FFT Program Generation for Shared Memory: SMP and Multicore

Franz Franchetti, Yevgen Voronenko, and Markus Püschel
Electrical and Computer Engineering
Carnegie Mellon University
{franzf, yvoronen, pueschel}@ece.cmu.edu

Abstract

The chip maker's response to the approaching end of CPU frequency scaling are multicore systems, which offer the same programming paradigm as traditional shared memory platforms but have different performance characteristics. This situation considerably increases the burden on library developers and strengthens the case for automatic performance tuning frameworks like Spiral, a program generator and optimizer for linear transforms such as the discrete Fourier transform (DFT). We present a shared memory extension of Spiral. The extension within Spiral consists of a rewriting system that manipulates the structure of transform algorithms to achieve load balancing and avoids false sharing, and of a backend to generate multithreaded code. Application to the DFT produces a novel class of algorithms suitable for multicore systems as validated by experimental results: we demonstrate a parallelization speed-up already for sizes that fit into L1 cache and compare favorably to other DFT libraries across all small and midsize DFTs and considered platforms.

CR Categories: F.2.1 [Analysis of algorithms and problem complexity]: Numerical algorithms and problems—Fast Fourier transform; D.1.3 [Programming techniques]: Concurrent programming—Shared memory; D.1.2 [Programming techniques]: Automatic programming—Program generators

Keywords: Fast Fourier transform, shared memory, multicore, chip multiprocessor, automatic parallelization

1 Introduction

After years of exponential growth, the CPU frequencies of recent generations of microprocessors have practically stalled, a consequence of the physical limits imposed by their power density. To keep Moore's Law on track, chip makers have started to follow a different route: multicore systems, also called chip multiprocessors (CMPs), that integrate multiple processor cores onto one chip. Dual core systems are currently sold by Intel, IBM, and AMD. IBM's Cell processor has eight special-purpose cores on one chip. In the future, concurrency will become mainstream and pose a major burden on compiler developers and programmers.

A mature body of work on parallelizing compilers exists, but targets mainly large applications for which moderate overhead is acceptable when they are mapped to a large number of processors. CMPs, on the other hand, offer a much better ratio of communication to computation speed, a property that changes the game, and, for example, should enable parallelization for much smaller problem sizes. In a sense this parallels the situation from a few years ago when SIMD vector instructions were introduced. Their underlying mathematical paradigm matched early vector processors, but different optimization techniques were necessary.

Programmers in charge of developing high performance libraries are already confronted with the difficult task of optimizing for deep memory hierarchies and extracting the fine-grain parallelism for vector instruction sets. Now, this challenge is compounded with multithreaded programming for a platform with new performance characteristics. This scenario strengthens the case for recent efforts on automatic performance tuning, program generation, and adaptive library frameworks that can offer high performance with greatly reduced development time. Examples include ATLAS [Whaley et al. 2001], Bebop/Sparsity [Im et al. 2004; Demmel et al. 2005], and FLAME [Gunnels et al. 2001; Bientinesi et al. 2005] for linear algebra, FFTW [Frigo and Johnson 2005] for the discrete Fourier transform (DFT), and Spiral [Püschel et al. 2005] for general linear transforms.

Contribution. In this paper we formally derive fast Fourier transform algorithms (FFTs) suitable for shared memory and, in particular, multicore platforms. The benefit of the formal approach is twofold. First, it enables us to reason about desirable properties; in particular, we can prove that the algorithms offer perfect load-balancing and avoid false sharing. Second, we implemented the framework in the form of a rewriting system as part of the Spiral program generator [Püschel et al. 2005] compatible with Spiral's for-

mal loop optimization [Franchetti et al. 2005], vectorization [Franchetti et al. 2006], and automatic tuning framework.

We evaluated the approach by generating FFT programs automatically tuned for a variety of shared memory platforms including classical SMPs and recent CMPs. The results show superior performance across a range of sizes compared to FFTW and the Intel vendor library. Further, on a CMP we demonstrate a speed-up through parallelization for a problem size as small as 2^8, which fits completely into L1 cache and runs at less than 10,000 cycles. In contrast, FFTW only takes advantage of the second processor for sizes larger than 2^{13}, running at more than 500,000 cycles.

Organization. In Section 2 we present the necessary background and related work for this paper. First, we discuss shared memory platforms and associated work on compilers. Then we explain the DFT, the Cooley-Tukey FFT, and its parallel versions derived previously. Finally, we overview the Spiral program generator. In Section 3 we formally derive parallel FFTs suitable for multicore systems and reason about their structure. Then, we explain the integration of the framework into Spiral. Section 4 presents experimental results and benchmarks with Spiral-generated FFTs for a variety of symmetric multiprocessor platforms. Finally, we offer conclusions in Section 5.

2 Background and Related Work

We provide background and discuss related work on shared memory platforms and programming, the DFT and the FFT algorithm for single and multiple processors, and the Spiral program generator.

2.1 Shared Memory Machines

Symmetric multiprocessing. In the late 1980's and early 1990's highly parallel shared-memory machines entered the scene. Smaller machines were symmetric multi processors (SMP) while larger machines used distributed shared memory (DSM), i.e., the memory was physically distributed but shared between all processors to allow programming without data transfer management. Today, some vendors still produce large shared memory machines; however, most highly parallel platforms are now clusters of SMPs with up to 4 processors per cluster node and explicit message passing between nodes.

A recent important change occurred when IBM, AMD, and Intel started producing shared memory chip-multiprocessors (CMPs): multiple CPU cores on the same piece of silicon. The integration ranges from two processors sharing no caches, such as Intel Pentium D processors, to two cores sharing a cache like Intel Core Duo and IBM Power 5 processors. AMD Opteron dualcore processors are in some sense a compromise using a fast on-chip cache coherency protocol. The first generation of multicore processors targeted servers while the latest generation targets consumer desktop and laptop computers, making multicore a mainstream technology.

Parallelizing compilers. Today's parallelizing compilers grew out of a vast body of research that started with the first shared memory machines in the late 1980's [Banerjee et al. 1993; Hiranandani et al. 1992; Wolfe 1996; Zima and Chapman 1990]. The resulting compilers are quite successful and provide good performance scaling for relative simple programs running on tightly coupled systems with more than 2 or 4 CPUs. However, they cannot achieve parallel speed-up for complicated programs targeting a small number of CPUs and small problem sizes. In particular, this applies to the DFT considered in this paper.

Generally, successful parallelization first requires optimization for the memory hierarchy, since accessing shared data is more expensive than accessing private data. Thus, the work on loop tiling, loop exchange, and loop interleaving are highly relevant to automatic parallelization and automatic performance tuning as these methods are fundamental program transformations to improve data locality in array computations. However, these transformations typically require expensive analysis [McKellar and E. G. Coffman 1969; Gatlin and Carter 1999; Wolf and Lam 1991].

Explicit programming of shared memory machines. Writing fast parallel programs is considerably more challenging than writing fast sequential programs. For problems that are data parallel (but not embarrassingly parallel) the programmer has to address the following issues: 1) *Load balancing:* All processors should have an equal amount of work assigned. In particular, sequential parts should be avoided since they limit the achievable speed-up due to Amdahl's law. 2) *Synchronization Overhead:* Synchronization should involve as little overhead as possible and threads should not wait unnecessarily at synchronization points. 3) *Avoiding false sharing:* Private data of different processors should not be in the same cache line, since this leads to cache thrashing and thus severe performance degradation. In addition, the programmer has to optimize single thread performance, and try to avoid excessive locking of shared variables, deadlocks, race conditions, and cache interference of multiple threads, among other things.

