# Discrete Fourier Transform on Multicores

Franz Franchetti, Markus Püschel, Yevgen Voronenko, Srinivas Chellappa, and José M. F. Moura

***Abstract*— This paper gives an overview on the techniques needed to implement the discrete Fourier transform (DFT) efficiently on current multicore systems. The focus is on Intel compatible multicores but we also discuss the IBM Cell, and briefly, graphics processing units (GPUs). The performance optimization is broken down into three key challenges: parallelization, vectorization, and memory hierarchy optimization. In each case, we use the Kronecker product formalism to formally derive the necessary algorithmic transformations based on a few hardware parameters. Further code level optimizations are discussed. The rigorous nature of this framework enables the complete automation of the implementation task as shown by the program generator Spiral. Finally, we show and analyze DFT benchmarks of the fastest libraries available for the considered platforms.**

## I. Introduction

The evolution of computing platforms is at a historic inflection point: after years of exponential growth, CPU frequency has stalled due to physical limitations. However, the theoretical floating point peak performance, a critical measure for the processing abilities of a platform, continues to increase at the pace predicted by Moore's Law. The reason is an increase in parallelism in the form of multiple processor cores and vector processing abilities [1].

The consequences for signal processing software are dramatic: it means the end of free speed-up for legacy software and a dramatically increased difficulty of writing high performance code, since the programmer now has to use multiple threads, vector instruction sets, and tune the code to the memory hierarchy. Unfortunately, this process is platform-specific: performance does not port easily. Failure to apply these optimizations by hand can result in dramatic performance losses as shown in Fig. 1. The figure shows the performance (in Gflop/s = gigafloating point operations per second; higher is better) for four implementations of the discrete Fourier transform (DFT), arguably the most important signal processing function and focus of this paper. All implementations use fast Fourier transform algorithms (FFTs) with roughly the same number of operations. Yet, the performance difference between the best and worst is a factor of 12 to 35. The bottom line shows the implementation from Numerical Recipes [2] based on a standard radix-2 iterative FFT. The next line is one of the best scalar (standard C code) implementations and is 5 times faster, since it is tuned to the memory hierarchy. The next one gains up to another factor of 3 by using SSE vector instructions. Finally, the fastest code uses up to four processor cores and speeds up the code by another factor of 3 for larger sizes. The performance drop for very large sizes is due to inevitable cache misses once the working set cannot be held in the largest cache.

The main reasons for the achieved speed-up are algorithmic: all top three lines are based on nonstandard FFT variants whose structure matches the architectural constraints imposed by this multicore platform. Equally notable, the source code for the fast implementations were not written by a human but automatically produced by a tool called Spiral [3], [4], [5] that we developed and whose framework underlies this paper.

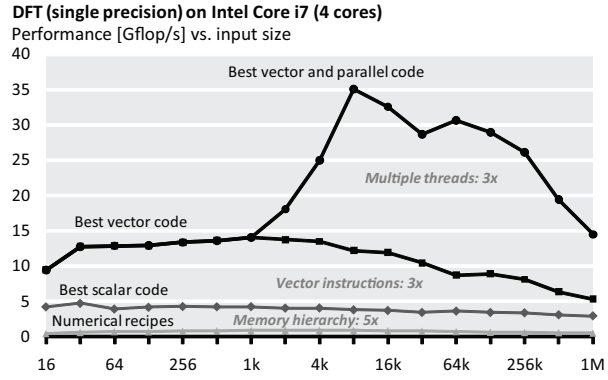



Fig. 1. Naive DFT implementations based on only minimizing the operations count underperform considerably on modern multicore CPUs.

We present a tutorial-style overview of the optimizations needed to achieve good performance on Intel multicore platforms (top line in Fig. 1), the IBM Cell, and briefly touch on GPUs. Starting from a standard recursive fast Fourier transform (FFT), we use the Kronecker product formalism to derive the necessary structural, or algorithmic optimizations. Necessary code level optimizations are also discussed. In this way, we address all three key problems identified in Fig. 1: multiple cores, vector instruction sets, and the memory hierarchy.

Finally, we show and analyze performance benchmarks of the fastest libraries for Intel Core, IBM Cell, and GPUs. Specifically, we show benchmarks for FFTW [6], [7], [8], Intel's IPP, Spiral-generated libraries, FFTC [9], and [10] for the GPU. Other libraries and related work will be introduced throughout the paper.

## II. Framework

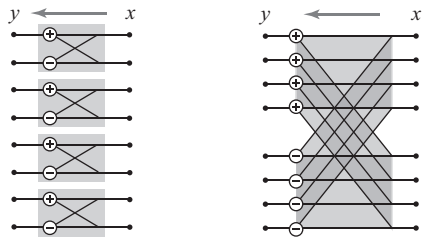
There are two fundamentally different ways of representing linear transforms such as the DFT: as matrix-vector products or using summations. Correspondingly, fast algorithms are represented either with a matrix formalism as in [11], [12] or as nested summations as in most signal processing books. In this section, we introduce the matrix formalism and use it to express various classical FFTs. Later in the paper, we use this formalism as a tool for structural manipulation of FFTs to derive variants that can be efficiently mapped to current multicore systems. The formal nature of the approach makes computer generation of FFT libraries possible as we have demonstrated with the tool Spiral [13] that we developed and briefly discuss later.

We focus on the DFT but note that the framework extends to a large class of linear transforms [3].

**Discrete Fourier transform.** The DFT of $n$ input samples

This work was supported by NSF through awards 0325687, 0702386, by DARPA (DOI grant NBCH1050009), the ARO grant W911NF0710416, and by Intel Corp. and Mercury Computer Systems, Inc.

The authors are with the Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh. E-mail: {franzf, yvoronen, schellap, pueschel, moura}@ece.cmu.edu. Ph: 412-268-6341; fax: 412-268-3890.




(a) $y = (I_4 \otimes \mathrm{DFT}_2)x$ (b) $y = (\mathrm{DFT}_2 \otimes I_4)x$

Fig. 2. Dataflow (right to left) of a parallel and its "dual" vector construct.

$x_0, \ldots, x_{n-1}$ is defined in summation form as

$$y_k = \sum_{0 \le \ell < n} \omega_n^{k\ell} x_\ell, \quad 0 \le k < n, \tag{1}$$

with $\omega_n = \exp(-2\pi j/n)$. Stacking the $x_\ell$ and $y_k$ into vectors $x$ and $y$ yields the equivalent form of a matrix-vector product:

$$y = \mathrm{DFT}_n\, x, \quad \mathrm{DFT}_n = [\omega_n^{k\ell}]_{0 \le k,\ell < n}. \tag{2}$$

We drop $x$ and $y$ and simply think of the matrix $\mathrm{DFT}_n$ as the transform, implicitly assuming that it is multiplied to $x$. Fast algorithms are now expressed as factorizations of $\mathrm{DFT}_n$ using the following formalism.

**Matrix formalism.** We denote with $I_n$ the $n \times n$ identity matrix, and the *butterfly matrix* with

$$\mathrm{DFT}_2 = \left[\begin{smallmatrix} 1 & 1 \\ 1 & -1 \end{smallmatrix}\right]. \tag{3}$$

The *Kronecker product* of matrices $A$ and $B$ is defined as

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

It replaces every entry $a_{k,\ell}$ of $A$ by the matrix $a_{k,\ell}B$. Most important are the cases where $A$ or $B$ is the identity. As examples, we show in Fig. 2 the structure and dataflow of the "dual" constructs $I_4 \otimes \mathrm{DFT}_2$ and $\mathrm{DFT}_2 \otimes I_4$. The former is obviously parallel, the latter has vector structure: it can be viewed as a single $\mathrm{DFT}_2$ operating on vectors of length 4 instead of scalars. This will be crucial later.

The *stride permutation* matrix $L^{mn}_m$ permutes the elements of the input vector as $in+j \mapsto jm+i$, $0 \le i < m$, $0 \le j < n$. If the vector $x$ is viewed as an $n \times m$ matrix, stored in row-major order, then $L^{mn}_m$ performs a transposition of this matrix. Further, if $P$ is a permutation (matrix), then $A^P = P^{-1}AP$ is the *conjugation* of $A$ with $P$. Note that $y = A^P x$ implies $Py = APx$, i.e., $A$ is performed on vectors permuted with $P$.

There are various identities connecting these constructs [14] as shown in Table I. For example, (9) shows how to translate the duals in Fig. 2 into each other.

