A PORTABLE SHORT VECTOR VERSION OF FFTW

Franz Franchetti, TU Vienna, Austria
Vienna University of Technology
Wiedner Hauptstrasse 8 – 10, A-1040 Wien, Austria
Phone: +43 1 58801-11524, Fax: + 43 1 58801-11599
email: franz.franchetti@tuwien.ac.at

Abstract. This paper presents a portable short vector extension for the popular FFT library FFTW. FFTW is a freely available portable FFT software-library that achieves top performance across a large number of platforms.
The newly developed extension enables the utilization of short vector extensions like Intel's SSE and SSE 2 as well as Motorola's AltiVec for any problem sizes. The method is independent of the machine's vector length. Experimental results show good speed-up for arbitrary problem sizes and excellent speed-up for problem sizes of powers of two and multiples of 16. The newly developed short vector version of FFTW compares favorably with the hand-tuned vendor library Intel MKL on IA-32 compatible machines (Intel Pentium III and 4, and AMD Athlon XP).

1. Introduction

The discrete Fourier transform (DFT) plays a central role in the field of scientific computing. DFT methods are an extremely powerful tool to solve a multitude of scientific and engineering problems. For example, the DFT is essential in digital signal processing and for solving partial differential equations.
Major vendors of *general purpose* microprocessors have included short vector SIMD (single instruction multiple data) extensions into their instruction set architecture (ISA) to improve the performance of multi-media applications. Examples of SIMD extensions supporting both integer operations and floating-point operations include the Intel Streaming SIMD Extensions (SSE and SSE 2) [1], AMD 3DNow! and its extensions [2], Motorola's AltiVec extension [3] and the Double Hummer floating-point unit for IBM's BG/L machine. Each of these ISA extensions is based on the packing of large registers (64 bits or 128 bits) with smaller data types and providing vector instructions operating on these registers.
SIMD extensions have the potential to speed up implementations in all areas where performance is crucial and the algorithms used exhibit the fine grain parallelism necessary for utilizing SIMD instructions. It is important to note that processors featuring short vector SIMD instructions are completely different from vector computers developed in the 1980's. Thus, solutions developed for conventional vector computers are not directly applicable to today's short vector extensions.
The structural complexity of FFT algorithms led to complicated algorithms which make it a challenging task to map them to standard hardware efficiently and an even harder problem to exploit special processor features satisfactorily [4]. The unprecedented complexity of today's computer systems implies that performance portable software—software that performs satisfactorily across platforms and generations—can only be achieved by means of automatic empirical performance tuning [5]. It is necessary to apply *search* techniques to find the best possible implementation on a given target machine using actual runtime as cost function, as modelling the machine's behavior accurately enough is impossible on today's computers. The development of FFTW was the first effort that uses special purpose compiler techniques in combination with hardware adaptation using the actual runtime as cost function [6].

Related Work. A first short vector extension for FFTW was presented in [7]. A hand-tuned radix-2 FFT algorithm for SSE and AltiVec is described in [8]. Automatic performance tuning for discrete linear transforms is the target of SPIRAL [9]. An extension to SPIRAL targeting short vector implementations for discrete linear transforms is presented in [10, 11]. SPIRAL is a library generator and optimizes a special program for each transform and problem size. In contrast, FFTW is a library which optimizes its implementation on a target machine in the initialization phase and handles *arbitrary* problem sizes that are specified at runtime. A code generator based extension to FFTW for two-way short vector extensions can be found at `http://www.fftw.org/~skral`.
This paper presents an extension to FFTW which portably supports short vector extension and leads to substantial speed-ups. The computation is sped up transparently for the FFTW user for *arbitrary* problem sizes, with special support for multiples of 16 for four-way short vector extensions and multiples

of four for two-way short vector extensions. Thus, the important power of two case is handled most efficiently.

2. Floating-Point Short Vector Extensions

All short vector SIMD extensions for general purpose processors are based on the packing of large registers with smaller datatypes (usually of 8, 16, 32, or 64 bits). Once packed into the larger register, operations are then performed in parallel on the separate data items within the vector register.
The first generation of short vector SIMD extensions only supported integer SIMD operations (for instance, Intel's MMX and HP's MAX). The second generation of short vector SIMD extensions features (two-way or four-way) single-precision floating-point vectors (for instance Intel's SSE, Motorola's AltiVec and AMD's 3DNow!) while the third generation features two-way double-precision vectors (for instance, Intel's SSE 2)
By introducing double-precision short vector SIMD extensions, this technology entered scientific computing. Conventional scalar codes become obsolete on machines featuring these extensions as such codes utilize only a fraction of the potential performance. But these extensions have strong implications on the development of algorithms as their efficient utilization is not straightforward.
The most important restriction of all SIMD extensions is the fact that only *naturally aligned vectors* can be accessed highly efficient. Although, loading subvectors or accessing unaligned vectors is supported by some extensions, these operations are more costly than the aligned vector access. On other extensions these operations feature prohibitive performance characteristics. This negative effect is the driving force behind the work presented in this paper.
It is important to note that vecorizing compilers for short vector SIMD extensions as the Intel C++ compiler cannot be used to vectorize FFTW automatically due to the dynamic structure of FFTW.

