

SIMD vector instructions. Recently, major vendors of general purpose microprocessors have included short vector SIMD (single instruction, multiple data) extensions into their instruction set architecture. Examples of SIMD extensions include Intel's MMX and SSE family, AMD's 3DNow! family, and Motorola's AltiVec extension. SIMD extensions have the potential to speed up implementations in areas where the relevant algorithms exhibit fine grain parallelism but are a major challenge to software developers.

In this paper we denote the vector length with ν. For example, SSE2 provides 2-way ($\nu = 2$) double and 4-way ($\nu = 4$) single precision floating point as well as 8-way ($\nu = 8$) 16-bit and 16-way ($\nu = 16$) 8-bit integer vector instructions.

Signal transforms. A (linear) signal transform is a matrix-vector multiplication $x \mapsto y = Mx$, where x is a real or complex input vector, M the transform matrix, and y the result. Examples of transforms include the discrete Fourier transform (DFT), multi-dimensional DFTs (MDDFT), the Walsh-Hadamard transform (WHT), and discrete Wavelet transforms (DWT) like the Haar wavelet. For example, for an input vector $x \in \mathbb{C}^n$, the DFT is defined by the matrix

$$\mathrm{DFT}_n = [\omega_n^{k\ell}]_{0 \le k,\ell < n}, \quad \omega_n = \exp(-2\pi i/n).$$

Algorithms for transforms can be written using the Kronecker product formalism [2, 3, 5] in the form of structured sparse matrix factorizations. In the following, we use I_n to denote an $n \times n$ identity matrix, and

$$A \otimes B = [a_{k\ell} B],\ A = [a_{k\ell}]$$

for the tensor product of matrices. Further we introduce the stride permutation matrix defined, for $m|n$, by

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

Equations (1)–(6) show examples of recursive transform algorithms, written in the form of rules:

$$\begin{aligned}
\mathrm{DFT}_{mn} &\to \big(\mathrm{DFT}_m \otimes \mathrm{I}_n\big) D_{m,n} \big(\mathrm{I}_m \otimes \mathrm{DFT}_n\big)\, \mathrm{L}_m^{nm} && (1)\\
\mathrm{DFT}_n &\to X_n\, \mathrm{RDFT}_n && (2)\\
\mathrm{WHT}_{mn} &\to \mathrm{WHT}_m \otimes \mathrm{WHT}_n && (3)\\
\mathrm{MDDFT}_{n_1 \times \cdots \times n_k} &\to \mathrm{MDDFT}_{n_1 \times \cdots \times n_r} \otimes \mathrm{MDDFT}_{n_{r+1} \times \cdots \times n_k} && (4)\\
\mathrm{MDDFT}_n &\to \mathrm{DFT}_n && (5)\\
\mathrm{Haar}_n &\to \mathrm{L}_2^n \big(\mathrm{I}_{n/2} \otimes \mathrm{DFT}_2\big) && (6)
\end{aligned}$$

In (1), $D_{m,n}$ is a complex diagonal matrix [2]. In (2), RDFT is the real version of the DFT (i.e., for real input) and X_n is an X-shaped matrix containing only the entries 0, ± 1, $\pm i$ [2]. The WHT is a real matrix and exists only for two-power size. $\mathrm{WHT}_2 = \mathrm{DFT}_2$ together with (3) defines the transform. In (4), the transform takes as input a $n_1 \times \cdots \times n_k$ array, stored linearized in a vector.

Spiral. Recursive application of rules like (1)–(6) yields many different algorithms for a given transform. Spiral [5] uses this fact to search for the fastest one on a given platform. A user-specified transform (like DFT_{256}) is expanded by Spiral using rules into a formula, which is then translated into a C program by a special formula compiler. The runtime of the program is measured and fed into a search module, which triggers, in a feedback loop, the generation of a modified formula based on a search strategy. Upon termination, Spiral outputs the fastest program found.

In this paper, we explain a crucial module in Spiral: A rewriting system that manipulates formulas to enable their direct compilation into SIMD vector code, which obviates the need for compiler vectorization.

Complex arithmetic. To describe complex transforms in terms of real arithmetic, we represent complex data vectors as real vectors using the interleaved complex format (alternating real and imaginary parts of the complex entries). Since the complex multiplication $(u + iv)(y + iz)$ is equivalent to the real multiplication $\left[\begin{smallmatrix} u & -v \\ v & u \end{smallmatrix}\right]\left[\begin{smallmatrix} y \\ z \end{smallmatrix}\right]$, we can write the complex matrix-vector multiplication $Mx \in \mathbb{C}^n$ as $\overline{M}x' \in \mathbb{R}^{2n}$, where we define $\overline{M}$ by replacing every entry $u + iv$ as $\left[\begin{smallmatrix} u & -v \\ v & u \end{smallmatrix}\right]$, and x' is x in the interleaved complex format.

3 Vectorization through Rewriting

Our goal is to take formulas obtained by the recursive application of rules like (1)–(6) and automatically manipulate them into a form that enables a direct mapping into SIMD vector code. Further, we also want to explore different vectorizations for the same formula. The solution is a suitably designed rewriting system that implements our previous ideas for formula-based vectorization in [1, 8].

Formula vectorization: The basic idea. The central formula construct that can be implemented on all ν-way short vector extensions is

$$A \otimes \mathrm{I}_\nu, \tag{7}$$

where A is an arbitrary real matrix. Vector code is obtained by generating scalar code for A (i.e., for $x \mapsto Ax$) and replacing all scalar operations by their respective ν-way vector operations. For example, `c=a+b` is replaced by `c=vadd(a,b)`.