The Kronecker product naturally arises in 2D and 3D DFTs, which respectively can be written as

$$\mathrm{DFT}_{m \times n} = \mathrm{DFT}_m \otimes \mathrm{DFT}_n, \tag{13}$$

$$\mathrm{DFT}_{k \times m \times n} = \mathrm{DFT}_k \otimes \mathrm{DFT}_m \otimes \mathrm{DFT}_n\,. \tag{14}$$

**Recursive DIT FFT.** The matrix formalism can be used to express FFTs as factorizations of the matrix $\mathrm{DFT}_n$ in (2). As an example, the recursive general-radix decimation in time (DIT) Cooley-Tukey FFT for $n = km$ is

$$\mathrm{DFT}_{km} = (\mathrm{DFT}_k \otimes I_m) T^n_m (I_k \otimes \mathrm{DFT}_m) L^n_k. \tag{15}$$

$$(BC)^\top = C^\top B^\top \tag{4}$$

$$(A \otimes B)^\top = A^\top \otimes B^\top \tag{5}$$

$$I_{mn} = I_m \otimes I_n, \tag{6}$$

$$A \otimes B = (A \otimes I_m)(I_n \otimes B) \tag{7}$$

$$A \otimes (BC) = (A \otimes B)(A \otimes C), \tag{8}$$

$$A \otimes B = L^{mn}_n (B \otimes A) L^{mn}_m, \tag{9}$$

$$(L^{mn}_m)^{-1} = L^{mn}_n \tag{10}$$

$$L^{kmn}_n = (L^{kn}_n \otimes I_m)(I_k \otimes L^{mn}_n), \tag{11}$$

$$L^{kmn}_{km} = (I_k \otimes L^{mn}_m)(L^{kn}_k \otimes I_m), \tag{12}$$

TABLE I

FORMULA IDENTITIES TO MANIPULATE FFT ALGORITHMS. $A$ IS $n \times n$, AND $B$ AND $C$ ARE $m \times m$. $A^\top$ IS THE TRANSPOSE OF $A$.

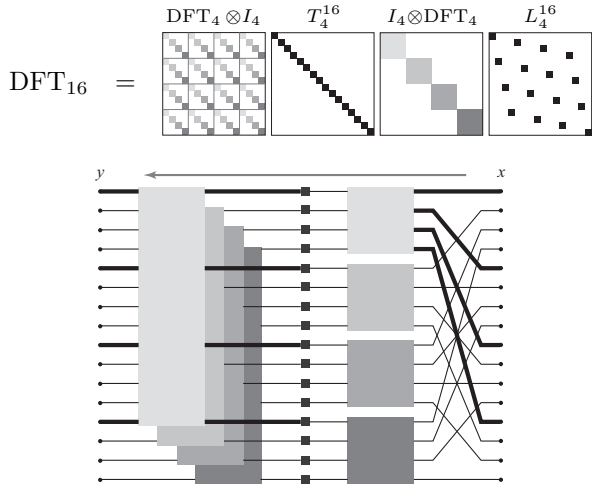



Fig. 3. Cooley-Tukey FFT (15) for $16 = 4 \times 4$ as matrix formula and as (complex) data-flow graph (from right to left). Some lines are bold to emphasize the strided access.

Here, $T^n_m$ is a diagonal matrix containing the *twiddle factors*. Fig. 3 shows the special case $16 = 4 \times 4$. Using this algorithm on an input vector $x$ means (reading Fig. 3 from left to right):

- $L^{16}_4$: Read $x$ at stride 4.
- $I_4 \otimes \mathrm{DFT}_4$: Apply 4 $\mathrm{DFT}_4$'s on consecutive chunks.
- $T^{16}_4$: Scale by the diagonal elements of $T^{16}_4$.
- $\mathrm{DFT}_4 \otimes I_4$: Apply 4 $\mathrm{DFT}_4$'s at stride 4.

**Recursive FFT variants.** Looking at Table I, it becomes clear that many variants of (15) can be derived. For example, simple transposition using that $\mathrm{DFT}^\top_n = \mathrm{DFT}_n$ yields the decimation in frequency (DIF) Cooley-Tukey FFT

$$\mathrm{DFT}_{km} = L^n_m (I_k \otimes \mathrm{DFT}_m) T^n_m (\mathrm{DFT}_k \otimes I_m). \tag{16}$$

The Four Step algorithm [15], [16], [11]

$$\mathrm{DFT}_{km} = (\mathrm{DFT}_k \otimes I_m) T^n_m L^n_k (\mathrm{DFT}_m \otimes I_k) \tag{17}$$

was originally developed for vector computers. It produces the longest possible unit stride vector operations at the cost of a transposition. The Six Step algorithm [17], [11]

$$\mathrm{DFT}_{km} = L^n_k (I_m \otimes \mathrm{DFT}_k) L^n_m T^n_m (I_k \otimes \mathrm{DFT}_m) L^n_k. \tag{18}$$

was originally developed for distributed memory machines. It produces fully local computation at the cost of three global transpositions (all-to-all data exchanges). Both (17) and (18)

are derived from (15) using (9)–(10). The required matrix transposition can be blocked using (11)–(12) for more efficient data movement.

For a 2D DFT, applying (7) to (13) yields the row-column algorithm

$$\mathrm{DFT}_{m\times n} = (\mathrm{DFT}_m \otimes I_n)(I_m \otimes \mathrm{DFT}_n). \qquad (19)$$

The 2D vector radix algorithm [18] is derived with (6)–(10):

$$\begin{aligned}\mathrm{DFT}_{mn\times rs} \;=\; & \big(\mathrm{DFT}_{m\times r}\otimes I_{ns}\big)^{I_m\otimes L_r^{rn}\otimes I_s}\big(T_n^{mn}\otimes T_s^{rs}\big)\\ & \big(I_{mr}\otimes \mathrm{DFT}_{n\times s}\big)^{I_m\otimes L_r^{rn}\otimes I_s}\big(L_m^{mn}\otimes L_r^{rs}\big).\end{aligned}$$

Higher-dimensional versions are derived similarly, and the associativity of $\otimes$ gives rise to more variants.

**Iterative FFT Algorithms.** A second class (historically earlier) of FFT algorithms based on (15) are the *iterative FFT algorithms*, obtained by recursively expanding (15) and again using Table I. The simplest are *radix-r* forms (usually $r = 2, 4, 8$), which require an FFT size of $n = r^k$ (more complicated mixed-radix radix variants always exist).

The decimation in time *triple-loop* FFT [19], [11],

$$\mathrm{DFT}_{r^k} = R_r^{r^k}\prod_{i=0}^{k-1} D_i^{r^k}\big(I_{r^{k-i-1}}\otimes \mathrm{DFT}_r\otimes I_{r^i}\big), \qquad (20)$$

is the simplest iterative algorithm. $R_r^{r^k}$ is the radix-$r$ digit reversal permutation and $D_i^{r^k}$ contains the twiddle factors in the $i$th stage. A radix-2 version is implemented by Numerical Recipes [2]. A variant of (20) is the *Pease* FFT [20], [11], which has constant geometry, i.e., the control flow is independent of the stage; however, it also requires the digit reversal permutation. It was originally developed for parallel computers, and its regular structure makes it a good choice for field-programmable gate arrays (FPGAs) or ASICs.

The *Stockham* FFT [21], [11],

$$\mathrm{DFT}_{r^k} = \prod_{i=0}^{k-1}\big(\mathrm{DFT}_r\otimes I_{r^{k-1}}\big)D_i^{r^k}\big(L_r^{r^{k-i}}\otimes I_{r^i}\big), \qquad (21)$$

is *self-sorting*, i.e., it does not have a digit reversal permutation. It was originally developed for vector computers.

**DFT variants and other FFTs.** In practice, several variants of the DFT in (2) are needed including forward/inverse, interleaved/split complex format, for complex/real input data, inplace/out-of-place ($y = x$ or not), and others [22]. Fortunately, most of these variants are close to the standard DFT in (2), so fast code for the latter can be adapted. An exception is the DFT for real input data, which has its own class of FFTs (see [4] for an overview using the above formalism). This paper focuses on the standard 1D interleaved (alternating real and imaginary parts) complex DFT in (2).