Vendor	Name	n-way	Prec.	Processor	Compiler
Intel	SSE	4-way	single	Pentium III Pentium 4	MS Visual C++ Intel C++ Compiler GNU C Compiler 3.0
Intel	SSE2	2-way	double	Pentium 4	MS Visual C++ Intel C++ Compiler GNU C Compiler 3.0
Intel	IPF	2-way	single	Itanium Itanium 2	Intel C++ Compiler
AMD	3DNow!	2-way	single	K6, K6-II	MS Visual C++ GNU C Compiler 3.0
AMD	Enhanced 3DNow!	2-way	single	Athlon (K7)	MS Visual C++ GNU C Compiler 3.0
AMD	3DNow! Professional (Enhanced 3DNow! and SSE compatible)	4-way	single	Athlon XP Athlon MP	MS Visual C++ Intel C++ Compiler GNU C Compiler 3.0
Motorola	AltiVec	4-way	single	Power PC G4	GNU C Compiler 3.0 Apple C Compiler 2.96
IBM	Hummer^2	2-way	double	BG/L processor	IBM `XLCentury`

Table 1: Short vector SIMD extensions providing floating-point arithmetic found in general purpose microprocessors.

2.1. Software Support

Currently, application developers have three common methods for accessing multi-media hardware. They can invoke vendor-supplied, multi-media libraries; rewrite key portions of the application in assembly language using the multi-media instructions; or code in a high-level language and use vendor-supplied macros that make available the functionality of the multi-media primitives through a simple function-call like interface.
Recognizing the tedious and difficult nature of assembly coding, most multi-media hardware vendors have developed programming-language abstractions. These give an application developer access to the

low-level multi-media primitives without having to actually write assembly language code. Typically, this approach results in a function-call-like abstraction that represents one-to-one mapping between a function call and a multi-media instruction.
The most common language extension for specifying multi-media primitives is to provide within the C programming language function-call like intrinsic (or built-in) functions and new data types to mirror the instructions and vector registers. For most SIMD extensions, at least one compiler featuring these language extensions exists. Examples include C compilers for Intel's SSE and SSE 2, Motorola's AltiVec, the AMD 3DNow! versions and for the IBM BG/L processors. Most short vector SIMD extensions are supported by the GNU C compiler via *built-in* functions. Table 1 summarizes the compiler support for current short-vector floating-point extensions.
Each intrinsic directly translates to a single multi-media instruction, and the compiler allocates registers and schedules instructions. This approach would be even more attractive to application developers if the industry agreed upon a common set of macros, rather than each vendor providing his own set. For the AltiVec architecture, Motorola has defined such an interface, and under Windows both the Intel C compiler and the Microsoft Visual Studio use the same language extension support for SSE and SSE 2.
Careful analysis of the instructions required to vectorize FFTW allows to define a set of C macros (a portable SIMD API) that can efficiently be implemented on all current architectures and provides all required operations. The main restriction turns out to be that across all short vector extensions only *naturally aligned vectors* can be loaded and stored highly efficient. All other memory access operations lead to low performance and may even feature prohibitive performance characteristics. All codes described in this paper use the portable SIMD API. The portable SIMD API has two main purposes.

Abstracting Machine Specifics. In the context of a short vector version of FFTW all short vector extensions basically feature the required functionality for the required building blocks. But the implementation of these building blocks depends on the target architecture. For instance, a complex reordering operation like a permutation has to be implemented using the register-register permutation instructions provided by the target architecture. In addition restrictions like aligned memory access have to be handled. Thus a set of building blocks is defined which can (*i*) be implemented on all current short vector architectures, and (*ii*) all discrete linear transforms can be built on top of these intermediate level building blocks. This set is called the *Portable SIMD API*.

Abstracting Compiler Specifics. All compilers featuring a short vector language extension for C provide the required functionality to implement the portable SIMD API. But syntax and semantics differ from platform to platform and from compiler to compiler. These specifics have to be hidden in the portable SIMD API.

3. FFTW

FFTW was the first effort to automatically generate FFT code using a special purpose compiler and use the empirically measured runtime as optimization criterion. Typically, FFTW performs faster than publicly available FFT codes and faster to equal with hand optimized vendor-supplied libraries across different machines. It provides comparable performance to other automatic performance tuning systems like SPIRAL. Currently, FFTW is the most popular portable high performance FFT library that is publicly available. It can be downloaded at `http://www.fftw.org`.
FFTW provides a recursive implementation of the Cooley-Tukey FFT algorithm. The matrix DFT_n is defined for any $n \in \mathbb{N}$ with $i = \sqrt{-1}$ by

$$\mathrm{DFT}_n = \left(e^{2\pi i k\ell/n} \mid k, \ell = 0, 1, \ldots, n-1 \right).$$

The values $\omega_n^{k\ell} = e^{2\pi i k\ell/n}$ are called twiddle factors. In this paper the *Kronecker* (or tensor) product formalism is used to express the Cooley-Tukey splitting [12, 4], i. e.,

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

and the required modifications. The algorithm derived from the Cooley-Tukey splitting computes a discrete Fourier transform (DFT) for a vector of size mn recursively by first reordering the data according

to L_m^{mn}, then computing m DFTs of size n followed by a scaling operation expressed by the *twiddle factor matrix* T_n^{mn} and finally computing n DFTs of size m.
The actual computation is done by automatically generated routines called *codelets* which restrict the computation to specially structured algorithms called right expanded trees [13]. The recursion stops when all remaining subproblem are solved using codelets. For a given problem size there are many different ways of solving the problem with potentially very different runtimes. FFTW uses dynamic programming with the empirically measured runtime of problems as cost function to find a fast implementation for a given problem size on a given machine. FFTW consists of the following fundamental parts.

