Parallelism in Spiral

Franz Franchetti
Carnegie Mellon University
franzf@ece.cmu.edu

Andreas Bonelli
Vienna University of Technology
andreas.bonelli@gmail.com

Ekapol Chuangsuwanich
Carnegie Mellon University
ekapolc@gmail.com

Yu-Chiang J. Lee
Carnegie Mellon University
yjlee@andrew.cmu.edu

Jürgen Lorenz
Vienna University of Technology
juergen.lorenz@gmail.com

Thomas Peter
ETH Zürich
petertho@ee.ethz.ch

Hao Shen
Technical University of Denmark
synchour@gmail.com

Marek Telgarsky
Carnegie Mellon University
marekt@andrew.cmu.edu

Yevgen Voronenko
Carnegie Mellon University
yvoronen@ece.cmu.edu

Markus Püschel
Carnegie Mellon University
pueschel@ece.cmu.edu

José M. F. Moura
Carnegie Mellon University
moura@ece.cmu.edu

Christoph W. Ueberhuber
Vienna University of Technology
c.ueberhuber@tuwien.ac.at

ABSTRACT

Spiral is a program generator for linear transforms such as the discrete Fourier transform. Spiral generates highly optimized code directly from a problem specification using a combination of techniques including optimization at a high level of abstraction using rewriting of mathematical expressions and heuristic search for platform adaptation. In this paper, we overview the generation of parallel programs using Spiral. This includes programs for vector architectures and programs for shared or distributed memory platforms.

1. INTRODUCTION

Programmers in charge of developing high performance libraries for current off-the-shelf computers are confronted with the difficult task of optimizing for deep memory hierarchies, extracting the fine-grain parallelism for vector instruction sets, and producing multithreaded code for multicore processors. The development of libraries for media processors like the Cell processor and powerful graphics processors (GPUs) supporting floating-point computation are even more difficult. Writing programs that take advantage of all the performance-enhancing hardware features in a modern commodity computer system is a nightmare.

This scenario strengthens the case for recent efforts on automatic performance tuning, program generation, and adaptive library frameworks that can offer high performance on a variety of platforms with greatly reduced development time. ATLAS [17] is a program generator for basic linear algebra subroutines (BLAS). For a given BLAS routine, ATLAS generates implementations with different degrees of loop unrolling and blocking to find the best match to the given microarchitecture. FLAME is a framework to systematically generate alternative programs from a problem specification for dense linear algebra problems [12, 2]. These programs are implemented on top of BLAS routines. FFTW [10] is a library for the discrete Fourier transform (DFT). For small DFT sizes, FFTW calls special code modules, called codelets. These are pregenerated and highly optimized using standard and DFT specific dataflow graph optimization techniques [9]. For large DFT sizes, FFTW uses heuristic search to find the best recursion strategy to break down into codelets. FFTW supports shared and distributed memory platforms.

Other examples of automatic tuning include [13] for sparse linear algebra and [1] for parallel tensor computations.

The above efforts make great strides towards automating the implementation and optimization task for important numerical functionality. However, the automation is mostly restricted to sequential code. For example, ATLAS does not generate vector or multithreaded implementations. FFTW includes generated codelets for vector architectures, but the multithreaded functionality, as the implementation of large DFTs, is hand-coded.

This is different in Spiral [15, 16], which is the topic of this paper. Spiral is a program generator for linear transforms and automates the entire implementation task from problem specification to program. This is possible because Spiral uses an internal mathematical language to express and optimize algorithms. This way, Spiral is extensible if new forms of code have to be generated such as vector code or multithreaded code for shared or distributed memory platforms. This paper gives an overview on the generation of parallel programs using Spiral.

2. SPIRAL

Spiral is a program generator for linear transforms such as the discrete Fourier transform (DFT), the Walsh-Hadamard transform (WHT), the discrete cosine and sine transforms, finite impulse response (FIR) filters, and the discrete wavelet transform, among others. The input to Spiral is a formally specified transform

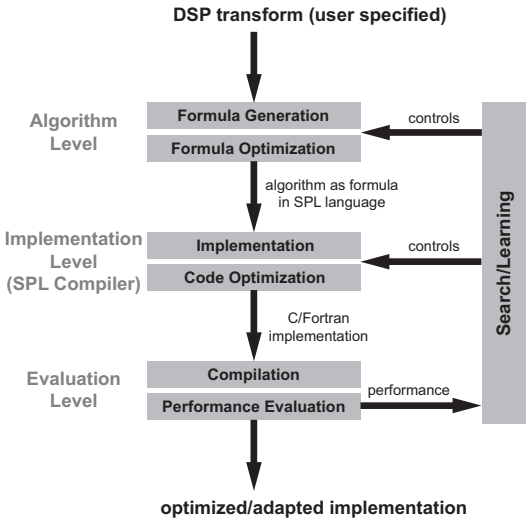

Figure 1: The program generator Spiral.

