SIMD VECTORIZATION OF NON-TWO-POWER SIZED FFTS

Franz Franchetti and Markus Püschel

Carnegie Mellon University
Department of Electrical and Computer Engineering
Pittsburgh, PA, United States

ABSTRACT

SIMD (single instruction multiple data) vector instructions, such as Intel's SSE family, are available on most architectures, but are difficult to exploit for speed-up. In many cases, such as the fast Fourier transform (FFT), signal processing algorithms have to undergo major transformations to map efficiently. Using the Kronecker product formalism, we rigorously derive a novel variant of the general-radix Cooley-Tukey FFT that is structured to map efficiently for any vector length ν and radix. Then, we include the new FFT into the program generator Spiral to generate actual C implementations. Benchmarks on Intel's SSE show that the new algorithms perform better on practically all sizes than the best available libraries Intel's MKL and FFTW.

Index Terms— Discrete Fourier transform, DFT, vector instructions, SSE, program generation, performance, Kronecker product

1. INTRODUCTION

Most current off-the-shelf and many embedded computing platforms offer SIMD (single instruction multiple data) vector instructions. A prominent example is Intel's SSE family, which enables parallel operation on 2-way double precision, 4-way single-precision, or 8-way 16-bit and 16-way 8-bit fixed-point data types. The potential speed-up is considerable, so these instructions should be used if highest performance is desired.

Unfortunately, achieving the possible speed-up with these instructions is difficult for all but the simplest computational problems since many constraints have to be obeyed to get performance. This includes proper data alignment during computation and the minimization of vector shuffle operations. As a consequence, existing fast algorithms often cannot be used directly but need to be restructured to map efficiently to vector architectures. This is certainly true for fast Fourier transform algorithms (FFTs) that possess a complex structure and pose a challenge even on platforms without special instructions.

In this paper we derive a novel variant of the general-radix Cooley-Tukey FFT, called general short-vector Cooley-Tukey FFT that has a structure that maps efficiently onto vector architectures. The new FFT applies to all radices, i.e., factorizations $N = mn$ of the input size and for all vector lengths ν. The derivation is done rigorously using the Kronecker product formalism [1]. In previous work [2] we had derived a vectorized FFT under the condition $\nu^2 \mid N$. The extension to arbitrary N is non-trivial.

Further, we included the new FFT into the program generation and optimization system Spiral [3, 4] to produce actual code for benchmarking. The benchmarks are against the best available libraries FFTW [5, 6] and Intel's Math Kernel Library (MKL). Both libraries offer vectorized code but the underlying algorithm is not

(fully) reported in the literature. The results show that our generated code, based on the new vectorized FFT, outperforms these libraries for most sizes. Further, we show that we can successfully (i.e, with obtaining speed-up) vectorize a larger class of sizes than FFTW.

Organization. Section 2 provides the necessary background on the FFT, the Kronecker product formalism, and Spiral. The main contribution of this paper, the new vectorized FFT, is derived in Section 3. Benchmarks with generated code conclude the paper in Section 4.

2. BACKGROUND

SIMD vector instructions. SIMD extensions like Intel's MMX and SSE, Motorola's AltiVec, and IBM's VMX (available on the Cell BE processor) operate on vector registers (64-bit or 128-bit wide) that are broken into ν-way vectors of smaller data types. For instance, SSE supports (among other data types) 4-way single-precision ($\nu = 4$) and 8-way 16-bit integer arithmetic ($\nu = 8$). SIMD extensions operate most efficiently if data is loaded and stored in full vectors that are naturally aligned (i.e., loading 16-byte aligned 128-bit words for SSE). Loading and storing of unaligned data and partial vectors incurs considerable overhead. To make things worse, loading or storing k out of the ν elements inside a vector, as required for handling arbitrary FFTs, requires the optimization of each combination of k and ν separately for each architecture.