The Planner. At runtime but as a one-time effort during the initialization phase, the *planner* uses dynamic programming to find a good decomposition of the problem size into a tree of computations according to the Cooley-Tukey recursion called *plan*. This tree encodes the order in which different the subproblems are solved using codelets which do the actual work.

The Executor. When computing a DFT of a data vector, the *executor* interprets the *plan* as generated by the planner in the initialization phase and calls the appropriate codelets with the respective parameters as required by the plan. This procedure computes the DFT recursively, thus leading to data access patterns which provide for memory access locality. The executor only supports the Cooley-Tukey splitting.

The Codelets. The actual computation of the FFT subproblems is done within *codelets.* These small routines come in two versions, (*i*) *twiddle codelets* which are used in intermediate recursion steps and additionally handle the *twiddle factors*, and (*ii*) *no-twiddle codelets* which are used in the leafs of the recursion and additionally handle the *stride permutations*. Within the codelets a larger variety of FFT algorithms is used, including the Cooley-Tukey algorithm, the split radix algorithm, the prime factor algorithm, and the Rader algorithm [4]. Twiddle-codelets operate in-place while no-twiddle codelets operate out-of-place.

The Codelet Generator GENFFT. At install time, all codelets are generated by a special purpose compiler called the *codelet generator* `genfft`. As an alternative, the pre-generated codelet library can be downloaded as well. In the standard distribution, codelets of sizes up to 64 (not restricted to powers of two) are included. But if special transform sizes are required, the codelets needed in this case can be generated by the user.

4. FFTs on Short Vector Hardware

One of the reasons for the exceptional performance of FFTW across a large number of platforms is due to the recursive implementation of the DFT computation. This kind of implementation provides for data locality and an efficient utilization of multiple level memory hierarchies. However, implementing recursive FFT algorithms on short vector SIMD architectures introduces two major problems.

Complex Numbers. The DFT is a complex-to-complex transform. To exhibit memory locality, the *interleaved complex* format has to be used where the real and imaginary parts of a vector of complex numbers are stored interleaved in memory. This leads to the intrinsic problem of loading and storing vectors of real or imaginary parts using vector memory access. However, this is required for efficient utilization of the arithmetic vector operations. The method used in this paper hides the required shuffle operations in the portable SIMD API and minimizes the number of such shuffle operations by using—whenever possible—a special vector interleaved format in the intermediate stages of the recursion.

Stride Permutations. The large number of possible decompositions of the DFT computation leads to complicated permutations. In the recursive implementation of FFTW each recursion stage introduces stride permutations L_m^{mn} applied to complex numbers. Especially in conjunction with vector arithmetics and especially for all short vector extensions that are not two-way this permutation cannot directly be implemented using vector memory access. The combination of multiple recursion steps may produce as complicated permutations as the bit-reversal permutation. The source of the problem is that when designing an FFT algorithm that supports *arbitrary* vector lengths (at the present two-way and four-way) of short vector extensions one cannot find a factorization of L_m^{mn}

that consists of a small number of of vector memory access operation and in-register permutations. The additional condition that all arithmetic operations have to be vector operations introduces problems for vector extensions regardless of their vector length.

In the following, three approaches to handle the stride permutations are discussed. The applicability and advantages of these approaches are summarized.

- The Cooley-Tukey recursion is changed such that the permutation L_m^{mn} is factorized into a sequence of permutations within vectors and permutations of vectors at the cost of changing the memory access pattern. All occurring permutations have to be handled using the portable SIMD API. This approach leads to minimum data reorganization operations (all done in register) and all memory access operations are done with vector operations. It holds for all short vector extensions regardless of the vector length. This approach is based on the FFTW *vector recursion* which is an experimental alternative recursion included into FFTW 2.1.3. This approach is used whenever possible (i. e., for multiples of 16 for four-way short vector extensions and for multiples of two for two-way short vector extensions) as this approach leads to the best performance.
- The permutation L_m^{mn} is decomposed into memory access operations for complex numbers. This follows the way the stride permutation is handled within the scalar FFTW. If pairs of real numbers cannot be accessed efficiently, this method is suboptimal. Nevertheless, this method has to be used whenever the first method is not applicable.

  In addition, using this method *arbitrary* problem sizes can be handled. Operations that cannot be performed using exclusively arithmetic vector operations are done using the scalar FPU whenever the short vector instructions cannot be used. This approach leads to a high SIMD utilization and considerable speed-ups for vector lengths that could not be handled otherwise.
- The permutation L_m^{mn} is performed explicitly in a separate computation stage requiring an iterative approach. One can use high-performance implementations of transpositions (blocked or cache oblivious) and machine specific versions that utilize special instructions. As this method requires a stagewise computation of the DFT it is not suitable for a short vector version of FFTW that keeps the recursive character.

The following section shows the first and second approach for a DFT computation that is broken into *four* recursive stages. This is the smallest example where the operation of the FFTW vector recursion can be studied.

4.1. Vector Codelets and Vector Executor

As a prerequisite the most natural vector construct is discussed. Suppose a short vector SIMD extension with the vector length denoted by ν. The construct

$$A \otimes \mathrm{I}_\nu \tag{2}$$

can be mapped onto short vector SIMD hardware using vector arithmetics exclusively. This construct is obtained by replacing any scalar in $a_{i,j}$ in A by

$$\mathrm{diag}(\underbrace{a_{i,j}, \ldots, a_{i,j}}_{\nu \text{ times}}).$$

Thus, vector code is obtained by replacing any scalar arithmetic operation in the code for construct A by the respective vector operation. According to

$$A \otimes \mathrm{I}_{k\nu} = (A \otimes \mathrm{I}_k) \otimes \mathrm{I}_\nu,$$

longer vectors of size $k\nu$ can be implemented using vector instructions as well.
Applying this approach, `genfft` can be used to generate vector codelets. In the codelets generated by `genfft` the scalar operations are replaced by vector instructions. In addition, the required in-register permutations are inserted to handle the interleaved complex format.