(e.g., DFT of size 245); the output is a highly optimized C program implementing the transform.

In Spiral (see Figure 1), recursive computation of larger transforms by smaller transforms is expressed using *rules*. For a given transform, Spiral recursively applies these rules to generate one out of many possible algorithms represented as a *formula* in a language called SPL (signal processing language). This formula is then structurally optimized using a rewriting system and finally translated into a C program (for computing the transform) using a special formula compiler. The C program is further optimized and then a native compiler is used to generate an executable. Its runtime is measured and fed into a search engine, which decides how to modify the algorithm; that is, the engine changes the formula, and thus the code, by using dynamic programming or other search methods. Eventually, this feedback loop terminates and outputs the fastest program found in the search. See [15, 6] for a complete description.

Mathematical foundation. A (linear) transform is a matrix-vector multiplication $x \mapsto y = Mx$, where x is a real or complex input vector, M the transform matrix, and y the result. For example, for an input vector $x \in \mathbb{C}^n$, the DFT is defined by the matrix

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

Algorithms for transforms can be written using the Kronecker product formalism [15] in the form of structured sparse matrix factorizations. We use I_n to denote an $n \times n$ identity matrix, and

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

for the tensor product of matrices. Then, for example, the Cooley-Tukey FFT algorithm can be written as

$$\mathrm{DFT}_{mn} \to \big(\mathrm{DFT}_m \otimes \mathrm{I}_n\big) D_{m,n} \big(\mathrm{I}_m \otimes \mathrm{DFT}_n\big) \mathrm{L}^{nm}_m \quad (1)$$

with the diagonal matrix $D_{m,n}$ and the stride permutation matrix L^{nm}_m.

3. FORMAL PARALLELIZATION

The key observation is that formula constructs can be related to properties of the target architecture. In particular, certain formula constructs can be implemented efficiently on a particular type of hardware while they are ill-suited for other types of hardware. As example, in (1) the construct

$$\mathrm{I}_m \otimes \mathrm{DFT}_n \quad (2)$$

has a perfect structure for m-way parallel (shared and distributed memory) machines. Similarly, the construct

$$\mathrm{DFT}_m \otimes \mathrm{I}_n \quad (3)$$

has a perfect structure for n-way vector SIMD (single instruction, multiple data) architectures [5]. However, (2) is ill-suited for vector SIMD architectures and (3) is ill-suited for distributed memory machines.

Formulas can be manipulated using algebraic identities [14]. For instance, the identity

$$\mathrm{DFT}_m \otimes \mathrm{I}_n = \mathrm{L}^{mn}_m \big(\mathrm{I}_n \otimes \mathrm{DFT}_m\big) \mathrm{L}^{mn}_n \quad (4)$$

replaces a vector formula by a parallel formula and introduces two stride permutations.

Optimization through rewriting. The basic idea in Spiral's formula optimization is to rewrite a generated formula into another formula that has a structure that maps well to a given target architecture. An example is parallelization: Spiral rewrites formulas to obtain the right form and the right degree of parallelism.

Spiral's rewriting system for parallelization consists of three crucial components to accomplish this goal:

- *Tags* encode target architecture types and parameters. Specifically, Spiral uses the tags "vec(ν)" for SIMD vector extensions, "smp(p, μ)" for shared memory, "mpi(p)" for message passing, and "gpu(n, t)" for graphics processors. The meaning of these parameters is explained later.

- *Base cases* encode formula constructs that can be mapped well to a given target architecture. For instance, we denote a p-way parallel base case generalizing (2) by $\mathrm{I}_p \otimes_\parallel A_n$, using the tagged operator "$\otimes_\parallel$" and the variable A_n matching any $n \times n$ matrix.