Historically, programming parallel machines was a very machine-dependent, painful, and non-portable process. In order to allow for portable multithreaded applications, thread libraries and parallel languages (or language extensions) were developed. These provide the means to start and synchronize threads, to protect shared variables, and to allocate and manage thread local data. Examples include the pthreads library and the OpenMP language extension [Chandra et al.

2000]. OpenMP extends C (or Fortran) by directives inlined into the source code as pragmas (C) or comments (Fortran) to pass parallelization information to the compiler and also includes a supporting runtime library.

Despite all efforts, however, producing portable, fast, stable, high-quality parallel software for shared memory machines is still a major challenge.

2.2 Discrete Fourier Transform

The discrete Fourier transform (DFT) of an input vector x of length n is defined as the matrix-vector product

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

Fast algorithms compute the DFT in $O(n \log n)$ operations and are referred to as fast Fourier transforms (FFTs). They can be described as recursive factorizations of the matrix DFT_n into structured sparse matrices using the Kronecker product formalism [Van Loan 1992]. In particular, the well-known Cooley-Tukey FFT can be written as (we write $\rightarrow$ instead of $=$ since later in Spiral we view these decompositions as rules)

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

In (1), I_k is the $k \times k$ identity matrix and $D_{m,n}$ a diagonal "twiddle factors" matrix. Particularly important is the tensor (or Kronecker) product of matrices,

$$A \otimes B = [a_{i,j}B]_{i,j} \quad \text{with} \quad A = [a_{i,j}]_{i,j}\,.$$

For example,

$$I_n \otimes A = \begin{bmatrix} A & & & \\ & A & & \\ & & \ddots & \\ & & & A \end{bmatrix}.$$

The iterative direct sum

$$\bigoplus_{i=0}^{n-1} A_i \quad \text{with} \quad A_i \in \mathbb{C}^{m\times m}$$

generalizes $I_n \otimes A$ to matrices A_i that depend on the iteration but are all of the same size $m \times m$. The stride permutation matrix L^{mn}_m permutes an input vector x of length mn as

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

If x is viewed as an $n \times m$ matrix, stored in row-major order, then L^{mn}_m performs a transposition of this matrix.

Actual DFT algorithms are obtained by applying the FFT (1) recursively to the subproblems DFT_m and DFT_n until the base case DFT_2 is reached. For instance one can factor $8 \rightarrow 2 \times 4 \rightarrow 2 \times (2 \times 2)$ using two recursive applications of (1). The complete FFT algorithm for this factorization can then be written as the following *formula*:

$$\mathrm{DFT}_8 = (\mathrm{DFT}_2 \otimes I_4) D_{8,4} \big(I_2 \otimes (\mathrm{DFT}_2 \otimes I_2)\, D_{4,2} (I_2 \otimes \mathrm{DFT}_2) L^4_2\big) L^8_2. \quad (2)$$

Programs implementing $y = Ax$ for some recursive formulas A are shown in Table 1. By applying these translation rules recursively to subexpressions of A one can translate complicated formula expressions like (2) into programs [Xiong et al. 2001]. This is the basic idea in Spiral (explained below); the formula language is called SPL (signal processing language).

SPL construct	code
$y = (A_n B_n)x$	`t[0:1:n-1] = B(x[0:1:n-1]);` `y[0:1:n-1] = A(t[0:1:n-1];)`

$y = (I_m \otimes A_n)x$	`for (i=0;i<m;i++)`
	`  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<m;i++)`
	`  y[i:n:i+m-1] =`
	`  A(x[i:n:i+m-1]);`

$y = \left(\bigoplus_{i=0}^{m-1} A_n^i\right) x$	`for (i=0;i<m;i++)`
	`  y[i*n:1:i*n+n-1] =`
	`  A(i, x[i*n:1:i*n+n-1]);`

$y = D_{m,n}x$	`for (i=0;i<m*n;i++)` `  y[i] = Dmn[i]*x[i];`

$y = L^{mn}_m x$	`for (i=0;i<m;i++)`
	`  for (j=0;j<n;j++)`
	`  y[i+m*j]=x[n*i+j];`

Table 1: Translating SPL constructs to code. x denotes the input and y the output vector. The subscript of A and B specifies the size of the (square) matrix. We use Matlab-like notation: `x[b:s:e]` denotes the subvector of x starting at `b`, ending at `e` and extracted at stride `s`.

Observe in Table 1 that the multiplication of a vector by a tensor product containing identity matrices can be computed using loops. The working set for each of the m iterations of $y = (I_m \otimes A_n)x$ is a contiguous block of size n and the base address is increased by n between iterations. In contrast, the working sets of size m of the n iterations of $y = (A_m \otimes I_n)x$ are interleaved, leading to stride n within one iteration and a unit stride base update across iterations. The iterations in both loops have no loop carried dependencies and thus can easily be parallelized on shared memory machines.

The SPL framework can be used to express a large class of linear transforms and its algorithms [Püschel et al. 2005] including multi-dimensional transforms, which are just tensor products of their one-dimensional counterparts.

Shared memory FFT algorithms. Early work by [Johnson et al. 1990] shows how to design parallel DFT algorithms for various architecture constraints using the Kronecker product formalism and is a major influence on our work. [Van Loan 1992] gives a good overview of sequential and parallel DFT algorithms. The major problem with using the standard Cooley-Tukey FFT algorithm (1) on shared memory machines is its memory access pattern: Large strides, and consecutive loop iterations touch the same cache lines, which leads to false sharing.

The governing idea of many parallel algorithms [Norton and Silberger 1987; Schwarztrauber 1987; Bailey 1990] is to reorder the data in explicit steps to remove false sharing introduced by strided memory access. For example, the six-step algorithm,

$$\mathrm{DFT}_{mn} \rightarrow L^{mn}_m (I_n \otimes \mathrm{DFT}_m) L^{mn}_n \\ D_{m,n} (I_m \otimes \mathrm{DFT}_n) L^{mn}_m \quad (3)$$

has embarrassingly parallel computation stages of the form $I_r \otimes \mathrm{DFT}_s$. The three stride permutations in (3) are executed separately as explicit matrix transpositions, i.e., data permutations. These transpositions are further optimized, e.g., through blocking [Al Na'mneh et al. 2005a], and partially folded into the adjacent computation stages [Takahashi 2002; Takahashi et al. 2003]. A different optimization approach reduces communication by increasing the computation by using $O(n^2)$ algorithms instead of fast $O(n \log n)$ algorithms to remove dependencies on small subproblems [Al Na'mneh et al. 2005b].

FFTW 3.1 [Frigo and Johnson 2005] offers a state-of-the-art multithreading DFT implementation, supporting many multithreaded programming interfaces across many operating systems. It parallelizes one- and multidimensional DFTs by allowing its search mechanism to parallelize many different loops that occur inside the algorithms. It uses advanced loop optimization to avoid cache problems (tiling and loop exchange) and it supports thread pooling to minimize the startup cost of parallel computation. (Thread pooling is experimental and turned off by default.) However, the infrastructure required for portability across machines and support for all problem sizes incurs considerable overhead. Because of this overhead, the authors of FFTW maintain that it may make sense to use multiple threads within FFTW only for problem sizes beyond several thousand data points.