Of course, most formulas do not match (7). In these cases we manipulate the formula using rewriting rules to consist of components that either match (7) or are among a small set of base cases. It turns out that for a large class of formulas the only base cases needed are

$$\mathrm{L}_2^{2\nu},\ \mathrm{L}_\nu^{2\nu},\ \mathrm{L}_\nu^{\nu^2},\ (\mathrm{I}_{n/\nu} \otimes \mathrm{L}_2^{2\nu})\overline{D_n}(\mathrm{I}_{n/\nu} \otimes \mathrm{L}_\nu^{2\nu}), \tag{8}$$

where D_n is any complex diagonal matrix. For $\nu = 2$ we also need the additional base case

$$\overline{[1, i]} = \begin{bmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & 1 & 0 \end{bmatrix}. \tag{9}$$

We assume that vectorized implementations of (8) are available. Note that $\mathrm{I}_m \otimes \mathrm{L}_2^{2\nu}$ converts a real vector $x' \in \mathbb{R}^{2m\nu}$ (which originates from a complex vector $x \in \mathbb{C}^{m\nu}$) from interleaved complex format into a block-interleaved complex format with block size ν. Analogously, $\mathrm{I}_m \otimes \mathrm{L}_\nu^{2\nu}$ converts back from block-interleaved into the interleaved complex format.

Definition 1. *We call a formula vectorized if it is either of the form* (7) *or one of the forms in* (8) *and* (9)*, or of the form*

$$\mathrm{I}_m \otimes A \textit{ or } AB, \tag{10}$$

where A and B are vectorized.

Formula manipulation. We vectorize formulas through formula manipulation using well-known mathematical identities such as (we assume that A is $n \times n$ and B and C are $m \times m$)

$$\begin{aligned} \mathrm{I}_{mn} &= \mathrm{I}_m \otimes \mathrm{I}_n && (11)\\ \mathrm{I}_m \otimes A &= \mathrm{L}^{mn}_m (A \otimes \mathrm{I}_m)\, \mathrm{L}^{mn}_n && (12)\\ \mathrm{L}^{kmn}_n &= (\mathrm{L}^{kn}_n \otimes \mathrm{I}_m)(\mathrm{I}_k \otimes \mathrm{L}^{mn}_n) && (13)\\ \mathrm{L}^{kmn}_{km} &= (\mathrm{I}_k \otimes \mathrm{L}^{mn}_m)(\mathrm{L}^{kn}_k \otimes \mathrm{I}_m) && (14)\\ A \otimes (BC) &= (A \otimes B)(A \otimes C) && (15)\\ A \otimes B &= (A \otimes \mathrm{I}_m)(\mathrm{I}_n \otimes B) = (\mathrm{I}_n \otimes B)(A \otimes \mathrm{I}_m) && (16) \end{aligned}$$

As a small example, we assume A is a real $n \times n$ matrix, and vectorize

$$\mathrm{I}_m \otimes A \qquad (17)$$

for a ν-way vector instruction set.

We first apply (11) to obtain

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

and then apply (12) to $\mathrm{I}_\nu \otimes A$ to get

$$\mathrm{I}_{m/\nu} \otimes \big(\mathrm{L}^{n\nu}_\nu (A \otimes \mathrm{I}_\nu)\, \mathrm{L}^{n\nu}_n \big). \qquad (18)$$

Note that in this formula $A \otimes \mathrm{I}_\nu$ is already vectorized, but the stride permutations are not. To vectorize the stride permutations, we apply (13) and (14) to get

$$\mathrm{I}_{m/\nu} \otimes \Big((\mathrm{L}^n_\nu \otimes \mathrm{I}_\nu)(\mathrm{I}_{n/\nu} \otimes \mathrm{L}^{\nu^2}_\nu)(A \otimes \mathrm{I}_\nu)(\mathrm{I}_{n/\nu} \otimes \mathrm{L}^{\nu^2}_\nu)(\mathrm{L}^n_{n/\nu} \otimes \mathrm{I}_\nu) \Big). \qquad (19)$$

Inspection shows that this formula is vectorized in the sense of Definition 1.

3.1 Rewriting System

Our goal is to *automatically* apply formula identities like (12)–(16) to transform given formulas into vectorized formulas. Note that the order and actual parameters chosen for each of the applied identities is a nontrivial choice. Only the correct choice will lead to vectorized formulas. Thus, automatic formula manipulation requires an appropriately designed rewriting system [4]. Specifically, it is a difficult problem to identify the right objects and rules in the system to guarantee that it is confluent and converges to fully vectorized formulas, when possible.

Vector tags. We introduce a set of tags to propagate vectorization information through the formulas and to perform algebraic simplification of permutations. Note that all objects remain matrices.

We tag a formula construct A to be translated into vector code for vector length ν by

$$\underbrace{A}_{\mathrm{vec}(\nu)} = A.$$

Further, we write

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

to stipulate that the tensor product is to be mapped into vector code as explained in Section 3 Using these tags, the base cases in (8) are expressed by

$$\underbrace{\mathrm{L}_2^{2\nu}}_{\mathrm{vec}(\nu)}, \; \underbrace{\mathrm{L}_\nu^{2\nu}}_{\mathrm{vec}(\nu)}, \; \underbrace{\mathrm{L}_\nu^{\nu^2}}_{\mathrm{vec}(\nu)}, \; \overline{\Big(\underbrace{D_n}_{\mathrm{vec}(\nu)}\Big)}^{\nu}, \; \underbrace{\overline{[1,i]}}_{\mathrm{vec}(2)}\,.$$

In addition we need variants of the operator $\overline{(.)}$ to handle the vectorization of complex formulas (A is assumed to be $n \times n$)

$$\overleftrightarrow{A}^{\nu} = \overline{A} \tag{20}$$

$$\overrightarrow{A}^{\nu} = (\mathrm{I}_{n/\nu} \otimes \mathrm{L}_2^{2\nu})\overline{A} \tag{21}$$

$$\overleftarrow{A}^{\nu} = \overline{A}(\mathrm{I}_{n/\nu} \otimes \mathrm{L}_\nu^{2\nu}) \tag{22}$$

$$\overline{A}^{\nu} = (\mathrm{I}_{n/\nu} \otimes \mathrm{L}_2^{2\nu})\overline{A}(\mathrm{I}_{n/\nu} \otimes \mathrm{L}_\nu^{2\nu}). \tag{23}$$