Discrete Fourier transform. Computing the discrete Fourier transform (DFT) of an input signal x of length N is equivalent to the matrix-vector multiplication $y = \mathrm{DFT}_N\, x$, where

$$\mathrm{DFT}_N = [\omega_N^{k\ell}]_{0 \le k, \ell < N}, \quad \omega_N = e^{-2\pi j/N}.$$

Fast Fourier transforms. Various fast Fourier transforms (FFTs) are available that enable the computation of the DFT with $O(N \log(N))$ operations for all sizes N. The most important one is the Cooley-Tukey FFT. It can be expressed as a factorization of the DFT_N into a product of structured sparse matrices provided that $N = mn$ factorizes. Using the Kronecker, or tensor product notation [1] this FFT is given by

$$\mathrm{DFT}_{mn} = (\mathrm{DFT}_m \otimes I_n) D_{m,n}(I_m \otimes \mathrm{DFT}_n) L_m^{mn}. \quad (1)$$

In (1), the *stride permutation* matrix L_m^{mn} permutes the input vector as

$$in + j \mapsto jm + i, \quad 0 \le i < m,\ 0 \le j < n.$$

If x is viewed as an $n \times m$ matrix, stored in row-major order, then L_m^{mn} performs a transposition of this matrix. Further, I_n is the $n \times n$ identity matrix, and the *tensor product* is defined as

$$A \otimes B = [a_{k,\ell} B], \quad \text{for } A = [a_{k,\ell}].$$

$D_{m,n}$ is a diagonal matrix containing the so-called twiddle factors.

Further matrix notation. For later derivations, we also define the *direct sum* of matrices as

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

and introduce the notation $A^M = M^T A M$ as well as a generalization of I_n, namely

$$I_{m \times n} = \begin{bmatrix} I_n \\ O_{m-n \times n} \end{bmatrix}, \quad m \geq n,$$

$$I_{m \times n} = \begin{bmatrix} I_m & O_{m \times n-m} \end{bmatrix}, \quad m < n.$$

Here, $O_{m \times n}$ is the $m \times n$ all-zero matrix. Clearly, $I_n = I_{n \times n}$.

We call any expression that is constructed from the above matrices and operators $\otimes, \oplus, \cdot$ (product) a *formula*. For example, the right hand side of (1) is a formula.

The DFT is a complex transform, but implemented in real arithmetic. This can be expressed formally using an operator $\overline{(\cdot)}$. Namely, if A is a complex $m \times n$ matrix, then $\overline{A}$ is the real $2m \times 2n$ matrix obtained by replacing every entry $a = a_1 + ja_2$ with the matrix $\begin{bmatrix} a_1 & -a_2 \\ a_2 & a_1 \end{bmatrix}$. This way $y = Ax$ becomes equivalent to $y' = \overline{A}x'$, where x', y' are in the interleaved complex format: x' contains alternating the real and imaginary parts of the elements in x. Implementing the DFT really means implementing $y' = \overline{\mathrm{DFT}}_N x'$.

Efficient software implementation. When implemented in software, then, for efficiency, (1) is not performed in the four stages suggested by the four factors (from right to left: permutation, loop over smaller DFTs, scaling, loop over smaller DFTs), but instead the permutation L_m^{mn} is fused with the subsequent loop, and the scaling by $D_{m,n}$ as well (as implemented, for example, in the library FFTW [5]). This increases locality and reduces cache misses.

On parallel or vector architectures, more involved manipulations of the FFT (1) are necessary to achieve high performance. These manipulations, and the resulting variants of (1), can also be expressed using the tensor product notation. Early examples can be found in [1]. For the latest generation of platforms, [7] optimizes for shared memory parallelism, and [2] for ν-way vector instructions. However, [2] requires $\nu^2 \mid N$. The generalization to arbitrary (composite) N is the subject of this paper.