The standard FFTW no-twiddle codelet of size n performs the operation DFT_n and the respective vector codelet performs $\mathrm{DFT}_n \otimes \mathrm{I}_\nu$ where the data elements are loaded and stored with strides according to the recursion applied by the executor depending on the used recursion.
The standard FFTW twiddle codelet of size m performs the operation

$$(\mathrm{DFT}_m \otimes \mathrm{I}_n)\, \mathrm{T}_n^{mn}$$

and the respective vector codelet performs

$$((\mathrm{DFT}_m \otimes \mathrm{I}_{\frac{n}{\nu}}) \otimes \mathrm{I}_\nu)\, \mathrm{T}_n^{mn}$$

Both operate in-place and with strides according to the recursion applied by the executor. In addition, the FFTW executor has to be adapted such that all operations that are carried out implicitly by the executor match $A \otimes \mathrm{I}_\nu$. This is discussed in the next section.

4.2. The FFTW Cooley-Tukey Recursion

As discussed throughout Section 4, the standard Cooley-Tukey recursion as given by equation (1) cannot be implemented using exclusively vector memory access. Consider the application of equation (1) to DFT_{rstu} leading to three stages of twiddle codelets and one stage of no-twiddle codelets.

Example 1 (FFTW Recursion for $\mathbf{N = rstu}$) Suppose a DFT computation using three twiddle codelet stages and one no twiddle stage. Using Equation 1 for $m = r$ and $n = stu$ leads to

$$\mathrm{DFT}_{rstu} = (\mathrm{DFT}_r \otimes \mathrm{I}_{stu})\, \mathrm{T}_{stu}^{rstu} (\mathrm{I}_r \otimes \mathrm{DFT}_{stu})\, \mathrm{L}_r^{rstu}\,.$$

Applying equation (1) again for $m = s$ and $n = tu$ leads to

$$\mathrm{DFT}_{rstu} = (\mathrm{DFT}_r \otimes \mathrm{I}_{stu})\, \mathrm{T}_{stu}^{rstu} (\mathrm{I}_r \otimes \big((\mathrm{DFT}_s \otimes \mathrm{I}_{tu})\, \mathrm{T}_{tu}^{stu} (\mathrm{I}_s \otimes \mathrm{DFT}_{tu})\, \mathrm{L}_s^{stu}\big)\, \mathrm{L}_r^{rstu}\,.$$

Applying equation (1) a last time with $m = t$ and $n = u$ leads to

$$\mathrm{DFT}_{rstu} = (\mathrm{DFT}_r \otimes \mathrm{I}_{stu})\, \mathrm{T}_{stu}^{rstu} (\mathrm{I}_r \otimes \big((\mathrm{DFT}_s \otimes \mathrm{I}_{tu})\, \mathrm{T}_{tu}^{stu} (\mathrm{I}_s \otimes \big((\mathrm{DFT}_t \otimes \mathrm{I}_u)\, \mathrm{T}_u^{tu} (\mathrm{I}_t \otimes \mathrm{DFT}_u)\, \mathrm{L}_t^{tu}\big)\, \mathrm{L}_s^{stu}\,\big)\, \mathrm{L}_r^{rstu} \qquad (3)$$

which is the final expansion.

Equation (3) features many constructs that do not match equation (2) and the stride permutations restrict the utilization of vector memory access. Thus, the second method has to be used to implement this formula using short vector extensions, i. e., it is not possible to implement this formula using *exclusively* vector memory access and arithmetic vector operation.

4.3. The FFTW Cooley-Tukey Vector Recursion

FFTW features an alternative—experimental—version of the Cooley-Tukey recursion, called *vector recursion*. It turned out to be of vital importance for the short vector vectorization within FFTW and for short vector implementations of FFTs in general. Based on this recursion the first method of handling the stride permutations was included into FFTW.
If the original problem size is mk_1k_2n, the right part of equation (1) can be factored alternatively into

$$\begin{aligned}(\mathrm{I}_m \otimes \mathrm{DFT}_{k_1k_2n})\, \mathrm{L}_m^{mk_1k_2n} \quad = \quad & \mathrm{I}_m \otimes (\mathrm{DFT}_{k_1} \otimes \mathrm{I}_{k_2n})\, \mathrm{T}_{k_2n}^{k_1k_2n} \\ & (\mathrm{L}_m^{mk_1} \otimes \mathrm{I}_{k_2n})(\mathrm{I}_{k_1} \otimes \underbrace{(\mathrm{I}_m \otimes \mathrm{DFT}_{k_2n})}_{(a)}\, \mathrm{L}_m^{mk_2n})(\mathrm{L}_{k_1}^{k_1k_2n} \otimes \mathrm{I}_m).\end{aligned} \qquad (4)$$

As construct (a) in equation (4) is of the same shape as the original construct, the vector recursion can be applied more than once. When finally $k_2 = 1$ is reached, the recursion stops and

$$(\mathrm{I}_m \otimes \mathrm{DFT}_n)\, \mathrm{L}_m^{mn} = \mathrm{L}_m^{mn} (\mathrm{DFT}_n \otimes \mathrm{I}_m) \qquad (5)$$

is applied. Thus, any operations apart from the leaf permutation L_m^{mn} is carried out on vectors of length m and n. Thus, if $m = n_1\nu$ and $n = n_1\nu$ all operations can be done using vector memory access. In that case the permutation L_m^{mn} can be implemented efficiently using vector memory access and the only required in-register permutation is $\mathrm{L}_\nu^{\nu^2}$ which can be implemented efficiently. These facts account for the restricted applicability of the vector recursion to multiples of ν^2, i. e., multiples of 16 for four-way short vector extensions and multiples of four for two-way vector extensions.