(20)–(23) are the four variants of $\overline{A}$ that have either interleaved or block-interleaved input and output format. The format conversions introduce the building blocks $\mathrm{L}_2^{2\nu}$ and $\mathrm{L}_\nu^{2\nu}$ defined in (8). A key idea in our rewriting system is to minimize these format conversions by applying the identity

$$\mathrm{L}_2^{2\nu}\,\mathrm{L}_\nu^{2\nu} = \mathrm{I}_{2\nu}\,.$$

To facilitate this simplification we introduce (20)–(23) as objects into our rewriting system. Rules (31)–(44) operate on the constructs (20)–(23) and encode the knowledge where to introduce $\mathrm{L}_2^{2\nu}$ and $\mathrm{L}_\nu^{2\nu}$ to minimize format conversion overhead.

Table 1. Stride permutation rules.

$$\underbrace{\mathrm{L}_n^{n\nu}}_{\mathrm{vec}(\nu)} \to \big(\mathrm{I}_{n/\nu} \otimes \underbrace{\mathrm{L}_\nu^{\nu^2}}_{\mathrm{vec}(\nu)}\big)\big(\mathrm{L}_{n/\nu}^{n} \otimes_\nu \mathrm{I}_\nu\big) \tag{24}$$

$$\underbrace{\mathrm{L}_\nu^{n\nu}}_{\mathrm{vec}(\nu)} \to \big(\mathrm{L}_\nu^{n} \otimes_\nu \mathrm{I}_\nu\big)\big(\mathrm{I}_{n/\nu} \otimes \underbrace{\mathrm{L}_\nu^{\nu^2}}_{\mathrm{vec}(\nu)}\big) \tag{25}$$

$$\underbrace{\mathrm{L}_m^{mn}}_{\mathrm{vec}(\nu)} \to \big(\mathrm{L}_m^{mn/\nu} \otimes_\nu \mathrm{I}_\nu\big)\big(\mathrm{I}_{mn/\nu^2} \otimes \underbrace{\mathrm{L}_\nu^{\nu^2}}_{\mathrm{vec}(\nu)}\big)\big((\mathrm{I}_{n/\nu} \otimes \mathrm{L}_{m/\nu}^{m}) \otimes_\nu \mathrm{I}_\nu\big) \tag{26}$$

Rules. The goal is to vectorize a given formula. In our rewriting system, this is done by tagging the formula with vec(ν) and apply rules that vectorize the formula, i.e., rewrite it such that the only components tagged with vec(ν) are vectorizable base cases. Rules are applied by matching the left side of a rule

Table 2. Tensor product rules. A is an $n \times n$ matrix.

$$\big(\underbrace{A \otimes \mathrm{I}_m}_{\mathrm{vec}(\nu)}\big) \rightarrow \big(A \otimes \mathrm{I}_{m/\nu}\big) \otimes_\nu \mathrm{I}_\nu \tag{27}$$

$$\big(\underbrace{\mathrm{I}_m \otimes A}_{\mathrm{vec}(\nu)}\big) \rightarrow \begin{cases} \mathrm{I}_{m/\nu} \otimes \underbrace{\big(\mathrm{I}_\nu \otimes A\big)}_{\mathrm{vec}(\nu)} \\ \underbrace{\mathrm{L}^{mn}_m}_{\mathrm{vec}(\nu)} \big(\underbrace{A \otimes \mathrm{I}_m}_{\mathrm{vec}(\nu)}\big) \underbrace{\mathrm{L}^{mn}_n}_{\mathrm{vec}(\nu)} \end{cases} \tag{28}$$

$$\underbrace{\big(\mathrm{I}_m \otimes A\big)\, \mathrm{L}^{mn}_m}_{\mathrm{vec}(\nu)} \rightarrow \begin{cases} \underbrace{\mathrm{L}^{mn}_m}_{\mathrm{vec}(\nu)} \big(\underbrace{A \otimes \mathrm{I}_m}_{\mathrm{vec}(\nu)}\big) \\ \Big(\mathrm{I}_{m/\nu} \otimes \underbrace{\mathrm{L}^{n\nu}_\nu}_{\mathrm{vec}(\nu)} \big(A \otimes_\nu \mathrm{I}_\nu\big)\Big)\big(\mathrm{L}^{mn/\nu}_{m/\nu} \otimes_\nu \mathrm{I}_\nu\big) \end{cases} \tag{29}$$

$$\underbrace{\Big(\mathrm{I}_k \otimes \big(\mathrm{I}_m \otimes A^{n \times n}\big)\, \mathrm{L}^{mn}_m\Big)\, \mathrm{L}^{kmn}_k}_{\mathrm{vec}(\nu)} \rightarrow \big(\underbrace{\mathrm{L}^{km}_k \otimes \mathrm{I}_n}_{\mathrm{vec}(\nu)}\big)\Big(\mathrm{I}_m \otimes \underbrace{\big(\mathrm{I}_k \otimes A^{n \times n}\big)\, \mathrm{L}^{kn}_k}_{\mathrm{vec}(\nu)}\Big)\big(\underbrace{\mathrm{L}^{mn}_m \otimes \mathrm{I}_k}_{\mathrm{vec}(\nu)}\big) \tag{30}$$

Table 3. Bar operator rules (left: recursive; right: base cases). A is an $n \times n$ matrix.