Spiral. Spiral is a program generation and optimization system for transforms including the DFT [3]. Given a DFT and its size N, Spiral expands DFT_N recursively using (1) or other FFTs until base cases ($N = 2$) are reached. The resulting formula (tensor product expression) is manipulated as explained in the previous paragraph and then translated into C code. Based on the runtime of this code, Spiral changes the initial recursive expansion (e.g., by choosing different factorizations $N = mn$) and repeats the process. This search eventually produces an implementation tuned to the given computer.

The derivation of a vectorized FFT in this paper is formalized to enable inclusion into Spiral. This way, we can generate code for this FFT automatically and search over alternatives for the fastest. More details are in the benchmark Section 4.

3. VECTORIZATION OF NON-TWOPOWER SIZES

Our goal is to derive a vectorized version of the Cooley-Tukey FFT (1) for any factorization $N = mn$ and vector length ν. Our approach is as follows:

- We identify *basic building blocks*, which are formulas that can be efficiently mapped to vector code. The idea is that then any formula that consists exclusively of these constructs can be mapped as well.

- We identify a set of *formula identities* such that the application to (1) transforms the FFT into a vectorized FFT.

- Finally, we show the obtained *vectorized FFT* and discuss its structure.

The goals for the final FFT are 1) to make sure all arithmetic is performed using vector instructions on properly aligned data; and 2) to minimize the required load/store and shuffle instructions.

We remind the reader that implementing or mapping to code of a matrix A means for the function $y = Ax$.

Basic building blocks. We list basic formulas that can be easily mapped to vector code and that are necessary to vectorize (1).

For any matrix A, the formula

$$A \otimes I_\nu \tag{2}$$

can be translated to vector code by generating scalar code for A and then replacing all scalar operations in that code by the corresponding vector operations and scalar variables by ν-way vector variables [2].

The permutations

$$L_2^{2\nu}, \quad L_\nu^{2\nu}, \quad L_\nu^{\nu^2} \tag{3}$$

are the only in-register permutations required to vectorize (1). For example, on Intel's SSE and $\nu = 4$, the first two require only two assembly instructions.

The construct

$$I_{m\nu \times n}, \quad (m-1)\nu < n \leq m\nu, \tag{4}$$

describes the loading of n consecutive values into m vector registers; the last $m\nu - n$ entries of the last vector register are set to zero. In words, this matrix extends the input vector by zeros to make the length divisible by ν, and, when implemented, performs the actual load operation.

The corresponding store operation is the construct

$$I_{m \times n\nu}, \quad (n-1)\nu < m \leq n\nu, \tag{5}$$

which writes m consecutive memory locations from n vector registers. The last $n\nu - m$ entries of the last vector register are not written.

We introduce the operator $(.)^{n,\nu}$ to transform complex diagonal matrices into vectorizable formulas:

$$D_{m,n}^{n,\nu} = \overline{D_{m,n}}^{(\overline{I_m \otimes I_{n \times \nu \lceil n/\nu \rceil}})}(I_{m\lceil n/\nu \rceil} \otimes L_\nu^{2\nu}). \tag{6}$$

This somewhat complicated looking transformation first extends D with zeros to have a size divisible by ν. Then it permutes rows and columns simultaneously so the non-zero pattern is the same as that of $I_{m \otimes \lceil n/\nu \rceil} \otimes A_2 \otimes I_\nu$, which matches (2) and can thus be implemented using vector operations only.

From the above formulas, we can build more complex ones that can still be mapped to ν-way vector code. This is captured in the following definition.

Definition 1 *We call a formula vectorized if it is either of the form (2)–(6), or of the form $I_m \otimes A$ or AB, where A and B are vectorized.*

As examples, consider $\nu = 4$ and a 2×2 matrix A_2. The formula $A_2 \otimes I_3$ is not vectorized; the formula $(A_2 \otimes I_4)L_2^8 I_{8 \times 7}$ is vectorized.

Formula identities. We list the necessary formula identities to vectorize (1). They are divided into 3 classes:

- The *vectorization rules* (7)–(9) (Table 1) perform the actual vectorization of formulas.