2.3 Spiral

Spiral [Püschel et al. 2005] is an automatic program generation and optimization system for linear digital signal processing (DSP) transforms. Spiral's internal structure is shown in Figure 1. The user formally specifies a DSP transform to be implemented as input to Spiral, e.g., $\mathrm{DFT}_{2^{10}}$. Spiral's output is a C program that computes the specified transform and

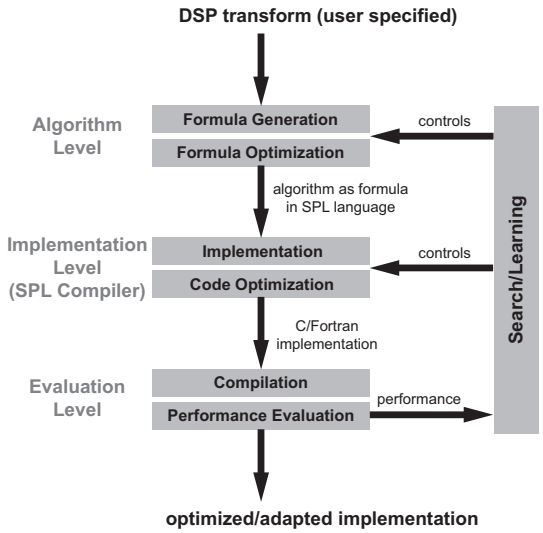


Figure 1: SPIRAL's architecture.

that is optimized to the platform Spiral is installed on. We briefly describe the generation process next.

Algorithm level. In the *formula generation* Spiral applies rules such as (1) to the given transform to generate one of many possible formulas, such as (2), represented in SPL. In the *formula optimization* stage Spiral optimizes the structure of the generated formula using a high-level approach to loop merging and index simplification [Franchetti et al. 2005]. It merges loops originating from tensor products with loops originating from permutations and diagonal matrices, reducing (1) to a sequence of two loops. Both formula generation and optimization are implemented through rewriting systems [Dershowitz and Plaisted 2001].

On platforms with vector instructions, Spiral takes the vector length into account for formula generation and optimization. The resulting formulas have a structure that maps directly into efficient vector code [Franchetti and Püschel 2002; Franchetti and Püschel 2003; Franchetti et al. 2006]. In this paper, we extend this approach to produce efficient formulas, and thus code, for shared memory platforms with focus on SMPs and multicore systems.

Implementation level. In the *implementation* stage, Spiral uses an extended version of Table 1 to translate the preoptimized formula into C code. For vector code and parallel code (this paper), the C programs include constructs like vector intrinsic functions or OpenMP parallel loops. In the *code optimization* stage, the obtained code is further optimized using standard compiler techniques including strength reduction and constant folding [Xiong et al. 2001; Franchetti et al. 2005].

Evaluation level. The final program is compiled using a standard C compiler in the *compilation* stage and the actual runtime is measured in the *performance evaluation* stage.

Search/learning. Besides the deterministic optimizations performed on the formula and the C code, Spiral optimizes for the target platform (in particular its memory hierarchy) through heuristic search in the formula space, such as dynamic programming or an evolutionary algorithm [Singer and Veloso 2001]. This search is controlled by the *search/learning* block based on the runtime of the previously generated implementations.

3 Parallel FFTs Through Formula Rewriting

Our goal is to extend Spiral (Section 2.3) to generate efficient transform code, in particular DFT code, for shared memory platforms including multicore systems. The approach is similar to the approach we took to generate vector code [Franchetti and Püschel 2002; Franchetti et al. 2006]. It is based on the observation that the formulas produced by Spiral have a direct interpretation in terms of parallel code. For example, the tensor products in (1) are essentially loops with fully independent iterations (no loop-carried dependencies) and known memory access patterns. Permutations express readdressing that Spiral will fuse into an adjacent computation loop in the formula optimization stage [Franchetti et al. 2005].

The basic idea is now to automatically rewrite a generated formula within Spiral to obtain a structure suitable for mapping into efficient multi-threaded code. This is possible, since a formula fully determines the memory access of the final program as functions of the loop variables. Therefore, using rewriting, we can statically schedule the loop iterations across p processors to ensure load balancing and minimize false sharing. For general programs, proving the independence of loop iterations and determining such a schedule is a hard problem that requires expensive analysis [Banerjee et al. 1993]. However, formula constructs like stride permutations and tensor products express very specific and regular memory access patterns and dependencies. Thus, as we show, we can find such a desired schedule very efficiently using the formula rewriting framework. Furthermore, we can prove that the solutions obtained this way do not incur false sharing.

We first explain this extension of the rewriting system in Spiral. Then we show the application to the DFT, effectively deriving a novel variant of the Cooley-Tukey FFT (1) different from (3) and suited for multicore systems.

3.1 Extending Spiral for Shared Memory

The extension of Spiral to support shared memory parallelism requires four steps:

- We identify relevant hardware parameters and include them as *shared memory tags* into the rewriting system.
- We identify *parallel formula constructs*, i.e., those subformulas that can be perfectly mapped to shared memory platforms.
- We identify and include *rewriting rules* into the rewriting system that transform general formulas into parallel formulas, i.e., formulas suitable for mapping to multithreaded code.
- We extend the implementation level of Spiral to map parallel formulas into C code including the C extensions required for multithreading to actually *generate multithreaded code*.

Shared memory tags. The two most important parameters of modern shared memory machines (SMPs and CMPs) with memory hierarchies are the number of processors p, and the cache line length μ of the most important cache level. In this paper, μ is measured in complex numbers. For instance, for a cache line length of 64 bytes and data type double (64 bit), μ=4.

We denote a formula construct A that should be rewritten into parallel formula constructs for a p-way shared memory machine with cache parameter μ by

$$\underbrace{A}_{\mathrm{smp}(p,\mu)}$$

and introduce $\mathrm{smp}(p, \mu)$ as tag to Spiral's rewriting system. We also assume that all shared data vectors are aligned at cache line boundaries in the final program.

Parallel formula constructs. For arbitrary matrices A and A_i the expressions

$$y = \big(I_p \otimes A\big)x \quad \text{with} \quad A \in \mathbb{C}^{m\mu \times m\mu}$$

and

$$y = \left(\bigoplus_{i=0}^{p-1} A_i\right) x \quad \text{with} \quad A_i \in \mathbb{C}^{m\mu \times m\mu}$$

express embarrassingly parallel computation on p processors as they express block diagonal matrices with p blocks [Johnson et al. 1990]. We assume the matrix dimensions to be multiples of μ; this ensures that during computation each cacheline is owned by exactly one processor. Under the assumption that all A_i have the same computational cost, programs implementing these constructs become embarrassingly parallel, load balanced, and free of false sharing.

Data shuffling of the form

$$y = \big(P \otimes I_\mu\big)x, \quad \text{with } P \text{ a permutation matrix,}$$

reorders blocks of μ consecutive elements and thus whole cache lines are reordered. On shared memory machines

this means that if in a computation stage each processor has the unique ownership of a cache line, then a subsequent data access as determined by $P \otimes I_\mu$ preserves this property, i.e., only the ownership of entire cachelines is exchanged (if at all). Thus, false sharing is avoided. Note, that in Spiral-generated programs permutations are usually not performed explicitly, but folded with adjacent computation blocks [Franchetti et al. 2005].