$$\underbrace{(\overline{A})}_{\mathrm{vec}(\nu)} \rightarrow \overleftrightarrow{(\underbrace{\ A\ }_{\mathrm{vec}(\nu)})}^{\nu} \tag{31}$$

$$\overleftrightarrow{AB}^{\nu} \rightarrow \overleftarrow{A}^{\nu}\overrightarrow{B}^{\nu} \tag{32}$$

$$\overleftarrow{AB}^{\nu} \rightarrow \overleftarrow{A}^{\nu}\overline{B}^{\nu} \tag{33}$$

$$\overrightarrow{AB}^{\nu} \rightarrow \overline{A}^{\nu}\overrightarrow{B}^{\nu} \tag{34}$$

$$\overline{AB}^{\nu} \rightarrow \overline{A}^{\nu}\overline{B}^{\nu} \tag{35}$$

$$\overleftrightarrow{\mathrm{I}_m \otimes A}^{\nu} \rightarrow \mathrm{I}_m \otimes \overleftrightarrow{A}^{\nu} \tag{36}$$

$$\overleftarrow{\mathrm{I}_m \otimes A}^{\nu} \rightarrow \mathrm{I}_m \otimes \overleftarrow{A}^{\nu} \tag{37}$$

$$\overline{\mathrm{I}_m \otimes A}^{\nu} \rightarrow \mathrm{I}_m \otimes \overline{A}^{\nu} \tag{38}$$

$$\overrightarrow{\mathrm{I}_m \otimes A}^{\nu} \rightarrow \mathrm{I}_m \otimes \overrightarrow{A}^{\nu} \tag{39}$$

$$\overleftarrow{A \otimes_\nu \mathrm{I}_\nu}^{\nu} \rightarrow (\mathrm{I}_{n/\nu} \otimes \underbrace{\mathrm{L}^{2\nu}_\nu}_{\mathrm{vec}(\nu)})(\overline{A} \otimes_\nu \mathrm{I}_\nu) \tag{40}$$

$$\overline{A \otimes_\nu \mathrm{I}_\nu}^{\nu} \rightarrow \overline{A} \otimes_\nu \mathrm{I}_\nu \tag{41}$$

$$\overrightarrow{A \otimes_\nu \mathrm{I}_\nu}^{\nu} \rightarrow (\overline{A} \otimes_\nu \mathrm{I}_\nu)(\mathrm{I}_{n/\nu} \otimes \underbrace{\mathrm{L}^{2\nu}_2}_{\mathrm{vec}(\nu)}) \tag{42}$$

$$\overline{\mathrm{L}^{mn}_m} \rightarrow \mathrm{L}^{mn}_m \otimes \mathrm{I}_2 \tag{43}$$

$$\overline{(\underbrace{\mathrm{L}^{\nu^2}_\nu}_{\mathrm{vec}(\nu)})}^{\nu} \rightarrow (\mathrm{L}^{2\nu}_\nu \otimes_\nu \mathrm{I}_\nu)(\mathrm{I}_2 \otimes \underbrace{\mathrm{L}^{\nu^2}_\nu}_{\mathrm{vec}(\nu)})(\mathrm{L}^{2\nu}_2 \otimes_\nu \mathrm{I}_\nu) \tag{44}$$

against a given expression, extracting the parameters defined in the left side, and replacing it with one of the choices in the right side parameterized by the extracted parameters. Most of our rewriting rules are shown in Tables 1–3.

The important difference between identities like (11)–(16) and rules like (24)–(44) is that the rules encode the decisions *how* to apply the identities, i.e., fix the choice of parameters. For instance, both identities (13) and (14) can be applied to $\mathrm{L}^{n\nu}_n$ for composite n and ta wo-power ν; however, only (14) with the specific choice $k = m/\nu$, $m = \nu$, and $n = \nu$ leads to a vectorized result:

$$\mathrm{L}^{n\nu}_n = (\mathrm{I}_{n/\nu} \otimes \mathrm{L}^{\nu^2}_\nu)(\mathrm{L}^{n}_{n/\nu} \otimes \mathrm{I}_\nu).$$

This knowledge is encoded in rule (24) which chooses the right parameters.

A similar degree of freedom for choosing m and n applies to identity (11) and the corresponding knowledge to choose the right factorization is encoded in first alternatives of rules (27) and (28).

Simple example. We return to our previous example (17) and explain how it is handled by our rewriting system. We start with the tagged formula

$$\underbrace{\mathrm{I}_m \otimes A}_{\mathrm{vec}(\nu)},$$

which means "$\mathrm{I}_m \otimes A$ is to be vectorized." The system can only apply one of the alternatives of rule (28). Suppose it chooses the first alternative, which yields

$$\mathrm{I}_{m/\nu} \otimes (\underbrace{\mathrm{I}_\nu \otimes A}_{\mathrm{vec}(\nu)})$$

and then applies the second alternative of (28) to $(\mathrm{I}_\nu \otimes A)$, which leads to

$$\mathrm{I}_{m/\nu} \otimes \big(\underbrace{\mathrm{L}^{n\nu}_{\nu}}_{\mathrm{vec}(\nu)} (A \otimes_\nu \mathrm{I}_\nu) \underbrace{\mathrm{L}^{n\nu}_{n}}_{\mathrm{vec}(\nu)}\big).$$

Next, only rules (24) and (25) match, which yields

$$\mathrm{I}_{m/\nu} \otimes \Big((\mathrm{L}^n_\nu \otimes_\nu \mathrm{I}_\nu)(\mathrm{I}_{n/\nu} \otimes \underbrace{\mathrm{L}^{\nu^2}_{\nu}}_{\mathrm{vec}(\nu)})(A \otimes_\nu \mathrm{I}_\nu)(\mathrm{I}_{n/\nu} \otimes \underbrace{\mathrm{L}^{\nu^2}_{\nu}}_{\mathrm{vec}(\nu)})(\mathrm{L}^n_{n/\nu} \otimes_\nu \mathrm{I}_\nu)\Big),$$

which is the properly tagged version of the vectorized formula (19).

Example: DFT. We now show how our rewriting system vectorizes DFT_{mn} with $\nu^2 \mid mn$. The vectorization process has to overcome three crucial problems for an arbitrary two-power ν: 1) handle the interleaved complex format, 2) vectorize the stride permutation, and 3) vectorize the complex diagonal matrix. Our example shows how to cope with these problems and how to get the short-vector FFT algorithm [8].