- The *complex arithmetic rules* (11)–(16) (Table 2) handle the data conversion from complex formulas to real formulas operating on vectors in the interleaved complex format and minimize the resulting data shuffling.

$$A_m \otimes I_n \rightarrow \left((A_m \otimes I_{\lceil n/\nu \rceil}) \otimes I_\nu \right)^{I_m \otimes I_{\nu \lceil n/\nu \rceil \times n}} \tag{7}$$

$$(I_m \otimes A_n) L_m^{mn} \rightarrow \left(I_{m \times \nu \lceil \frac{m}{\nu} \rceil} \otimes I_{n \times \nu \lceil \frac{n}{\nu} \rceil} \right) L_{\nu \lceil \frac{n}{\nu} \rceil}^{\nu^2 \lceil \frac{m}{\nu} \rceil \lceil \frac{n}{\nu} \rceil} \left(\left(I_{\nu \lceil \frac{m}{\nu} \rceil \lceil \frac{n}{\nu} \rceil \times n \lceil \frac{m}{\nu} \rceil} (A_n \otimes I_{\lceil \frac{m}{\nu} \rceil}) \right) \otimes I_\nu \right) \left(I_n \otimes I_{\nu \lceil \frac{m}{\nu} \rceil \times m} \right) \tag{8}$$

$$L_m^{mn} \rightarrow \left(L_m^{mn/\nu} \otimes I_\nu \right) \left(I_{mn/\nu^2} \otimes L_\nu^{\nu^2} \right) \left((I_{n/\nu} \otimes L_{m/\nu}^m) \otimes I_\nu \right) \tag{9}$$

Table 1. Vectorization rules.

$$\overline{AB} \rightarrow \overline{A}\,\overline{B} \tag{11}$$

$$\overline{I_m \otimes A_n} \rightarrow I_m \otimes \overline{A}_n \tag{12}$$

$$\overline{A_m \otimes I_\nu} \rightarrow \left(\overline{A}_m \otimes I_\nu \right)^{I_m \otimes L_2^{2\nu}} \tag{13}$$

$$\overline{D}_{m,n} \rightarrow \left(D_{m,n}^{n,\nu} \right)^{I_m \otimes \left((I_{\lceil n/\nu \rceil} \otimes L_2^{2\nu}) \overline{(I_{\nu \lceil n/\nu \rceil \times n})} \right)} \tag{14}$$

$$\overline{I_{m \times n}} \rightarrow I_{2m \times 2n} \tag{15}$$

$$\overline{L_\nu^{\nu^2}} \rightarrow \left(I_2 \otimes L_\nu^{\nu^2} \right)^{(L_2^{2\nu} \otimes I_\nu)(I_\nu \otimes L_2^{2\nu})} \tag{16}$$

Table 2. Complex arithmetic rules.

$$(A_m \otimes B_n)(A_m \otimes C_n) \rightarrow A_m \otimes B_n C_n \tag{17}$$

$$L_2^{2\nu} L_\nu^{2\nu} \rightarrow I_2 \otimes I_\nu \tag{18}$$

$$(A_m \oplus O_n) I_{m+n \times m}\, I_{m \times m+n} \rightarrow (A_m \oplus O_n) \tag{19}$$

Table 3. Simplification rules.

$$\overline{\mathrm{DFT}}_{mn} = \left(I_m \otimes I_{2n \times 2\nu \lceil \frac{n}{\nu} \rceil} \right) \tag{21}$$

$$\left(I_{m \lceil \frac{n}{\nu} \rceil} \otimes L_\nu^{2\nu} \right) \tag{22}$$

$$\left((\overline{\mathrm{DFT}_m} \otimes I_{\lceil \frac{n}{\nu} \rceil}) \otimes I_\nu \right) D_{m,n}^{n,\nu} \tag{23}$$

$$\left((I_{m \times \nu \lceil \frac{m}{\nu} \rceil} \otimes I_{2 \lceil \frac{n}{\nu} \rceil}) \otimes I_\nu \right) \tag{24}$$