Example 2 (FFTW Vector Recursion for $\mathrm{DFT}_{\mathbf{rstu}}$) Suppose a DFT computation using three twiddle codelet stages and one no twiddle stage. Using Equation 1 leads to

$$\mathrm{DFT}_{rstu} = (\mathrm{DFT}_r \otimes \mathrm{I}_{stu})\, \mathrm{T}^{rstu}_{stu} \underbrace{(\mathrm{I}_r \otimes \mathrm{DFT}_{stu})\, \mathrm{L}^{rstu}_{r}}_{(b)} .$$

Now applying equation (4) for $m = r$, $k_1 = s$, $k_2 = t$, and $n = u$ to construct (b) in the above equation leads to

$$\begin{aligned}\mathrm{DFT}_{rstu} \;=\;& (\mathrm{DFT}_r \otimes \mathrm{I}_{stu})\, \mathrm{T}^{rstu}_{stu} \\ & (\mathrm{I}_r \otimes (\mathrm{DFT}_s \otimes \mathrm{I}_{tu})\, \mathrm{T}^{stu}_{tu})\, (\mathrm{L}^{rs}_r \otimes \mathrm{I}_{tu})(\mathrm{I}_s \otimes (\mathrm{I}_r \otimes \mathrm{DFT}_{tu})\, \mathrm{L}^{rtu}_r)(\mathrm{L}^{stu}_s \otimes \mathrm{I}_r).\end{aligned}$$

Applying equation (4) a second time with $m = r$, $k_1 = t$, $k_2 = 1$, and $n = u$ leads to

$$\begin{aligned}\mathrm{DFT}_{rstu} \;=\;& (\mathrm{DFT}_r \otimes \mathrm{I}_{stu})\, \mathrm{T}^{rstu}_{stu} \\ & \big(\mathrm{I}_r \otimes (\mathrm{DFT}_s \otimes \mathrm{I}_{tu})\, \mathrm{T}^{stu}_{tu}\big) \\ & (\mathrm{L}^{rs}_r \otimes \mathrm{I}_{tu})(\mathrm{I}_s \otimes \big(\mathrm{I}_r \otimes (\mathrm{DFT}_t \otimes \mathrm{I}_u)\, \mathrm{T}^{tu}_u\big)\, (\mathrm{L}^{rt}_r \otimes \mathrm{I}_u)(\mathrm{I}_t \otimes (\mathrm{I}_r \otimes \mathrm{DFT}_u)\, \mathrm{L}^{ru}_r)(\mathrm{L}^{tu}_t \otimes \mathrm{I}_r))(\mathrm{L}^{stu}_s \otimes \mathrm{I}_r).\end{aligned}$$

Applying the leaf transformation equation (5) leads to

$$\begin{aligned}\mathrm{DFT}_{rstu} \;=\;& (\mathrm{DFT}_r \otimes \mathrm{I}_{stu})\, \mathrm{T}^{rstu}_{stu} \\ & \big(\mathrm{I}_r \otimes (\mathrm{DFT}_s \otimes \mathrm{I}_{tu})\, \mathrm{T}^{stu}_{tu}\big) \\ & (\mathrm{L}^{rs}_r \otimes \mathrm{I}_{tu})(\mathrm{I}_s \otimes \big(\mathrm{I}_r \otimes (\mathrm{DFT}_t \otimes \mathrm{I}_u)\, \mathrm{T}^{tu}_u\big)\, (\mathrm{L}^{rt}_r \otimes \mathrm{I}_u)(\mathrm{I}_t \otimes (\mathrm{L}^{ru}_r (\mathrm{DFT}_u \otimes \mathrm{I}_r))(\mathrm{L}^{tu}_t \otimes \mathrm{I}_r))(\mathrm{L}^{stu}_s \otimes \mathrm{I}_r).\end{aligned}$$

which is the final expansion. Note, that except from L^{ru}_r any expressions in this formula are (i) twiddle factors, (ii) of form $A \otimes \mathrm{I}_r$, or (iii) of form $A \otimes \mathrm{I}_u$ and thus of form $(A \otimes \mathrm{I}_k) \otimes \mathrm{I}_\nu$ if $m = n_1\nu$ and $n = n_1\nu$.

5. Experimental Results

The methods described above were included into and tested with an experimental version of FFTW, `nfftw2` by Matteo Frigo and Steven Johnson. The development is intended to be a short vector add-on for the next version of FFTW. The newly developed short vector framework was tested for four-way SIMD on an 650 MHz Pentium III, an 1.8 GHz Pentium 4, an Athlon XP 1800+ running at 1533 MHz for SSE, and on an 400 MHz Motorola PPC 740 G4 for AltiVec. The Intel C++ compiler 6.0 was used for all experiments. Two-way SIMD was tested using SSE 2 on the 1.8 GHz Pentium 4. The performance is displayed in pseudo Gflop/s (runtime $/5N \log N$) which is a scaled inverse of the runtime and thus preserves the runtime relations and additionally gives an indication of the absolute performance [6]. The SIMD version is compared to the scalar version and—whenever possible—to the hand-optimized vendor library Intel MKL 5.1. [1] It is important to note, that `nfftw2` is not an official release and thus only the performance relations are relevant.