The DFT is a complex transform, but vector instructions operate on real vectors. Thus, we have to start with $\overline{\mathrm{DFT}_{mn}}$, tagged for vectorization. First the system commutes the vector tag using (31) and the $\overleftrightarrow{(.)}^{\nu}$ operator and breaks the product using (32)–(39):

$$\begin{aligned}
\underbrace{(\overline{\mathrm{DFT}_{mn}})}_{\mathrm{vec}(\nu)} &\to \underbrace{\Big(\overline{(\mathrm{DFT}_m \otimes \mathrm{I}_n) D_{m,n} (\mathrm{I}_m \otimes \mathrm{DFT}_n)\, \mathrm{L}^{mn}_m}\Big)}_{\mathrm{vec}(\nu)} \\
&\to \overleftrightarrow{\big(\underbrace{(\mathrm{DFT}_m \otimes \mathrm{I}_n) D_{m,n} (\mathrm{I}_m \otimes \mathrm{DFT}_n)\, \mathrm{L}^{mn}_m}_{\mathrm{vec}(\nu)}\big)}^{\nu} \\
&\to \overleftrightarrow{\big((\underbrace{\mathrm{DFT}_m \otimes \mathrm{I}_n}_{\mathrm{vec}(\nu)})\, \underbrace{D_{m,n}}_{\mathrm{vec}(\nu)}\, \underbrace{(\mathrm{I}_m \otimes \mathrm{DFT}_n)\, \mathrm{L}^{mn}_m}_{\mathrm{vec}(\nu)}\big)}^{\nu} \\
&\to \overleftarrow{(\underbrace{\mathrm{DFT}_m \otimes \mathrm{I}_n}_{\mathrm{vec}(\nu)})}^{\nu}\, \overline{(\underbrace{D_{m,n}}_{\mathrm{vec}(\nu)})}^{\nu}\, \overrightarrow{\underbrace{(\mathrm{I}_m \otimes \mathrm{DFT}_n)\, \mathrm{L}^{mn}_m}_{\mathrm{vec}(\nu)}}^{\nu}
\end{aligned}$$

We now continue with the three factors separately. The system applies (27) and (40) to the first factor

$$\begin{aligned}
\overleftarrow{(\underbrace{\mathrm{DFT}_m \otimes \mathrm{I}_n}_{\mathrm{vec}(\nu)})}^{\nu} &\to \overleftarrow{(\mathrm{DFT}_m \otimes \mathrm{I}_{n/\nu}) \otimes_\nu \mathrm{I}_\nu}^{\nu} \\
&\to (\mathrm{I}_{mn/\nu} \otimes \underbrace{\mathrm{L}^{2\nu}_{\nu}}_{\mathrm{vec}(\nu)})(\overline{\mathrm{DFT}_m \otimes \mathrm{I}_{n/\nu}} \otimes_\nu \mathrm{I}_\nu)
\end{aligned}$$

which is vectorized. The second factor is already vectorized. For the third factor, suppose the system chooses the second alternative of (30) and then breaks and propagates $\overrightarrow{(.)}^{\nu}$ using (32)–(39):

$$\begin{aligned}
\underbrace{\overrightarrow{(\mathrm{I}_m \otimes \mathrm{DFT}_n)\, \mathrm{L}^{mn}_m}^{\nu}}_{\mathrm{vec}(\nu)} &\to \overrightarrow{\big(\mathrm{I}_{m/\nu} \otimes \underbrace{\mathrm{L}^{n\nu}_{\nu}}_{\mathrm{vec}(\nu)} (\mathrm{DFT}_n \otimes_\nu \mathrm{I}_\nu)\big)(\mathrm{L}^{mn}_m \otimes_\nu \mathrm{I}_\nu)}^{\nu} \\
&\to \big(\mathrm{I}_{m/\nu} \otimes \overline{\underbrace{\mathrm{L}^{n\nu}_{\nu}}_{\mathrm{vec}(\nu)} (\mathrm{DFT}_n \otimes_\nu \mathrm{I}_\nu)}^{\nu}\big)\overrightarrow{(\mathrm{L}^{mn}_m \otimes_\nu \mathrm{I}_\nu)}^{\nu} \\
&\to \big(\mathrm{I}_{m/\nu} \otimes \overline{\underbrace{\mathrm{L}^{n\nu}_{\nu}}_{\mathrm{vec}(\nu)}}^{\nu}\overline{(\mathrm{DFT}_n \otimes_\nu \mathrm{I}_\nu)}^{\nu}\big)\overrightarrow{(\mathrm{L}^{mn}_m \otimes_\nu \mathrm{I}_\nu)}^{\nu}
\end{aligned}$$

We now continue with the factors of the tensor product. One difficult part of the vectorization is the interplay of $\overline{(.)}$ and the stride permutation $\mathrm{L}^{n\nu}_{\nu}$. First, rule (25) factors the stride permutation and then rules (32)–(39) handle $\overline{(.)}^{\nu}$. Finally rules (40)–(44) encode the rather involved manipulation required to completely vectorize:

$$\begin{aligned}
\underbrace{\overline{\mathrm{L}^{n\nu}_{\nu}}^{\nu}}_{\mathrm{vec}(\nu)} &\to \overline{(\underbrace{\mathrm{L}^{n}_{\nu} \otimes \mathrm{I}_\nu}_{\mathrm{vec}(\nu)})(\mathrm{I}_{n/\nu} \otimes \underbrace{\mathrm{L}^{\nu^2}_{\nu}}_{\mathrm{vec}(\nu)})}^{\nu} \\
&\to \overline{(\mathrm{L}^{n}_{\nu} \otimes_\nu \mathrm{I}_\nu)}^{\nu}\Big(\mathrm{I}_{n/\nu} \otimes \overline{(\underbrace{\mathrm{L}^{\nu^2}_{\nu}}_{\mathrm{vec}(\nu)})}^{\nu}\Big) \\
&\to (\overline{\mathrm{L}^{n}_{\nu}} \otimes_\nu \mathrm{I}_\nu)\Big(\mathrm{I}_{n/\nu} \otimes (\mathrm{L}^{2\nu}_{\nu} \otimes_\nu \mathrm{I}_\nu)(\mathrm{I}_2 \otimes \underbrace{\mathrm{L}^{\nu^2}_{\nu}}_{\mathrm{vec}(\nu)})(\mathrm{L}^{2\nu}_{2} \otimes_\nu \mathrm{I}_\nu)\Big)
\end{aligned}$$