We introduce tagged versions of the tensor product and direct sum operators in Spiral:

$$I_p \otimes_\parallel A, \quad \bigoplus_{i=0}^{p-1}{}_\parallel A_i, \quad P \bar{\otimes} I_\mu, \quad \text{with } A, A_i \in \mathbb{C}^{m\mu \times m\mu}. \tag{4}$$

These are the same matrix operators as their untagged counterparts, but declare that a construct is fully optimized for shared memory machines and does not require further rewriting. By fully optimized we mean that the formula is load-balanced for p processors (provided the A_i have equal computational cost) and avoids false sharing. This property is preserved for products of these constructs.

Definition 1 We say that a formula is *load-balanced* (*avoids false sharing*) if it is of the form (4) or of the form

$$I_m \otimes A \quad \text{or} \quad AB, \tag{5}$$

where A and B are load-balanced formulas (formulas that avoid false sharing). A formula is fully optimized (for shared memory) if it is load-balanced and avoids false sharing.

The goal of the rewriting system (explained next) is to transform formulas into fully optimized formulas.

Rewriting rules. Table 2 summarizes the rewriting rules sufficient for parallelizing the FFT (1). Identifying these rules is a major contribution of the paper. Spiral's rewriting system matches the left side of a rule against a given formula and replaces the matched expression by the right-hand side of the rule. All matrix parameters in the rules are integers; thus, an expression n/p on the right-hand side of a rule implies that the precondition $p|n$ must hold for the rule to be applicable.

As an example, consider rule (7) that encodes a form of loop tiling and scheduling. Namely, according to Table 1, the construct

$$A_m \otimes I_n \tag{12}$$

encodes a loop with unit stride between two iterations. Application of (7) leads to

$$\big(L^{mp}_m \otimes I_{n/p}\big)\big(I_p \otimes (A_m \otimes I_{n/p})\big)\big(L^{mp}_p \otimes I_{n/p}\big). \tag{13}$$

The construct $I_p \otimes (A_m \otimes I_{n/p})$ in (13) encodes a double loop: the outer loop running from 0 to $p-1$ and the inner loop running from 0 to $n/p-1$. Spiral's loop merging stage [Franchetti et al. 2005] regards this tensor product as

$$\underbrace{AB}_{\mathrm{smp}(p,\mu)} \rightarrow \underbrace{A}_{\mathrm{smp}(p,\mu)}\underbrace{B}_{\mathrm{smp}(p,\mu)} \tag{6}$$

$$\underbrace{A_m \otimes I_n}_{\mathrm{smp}(p,\mu)} \rightarrow \underbrace{\big(L^{mp}_m \otimes I_{n/p}\big)\big(I_p \otimes (A_m \otimes I_{n/p})\big)\big(L^{mp}_p \otimes I_{n/p}\big)}_{\mathrm{smp}(p,\mu)} \tag{7}$$

$$\underbrace{L^{mn}_m}_{\mathrm{smp}(p,\mu)} \rightarrow \begin{cases} \underbrace{\big(I_p \otimes L^{mn/p}_{m/p}\big)}_{\mathrm{smp}(p,\mu)}\underbrace{\big(L^{pn}_p \otimes I_{m/p}\big)}_{\mathrm{smp}(p,\mu)} \\ \underbrace{\big(L^{pm}_m \otimes I_{n/p}\big)}_{\mathrm{smp}(p,\mu)}\underbrace{\big(I_p \otimes L^{mn/p}_m\big)}_{\mathrm{smp}(p,\mu)} \end{cases} \tag{8}$$

$$\underbrace{I_m \otimes A_n}_{\mathrm{smp}(p,\mu)} \rightarrow I_p \otimes_\parallel \big(I_{m/p} \otimes A_n\big) \tag{9}$$

$$\underbrace{(P \otimes I_n)}_{\mathrm{smp}(p,\mu)} \rightarrow \big(P \otimes I_{n/\mu}\big)\bar{\otimes} I_\mu, \tag{10}$$

$$\underbrace{D}_{\mathrm{smp}(p,\mu)} \rightarrow \bigoplus_{i=0}^{p-1}{}_\parallel D_i, \tag{11}$$

Table 2: Shared memory parallelization rules. P is any permutation, D, D_i are diagonal matrices.

the *skeleton* which fixes the loop order and loop bounds. The *decorations* $L^{mp}_m \otimes I_{n/p}$ and $L^{mp}_p \otimes I_{n/p}$ are not performed explicitly, but merged with the skeleton loops. To produce the final code, Spiral further applies rules (6) and (8)–(10) and performs loop merging. The resulting code for (13) is shown below.

```
parallel for (i=0; i<p; i++)
   for (j=0; j<n/p; j++)
      y[i*n/p+j:n:i*n/p+j+m-1] =
         A(x[i*n/p+j:n:i*n/p+j+m-1]);
```

Inspection shows that n/p consecutive iterations of the original loop given by (12) are executed on the same processor and touch m contiguous memory areas of n/p complex numbers. If $\mu|m$ and $p|n$, each processor "owns" $mn/p\mu$ cache lines.

Similarly, the other rules in Table 2 encode variants of loop tiling, loop interchange, parallelization, or propagate the tags $\mathrm{smp}(p,\mu)$. Rule (6) expresses that in products of matrices each factor will be rewritten separately. (7) and (9) handle tensor products with identity matrices. Both rules distribute the computational load evenly among the p processors and execute as many consecutive iterations as possible on the same processor (as shown above). Rule (8) breaks stride permutations into two stages: one performs stride permutations locally for each processor, the other permutes consecutive chunks of data. (7) and (8) require the subsequent application of (6), (9), and (10) to fully break down to parallel formula constructs (4). Tensor products of a permutation and a sufficiently large identity matrix are broken into cache line resolution by (10). Rule (11) handles the twiddle factors by breaking a diagonal matrix into a direct sum of diagonal matrices.

implementation	program for $y = (I_p \otimes_\parallel A_n)x$

pthreads	`// parallel on p threads`
	`i = get_thread_id();`
	`y[i*n:1:i*n+n-1] =`
	`  A(x[i*n:1:i*n+n-1]);`
	`barrier();`

OpenMP	`// one iteration per thread`
	`#pragma omp parallel for \`
	` schedule(static) shared(x, y)`
	`for (i=0; i<p; i++)`
	`  y[i*n:1:i*n+n-1] =`
	`  A(x[i*n:1:i*n+n-1]);`

Table 3: Implementation of $y = (I_p \otimes_\parallel A_n)x$ using pthreads (SPMD) and OpenMP (loop parallelization). The parallel iterative direct sum in (4) is handled analogously.

The rules in Table 2 are based on known formula identities summarized in [Johnson et al. 1990; Franchetti and Püschel 2003; Franchetti et al. 2006]. They replace the usually expensive analysis required for the associated loop transformations by cheap pattern matching and also encode the actual transformation.

Generating multithreaded code. Extending Spiral's implementation level to support shared memory parallel code is straightforward. The only thing we have to add is the translation of the constructs

$$I_p \otimes_\parallel A \quad \text{and} \quad \bigoplus_{i=0}^{p-1}{}_\parallel A_i$$

into parallel code for p threads. We do not need to add support for $P\bar{\otimes}I_\mu$ as these permutations encode memory access with special indexing properties and are already handled within the formula optimization level. Namely, they are merged with the adjacent loops implementing tensor products [Franchetti et al. 2005].