Concerning four-way short vector extensions, these experiments show that that the vector recursion is superior whenever applicable. It is important to note that SSE has hardware support for loading subvectors of size two while that is an costly operation on the AltiVec extension. On the other hand AltiVec internally operates on vectors of size four while on all machines featuring SSE the four-way operations are broken internally into two two-way operations thus limiting the vectorization speed-up. However, due to other architectural implications the theoretical speed-up limit achievable due to vectorization (thus ignoring effects like smaller program size due to less instructions) is a factor of four for SSE on the Pentium III, Pentium 4 and for AltiVec on the PPC 7400 G4. For SSE on the Athlon XP and SSE 2 on the Pentium 4 the limit is a factor of two.

Figures 1 and 3 show the vector recursion applied to power of two problem sizes tested across all IA-32 compatible machines within the test pool. Both SSE and SSE 2 was tested. Figure 1 focusses on problem sizes of powers of two. On the Pentium 4 speed-ups of more than three are achieved for SSE and more than 1.8 for SSE 2 for data sets that fit into the L1 data cache. For data sets that fit into the L2 cache but not into the L1 cache, speed-ups of 2.5 for SSE and 1.7 for SSE 2 were achieved. On the Pentium III speed-up factors up to 2.8 are achieved The short vector version achieves speed-ups of up to 1.8 on the Athlon XP. Typically, the short vector version of FFTW is faster than the Intel MKL with the exception of out of L1 cache computations on the Pentium 4 where the MKL uses *prefetching.* Figure 3 focusses on non-powers of two (the problem sizes are multiples of 16) to all IA-32 compatible machines. Depending on the actual problem size good speed-ups can be achieved. However, for some problem sizes (featuring large prime factors) the performance gain breaks down significantly. On the Pentium 4 speed-ups of 2.2 are achieved for SSE and 1.7 for SSE 2. On the Pentium III speed-ups between 2.5 and 3 are achieved while on the Athlon XP speed-ups around 1.8 are achieved.

[1] `http://www.intel.com/software/products/mkl`

Figures 2 and 4 show the standard Cooley-Tukey recursion with subvector memory access using SSE on the Pentium III and AltiVec on the PPC 7400 G4. Figure 2 focusses on problem sizes of powers of two. Speed-up factors of up to two are achieved on the Pentium III and speed-up factors of up to 2.5 on the G4. Figure 4 focusses on non-power of two sizes which are *not* multiples of 16. Thus the standard Cooley-Tukey recursion using subvector memory access has to be used. Speed-up factors of more than two are achieved on both machines.

Conclusion

An experimental short vector version of FFTW was presented and analyzed. The two variants of the Cooley-Tukey recursion provided by FFTW were used and extended to support short vector instructions efficiently. The implementation provides good speed-ups for arbitrary problem sizes and excellent speed-up for special problem sizes, i. e., multiples of 16 and powers of two.
Future work will include further optimization and the development of a short vector extension publicly available in the next official release of FFTW.

Acknowledgements

The author wants to thank Matteo Frigo and Steven Johnson for the fruitful ongoing collaboration and for pointing out the vector recursion of FFTW. In addition, the author wants to thank Christoph Ueberhuber and the Austrian Science Fund FWF for their support.

References

[1] Intel Corporation, *Intel C/C++ Compiler User's Guide*, 2002.

[2] Advanced Micro Devcies Corporation, *3DNow! Technology Manual*, 2000.

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

[4] C. F. Van Loan, *Computational Frameworks for the Fast Fourier Transform*, SIAM Press, Philadelphia, 1992.

[5] C. Fabianek, F. Franchetti, M. Frigo, H. Karner, L. Meirer, and C.W. Ueberhuber, "Survey of Self-adapting FFT Software," Technical Report AURORA TR-2002-01, Institute for Applied Mathematics and Numerical Analysis, Vienna University of Technology, 2002.

[6] Matteo Frigo and Steven G. Johnson, "FFTW: An Adaptive Software Architecture for the FFT," in *ICASSP 98*, 1998, vol. 3, pp. 1381–1384, `http://www.fftw.org`.

[7] F. Franchetti, H. Karner, S. Kral, and C. W. Ueberhuber, "Architecture Independent Short Vector FFTs," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP'01)*, pp. 1109–1112. IEEE Press, New York, 2001.

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

[9] 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`.

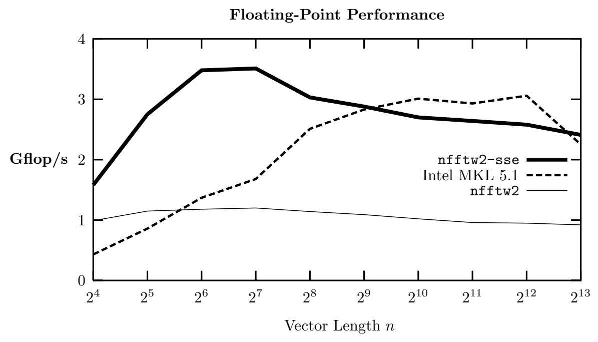
[10] Franz Franchetti and Markus Püschel, "A SIMD Vectorizing Compiler for Digital Signal Processing algorithms," in *Proc. International Parallel and Distributed Processing Symposium (IPDPS'02)*, 2002.