The vectorization of the remaining constructs is straight-forward using rules (40)–(44):

$$\overline{(\mathrm{DFT}_n \otimes_\nu \mathrm{I}_\nu)}^{\nu} \to \big(\overline{\mathrm{DFT}_n} \otimes_\nu \mathrm{I}_\nu\big)$$

and

$$\begin{aligned}
\overrightarrow{(\mathrm{L}^{mn}_m \otimes_\nu \mathrm{I}_\nu)}^{\nu} &\to (\overline{\mathrm{L}^{mn}_m} \otimes_\nu \mathrm{I}_\nu)\big(\mathrm{I}_{mn/\nu} \otimes \underbrace{\mathrm{L}^{2\nu}_{2}}_{\mathrm{vec}(\nu)}\big) \\
&\to \big((\mathrm{L}^{mn}_m \otimes \mathrm{I}_2) \otimes_\nu \mathrm{I}_\nu\big)\big(\mathrm{I}_{mn/\nu} \otimes \underbrace{\mathrm{L}^{2\nu}_{2}}_{\mathrm{vec}(\nu)}\big).
\end{aligned}$$

Collecting the vectorized formulas yields a completely vectorized FFT

$$
\begin{aligned}
&(\mathrm{I}_{mn/\nu} \otimes \underbrace{\mathrm{L}^{2\nu}_{\nu}}_{\mathrm{vec}(\nu)})(\overline{\mathrm{DFT}_m \otimes \mathrm{I}_{n/\nu}} \otimes_\nu \mathrm{I}_\nu)\overline{\underbrace{(D_{m,n})}_{\mathrm{vec}(\nu)}}^{\nu} \\
&\Big(\mathrm{I}_{m/\nu} \otimes (\overline{\mathrm{L}^{n}_{\nu}} \otimes_\nu \mathrm{I}_\nu)(\mathrm{I}_{n/\nu} \otimes (\mathrm{L}^{2\nu}_{\nu} \otimes_\nu \mathrm{I}_\nu)(\mathrm{I}_2 \otimes \underbrace{\mathrm{L}^{\nu^2}_{\nu}}_{\mathrm{vec}(\nu)})(\mathrm{L}^{2\nu}_{2} \otimes_\nu \mathrm{I}_\nu))(\overline{\mathrm{DFT}_n} \otimes_\nu \mathrm{I}_\nu)\Big) \\
&\big((\mathrm{L}^{mn}_{m} \otimes \mathrm{I}_2) \otimes_\nu \mathrm{I}_\nu\big)(\mathrm{I}_{mn/\nu} \otimes \underbrace{\mathrm{L}^{2\nu}_{2}}_{\mathrm{vec}(\nu)}).
\end{aligned}
$$

Inspection shows that this formula is indeed vectorized in the sense of Definition 1.

Note that there are degrees of freedom in applying our rule set, which thus yield different vectorizations. The search in the Spiral system will select the best for the given platform.

4 Experimental Results

We incorporated our rewriting system into the Spiral code generator to automatically generate vector code and search over alternative algorithms or formulas. We show runtime benchmarks on a 3 GHz Intel Pentium 4 running Windows XP and a 3.6 GHz Intel Pentium 4 running Linux kernel 2.6. We used the Intel C++ compiler 9.0 with options "/QxKW /O3 /G7 /Qc99 /Qrestrict" for vector code and "/O3 /G7 /Qc99 /Qrestrict" for scalar x86 and x87 code. These options turned out to produce the fastest code. DFT_n performance is measured in pseudo Mflop/s for floating-point code and in pseudo Mfpop/s for fixed-point code, both computed as $5n \log_2 n/(\text{runtime [ms]})$. The Haar_n wavelet performance is measured in Mfpop/s $= 2N/(\text{runtime [ms]})$. For all performance results higher is better. We compare our generated code with the Intel MKL 8.0 (DFTI functions) and IPP 5.0 library and with FFTW 3.1 [6] for both two-powers and multiples of ν.

2-way double-precision. Figure 1 evaluates our approach for two-way vectorization. We compare two-power DFTs of sizes $2^7 \leq n \leq 2^{14}$: 1) Spiral generated scalar x87 code; 2) Spiral generated SSE2 code; 3) FFTW 3.1 with enabled SSE2 support; and 4) Intel IPP 5.0 using SSE2. Spiral and FFTW achieve similar performance with FFTW being slightly faster, and IPP is between 5% and 15% faster than both. For Spiral generated code, SSE2 vectorization provides around 50% speed-up over scalar code.