All temporary variables used outside of these parallel constructs must be shared between the threads while all temporary variables solely used inside the parallel constructs must be thread local. We alternatively use two approaches: 1) the single program multiple data (SPMD) paradigm using pthreads (and a very fast barrier implemented using busy waiting that requires about 15 x86 assembly instructions); and 2) the loop level parallelization paradigm using OpenMP. Table 3 shows the actual parallel code.

3.2 Multicore Cooley-Tukey FFT

We apply the rewriting framework to derive a parallel version of the Cooley-Tukey FFT (1) using the rules (6)–(11). In Spiral these steps are performed automatically through rewriting. The result is a version of the Cooley-Tukey FFT that is fully optimized for shared memory in the sense of Definition 1.

Derivation. We start by specifying that we want to compute $y = \mathrm{DFT}_N\, x$ on p processors with cache line size μ,

$$\underbrace{\mathrm{DFT}_N}_{\mathrm{smp}(p,\mu)}.$$

In the first step rule (1) chooses a factorization of $N = mn$ and breaks DFT_N into DFT_m and DFT_n. Next, rule (6) propagates the parallelization tag to all factors:

$$\underbrace{\mathrm{DFT}_{mn}}_{\mathrm{smp}(p,\mu)} \to \underbrace{\big(\mathrm{DFT}_m \otimes I_n\big) D_{m,n} \big(I_m \otimes \mathrm{DFT}_n\big) L^{nm}_m}_{\mathrm{smp}(p,\mu)}$$
$$\to \underbrace{\big(\mathrm{DFT}_m \otimes I_n\big)}_{\mathrm{smp}(p,\mu)} \underbrace{D_{m,n}}_{\mathrm{smp}(p,\mu)} \underbrace{\big(I_m \otimes \mathrm{DFT}_n\big)}_{\mathrm{smp}(p,\mu)} \underbrace{L^{nm}_m}_{\mathrm{smp}(p,\mu)}. \quad (14)$$

We now consider each of the four factors on the right-hand side of (14) separately. The first factor is rewritten by applying (7) (requiring $p|n$),

$$\underbrace{\mathrm{DFT}_m \otimes I_n}_{\mathrm{smp}(p,\mu)} \to \underbrace{\big(L^{mp}_m \otimes I_{n/p}\big)}_{\mathrm{smp}(p,\mu)} \underbrace{\big(I_p \otimes (\mathrm{DFT}_m \otimes I_{n/p})\big)}_{\mathrm{smp}(p,\mu)} \underbrace{\big(L^{mp}_p \otimes I_{n/p}\big)}_{\mathrm{smp}(p,\mu)},$$

followed by (6), (9), and (10) (requiring $\mu|n/p$),

$$\underbrace{\mathrm{DFT}_m \otimes I_n}_{\mathrm{smp}(p,\mu)} \to \big((L^{mp}_m \otimes I_{n/p\mu})\bar{\otimes} I_\mu\big) \big(I_p \otimes_\parallel (\mathrm{DFT}_m \otimes I_{n/p})\big)\big((L^{mp}_p \otimes I_{n/p\mu})\bar{\otimes} I_\mu\big). \quad (15)$$

The second factor in (14) is parallelized using (11) (requiring $p|mn$) and the third factor using (9) (requiring $p|m$),

$$\underbrace{D_{m,n}}_{\mathrm{smp}(p,\mu)} \to \bigoplus_{i=0}^{p-1}{}_\parallel D^i_{m,n} \quad (16)$$

and

$$\underbrace{I_m \otimes \mathrm{DFT}_n}_{\mathrm{smp}(p,\mu)} \to I_p \otimes_\parallel \big(I_{m/p} \otimes \mathrm{DFT}_n\big). \quad (17)$$

The remaining fourth factor in(14) is parallelized by the first choice of rule (8) (requiring $p|m$) followed by (6), (9), and (10) (requiring $\mu|m/p$),

$$\underbrace{L^{mn}_m}_{\mathrm{smp}(p,\mu)} \to \underbrace{\big(I_p \otimes L^{mn/p}_{m/p}\big)}_{\mathrm{smp}(p,\mu)} \underbrace{\big(L^{pn}_p \otimes I_{m/p}\big)}_{\mathrm{smp}(p,\mu)}$$
$$\to \big(I_p \otimes_\parallel L^{mn/p}_{m/p}\big)\big((L^{pn}_p \otimes I_{m/p\mu})\bar{\otimes} I_\mu\big). \quad (18)$$

Collecting (15)–(18) and the constraints required for applying the rules leads to the final expression output by our

$$\underbrace{\mathrm{DFT}_{mn}}_{\mathrm{smp}(p,\mu)} \rightarrow \big((L^{mp}_m \otimes I_{n/p\mu})\bar{\otimes} I_\mu\big)\big(I_p \otimes_\parallel (\mathrm{DFT}_m \otimes I_{n/p})\big)\big((L^{mp}_p \otimes I_{n/p\mu})\bar{\otimes} I_\mu\big)$$

$$\left(\bigoplus_{i=0}^{p-1}{}_\parallel D^i_{m,n}\right)\Big(I_p \otimes_\parallel (I_{m/p} \otimes \mathrm{DFT}_n)\big)\big(I_p \otimes_\parallel L^{mn/p}_{m/p}\big)\big((L^{pn}_p \otimes I_{m/p\mu})\bar{\otimes} I_\mu\big) \quad (19)$$

Figure 2: Multicore Cooley-Tukey FFT for p processors and cache line length μ.

rewriting system, (19) displayed in Figure 2, with the requirement $p\mu|m$ and $p\mu|n$. Inspection shows that (19) is fully optimized for shared memory in the sense of Definition 1. We call (19) the *multicore Cooley-Tukey FFT*.

```
// C99 OpenMP DFT_8
// call by a sequential function

#include <omp.h>

static _Complex double D[8] = {
  1, 1, 1,
  0.70710678118654+__I__*0.70710678118654,
  1, 1, 1,
  -0.70710678118654+__I__*0.70710678118654
};

void DFT_8(_Complex double *Y,
           _Complex double *X) {
  int i1, i3;
  static _Complex double T[8];
  #pragma omp parallel for \
   schedule(static) shared(T, X, Y)
  for(i1 = 0; i1 <= 1; i1++) {
    _Complex double s1, s2;
    int i2;
    for(i2 = 0; i2 <= 1; i2++) {
      s1 = X[2*i1 + i2];
      s2 = X[4 + 2*i1 + i2];
      T[4*i1 + 2*i2] = s1 + s2;
      T[4*i1 + 2*i2 + 1] = s1 - s2;
    }
  }
  #pragma omp parallel for \
   schedule(static) shared(T, X, Y)
  for(i3 = 0; i3 <= 1; i3++) {
    _Complex double s3, s4, s5, s6,
      s7, s8, s9, s10;
    s10 = D[i3]*T[i3];
    s9 = D[4 + i3]*T[4 + i3];
    s8 = s10 + s9;
    s4 = D[6 + i3]*T[6 + i3];
    s7 = D[2 + i3]*T[2 + i3];
    s6 = s7 + s4;
    s5 = s10 - s9;
    s3 = __I__*(s7 - s4);
    Y[i3] = s8 + s6;
    Y[4 + i3] = s8 - s6;
    Y[2 + i3] = s5 + s3;
    Y[6 + i3] = s5 - s3;
  }
}
```

Figure 3: Multithreaded C99 OpenMP function computing $y = \mathrm{DFT}_8\, x$ using 2 processors, called by a sequential program.