$$P_{m,n} \tag{25}$$

$$\left(\left(I_{\nu \lceil \frac{m}{\nu} \rceil \lceil \frac{n}{\nu} \rceil \times n \lceil \frac{m}{\nu} \rceil} (\overline{\mathrm{DFT}_n} \otimes I_{\lceil \frac{m}{\nu} \rceil}) \right) \otimes I_\nu \right) \tag{26}$$

$$\left(I_{n \lceil \frac{m}{\nu} \rceil} \otimes L_2^{2\nu} \right) \tag{27}$$

$$\left(I_n \otimes I_{2\nu \lceil \frac{m}{\nu} \rceil \times 2m} \right) \tag{28}$$

Table 4. General short-vector Cooley-Tukey FFT for any ν and $N = mn$. The permutation $P_{m,n}$ is shown in Table 5.

- The *simplification rules* (17)–(19) (Table 3) minimize the number of shuffle operations and partial vector loads/stores.

As a simple example of formula vectorization, consider again $\nu = 4$ and $\mathrm{DFT}_2 \otimes I_3$, which expresses a loop with 3 iterations (here, DFT_2 is considered as operating on a real input). We use (7) to vectorize:

$$\mathrm{DFT}_2 \otimes I_3 \rightarrow (I_2 \otimes I_{3 \times 4})(\mathrm{DFT}_2 \otimes I_4)(I_2 \otimes I_{4 \times 3}). \tag{10}$$

$(I_2 \otimes I_{4 \times 3})$ loads two times 3 elements into a 4-way vector, zeroing the last element. $(\mathrm{DFT}_2 \otimes I_4)$ adds and subtracts both 4-way vectors. $(I_2 \otimes I_{3 \times 4})$ stores the first 3 elements of each 4-way vector consecutively to memory.

Vectorized FFT. We now derive a vectorized Cooley-Tukey FFT for any ν and factorization $N = mn$. Due to space limitations, we only sketch the derivation and spend more time in interpreting the result.

The starting point is

$$\overline{\mathrm{DFT}}_N. \tag{20}$$

First, we expand (20) using (1) to expand the larger DFT into smaller DFTs. In the next step the vectorization rules (7)–(9) vectorize all factors in (1). Then, rules (11)–(16) translate the complex formula into its corresponding real version. Finally, rules (17)–(19) simplify the formula and drop base cases involving loads/stores or shuffles wherever possible. The result is the FFT in Table 4, which we call the *general short vector Cooley-Tukey FFT algorithm*. It is parameterized by the vector length ν. Inspection shows that each of the factors (21)–(28) is indeed vectorized in the sense of Definition 1. Note, that (21)–(28) is vectorized *independent* of how the smaller DFT_m and DFT_n are further expanded using, for example, again (1) or a prime-factor or Rader FFT. In other words, (21)–(28), as (1), spans an entire set of algorithms that can be searched for the fastest, which is exactly what Spiral does (see also the discussion below). Also note that, as in (1), the occurring shuffle operations are never

explicitly performed but converted into readdressing or fused with vector load and store operations.

We discuss the factors in Table 4 in the order they are performed from bottom to top. The input vector (of length $2N = 2mn$) is initially in the interleaved format. (28) expands m, and thus the length of the input vector to be divisible by ν. (27) converts into vector interleaved format (ν real parts followed by ν imaginary parts and so on). (26) performs a perfect vectorized computation on aligned vectors and in its last step expands n to be divisible by ν. (25) is a permutation (shown in Table 5) that operates on entire vectors except for $L_\nu^{\nu^2}$ at its core. (24) contracts m. (23) performs a perfect vectorized computation on aligned vectors. (22) converts from vector interleaved to interleaved format. (21) extracts and stores the relevant parts of the results, contracting n.