DFT algorithms fundamentally different from (15) include prime-factor ($n$ is a product of coprime factors), Rader ($n$ is prime), and Bluestein or Winograd (any $n$) FFTs and can also be expressed in the above formalism [11]. In practice these are mostly used for small sizes $< 32$, which then serve as building blocks for large composite sizes via (15).

**From matrix formulas to implementations.** Table II shows how to translate matrix formulas into basic sequential loop code. However, strictly applying the upper part of the table will lead to low performance. The last three entries show optimized translations. The last two show how a diagonal scaling is always fused with subsequent loops and how permutations are (almost) always done as readdressing in the subsequent loop. The shown translation of $y = (L_m^{mn}\otimes I_k)x$ replaces every scalar in the shown translation of $y = L_m^{mn}x$ by a vector of length $k$, similar as in Fig. 2(b).

| Matrix formula | Matlab pseudo code |
|---|---|
| $y = (A_nB_n)x$ | `t[0:1:n-1] = B(x[0:1:n-1]);`<br>`y[0:1:n-1] = A(t[0:1:n-1];)` |
| $y = \left(\prod_{i=0}^{k-1} A_i\right)x$ | `y = x;`<br>`for (i=0; i<k; i++)`<br>`  {x = y; y = A(i, x);}` |
| $y = (I_m \otimes A_n)x$ | `for (i=0; i<m; i++)`<br>`  y[i*n:1:i*n+n-1] = A(x[i*n:1:i*n+n-1]);` |
| $y = (A_m \otimes I_n)x$ | `for (i=0; i<n; i++)`<br>`  y[i:n:i+m*n-n] = A(x[i:n:i+m*n-n]);` |
| $y = D_nx$ | `for (i=0; i<n; i++)`<br>`  y[i] = Dn[i]*x[i];` |
| $y = L_m^{mn}x$ | `for (i=0; i<m; i++)`<br>`  for (j=0; j<n; j++)`<br>`  y[i+m*j:1:i+m*j] = x[n*i+j:1:n*i+j];` |
| $y = (L_m^{mn}\otimes I_k)x$ | `for (i=0; i<m; i++)`<br>`  for (j=0; j<n; j++)`<br>`  y[k*(i+m*j):1:k*(i+m*j)+k-1] =`<br>`  x[k*(n*i+j):1:k*(n*i+j)+k-1];` |
| $y = (A_m \otimes I_n)D_{mn}x$ | `for (i=0; i<n; i++)`<br>`  t = Dmn[i:n:i+m*n-n]*x[i:n:i+m*n-n];`<br>`  y[i:n:i+m*n-n] = A(t);` |
| $y = (I_m \otimes A_n)L_m^{mn}x$ | `for (i=0; i<m; i++)`<br>`  y[i*n:1:i*n+n-1] = A(x[i:m:i+n*m-m]);` |

TABLE II

FROM MATRIX FORMULAS TO CODE. THE SUBSCRIPT OF $A, B$ SPECIFIES THE (SQUARE) MATRIX SIZE. `x[b:s:e]` DENOTES THE SUBVECTOR OF $x$ STARTING AT `b`, ENDING AT `e`, EXTRACTED AT STRIDE `s`. THE DIAGONAL ELEMENTS OF $D$ ARE STORED IN AN ARRAY WITH THE SAME NAME.

Further, different types of implementations are possible, briefly discussed next.

*Single-size kernels:* For small input sizes $\leq$ 32 or 64, matrix formulas are often implemented as fully unrolled code blocks. In this case array scalarization and, to a lesser extent, algebraic optimizations and scheduling are used to achieve best performance and can be completely automated [7], [23], [3], [24]. For example, a $\mathrm{DFT}_8$ kernel is implemented in a few tens of lines of code.

*Single-size loop code:* If the input size is known in advance, a fixed formula or data flow can be chosen and implemented using nested loops arising from tensor products and iterative products. As discussed before, scaling and readdressing is merged into kernels for high performance. These implementations can be generated automatically using Spiral [23], [3], [25]. One such single-size implementation can require up to a few hundreds of lines of code.

*General-size loop code:* Only iterative algorithms lend themselves to general-size loop code implementation. Again, readdressing is folded into the computational kernel. An example is the Numerical Recipes code [2], which implements (20) for $r = 2$ as a triple loop in about one page of C code.

*General-size recursive code:* Translating formulas into recursive code is complicated, but is at the heart of several high-performance portable general-size FFT libraries [8], [26], [27],

[28]. A tutorial for a recursive radix-4 FFT is given in [29], leading to about two pages of C code. Extension to vector and multicore platforms considerably increases the code size: for example, FFTW contains more than 200,000 lines of code. The implementation of such libraries was also automated using Spiral [28], [4].

## III. Mapping FFTs to Multicore CPUs

Historically, the Kronecker product formalism was used to develop FFTs for parallel target platforms such as small-scale and massive multiprocessors, and vector computers [30], [16], [11]. We now discuss how to extend this approach to state-of-the-art multicore CPUs. The new hardware characteristics that need to be captured are: 1) multiple cores communicating through shared caches or explicit messages, 2) SIMD short vector instructions, and 3) the memory hierarchy and its transfer restrictions, such as caches and DMA-based streaming memory. We will address each of these features in three steps. First, we identify relevant hardware parameters. Second, we identify a set of matrix formulas that can be mapped efficiently for these parameters. Third, we derive a variant of the recursive Cooley-Tukey FFT (15) that is a member of this set. In each case, we also briefly discuss the mapping to actual code including further relevant code level optimizations. In Section VI, we then instantiate the concepts and algorithms to an Intel Core and the Cell BE and briefly discuss GPUs and FPGAs.

Choosing recursive algorithms is not a requirement. It is possible to start from iterative algorithms or combine one or two steps of recursion with an iterative algorithm and achieve reasonable performance (as demonstrated by several vendor libraries). However, many current high-performance libraries for cache-based machines implement the recursive FFT algorithms [6], [3], [4], [27], [26], [31] discussed here.

### A. Parallelism: Multiple Cores

The multicore CPUs we target may have shared caches (Fig. 4), private caches, or scratchpads (local stores) with data being transferred in packets. Cache coherent architectures transfer data implicitly between private and shared caches as required. Data transfer between scratchpads has to be managed explicitly by the programmer. In each case, to obtain best performance it is crucial to ensure that the whole data content in a transfer (e.g., cache line or DMA packet) is used by the receiver (spatial locality) and that the number of transfers is minimized (temporal locality).

**Machine model.** We assume that the packet size is a multiple of an atomic packet size of $\mu$ complex numbers. For instance, on a cache-based memory hierarchy, a cache coherency event, a cache miss, or an eviction always transmits a whole cache line (e.g., 64 bytes translates into $\mu$ = 8 for complex single-precision). In scratchpad based systems like the Cell, DMA packets need to be of sufficient size for performance; to yield reasonable performance, a Cell DMA transfer between SPEs should be at least 128 bytes ($\mu = 16$ for complex single-precision numbers) and a multiple of 16.

We consider CPUs with $p$ cores. Well designed parallel code is load balanced (all cores have the same amount of work), with minimal data transmission between cores, performed in packets of size $\mu$. On shared memory multicores this implies that the code is free of false sharing (two cores accessing different elements in the same cache line).

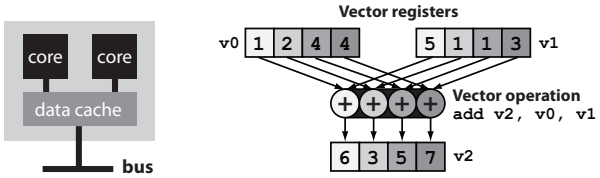



Fig. 4. Shared cache in a multicore CPU and SIMD vector extensions.

Matrix formulas solely built from

$$I_p \otimes A, \quad D_n, \quad P \otimes I_\mu \quad (P \text{ permutation, } D_n \text{ diagonal})$$

with $A$ a $m \times n$ matrix and $\mu \mid m, n$ can be implemented efficiently as parallel code; we call them *parallel constructs*. Namely, $I_p \otimes A$ is load balanced and embarrassingly parallel (see Fig. 2(a)), i.e., it does not require any communication. The same holds for scaling by $D_n$. Finally, the communication pattern $P_n \otimes I_\mu$ transmits entire packets of size $\mu$ between cores. Note that products of parallel constructs are again parallel constructs.