As a small example, Figure 3 shows the C99 OpenMP program generated by Spiral from (19) with $m = 4$, $n = 2$, $p = 2$, and $\mu = 2$. Note, that all permutations are fused into the skeleton loops [Franchetti et al. 2005] and that the smaller DFT_2 and DFT_4 are fully unrolled.

Discussion. Traditional shared memory FFT algorithms [Norton and Silberger 1987; Schwarztrauber 1987; Bailey 1990] optimize for a large number of processors with the actual number not known in advance. The cost of memory access is assumed to be small compared to arithmetic operations. Under this assumptions the six-step algorithm (3) with the stride permutations implemented explicitly is a good solution, in particular, when assuming NUMA architectures. Adapting the explicit stride permutation to modern memory hierarchies makes blocking the stride permutation [Al Na'mneh et al. 2005a] and partially folding it into the computation a good choice [Takahashi 2002].

Studying the source code of FFTW 3.1 [Frigo and Johnson 2005] reveals that it implements a parallel Cooley-Tukey FFT obtained by parallelizing loops inside the algorithm and scheduling these loops block-cyclically. It also includes experimental thread pooling. Their algorithm space overlaps the space spanned by formula (19), albeit μ and the interplay of p and μ is not explicitly used. Thus, more possible algorithms are considered, only some of which are suited for the particular parallel target architecture. Further, FFTW requires a large infrastructure, which makes low-overhead parallelization difficult.

In contrast, the multicore Cooley-Tukey FFT (19) exists for all DFT_N with $(p\mu)^2|N$, independently of the further decomposition of DFT_m and DFT_n. It spans a set of shared memory DFT algorithms that are load balanced and free of false-sharing for p processors and cache line length μ. Further, by automatically implementing instances of (19) for fixed N, p, and μ, we can use low-latency minimal overhead synchronization. In addition, the fact that (19) breaks down to smaller DFTs with alignment guarantees for their input and output vectors makes it possible to use (19) in tandem with the efficient short vector Cooley-Tukey FFT [Franchetti and Püschel 2003; Franchetti et al. 2006] on machines with SIMD extensions.

	CPU			*cache*				
	processor	p-way	f [GHz]	L1D	L2 unified	sync.	line	μ
traditional SMP	Xeon MP	4	2.8	8kB / 32kB	512kB / 2MB	bus	64B	4
transitional	Pentium D	2	3.6	16kB / 32kB	2MB / 4MB	bus	64B	4
multicore	Opteron Dual-Core	4 (2×2)	2.2	64kB / 256kB	1MB / 4MB	fast	64B	4
	Core Duo	2	2.0	32kB / 64kB	2MB / 2MB	shared	64B	4

Table 4: SMP test platforms: traditional (1 CPU per chip), and chip multiprocessors (multicore and transitional, 2 CPUs per chip). Cache sizes are given for 1 CPU / all CPUs.

		cache capacity		*break even or first speed-up*		
	CPU	L1	L2	Spiral (1D)	FFTW (1D)	MKL (2D)
traditional SMP	Xeon MP	2^7 / 2^9	2^{13} / 2^{15}	2^{11} / 117,000	2^{14} / 1,320,000	$2^7 \times 2^7$ / 3,110,000
transitional	Pentium D	2^8 / 2^9	2^{15} / 2^{16}	2^{10} / 43,300	2^{13} / 516,000	$2^7 \times 2^7$ / 2,640,000
multicore	Opteron Dual-Core	2^{10} / 2^{12}	2^{14} / 2^{16}	2^8 / 9,200	2^{14} / 1,270,000	–
	Core Duo	2^9 / 2^{10}	2^{15} / 2^{15}	2^8 / 9,610	2^{14} / 718,000	$2^6 \times 2^7$ / 1,130,000

Table 5: Cache capacity (largest DFT size with memory footprint fitting into the cache level of one / all processors), and smallest DFT sizes where parallel code is at least as fast as the sequential code / runtime of these DFTs in cycles.

4 Experimental Results

Experimental setup. We evaluated our approach on the following 4 SMP machines shown in Table 4.

- 2.0 GHz Intel *Core Duo*: dualcore CPU with shared L2 cache, laptop;
- 3.6 GHz Intel *Pentium D*: two CPUs on one chip, synchronization through bus, desktop;
- 2.2 GHz AMD *Opteron Dual Core*: four CPUs (two per dualcore chip) with fast on-chip cache synchronization but no shared cache, workstation; and
- 2.8 GHz Intel *Xeon MP*: four processors communicating through the bus, rackmount server.

The Core Duo and the Opteron are "real" multicore CPUs with fast on-chip communication while the Xeon MP is a traditional SMP with interprocessor communication through the bus. The Pentium D is a transition between traditional SMP and multicore CPU (two CPUs on the same chip, but bus communication). All machines run Linux (SMP kernel 2.6, SMP kernel 2.4 for the Xeon MP). We used the 32-bit Intel C++ compiler 9.0 with options "-O3 -xWP -tpp7" on all machines ("-xW" for the Xeon MP).

We compared Spiral-generated code to FFTW 3.1 and the Intel MKL 8.0.

We built FFTW using the Intel C++ compiler. We used FFTW's benchmark utility, called bench, with the options "-opatient -onthreads=<n>." As experimental option to be turned on by hand FFTW supports thread pooling using semaphores and spin locks. We ran FFTW both with and without experimental thread-pooling and took the better performance. However, thread pooling only worked for two threads using semaphores. Spin lock support did not compile and for four threads thread pooling was hanging. We enabled FFTW's SSE2 support for all experiments and used a similar vectorization method within Spiral to produce SSE2 code. The performance comparison between FFTW and Spiral is summarized in Figure 4. We measure performance in *pseudo Mflop/s*, which is proportional to inverse runtime and defined as $5N \log_2 N/t$, where t is the runtime in μs.

The Intel MKL is parallelized using OpenMP. It does not support one-dimensional parallel FFTs, so we compare to two-dimensional FFTs (only in Table 5) of the same data size (e.g., DFT_{1024} vs. $\mathrm{DFT}_{32\times32}$). This gives the MKL a slight advantage (better structure and less arithmetic).

General behavior. In Figure 4 we see the same general behavior across all machines.

Spiral-generated sequential code is within 10% of FFTW's performance.

FFTW's benchmark utility cannot be forced to run with a certain number of threads. One can only specify a maximum number of threads to be used and FFTW will pick the number of threads that yield the highest performance. So for parallel performance we always display the maximum performance of 1, 2, and 4 threads (we run 4 threads only on the 4-processor machines). This results in a branching of the sequential and parallel line at the first problem size where parallelization improves performance.

Spiral pthreads code consistently gets a parallelization speed-up earlier than FFTW and gets higher top performance for in-cache problem sizes. On the two-processor

machines (Core Duo and Pentium D) and for out-of-cache sizes, Spiral-generated parallel code is running within 75% of FFTW's performance. The relative gain of FFTW is due to extensive optimizations that specifically target large problem sizes [Frigo and Johnson 2005]. Spiral currently does not support all of these optimizations. On the four-processor machines and for out-of-cache sizes, Spiral-generated parallel code is equally fast (Xeon MP) and up to 25% faster (Opteron) than FFTW. The breakdown of FFTW's experimental thread pooling on the four-processor machines may contribute to Spiral's higher relative performance for large sizes on four processors as compared to two processors. FFTW starts using all 4 processors at $N = 2^{20}$ compared to $N = 2^9$ for Spiral.