[11] F. Franchetti, M. Püschel, J. Moura, and C. W. Ueberhuber, "Short Vector SIMD Code Generation for DSP Algorithms.," Proc. of the 2002 High Performance Embedded Computing, 2002.

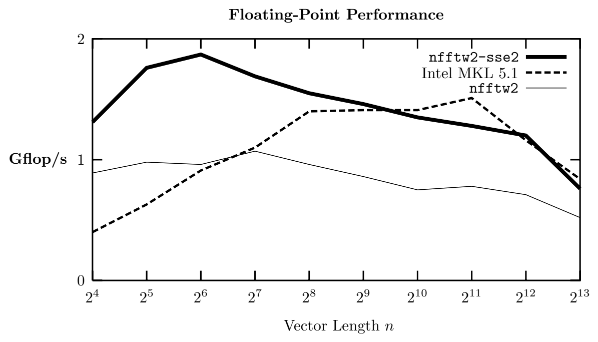
[12] J. Johnson, R. W. Johnson, D. Rodriguez, and R. Tolimieri, "A Methodology for Designing, Modifying, and Implementing FFT Algorithms on Various Architectures," *Circuits Systems Signal Process.*, vol. 9, pp. 449–500, 1990.

[13] G. Haentjens, "An Investigation of Cooley-Tukey Decompositions for the FFT," Masters thesis, Electrical and Computer Engineering Department, Carnegie Mellon University, Pittsburgh, PA, 2000.

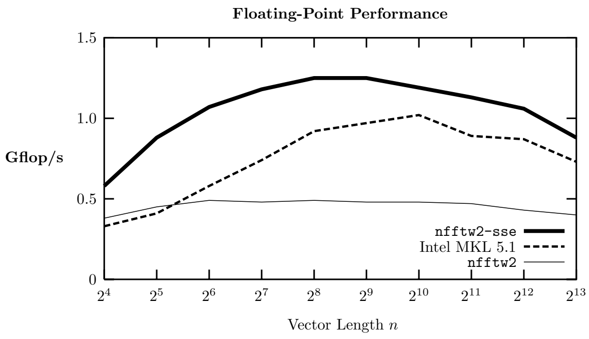
(a)



(b)



(c)



(d)

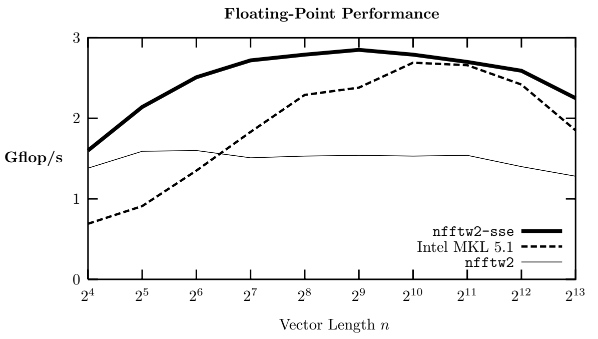
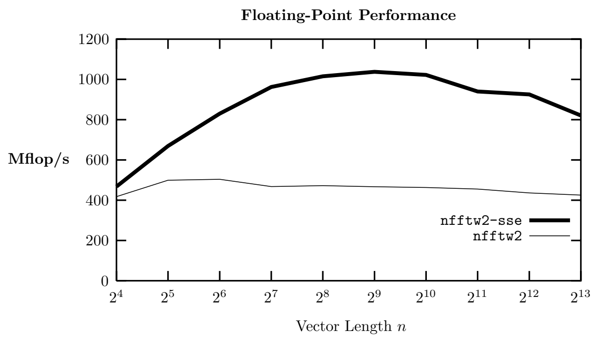


Figure 1: Performance results for DFT_N, $N = 2^4, \ldots, 2^{13}$, using the vector recursion for (a) single-precision and SSE, (b) double precision and SSE 2 on an Intel Pentium 4 running at 1.8 GHz, (c) single-precision and SSE on an Intel Pentium III running at 650 MHz, and (d) single-precision and SSE on an AMD Athlon XP 1800+ running at 1533 MHz. The FFT programs compared are (i) FFTW-SIMD, (ii) the Intel MKL 3.5, and (iii) the scalar FFTW version.

(a)



(b)