**Multicore Cooley-Tukey FFT.** We now state a multicore FFT built exclusively from parallel constructs, derived using Table I [5]:

$$\mathrm{DFT}_{mn} = \big(I_p \otimes (\mathrm{DFT}_m \otimes I_{n/p})\big)^{\left((L_p^{mp} \otimes I_{n/p\mu}) \otimes I_\mu\right)} T_m^{mn}$$
$$\big(I_p \otimes (I_{m/p} \otimes \mathrm{DFT}_n) L_{m/p}^{mn/p}\big)\big((L_p^{pn} \otimes I_{m/p\mu}) \otimes I_\mu\big). \quad (22)$$

Implementation of (22) on a cache-based system relies on the cache coherency protocol to transmit cache lines of length $\mu$ between cores and requires a global barrier. Implementation on a scratchpad based system requires explicit sending and receiving of the data packets, and depending on the communication interface additional synchronization may be required.

Equation (22) can be used as outermost recursion to enable multicore parallelization. The smaller DFTs are then expanded using the short vector Cooley-Tukey FFT (23) or the vector recursion (29) shown later in this section.

Historically, the Pease and the Six Step FFT (18) were starting points for parallel iterative or recursive implementations, but due to changed trade-offs these algorithms are no longer a good choice in many cases.

**Mapping to C code.** OpenMP [32] is a good choice for parallel code if it is well supported by the target platform's compiler. OpenMP allows the programmer to declare certain loops to be parallel, and to specify variables as shared or private. It enables the programming of sophisticated parallel software without needing to deal with lower-level threading details. As example, the formula $I_4 \otimes \mathrm{DFT}_2$ in Fig. 2(a) is translated into the OpenMP program snippet below. Note that only a C `#pragma` is inserted to instruct the compiler to parallelize the `for` loop. If OpenMP is turned off, these pragmas are ignored and the program becomes sequential.

```
double x[8], y[8];
#pragma omp parallel for
for (int i=0; i<4; i++)
{  y[2*i]   = x[2*i] + x[2*i+1];
   y[2*i+1] = x[2*i] - x[2*i+1];
}
```

Because of the regular structure of FFTs only a few more issues have to be addressed for efficient OpenMP parallelization. For correctness, the sharing and privatization of variables with the OpenMP `shared` and `private` clauses has to be done properly to avoid race conditions and other problems. For performance, scheduling hints can be provided (e.g., `schedule(static)`). In addition, if only a subset of the available cores is to be used, affinity can be set to choose the subset. For example, two threads operating on the same data should be physically close, i.e., share a high level of the memory hierarchy.

On some target platforms no OpenMP compiler may be available, in which case one must use threading libraries like the portable Posix threads (pthreads) [33] library or use operating system threading interfaces to build the required parallel loops and barriers. While this approach can yield a slight performance advantage, it requires understanding of the target architecture, its memory consistency model, and cache coherency protocols.

On the Cell processor the programmer needs to manage and synchronize threads on the PPE and the SPEs, and perform data movements via DMA transfers. Due to the Cell's unconventional architecture, libraries for it must be adapted to take advantage of its features. While there have been some programming paradigms ported to the Cell (including some function offloading interfaces), for the class of programs discussed in this paper, performance is best obtained by avoiding the overhead of such interfaces.

### *B. SIMD Vectorization*

Most multicore CPUs include vector instruction sets. SIMD vector extensions add vector registers (2-way double or 4-way float on the Core i7 and the Cell), and much longer vectors will be available in the near future (e.g., 16-way single precision on Intel's upcoming Larrabee GPU, and 4-way double and 8-way single precision in AVX on the next generation of Intel multicore CPUs). Vector instructions then operate on these registers in parallel, providing high potential speed-up. A 4-way vector addition is shown in Fig. 4.

**Machine model.** To obtain best performance on vector extensions, data should be loaded and stored with vector memory operations that transfer complete, naturally aligned vectors. Unaligned and subvector accesses are expensive. All operations on the vector registers should be vector operations (vector addition, subtraction, and multiplication). Data reorganization within registers (shuffles) are needed for FFTs but should be minimized.

For this paper, we restrict ourselves to what we call *complex vectorization*. We denote the machine vector length with $\nu$, meaning $\nu$ complex numbers are packed into a vector register of length $2\nu$; e.g., for 4-way float SSE, $\nu = 2$.

All formulas built solely from

$$A \otimes I_\nu, \quad D_n \text{ (complex diagonal)}, \quad \text{and} \quad L^{\nu^2}_\nu$$

can be implemented efficiently with vector instructions; we call them vector constructs. Moreover, if $A$ and $B$ are vector constructs, then $AB$ and $I_n \otimes A$ are vector constructs.

First, $A \otimes I_\nu$ is naturally vectorized (e.g., Fig. 2(b)): vector code for $y = (A \otimes I_\nu)x$ can be obtained from scalar code implementing $y = Ax$ by simply replacing all scalar operations by the corresponding vector operations, and all scalar variables by vector variables. Second, we assume that $y = D_n x$ can be implemented efficiently for $\nu \mid n$. This is a reasonable assumption: for instance, the SSE4.2 instruction set implemented by the Core i7 contains instructions for the efficient mapping of complex multiplications. Third, $y = L^{\nu^2}_\nu x$ can always be implemented with a small number of vector instructions [34].

**Short vector Cooley-Tukey FFT.** We show a short vector FFT algorithm that is built from vectorizable constructs, derived from (15) using Table I. It requires only a small number of in-register shuffles [35]:

$$\begin{aligned}\mathrm{DFT}_{mn} = \big((\mathrm{DFT}_m \otimes I_{n/\nu}) \otimes I_\nu\big) T^{mn}_n \big(I_{m/\nu} \otimes (I_{n/\nu} \otimes L^{\nu^2}_\nu)\\ (L^n_{n/\nu} \otimes I_\nu)(\mathrm{DFT}_n \otimes I_\nu)\big)\big(L^{mn/\nu}_{m/\nu} \otimes I_\nu\big). \quad (23)\end{aligned}$$

This FFT is composable with memory hierarchy optimized FFTs. Namely, inserting (23) into the vector recursion (29) shown later yields again a vector construct.

A somewhat more complicated *real* (using a real representation of matrices) short vector FFT is derived in [36], [37].

Traditional vector algorithms like the Four Step algorithm (17) or the Stockham algorithm (21) were designed for traditional vector computers with much longer vectors. Due to the expensive permutations, they are not a good choice for short-vector SIMD architectures.

**Mapping to C code.** The most convenient way to efficiently use SIMD extensions is through intrinsic function interfaces provided by most high-performance compilers. For instance, the Intel C++ compiler, Microsoft's VisualStudio C compiler, IBM's XL C compiler, and the GNU C compiler provide such an interface for the supported SIMD extensions.

The programmer uses a data type and function abstraction of the SIMD extensions to implement C code. The compiler understands the data types and special functions and maps the C program to the respective instructions. In this scenario the programmer must select the appropriate instructions and make sure machine restrictions like data alignment are met. However, the programmer does not have to directly use assembly and thus is spared from register allocation and instruction scheduling. For example, the formula $\mathrm{DFT}_2 \otimes I_4$ in Fig. 2(b) is implemented by the following C program snippet using intrinsics for the Intel C++ compiler. `__m128` is a built-in data type to abstract XMM vector registers, and `_mm_add_ps()` abstracts the SSE instruction `addps` through a function call:

```
__m128 x[2], y[2];
y[0] = _mm_add_ps(x[0], x[1]);
y[1] = _mm_sub_ps(x[0], x[1]);
```

### *C. Memory Hierarchy*

Our target multicores CPUs have a memory hierarchy with multiple cores sharing the off-chip bandwidth. Machines with memory hierarchies present algorithm designers with two challenges:

- *Temporal locality:* Faster memory levels are smaller, and the working set must be blocked to fit into that level to minimize data transfers.
- *Spatial locality:* Data transfer between memory hierarchy levels happens in packets. This implies that transferred

packets should be fully used to avoid wasting memory bandwidth.

**Machine model.** A memory hierarchy can have multiple levels. For a given level, we call the capacity $N$ if it can hold the working set for the computation of $y = Ax$ for an $N \times N$ matrix $A$. This implies that the input vector $x$, the output vector $y$, and all necessary temporary arrays and constants fit into that cache level. For instance, if we consider double-precision, one (complex) value is 16 bytes. If $A$ is a DFT, $N$ is the cache size divided by 64 (assuming a factor of 4 space overhead). As before, we assume that data is transferred between the current level and the next lower level in the memory hierarchy in packets of $\mu$ complex numbers. Moreover, if it is a set-associative cache, it can hold $\alpha$ lines of $\mu$ elements in the same set, and hence there are $\sigma = N/(\alpha\mu)$ sets.