4-way single-precision. Figure 2 compares DFT code for two-power sizes $2^4 \leq n \leq 2^{12}$: 1) Spiral generated scalar x87 code; 2) Spiral generated scalar x87 code vectorized by Intel's compiler (option "/QxKW"); 3) Spiral generated 4-way SSE code; 4) FFTW 3.1 with enabled SSE support; and 5) Intel IPP 5.0 using SSE. Spiral generated SSE code is up to 3 times faster as Spiral generated scalar x87 code. Using the Intel C++ compiler to vectorize code leads only to around 50% speed-up. For $16 \leq n \leq 128$, Spiral generated SSE code is clearly the

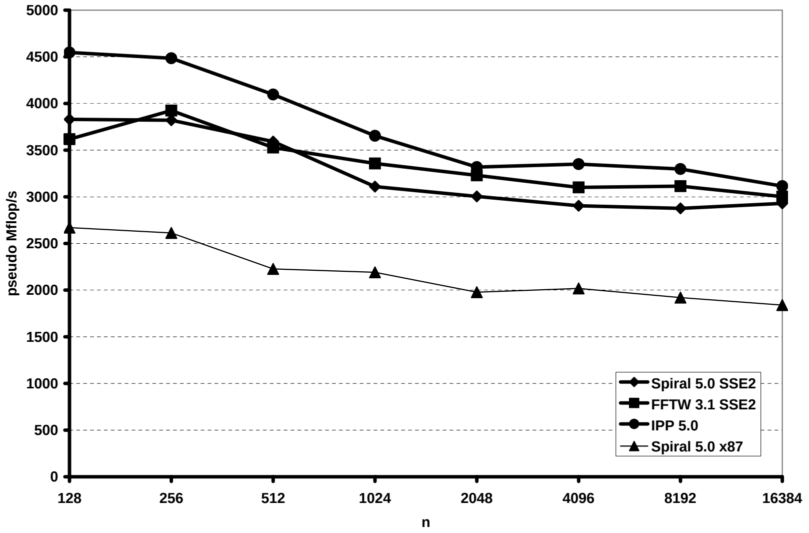

Fig. 1. Performance of DFT_n with $n = 2^k$, implemented in double-precision on a 3.6 GHz Pentium 4.

fastest. For n=256 both FFTW and IPP are slightly faster as Spiral generated SSE code. For $512 \leq n \leq 2048$, Spiral generated SSE code is within 10% of IPP.

8-way 16-bit fixed-point. Figure 3 compares two-power DFT fixed-point code. It shows 1) Spiral generated SSE2 code (8-way, 16-bit), 2) scalar 16-bit x86 code, and 3) Intel IPP 5.0 (16-bit). Spiral's SSE2 vectorization consistently provides speed-up of 5 times over scalar x86 code. Spiral generated SSE2 code is 2 to 2.5 times faster as the IPP and 5 to 6 times faster than Spiral generated scalar code. Figure 4 shows that for DFTs of size $n = 64 \times 2^k 3^\ell 5^m$ SSE2 code generated by Spiral maintains the speed-up of 5 times over scalar x86 code. IPP does only provide two-power FFTs for 16-bit fixed-point. FFTW does not provide fixed-point code.

16-way 8-bit fixed-point. Figure 5 compares implementations of the Haar wavelet: 1) Spiral generated SSE2 code, 2) Spiral generated scalar 8-bit x86 code (16-way, 8-bit), and 3) the Intel IPP 5.0 (8-bit). For Spiral, SSE2 vectorization provides a speed-up of up to 6 times over scalar x86 code. Spiral generated SSE2 code is 2 to 2.5 times faster than the IPP. FFTW does not implement Haar wavelets.

The lack of precision makes large size DFT implementations unpractical in this case.

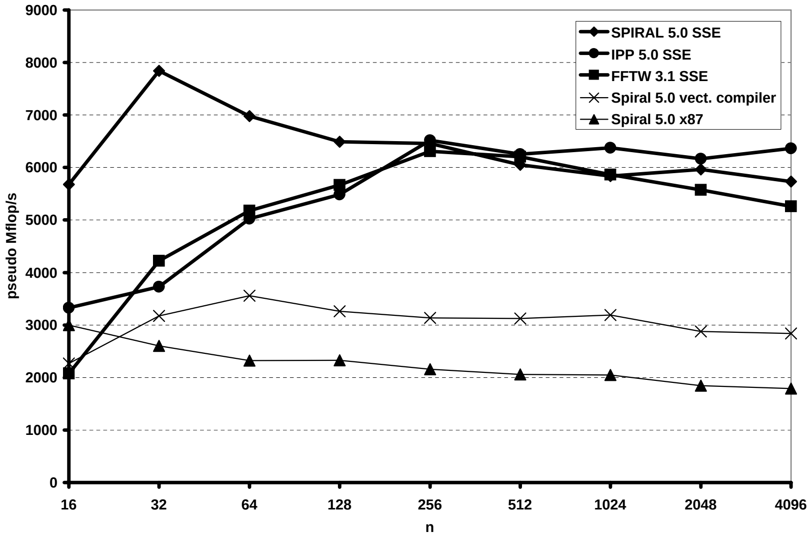

Fig. 2. Performance of DFT_n with $n = 2^k$, implemented in single-precision on a 3 GHz Pentium 4.

5 Conclusion

SIMD vector instructions have a huge potential to speed up performance critical computational kernels with fine-grain parallelism. However, compiler support is limited and typically programmers have to resort to low-level C extensions or to assembly language programming to realize the potential of SIMD extensions. To overcome these problems for the domain of signal transforms, we presented a domain-specific vectorization framework for signal transform algorithms and in particular FFTs. The basic idea is to vectorize at a high mathematical level of abstraction, where more structural information is available as in the corresponding C code. The suitable tool for implementing this technique is a rule based rewriting system, which we included in Spiral to enable search in tandem with vectorization. Experiments with the DFT show the viability of the approach. We are currently exploring similar strategies for shared and distribute memory parallelization.