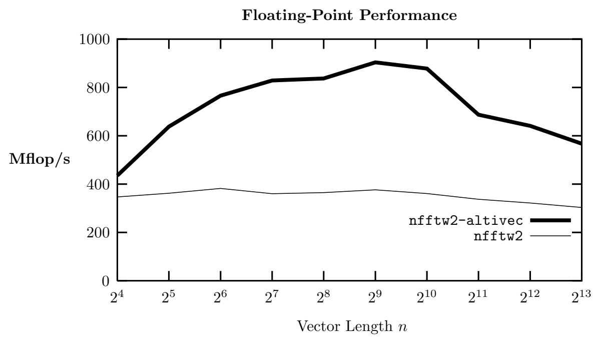
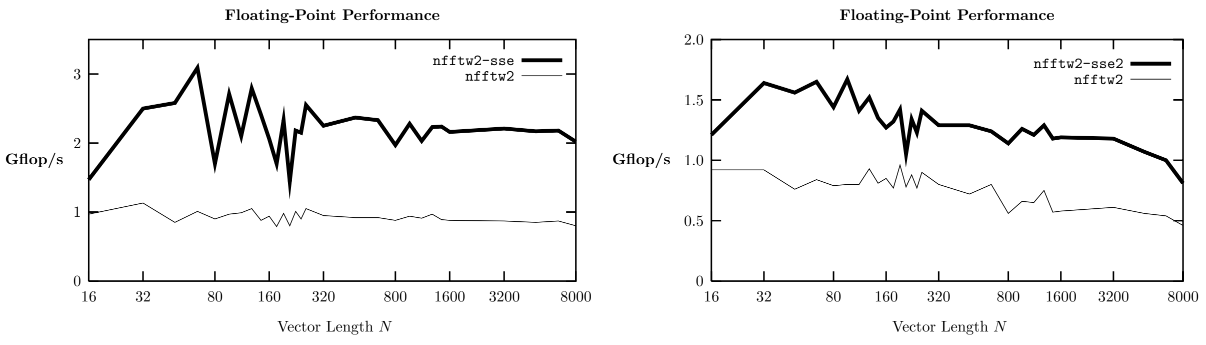


Figure 2: Performance results for DFT_N, $N = 2^4, \ldots, 2^{13}$, using the standard recursion and subvector memory access for (a) single-precision and SSE on an Intel Pentium III running at 650 MHz, and (b) single-precision and AltiVec on a Motorola PPC 7400 G4 running at 400 MHz. The FFT programs compared are (i) FFTW-SIMD, and (ii) the scalar FFTW version.

(*a*) (*b*)



(*c*) (*d*)

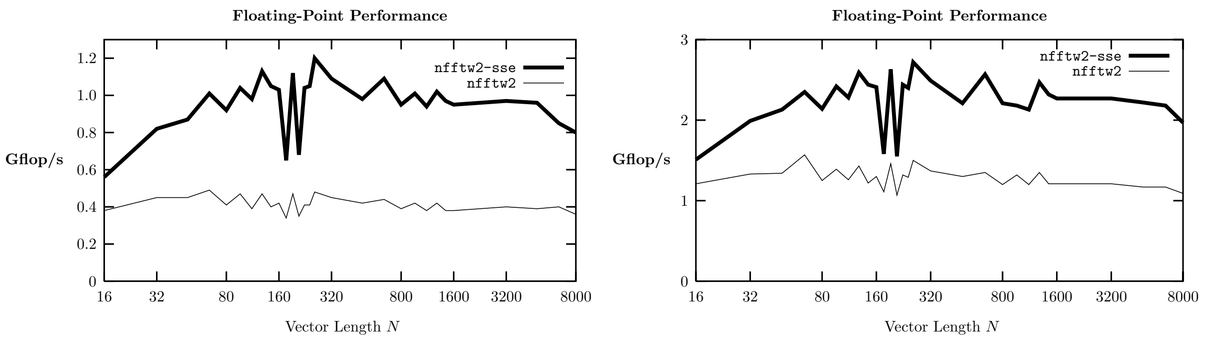


Figure 3: Performance results for DFT_N, $N = 16, \ldots, 8000$, $N = 16k$, using the vector recursion for (*a*) single-precision and SSE and (*b*) double precision and SSE 2 on an Intel Pentium 4 running at 1.8 GHz, (*c*) for single-precision and SSE on an Intel Pentium III running at 650 MHz, and (*d*) single-precision and SSE on an AMD Athlon XP 1800+ running at 1533 MHz. The FFT programs compared are (*i*) Fftw-Simd, and (*ii*) the scalar Fftw version.

(*a*) (*b*)

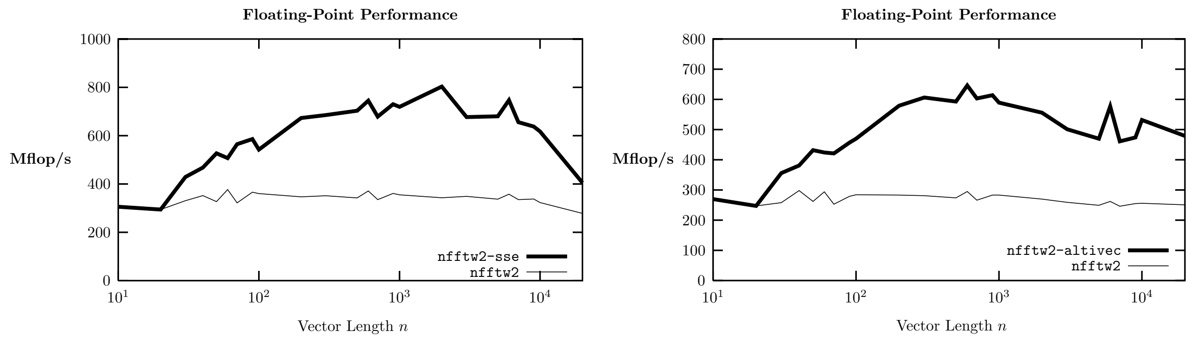


Figure 4: Performance results for DFT_N, $N = 10, \ldots, 20\,000$, using the standard recursion and subvector memory access for (*a*) single-precision and SSE on an Intel Pentium III running at 650 MHz, and (*b*) single-precision and AltiVec on a Motorola PPC 7400 G4 running at 400 MHz. The FFT programs compared are (*i*) Fftw-Simd, and (*ii*) the scalar Fftw version.