We call a formula $A$ a *memory construct* if during the computation of $y = Ax$, elements of $x$ are loaded once and never stored, and elements of $y$ are never loaded and stored once. Obvious memory constructs are

$$A_n \ (n \le N), \quad P \otimes I_\mu \ (P \text{ a permutation}), \quad D_n. \tag{24}$$

The first has a sufficiently small working set. The second loads or stores complete packets. The last, diagonal scaling, poses no problems.

One problematic construct in (15) has the form $A_m \otimes I_n$. As Table II shows, the loop body accesses data at stride $n$ yielding poor spatial locality unless $m \le \alpha$, which is very restrictive. However, at the expense of some overhead, this condition can be relaxed to $m \le N/\mu = \sigma\alpha$ through *buffering*. It is done by first tiling the loop by $\mu$ and then copying the working set for the innermost loop into contiguous memory. The tiled loop corresponds to the formula

$$A_m \otimes I_n = (I_{n/\mu} \otimes (A_m \otimes I_\mu))^{(L^{mn/\mu}_{n/\mu} \otimes I_\mu)}. \tag{25}$$

Buffering means that in the above formula the conjugation is implemented using actual copy operations (in contrast to translating them into re-indexing) based on the third-last entry in Table II. The resulting pseudo code snippet is shown below. On the Cell the copy operations are translated into DMA instructions instead.

```
double x[m*n], y[m*n];
for (j=0;j<n/mu;j++)
{  // allocate buffers
   double u[m*mu], v[m*mu];
   // copy into buffer
   for (k=0;k<m;k++)
      u[k*mu:1:k*(mu+1)-1] =
         x[j*mu+k*n:1:j*(mu+1)-1+k*n];
   // compute A on buffered contiguous data
   for (i=0;i<mu;i++)
      v[i:mu:i+m*mu-mu] = A(u[i:mu:i+m*mu-mu]);
   // copy data back
   for (k=0;k<m;k++)
      y[j*mu+k*n:1:j*(mu+1)-1+k*n] =
         v[k*mu:1:k*(mu+1)-1];
}
```

The other problematic construct in (15) has the form $(I_m \otimes A_n)L^{mn}_m$ and can be handled similarly: the loop is again tiled but only the load-side has a strided access and needs to be buffered. Formally,

$$(I_m \otimes A_n)L^{mn}_m = \big(I_{m/\mu} \otimes L^{n\mu}_\mu (A_n \otimes I_\mu)\big)(L^{mn/\mu}_{m/\mu} \otimes I_\mu) \tag{26}$$

expresses the tiling and again the permutation $L^{mn/\mu}_{m/\mu} \otimes I_\mu$ is implemented using explicit copy operations.

**Memory hierarchy optimizations.** To obtain FFT algorithms suitable for the memory hierarchy (i.e., the algorithm is a memory construct), we start with (15):

$$\mathrm{DFT}_{mn} = (\mathrm{DFT}_m \otimes I_n) T^{mn}_n (I_m \otimes \mathrm{DFT}_n) L^{mn}_m. \tag{27}$$

$\mathrm{DFT}_m \otimes I_n$ is a memory construct for $m \le \alpha$ or $m \le \sigma\alpha$ if buffering is applied. This explains why relatively small values of $m$ (the radix) work well in practice. $I_m \otimes \mathrm{DFT}_n$ is a memory construct for $n \le N$. If $n > N$, recursive application of (27) will eventually yield $n \le N$, producing another memory construct $\mathrm{DFT}_m \otimes I_n$ at each recursion step along the way. This suggests that the largest possible radix $m$ is a good choice in each step. We show a two-level recursion for further discussion:

$$\begin{aligned}\mathrm{DFT}_{kmn} &= (\mathrm{DFT}_k \otimes I_{mn}) T^{kmn}_{mn} \\ &\big(I_k \otimes (\mathrm{DFT}_m \otimes I_n) T^{mn}_n (I_m \otimes \mathrm{DFT}_n) L^{mn}_m\big) L^{kmn}_k.\end{aligned} \tag{28}$$

In (27), the only non-memory construct is one $L^{mn}_m$ at every recursion level with $mn > N$. Using buffering at each step incurs too much overhead. Buffering the rightmost $L$'s in (28) jointly is not possible. One solution is to give up on spatial locality: all rightmost $L$'s are fused and merged into the first loop as explained before. A better solution is to translate it into a memory construct, which is indeed possible and done, e.g., in FFTW [6]. Namely, the entire second line in (28) is translated into the *vector recursion*

$$\begin{aligned}&\big(I_k \otimes (\mathrm{DFT}_m \otimes I_n) T^{mn}_n\big) \\ &\big(L^{km}_k \otimes I_n\big)\Big(I_m \otimes \big(I_k \otimes \mathrm{DFT}_n\big) L^{kn}_k\Big)\big(L^{mn}_m \otimes I_k\big)\end{aligned} \tag{29}$$

with $\mu \mid n, k$. This is repeated until the problematic $(I_k \otimes \mathrm{DFT}_n)L^{kn}_k$ is small enough to be a memory construct. The formula manipulation leading to (29) manifests itself as loop splitting and loop exchange in the equivalent code [6].

Historically, the iterative triple loop algorithm (20) was used to compute FFTs on a single CPU. However, once the data set does not fit in cache, cache thrashing occurs and the performance drops drastically.

**Mapping to C code.** The structural optimization ensures that the algorithm has good cache locality. When mapping to code, the following additional issues have to be addressed: 1) how to create efficient basic blocks, 2) how to exploit degrees of freedom, and 3) how to handle constants (twiddle factors).

On modern deeply pipelined superscalar processors, the recursive FFT has to be terminated with a basic block that is sufficiently large but does not cause instruction cache misses. Experiments show that a DFT of a size between $4\nu$ and $32\nu$ ($\nu$ is the SIMD vector length) is a good choice. The basic block is obtained by unrolling an FFT with minimal operations count and performing scalar replacement to enable efficient register allocation and scheduling. Additional small gains may be achievable by C code scheduling, reduction of the needed constants, and a few other techniques. This process was automated in [7], [23]. The downside is considerably increased code size. For example, FFTW [6] requires several megabytes of C code to implement 1D FFTs based on (15).

The FFTs (22), (23), (25), (26), (29) contain degrees of freedom (mainly the respective radix, if and where to buffer, and when to terminate with a basic block) that can be searched over for further platform adaptation. Dynamic programming has been proven efficient in most cases [6], [3].

Finally, the twiddle constants are usually precomputed except for possibly very large sizes for which the FFT becomes memory bound. In this case computation on the fly can yield considerable improvements. The decision is again handled by search in FFTW and Spiral-generated libraries [4].

## IV. Mapping FFTs to GPUs and FPGAs

On early machines, large programs with complex loop structures were expensive, and memory access was relatively cheap. Multiple passes through the entire data set were acceptable while recursive functions were hard to implement and expensive. Thus, iterative FFT algorithms like (20)–(21) were the best choice and were developed first. Current machines with streaming memory (GPUs) or small memories (DSP processors or embedded processors) operate in a similar trade-off spot. On FPGAs, the simple loop structure of iterative algorithms makes them the preferred choice. For example, the regular Pease FFT and variants of the iterative FFT are good choices for latency and throughput optimized FPGA implementations, respectively [20], [11], [38]. In the following we very briefly discuss GPUs and provide references for more details.

Since the advent of programmable pixel shaders, general purpose programming on GPUs became an increasingly viable option. Earlier GPUs like Nvidia's G79 series were a first step towards that goal. However, with the Nvidia G80 series GPUs have become truly programmable. While they are still optimized for graphics-like workloads, mapping non-graphics applications with similar structure can result in astonishing performance including for the DFT. The caveat is that the performance is often only achievable for data resident in GPU memory; data transfer between GPU and host CPU is still very expensive and may nullify any speed-up obtained through the GPU's high raw performance (we discuss this issue to greater detail in Section VI-C). Intel's upcoming Larrabee platform may improve this situation.

**Machine Model.** State-of-the-art GPUs like the Nvidia G80 series applies ideas from symmetrically multithreaded (e.g., Tera MTA) and vector computers (e.g., Cray T90) to achieve high streaming performance [39]. In addition, minimal control flow, small computational kernels, and spatial locality are a requirement to achieve high performance.