References

1. Franchetti, F., Püschel, M.: A SIMD vectorizing compiler for digital signal processing algorithms. In: Proc. IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS). (2002) 20–26

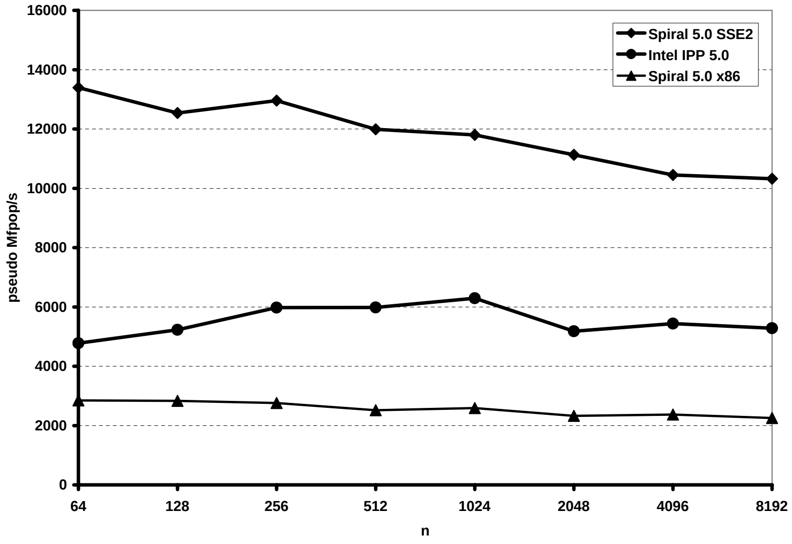

Fig. 3. Performance of DFT_n with $n = 2^k$, implemented in 16-bit fixed-point on a 3.6 GHz Pentium 4.

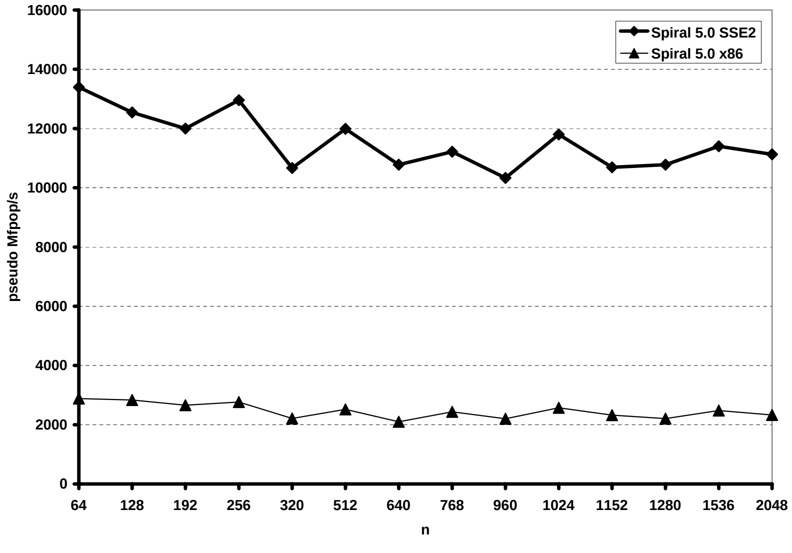

Fig. 4. Performance of DFT_n with $n = 64 \times 2^k 3^\ell 5^m$, implemented in 16-bit fixed-point on a 3.6 GHz Pentium 4.

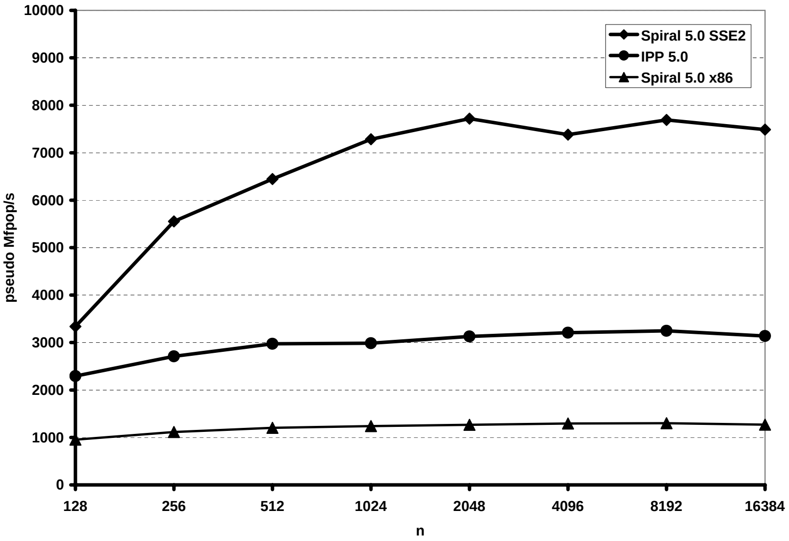

Fig. 5. Performance of the wavelet Haar_n with $n = 2^k$, implemented in 8-bit fixed-point on a 3.6 GHz Pentium 4.

2. Van Loan, C.: Computational Framework of the Fast Fourier Transform. SIAM (1992)
3. Johnson, J.R., Johnson, R.W., Rodriguez, D., Tolimieri, R.: A methodology for designing, modifying, and implementing Fourier transform algorithms on various architectures. IEEE Trans. Circuits, Systems, and Signal Processing **9**(4) (1990) 449–500
4. Dershowitz, N., Plaisted, D.A.: Rewriting. In Robinson, A., Voronkov, A., eds.: Handbook of Automated Reasoning. Volume 1. Elsevier (2001) 535–610
5. Püschel, M., Moura, J.M.F., Johnson, J., Padua, D., Veloso, M., Singer, B.W., Xiong, J., Franchetti, F., Gačić, A., Voronenko, Y., Chen, K., Johnson, R.W., Rizzolo, N.: SPIRAL: Code generation for DSP transforms. Proc. of the IEEE **93**(2) (2005) 232–275 Special issue on *Program Generation, Optimization, and Adaptation*.
6. Frigo, M., Johnson, S.G.: The design and implementation of FFTW3. Proceedings of the IEEE **93**(2) (2005) 216–231 special issue on "Program Generation, Optimization, and Adaptation".
7. Robelly, J., Cichon, G., Seidel, H., Fettweis, G.: A HW/SW design methodology for embedded SIMD vector signal processors. International Journal of Embedded Systems IJES (2005)
8. Franchetti, F., Püschel, M.: Short vector code generation for the discrete Fourier transform. In: Proc. IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS). (2003) 58–67