Traditional SMP vs. multicore. Table 5 shows that multicore processors make it possible to gain performance by parallelization for much smaller problem sizes than on traditional SMPs. In our experiments only Spiral generated code could take advantage from the multicore processors' faster communication, while FFTW's and Intel MKL's performance characteristics was the same on both traditional SMPs and multicore processors.

To see the difference between traditional SMPs and multicore CPUs, we need to analyze Table 5 carefully. We estimate the memory footprint of a double precision complex out-of-place DFT_N as being between $32N$ and $64N$ bytes. Now we compute the largest DFTs for which the working set fits completely into the L1 cache of one processor and the combined L1 cache of all processors. We do the same for the L2 cache. The result is displayed in the columns "cache capacity: L1, L2" of Table 5. Columns "break even or first speed-up" of Table 5 shows the smallest problem size for which parallel computation is at least as fast as sequential computation. It also shows the runtime of these DFTs in cycles.

The difference between SMP and multicore becomes clear as we relate these break even points to the L1 cache size of our target machines. It turns out that on multicore CPUs Spiral code breaks even for $N = 2^8$, which runs in less than 10,000 cycles. Surprisingly, we see speed-up even though the working set fits into the L1 cache of *one* CPU, which is due to the fast on-chip communication. In contrast, on the Pentium D and the traditional SMP Xeon MP the break point occurs at $N = 2^{10}$ and $N = 2^{11}$, respectively. It requires 5 to 10 times longer running programs to account for the higher overhead. The break even point occurs for the smallest problem sizes where the working set does not fit into the combined L1 cache of all CPUs any more. This shows the much higher communication cost on traditional SMPs.

In contrast, FFTW and MKL break even on all machines at around $N = 2^{13}$ or $N = 2^{14}$, running in the range of 500,000 to 2,500,000 cycles. On multicore processors FFTW and MKL have to run programs with more than 100 times longer runtime before parallelization pays off compared to Spiral. On the traditional SMP this ratio is around 10 times. For all machines, the working set for $N = 2^{13}$ does not fit into the combined L1 cache of all processors but fits into the L2 cache.

OpenMP vs. pthreads. Using Intel's newest OpenMP library one can achieve good parallel performance on both traditional SMPs and multicore CPUs. The Spiral OpenMP performance is within 15% of our low-latency pthreads implementation. The OpenMP overhead is low enough to use it even for small problem sizes. However, to achieve highest performance we need to resort to pthreads and our custom low-latency synchronization. Our experiments also show that it is absolute essential to use the newest available OpenMP version. Older versions may not recognize the target processor correctly and can quickly deteriorate performance.

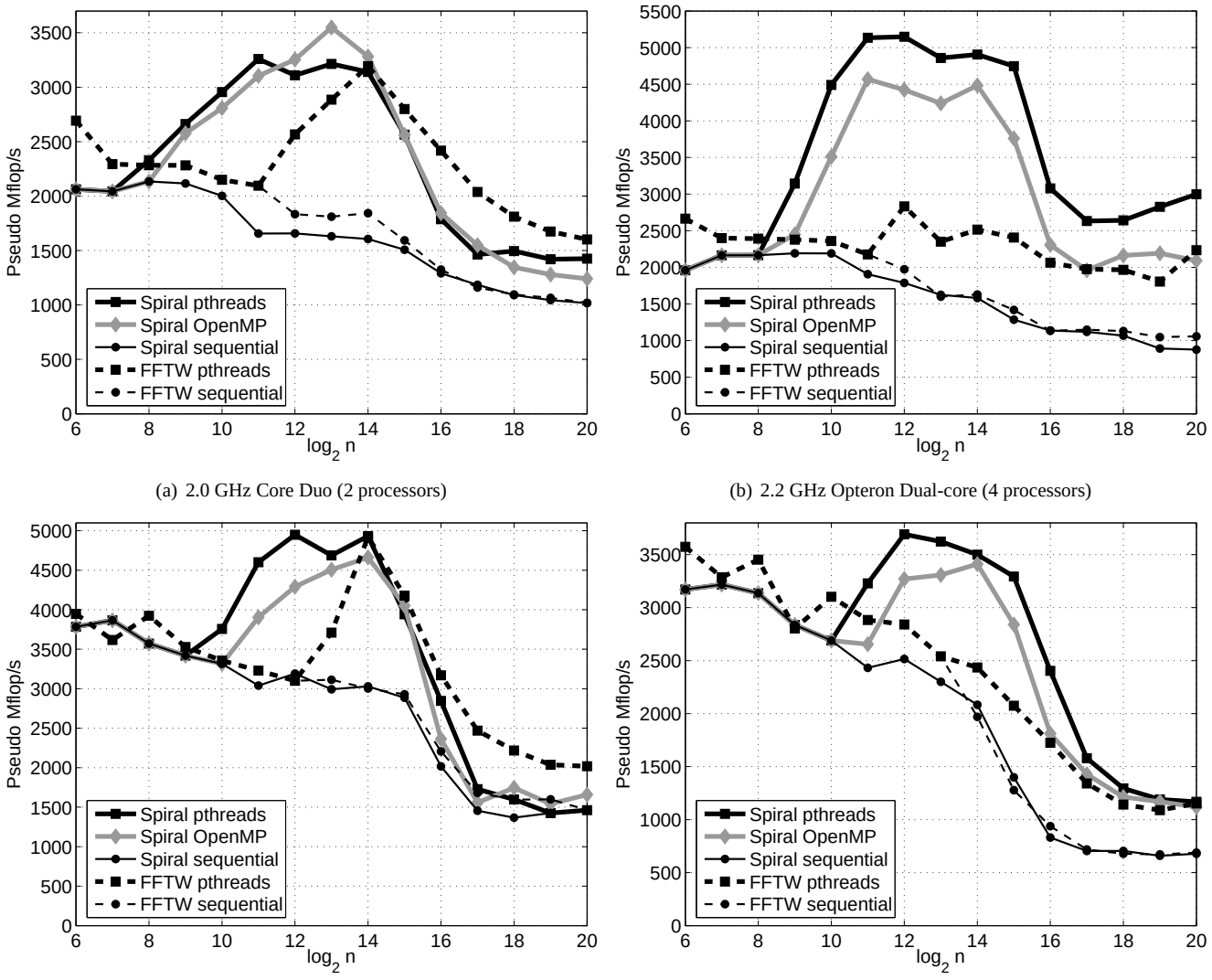
Compiler parallelization. We experimented extensively with the Intel C++ compiler's automatic parallelization capabilities. However, the compiler failed to produce any speed-up even though it sometimes did parallelize. This behavior for DFT code is similar as for SIMD vectorization [Franchetti and Püschel 2003] and compiler vectorization and shows the importance of writing or generating explicit parallel and vector code.