**GPU FFT algorithms.** The above analysis suggests the Stockham FFT algorithm (21), originally developed for vector computers. Indeed, most FFT implementations developed for GPUs [40], [41], [42], [10] are based on (21), and the radix is chosen to match the GPU's hardware parameters.

**Mapping to code.** In early GPU computing the computation had to be mapped to pixel shaders using graphics languages like OpenGL and DirectX. The language Cg was a first step towards more general-purpose shaders that could be programmed in a C-like language.

With the G80 series, Nvidia introduced CUDA [43] which makes it possible to run more general compute-intensive algorithms on GPUs. The program has to be broken into host code (implemented in C with CUDA library calls) and CUDA kernels; these kernels are programmed in a special C dialect. The kernels are run in a data-parallel SPMD fashion on a grid; a sufficiently large grid and homogeneous kernel code allows for high throughput performance. It is paramount to structure the data access pattern such that kernels that are grid neighbours operate on contiguous data (spatial locality), to enable coalesced memory access. CUDA kernels are compiled to a platform-independent byte code that the CUDA driver translates into actual GPU code on its first invocation, introducing a significant overhead.

OpenCL [44] is an emerging open standard for parallel programming of heterogeneous systems; one of its targets is GPU computing and partitioning of computation across CPUs and GPUs.

## V. Computer Generation of Libraries

The framework presented in this paper concisely describes FFTs and enables structural optimization to efficiently match algorithms to multicore platforms based on a few crucial parameters. The formal nature of the approach has another major advantage: it serves as a blueprint for the computer generation of transform libraries. We have demonstrated this with Spiral, a system that generates high-performance libraries for linear transforms including the DFT. Given only textbook algorithms (as in Section II), Spiral generates multithreaded, vectorized source code. In Spiral, the matrix formalism is used as domain-specific declarative language called SPL, on which structural optimizations are performed by rewriting systems.[1] The source code level optimizations are performed by Spiral's backend compiler (an extension of [23]). Spiral can generate code for transforms of fixed input size [3], [5], [37], [45] or general input size transform libraries [28], [4] that are similar to FFTW.

Major advantages of library generation include the efficient handling of the implementation complexity and the ability to quickly port to new platforms. The complexity is due to the combination of many non-trivial transformations including those in this paper, the need for further code level optimizations (such as the unrolling of small kernels as was briefly discussed), the degrees of freedom in these optimizations (such as the choice of radix or the use or not use of buffering), the need for specialization (e.g., for small code sizes, FFTW provides many variants), the number of transforms needed (the DFT variants discussed in Section II and other transforms), and the set of available algorithms.

Problems with porting include different programming models (vector instructions, explicit DMA, OpenMP) and the difficulty of maintaining performance, both exacerbated by the fast evolution of platforms.

A library generator greatly alleviates these problems, and, as shown in the next section, often without losses and sometimes even gains in performance. For example, an increasing number of transform routines in Intel's IPP (starting with 6.0) are generated by Spiral, the main reason being superior performance.

[1] To perform all necessary optimizations, Spiral uses in addition an extension called $\Sigma$-SPL [25], [28] not described here due to lack of space.

## VI. Benchmarks on Multicore CPUs

In this section we show FFT benchmarks of the fastest libraries on multicores that are state-of-the-art at the time of this writing. The focus is on an Intel Core quadcore system and the IBM Cell BE with 9 cores, but we also include results for the Nvidia GPU GTX280 with 240 cores. For the Core and the Cell we consider Spiral generated libraries, which implement the exact techniques discussed and the similar FFTW 3.2.[2] For the Core we also show Intel's IPP 6.0 [22] and for the Cell FFTC [9]. For the GPU, we extracted the results from [10].

The performance for input size $n$ is computed as $5n\log_2(n)/t$, where $t$ is the runtime in seconds. This is a slight overestimate since the true flop count is closer to $4n\log_2(n)$ and depends on the exact algorithm and recursion strategy chosen.

### A. Intel Multicore

**Platform.** We consider a 2.66 GHz Intel Core i7 quadcore processor (Nehalem microarchitecture, 45 nm process) with SSE 4.2 instruction set. It has three levels of cache and 25.6 GB/s memory bandwidth (using all three on-chip memory controllers). Each core supports hyperthreading but for programs with high arithmetic density (like FFTs), SMT does not provide any benefit, so in Spiral we use no more than 4 threads using OpenMP.

The Core i7 implements SSE 4.2, providing 2-way double precision and 4-way single precision floating point vector support. Moreover, it supports complex arithmetic operations packing one complex double-precision number or 2 complex single-precision numbers into vector registers. While unaligned memory access is supported, the best performance is achieved with 16-byte aligned vector loads and stores.

The Core i7 implements the shared memory paradigm. Each core has a private 64 kB L1 cache and 256 kB L2. The 8 MB L3 cache is shared among cores. All caches have 64 byte cache lines (4 complex double precision numbers or 8 complex single-precision numbers).

The theoretical peak performance is 85.12 Gflop/s for single and 42.56 Gflop/s for double precision.

**Results.** Figs. 5(a) and (b) show results for out-of-place double-precision and single-precision 2-power FFTs. FFTW and the Spiral generated library [4] are compiled with the Intel C++ compiler 11.0 and flags "-O3 -xS"; IPP is provided as binary. The measurements are with "warm" cache.

For the Spiral generated libraries, the working set for input size $n$ is $6n$ real numbers or $4\frac{1}{16}n$ real numbers if the twiddle factors are computed on the fly. Figs. 5(a) and (b) indicate maximal cache resident sizes. For example, in Fig. 5(a), an FFT is L3 cache resident up to $n = 2^{17}$.

Overall in Figs. 5(a) and (b), Spiral is about equal and often faster than the hand-written libraries. For small sizes, the performance reaches up to 10 Gflop/s (double precision) or 14 Gflop/s (single precision); In double precision a slight drop occurs for the first size (256) that is not completely unrolled. This is due to the occurring index expressions that can be fully precomputed and inlined only if the code is unrolled. The first speed-up through threading occurs already for a working set that fits into the L1 cache of one core. Subsequently, the performance ramps up as four cores are used on L1, L2, or L3 cache resident data yielding about 3x speedup over 1 thread. The performance peaks at 15 Gflop/s (double precision) and 35 Gflop/s (single precision) for vectorized code running on all 4 cores. The drop for single precision and input size 16K may be an artifact of imperfect search. For sizes outside the shared L3 cache, the performance drops as the computation becomes memory bound. At this point buffering, vector recursion, and on-the-fly twiddle computation become crucial.

Since the FFTs used by the Spiral-generated library are already adapted as explained in this paper, even a random choice of recursion will yield reasonable performance (within 2x say). The detailed shape of the best recursion for a given DFT size on a given multicore CPU is difficult to predict. However, we made the following observations.

The multicore Cooley-Tukey FFT (22) is used as top-level algorithm sizes that are large enough to benefit from parallelization. Typically, $m = \mu$ in (22) leads to a yields good performance. Further, the short vector Cooley-Tukey FFT (23) provides perfect SIMD vectorization. On the Core i7 $n = \nu$ is a good choice.

For cache-resident sizes, the standard Cooley-Tukey FFT (15) is a good choice with $k$ small enough such that $\mathrm{DFT}_k$ can be implemented fully unrolled, and the machine has enough registers to support the computation. In practice, $8\nu \le k \le 32\nu$ (CPU-dependent) is a good choice. Once the working set no longer fits into the last cache level (or into the local store on the Cell) the involved trade-offs become tricky. In addition to (15), the vector recursion (29) and buffering (25)–(26) become fastest. On the Core i7 for large enough $\mathrm{DFT}_{kmn}$, a typical out-of-cache decomposition applies the vector recursion (29) with $8 \le k, m \le 32$ until the working set fits into the last cache level. In addition, $\mathrm{DFT}_k \otimes I_{mn}$ and $\mathrm{DFT}_m \otimes I_n$ are buffered and the twiddle factors computed on the fly. This coincides with our abstract analysis in Section III-C.