Spiral. For implementation purposes, the main task was to identify the transformations in Tables 1–3. We included them into Spiral's formula rewriting system, so Spiral derives autonomously the vectorized FFT in Table 4, further expands the smaller DFTs in different ways, generates the actual C code including vector instructions, and searches for the fastest implementation. As m and n may be prime numbers, the general short vector Cooley-Tukey FFT (Table 4) does not exactly specialize to the short vector Cooley-Tukey FFT [2] which requires $\nu | m, n$. Benchmarks with the generated code using our new vectorized FFT are shown next.

4. EXPERIMENTAL RESULTS

Benchmark setup. We evaluated our vectorization method on a 3.6 GHz Intel Pentium 4 running Windows XP, and using the Intel C++ compiler 8.1 with options "-O3 -QxKWP". We evaluated both the 4-way single-precision float and the 8-way 16-bit integer mode of the SSE vector instruction set.

All Spiral DFT programs are automatically generated and adapted through Spiral's search mechanism. Specifically for SSE,
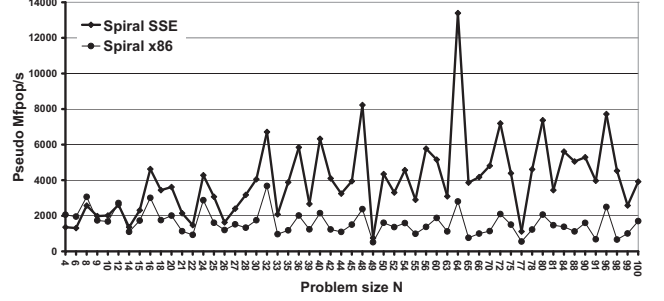
$$P_{m,n} = \left(\left(L_{\nu\lceil\frac{m}{\nu}\rceil}^{\nu\lceil\frac{m}{\nu}\rceil\lceil\frac{n}{\nu}\rceil} \otimes I_2 \right) \otimes I_\nu \right) \left(I_{\lceil\frac{m}{\nu}\rceil\lceil\frac{n}{\nu}\rceil} \otimes \left((L_\nu^{2\nu} \otimes I_\nu)(I_2 \otimes L_\nu^{\nu^2})(L_2^{2\nu} \otimes I_\nu) \right) \right) \left(\left((I_{\lceil\frac{n}{\nu}\rceil} \otimes L_{\lceil\frac{m}{\nu}\rceil}^{\nu\lceil\frac{m}{\nu}\rceil}) \otimes I_2 \right) \otimes I_\nu \right)$$

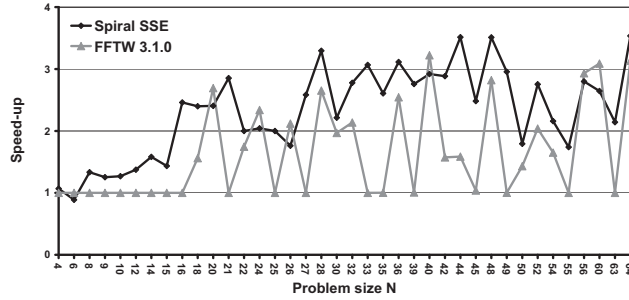**Table 5**. Intermediate shuffle in the general short-vector Cooley-Tukey FFT (Table 4).

(a) Performance: 4-way single-precision floating-point SSE
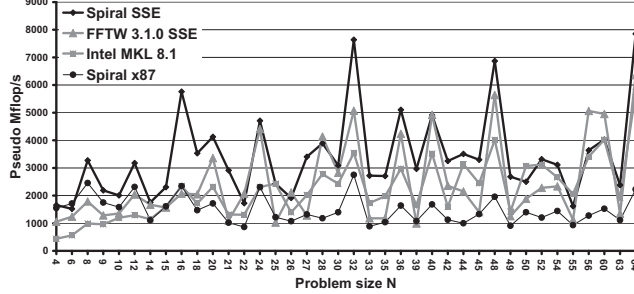
(c) Performance: 8-way 16-bit integer SSE