5 Conclusion

As multicore CPUs become mainstream, programming for performance may finally be pushed over the edge from general computer science knowledge to specialized expert skill. To facilitate code development and optimization, at least for well understood library functionality, a new breed of tools is necessary in the form of formal frameworks, program generators, or adaptive libraries. A few of these exist but more work is needed. This paper aims to be a contribution in this direction. The generation of fast FFTs for SMPs and multicore systems is useful, but we believe the ideas and concepts presented (and underlying Spiral in general) are of equal value: a high-level domain-specific framework that enables us to reason about algorithms before they are implemented, that enables optimizations unpractical at the program level, and that completely automates the implementation task.

6 Acknowledgement

This work was supported by DARPA through the Department of Interior grant NBCH1050009 and by NSF through awards 0234293 and 0325687. The authors also wish to thank Paul Petersen (Intel) for helpful suggestions on the use of OpenMP.



(a) 2.0 GHz Core Duo (2 processors)

(b) 2.2 GHz Opteron Dual-core (4 processors)

(c) 3.6 GHz Pentium D (2 processors)

(d) 2.8 GHz Xeon MP (4 processors)

Figure 4: Results for DFT_N on four symmetric multiprocessing machines. The performance is given in *pseudo Mflop/s* defined as $5N \log N/\text{runtime}$. Higher is better. Note that the scales in the plots differ.

References

AL NA'MNEH, R. A., PAN, W. D., AND ADHAMI, R. 2005. Communication efficient adaptive matrix transpose algorithm for FFT on symmetric multiprocessors. In *Proc. Southeastern Symposium on System Theory (SSST)*, 312–315.

AL NA'MNEH, R. A., PAN, W. D., AND ADHAMI, R. 2005. Parallel implementation of 1-D fast Fourier transform without inter-processor communications. In *Proc. Southeastern Symposium on System Theory (SSST)*, 307–311.

BAILEY, D. H. 1990. FFTs in external or hierarchical memory. *J. Supercomputing 4*, 23–35.

BANERJEE, U., EIGENMANN, R., NICOLAU, A., AND PADUA, D. A. 1993. Automatic program parallelization. *Proceedings of the IEEE 81*, 2, 211–243.

BIENTINESI, P., GUNNELS, J. A., MYERS, M. E., QUINTANA-ORTI, E., AND VAN DE GEIJN, R. 2005. The science of deriving dense linear algebra algorithms. *TOMS 31*, 1 (March), 1–26.

CHANDRA, R., MENON, R., DAGUM, L., KOHR, D., MAYDAN, D., AND MCDONALD, J. 2000. *Parallel Programming in OpenMP*. Elsevier.

DEMMEL, J., DONGARRA, J., EIJKHOUT, V., FUENTES, E., PETITET, A., VUDUC, R., WHALEY, C., AND YELICK, K. 2005. Self adapting linear algebra algorithms and software. *Proceedings of the IEEE 93*, 2, 293–312.

Special issue on "Program Generation, Optimization, and Adaptation".

DERSHOWITZ, N., AND PLAISTED, D. A. 2001. Rewriting. In *Handbook of Automated Reasoning*, A. Robinson and A. Voronkov, Eds., vol. 1. Elsevier, ch. 9, 535–610.

FRANCHETTI, F., AND PÜSCHEL, M. 2002. A SIMD vectorizing compiler for digital signal processing algorithms. In *Proc. IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS)*, 20–26.

FRANCHETTI, F., AND PÜSCHEL, M. 2003. Short vector code generation for the discrete Fourier transform. In *Proc. IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS)*, 58–67.

FRANCHETTI, F., VORONENKO, Y., AND PÜSCHEL, M. 2005. Loop merging for signal transforms. In *Proc. Programming Language Design and Implementation (PLDI)*, 315–326.

FRANCHETTI, F., VORONENKO, Y., AND PÜSCHEL, M. 2006. A rewriting system for the vectorization of signal transforms. In *Proc. High Performance Computing for Computational Science (VECPAR)*.

FRIGO, M., AND JOHNSON, S. G. 2005. The design and implementation of FFTW3. *Proceedings of the IEEE 93*, 2, 216–231. Special issue on "Program Generation, Optimization, and Adaptation".

GATLIN, K. S., AND CARTER, L. 1999. Architecture-cognizant divide and conquer algorithms. In *Proc. Supercomputing (CDROM)*.

GUNNELS, J. A., GUSTAVSON, F. G., HENRY, G. M., AND VAN DE GEIJN, R. A. 2001. FLAME: Formal linear algebra methods environment. *TOMS 27*, 4 (December), 422–455.

HIRANANDANI, S., KENNEDY, K., AND TSENG, C.-W. 1992. Compiling Fortran D for MIMD distributed-memory machines. *Commun. ACM 35*, 8, 66–80.

IM, E.-J., YELICK, K., AND VUDUC, R. 2004. Sparsity: Optimization framework for sparse matrix kernels. *Int'l J. High Performance Computing Applications 18*, 1.

JOHNSON, J. R., JOHNSON, R. W., RODRIGUEZ, D., AND TOLIMIERI, R. 1990. A methodology for designing, modifying, and implementing FFT algorithms on various architectures. *Circuits Systems Signal Processing 9*, 449–500.

MCKELLAR, A. C., AND E. G. COFFMAN, J. 1969. Organizing matrices and matrix operations for paged memory systems. *Communications ACM 12*, 3, 153–165.

NORTON, A., AND SILBERGER, A. J. 1987. Parallelization and performance analysis of the Cooley-Tukey FFT algorithm for shared-memory architectures. *IEEE Trans. Comput. 36*, 5, 581–591.

PÜSCHEL, M., MOURA, J. M. F., JOHNSON, J., PADUA, D., VELOSO, M., SINGER, B. W., XIONG, J., FRANCHETTI, F., GAČIĆ, A., VORONENKO, Y., CHEN, K., JOHNSON, R. W., AND RIZZOLO, N. 2005. SPIRAL: Code generation for DSP transforms. *Proc. of the IEEE 93*, 2, 232–275. Special issue on *Program Generation, Optimization, and Adaptation*.

SCHWARZTRAUBER, P. N. 1987. Multiprocessor FFTs. *Parallel Computing 5*, 197–210.

SINGER, B., AND VELOSO, M. 2001. Stochastic search for signal processing algorithm optimization. In *Proc. Supercomputing*.

TAKAHASHI, D., SATO, M., AND BOKU, T. 2003. An OpenMP implementation of parallel FFT and its performance on IA-64 processors. *Lecture Notes in Computer Science 2716*, 99–108.

TAKAHASHI, D. 2002. A blocking algorithm for parallel 1-D FFT on shared-memory parallel computers. *Lecture Notes in Computer Science 2367*, 380–389.

VAN LOAN, C. 1992. *Computational Framework of the Fast Fourier Transform*. SIAM.

WHALEY, R. C., PETITET, A., AND DONGARRA, J. J. 2001. Automated empirical optimization of software and the ATLAS project. *Parallel Computing 27*, 1–2, 3–35.

WOLF, M. E., AND LAM, M. S. 1991. A data locality optimizing algorithm. In *Proc. Programming Language Design and Implementation (PLDI)*, 30–44.

WOLFE, M. 1996. *High performance compilers for parallel computing*. Addison-Wesley, Redwood City, CA.

XIONG, J., JOHNSON, J., JOHNSON, R., AND PADUA, D. 2001. SPL: A language and compiler for DSP algorithms. In *Proc. Programming Language Design and Implementation (PLDI)*, 298–308.

ZIMA, H., AND CHAPMAN, B. 1990. *Supercompilers for parallel and vector computers*. ACM Press, New York.