### B. Cell BE

**Platform.** We consider a 3.2 GHz Cell BE with 9 cores, including one traditional PowerPC core and 8 SIMD vector cores (called synergistic processing elements, or SPEs). Each SPE includes its own fast on-chip 256 KB local memory (local store) that is designed to be explicitly managed by the programmer. This means inter-core and main memory-local store transfers must be performed via DMA; the achieved DMA bandwidth increases with DMA packet size. The Cell includes a set of 4-way single precision SIMD instructions for the SPEs, accessible via C intrinsics. The vectorization, however, is very similar to Intel's SSE. The peak performance of the Cell is 204.8 Gflop/s single precision (SPEs only) and 14.4 Gflop/s double precision.

**Results.** Fig. 5(c) shows the (latency) performance of Spiral generated code (separate functions for every size [45] in this case), FFTC [9], and FFTW 3.2, each using the interleaved-complex data format. Spiral-generated code is compiled with spu-gcc (flags: "-O2"), the other data is extracted from the respective papers. In addition we include the $2^{24}$-sized DFT from [46] and the $2^{16}$-sized DFT from [47] (both use split-

[2]FFTW implements the recursive Cooley-Tukey FFT, buffering, vector recursion, and SIMD vectorization, using algorithms similar but not equal to (22) and (23).

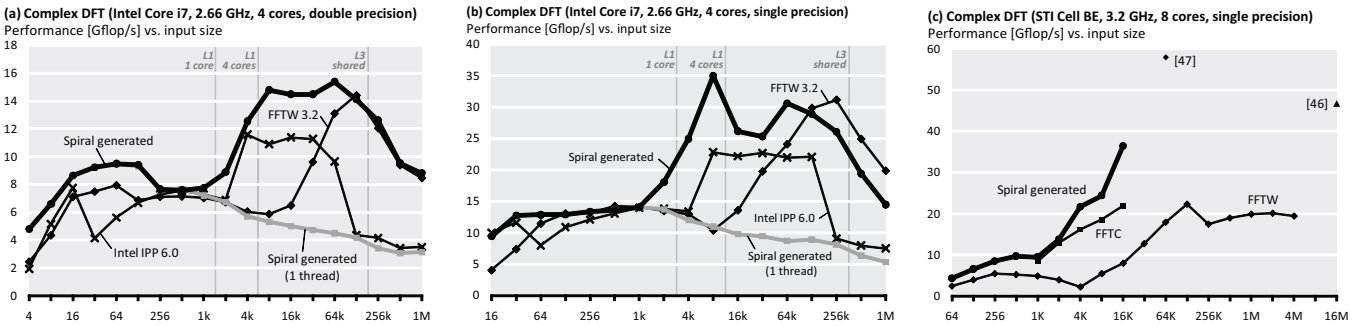



Fig. 5. DFT performance on the 2.66 GHz Intel Core i7 (a), (b) and the 3.2 GHz Cell BE (c). Higher is better.

complex data format). The latter has been estimated to achieve a throughput of 116 Gflop/s.

Spiral-generated code to date is limited to sizes for which the working set fits into the union of all local stores; the same seems to hold for FFTC. Both perform better than FFTW for these sizes. The excellent performance of both [47] and [46] is due to a highly optimized $2^8$-sized kernel.

All loads and stores from main memory, and all inter-core communication (permutations in (22)) are performed explicitly using DMA instructions. We use the Cell's inter-core messaging mechanisms for synchronization barriers.

The Cell allows initiated DMA instructions to proceed in the background along with active computation. Although not currently used in our code in Fig. 5(c), large out-of-chip DFT sizes can use a multibuffering technique based on (25) to (partially) hide memory costs. Data can be stored and loaded for the previous and next iterations in separate buffers while computation progresses for the current iteration. The explicit move operations in (25) would become DMA instructions.

For parallel code, in contrast to the Core i7, the best $m, n$ in (22) found are both close to $\sqrt{mn}$, since this maximizes the packet size $\mu$. The remaining choices found are similar to the Core i7; inside (22) and (23), a (15) with $m \approx 128$ is chosen.

Other results include [48], who achieve about 22 Gflop/s on a single SPE for DFTs of input sizes $2^{10}$ and $2^{13}$ resident in the SPE's local store. [49] implement 2D and 3D parallel SPE-resident FFT kernels achieving up to 30 Gflop/s.

### C. GPU

**Platform.** We consider the Nvidia 280GTX with 240 cores grouped into 30 multiprocessors, 1 GB of on-GPU main memory and a GPU memory bandwidth of 140 GB/s. The shader clock is 1.3 GHz, and each core can perform 1 fused multiply-add and a multiply operation per cycle, leading to 936 Gflop/s peak performance. The connection between CPU memory and GPU uses PCIe 2.0, which for 16 lanes has a bandwidth of 16 GB/s (8 in each direction). The theoretical (single precision) peak performance is 936 Gflop/s.

**Results.** The runtime results in Fig. 6 are taken from [10], and appear to be the fastest at the time of this writing. The memory configuration limits the achievable floating-point performance for FFTs to 43.75 $\log_2 n$ Gflop/s out of GPU memory (obtained from 140 GB/s GPU memory bandwidth), and to 5 $\log_2 n$ Gflop/s out of host memory (obtained from 16 GB/s PCIe bandwidth). These bounds are included as gray lines in Fig 6. The plot shows latency (one DFT is performed)

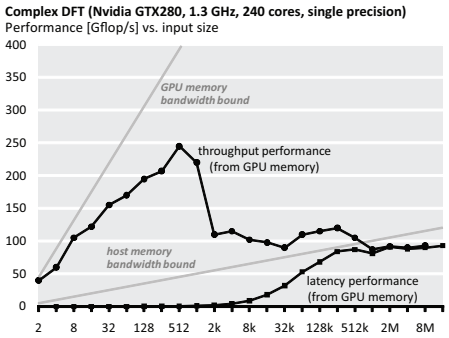



Fig. 6. DFT performance on a Nvidia GTX 280 GPU. Higher is better.

and throughput ($2^{23}/n$ DFTs of size $n$ are performed in batch mode), both with GPU memory resident data.

Throughput performance plays well to the strength of the GPU and ramps up almost parallel to the memory bandwidth bound until $n = 512$; then cache limitations cause the performance to drop to about 100 Gflop/s. We note that it is possible (not specified in [10]) that the batch mode computes $\mathrm{DFT}_n \otimes I$ (interleaved DFTs) rather than $I \otimes \mathrm{DFT}_n$. Not surprisingly, latency performance in contrast can amortize memory latency only for large sizes.

It seems attractive to transparently utilize the GPU as accelerator for batched single precision DFTs in CPU computations. The problem is the PCIe bandwidth between host memory and GPU memory, which yields the lower gray line as performance bound, and realistically probably half of that. Consequently, only very large sizes would benefit. However, if the entire application can be implemented on the GPU, the full performance can be harnessed, and can yield for the DFT up to an eightfold performance improvement compared to a CPU and up to fivefold compared to the cell.

### D. FPGA

While not directly in the scope of this paper, we mention for comparison that the FFT in [38], [13] achieves up to 40 Gflop/s throughput performance on a Virtex-4 (V4-FX140) for a $\mathrm{DFT}_{256}$ in single precision floating point using all logic available (more data is not readily available). On Virtex-6 twice the performance is possible using twice the resources. This performance requires the data to be on-chip; the off-chip bandwidth is typically about 10 GB/s in each direction. Note that usually fixed point is used on FPGAs and that most applications leave only a small part of the available logic for FFTs. In summary, the main appeal of FPGAs for DFTs are in saving power rather than as mere accelerator.

For commercial state-of-the-art FFTs on FPGAs see, e.g., [50], [51].

## VII. Conclusion

The end of CPU frequency scaling and advent of multicore systems has two major consequences for compute intensive signal processing applications. First, it marks the end of free speed-up for legacy software. Second, the software development skill required to achieve optimal performance is dramatically increased. As we have shown for the DFT, minimizing operations count alone does not yield optimal or even close-to-optimal performance. Instead, the structure of algorithms becomes crucial and has to be matched to the target architecture. Specifically, the check list for high performance is efficient parallelization, vectorization, and memory hierarchy optimization. The necessary transformations are likely to stay out of reach for compilers since they require domain knowledge and the ability to assess the many available choices. To handle the implementation complexity we believe it is important to develop rigorous approaches that formalize algorithmic optimizations by connecting the algorithm structure with architecture parameters. We have presented such a framework for the DFT and used it to give an overview on FFTs and optimizations for current multicores. Further, as we demonstrated with Spiral, the rigorous nature of the framework enables automation: the computer generation of DFT libraries that often achieve excellent performance compared to their hand-written counterparts.