(b) Speed-up: 4-way single-precision floating-point SSE

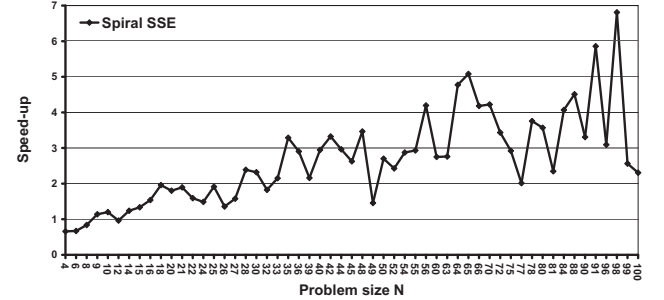**(d) Speed-up: 8-way 16-bit integer SSE**

Fig. 1. Complex DFT on a 3.6 GHz Pentium 4. (a) and (c): Performance of scalar (x86/x87) and vectorized (SSE) Spiral-generated code, FFTW 3.1.0, and Intel MKL 8.1 (the latter two are not available for integer code); (b) and (d): Speed-up of SSE code over the respective scalar code for FFTW 3.1.0 and Spiral-generated code. MKL scalar code is not available. Higher is better in all plots.

this means that Spiral first chooses a split (1), then rewrites into the form in Table 4, and then has further freedom in expanding the smaller DFTs in (23) and (26) using also prime-factor and Rader FFTs.

We compare our generated programs to the SSE version of FFTW 3.1.0 (pre-built Windows library available at fftw.org), and to the vendor library Intel Math Kernel Library (MKL) 8.1. Both libraries only provide vectorized floating-point code (except two-power sizes in MKL). We report performance in "pseudo Mflop/s" (float) and "pseudo Mfpop/s" (integer). Both are computed for DFT_N by $(5N \log_2 N)/t$, where t is the runtime in microseconds. Speed-up is computed by $t_{\mathrm{scalar}}/t_{\mathrm{vector}}$. Thus, in all performance metrics, higher is better.

Fig. 1 summarizes the results. We evaluated all composite problem sizes $N \leq M$, where $M = 64$ for float and $M = 100$ for 16-bit integer, with all prime factors $p < 16$.

Comparison to FFTW and MKL. Spiral-generated 4-way vector code is faster than FFTW and MKL for almost all considered sizes (Fig. 1(a)) and achieves a speed-up over the scalar (x87) code for all expect the smallest sizes. The same holds for the integer code (Fig. 1(c)). Thus, our vectorization methods works in general.

The actual speed-up obtained through vectorization over the scalar code is shown in Figs. 1(b) and (d). Again, they show a speed-up for Spiral for all except the smallest sizes. FFTW, for 4-way float, achieves a vectorization speed-up only for even sizes and is roughly comparable to Spiral only for sizes divisible by 4.

5. REFERENCES

[1] C. Van Loan, *Computational Framework of the Fast Fourier Transform*, SIAM, 1992.

[2] F. Franchetti and M. Püschel, "Short vector code generation for the discrete Fourier transform," in *Proc. Int'l Parallel and Distributed Processing Symposium (IPDPS)*, 2003, pp. 58–67.

[3] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, "SPIRAL: Code generation for DSP transforms," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 232–275, 2005, special issue on *Program Generation, Optimization, and Adaptation*.

[4] "Spiral web site," 1998, www.spiral.net.

[5] M. Frigo and S. G. Johnson, "FFTW: An adaptive software architecture for the FFT," in *Proc. IEEE Int'l Conf. Acoustics, Speech, and Signal Processing (ICASSP)*, 1998, vol. 3, pp. 1381–1384, www.fftw.org.

[6] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005, special issue on "Program Generation, Optimization, and Adaptation".

[7] F. Franchetti, Y. Voronenko, and M. Püschel, "FFT program generation for shared memory: SMP and multicore," in *Proc. Supercomputing (SC)*, 2006.