## References


[1] G. Blake, R. G. Dreslinski, and T. Mudge, "A survey of multicore architectures," *IEEE Signal processing Magazine*, 2009.

[2] W. H. Press, B. P. Flannery, Teukolsky S. A., and Vetterling W. T., *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge University Press, 2nd edition, 1992.

[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] Y. Voronenko, F. de Mesmay, and M. Püschel, "Computer generation of general size linear transform libraries," in *Proc. Code Generation and Optimization (CGO)*, 2009, pp. 102–113.

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

[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] M. Frigo, "A fast Fourier transform compiler," in *Proc. Programming Language Design and Implementation (PLDI)*, 1999.

[8] "FFTW 3.2," `www.fftw.org`.

[9] David A. Bader and Virat Agarwal, "FFTC: Fastest Fourier transform for the IBM Cell Broadband Engine," in *Proc. Intl. Conference on High Performance Computing (HiPC)*, 2007, pp. 172–184.

[10] Naga K. Govindaraju, Brandon Lloyd, Yuri Dotsenko, Burton Smith, and John Manferdelli, "High performance discrete fourier transforms on graphics processors," in *Proc. Supercomputing (SC)*, 2008, pp. 1–12.

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

[12] R. Tolimieri, M. An, and C. Lu, *Algorithms for Discrete Fourier Transforms and Convolution*, Springer, 2nd edition, 1997.

[13] "Spiral web site," `www.spiral.net`.

[14] J. Johnson, R. W. Johnson, D. Rodriguez, and R. Tolimieri, "A methodology for designing, modifying, and implementing Fourier transform algorithms on various architectures," *IEEE Trans. Circuits and Systems*, vol. 9, pp. 449–500, 1990.

[15] A. Norton and A. J. Silberger, "Parallelization and performance analysis of the Cooley-Tukey FFT algorithm for shared-memory architectures," *IEEE Trans. Comput.*, vol. 36, no. 5, pp. 581–591, 1987.

[16] M. Hegland, "Block algorithms for FFTs on vector and parallel computer," in *Parallel Computing: Trends and Applications*, pp. 129–136. 1994.

[17] D. H. Bailey, "FFTs in external or hierarchical memory," *J. Supercomputing*, vol. 4, pp. 23–35, 1990.

[18] D. B. Harris, J. H. Mc Clellan, D. S. K. Chan, and H. W. Schuessler, "Vector radix fast fourier transform," in *Proc. Int. Conf. on Acoustics, Speech, and Signal Processing (ICASSP)*, 1977, pp. 548–551.

[19] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Math. of Computation*, vol. 19, pp. 297–301, 1965.

[20] M. C. Pease, "An adaptation of the fast Fourier transform for parallel processing," *Journal of the ACM*, vol. 15, no. 2, April 1968.

[21] Paul N. Schwarztrauber, "Multiprocessor FFTs," *Parallel Computing*, vol. 5, pp. 197–210, 1987.

[22] Website, "Intel integrated performance primitives (ipp) 6.0," `software.intel.com/en-us/intel-ipp`.

[23] J. Xiong, J. Johnson, R. Johnson, and D. Padua, "SPL: A language and compiler for DSP algorithms," in *Proc. Programming Language Design and Implementation (PLDI)*, 2001, pp. 298–308.

[24] I. W. Selesnick and C. S. Burrus, "Automatic generation of prime length FFT programs," *IEEE Trans. Signal Processing*, vol. 44, pp. 14–24, 1996.

[25] F. Franchetti, Y. Voronenko, and M. Püschel, "Loop merging for signal transforms," in *Proc. Programming Language Design and Implementation (PLDI)*, 2005, pp. 315–326.

[26] D. Mirković and S. L. Johnsson, "Automatic performance tuning in the UHFFT library," in *Proc. Int'l Conf. Computational Science (ICCS)*. 2001, vol. 2073 of *LNCS*, pp. 71–80, Springer.

[27] D. Takahashi, "An implementation of parallel 1-D FFT using SSE3 instructions on dual-core processors," in *Proc. Int'l Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA)*, 2006, pp. 1178–1187.

[28] Y. Voronenko, *Library Generation for Linear Transforms*, Ph.D. thesis, Electrical and Computer Engineering, Carnegie Mellon University, 2008.

[29] Srinivas Chellappa, Franz Franchetti, and Markus Püschel, "How to write fast numerical code: A small introduction," in *Lecture Notes in Computer Science*. 2008, vol. 5235, pp. 196–259, Springer.

[30] J. R. Johnson, R. W. Johnson, D. Rodriguez, and R. Tolimieri, "A methodology for designing, modifying, and implementing Fourier transform algorithms on various architectures," *IEEE Trans. Circuits, Systems, and Signal Processing*, vol. 9, no. 4, pp. 449–500, 1990.

[31] A. Ali, L. Johnsson, and J. Subhlok, "Scheduling FFT computation on SMP and multicore systems," in *Proc. Int'l Conf. Supercomputing (ICS)*, 2007.

[32] OpenMP, *OpenMP C and C++ Application Pragram Interface, Version 1.0*, 1998, `www.openmp.org`.

[33] Bill Gallmeister, *POSIX.4*, O'Reilly, 1994.

[34] Franz Franchetti and Markus Püschel, "Generating SIMD vectorized permutations," in *Proc. Int'l Conf. Compiler Construction (CC)*. 2008, vol. 4959 of *Lecture Notes in Computer Science*, pp. 116–131, Springer.

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

[36] F. Franchetti and M Püschel, "A SIMD vectorizing compiler for digital signal processing algorithms," in *Proc. IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS)*, 2002, pp. 20–26.

[37] F. Franchetti, Y. Voronenko, and M. Püschel, "A rewriting system for the vectorization of signal transforms," in *Proc. High Performance Computing for Computational Science (VECPAR)*, 2006.

[38] P. A. Milder, F. Franchetti, J. C. Hoe, and M. Püschel, "Formal datapath representation and manipulation for implementing DSP transforms," in *Proc. Design Automation Conference (DAC)*, 2008, pp. 385–390.

[39] Allan Snavely, Larry Carter, Jay Boisseau, Amit Majumdar, Kang Su Gatlin, Nick Mitchell, John Feo, and Brian Koblenz, "Multi-processor performance on the Tera MTA," in *Proc. Supercomputing (SC)*, 1998, pp. 1–8.

[40] Kenneth Moreland and Edward Angel, "The FFT on a GPU," in *Proc. ACM SIGGRAPH/EUROGRAPHICS Conf. on Graphics hardware*, 2003, pp. 112–119.

[41] Naga K. Govindaraju and Dinesh Manocha, "Cache-efficient numerical algorithms using graphics hardware," *Parallel Comput.*, vol. 33, no. 10-11, pp. 663–684, 2007.

[42] Akira Nukada, Yasuhiko Ogata, Toshio Endo, and Satoshi Matsuoka, "Bandwidth intensive 3-d FFT kernel for GPUs using CUDA," in *Proc. Supercomputing (SC)*, 2008, pp. 1–11.

[43] "Nvidia CUDA," `www.nvidia.com/cuda`.

[44] "OpenCL," `www.khronos.org/opencl/`.

[45] S. Chellappa, F. Franchetti, and Markus Püschel, "Computer generation of fast Fourier transforms for the Cell Broadband Engine," in *Proc. Int'l Conf. Supercomputing (ICS)*, 2009.
[46] Alex C. Chow, Gordon C. Fossum, and Daniel A. Brokenshire, "A programming example: Large FFT on the Cell Broadband Engine," Tech. Rep., IBM, May 2005.
[47] Jon Greene and Robert Cooper, "A parallel 64K complex FFT algorithm for the IBM/Sony/Toshiba Cell Broadband Engine processor," in *Global Signal Processing Expo (GSPx)*, 2005.
[48] L. Cico, R. Cooper, and J. Greene, "Performance and programmability of the IBM/Sony/Toshiba Cell Broadband Engine Processor," in *Proc. (EDGE) Workshop*, 2006.
[49] Paolo Bientinesi, Nikos Pitsianis, and Xiaobai Sun, "Multi-dimensional array operations for signal processing algorithms," in *Proc. Int'l Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA)*, 2008.
[50] "4DSP," `www.4dsp.com/fft.htm`.
[51] "Dillon FFT," `www.dilloneng.com/fft_ip`.