- *Rewriting rules* encode how to translate general formulas into base cases. For instance, we generalize (4) into the rewriting rule

$$\underbrace{A_m \otimes \mathrm{I}_n}_{\mathrm{mpi}(p)} \to \underbrace{\mathrm{L}^{mn}_m}_{\mathrm{mpi}(p)} \big(\mathrm{I}_p \otimes_\parallel (\mathrm{I}_{n/p} \otimes A_m)\big) \underbrace{\mathrm{L}^{mn}_n}_{\mathrm{mpi}(p)}. \quad (5)$$

(5) applies identity (4), but "knows" (due to the tag mpi(p)) that the target architecture is a p-way parallel message passing system, and thus introduces the matching base case $\mathrm{I}_p \otimes_\parallel (\mathrm{I}_{n/p} \otimes A_m)$. The stride permutations L^{mn}_m and L^{mn}_n will be handled by further rewriting.

To identify the above components, we follow the same procedure for all supported architectures. 1) We identify the most important platform parameters and encode these as *tags*. 2) We identify formula constructs

that can be mapped well to the target architecture, thus defining a set of *base cases*. We also specify their efficient implementation. 3) We identify a set of rewriting rules, parameterized by hardware parameters, which translate general constructs into base cases.

In the remainder of this section we provide additional details on this approach for vector SIMD extensions, multicore CPUs, message passing, the Cell processor, and GPUs.

Vector SIMD instructions. To generate efficient SIMD vector code we need to guarantee that all memory accesses are properly aligned and load/store whole vectors. All arithmetic should be done using vector operations and all data shuffling should take place in vector registers using efficient shuffle instructions (which may differ across supported SIMD architectures). The number of shuffle operations should be minimized.

We introduce the tag "vec(ν)" for vector SIMD operation using ν-way vector instructions. The construct

$$A \vec{\otimes} \mathrm{I}_\nu \tag{6}$$

(using the tagged operator "$\vec{\otimes}$") can be implemented solely using vector arithmetic and guarantees aligned memory access of whole vectors. Further, Spiral automatically builds a library of vectorized implementations of permutations including

$$\underbrace{\mathrm{L}^{2\nu}_2}_{\mathrm{vec}(\nu)}, \quad \underbrace{\mathrm{L}^{2\nu}_\nu}_{\mathrm{vec}(\nu)}, \quad \text{and} \quad \underbrace{\mathrm{L}^{\nu^2}_\nu}_{\mathrm{vec}(\nu)} \tag{7}$$

for every supported vector architecture. The constructs (6) and (7) constitute some of the base cases for vector architectures.

Vector rewriting rules like

$$\underbrace{A \otimes \mathrm{I}_n}_{\mathrm{vec}(\nu)} \quad \rightarrow \quad (A \otimes \mathrm{I}_{n/\nu}) \vec{\otimes} \mathrm{I}_\nu \tag{8}$$

$$\underbrace{\mathrm{L}^{n\nu}_n}_{\mathrm{vec}(\nu)} \quad \rightarrow \quad \big(\mathrm{I}_{n/\nu} \otimes \underbrace{\mathrm{L}^{\nu^2}_\nu}_{\mathrm{vec}(\nu)}\big)\big(\mathrm{L}^n_{n/\nu} \vec{\otimes} \mathrm{I}_\nu\big) \tag{9}$$

are parameterized by ν and encode how to turn general constructs into vector base cases. Using vector base cases and vector breakdown rules, Spiral successfully vectorizes a large class of linear transform algorithms. Further details on Spiral's vectorization process can be found in [5, 8].

Multicore and SMP. The goal on symmetric multiprocessors and multicore CPUs is to balance the computational load and to avoid false sharing (private data of multiple processors stored in the same cache line). We aim at generating programs in which each cache line is accessed only by one processor at a time (the processor "owns" that cache line), and change of ownership happens as little as possible, thus minimizing communication invoked by the cache coherency protocol.

We introduce the tag "smp(p, μ)" for shared memory computation on p processors with cache line size μ. The construct

$$\mathrm{I}_p \otimes_\parallel A \tag{10}$$

expresses embarrassingly parallel, load balanced operation on p processors and can be easily translated into a multithreaded program. The constructs

$$P \bar{\otimes} \mathrm{I}_\mu, \quad P \text{ a permutation on } mp \text{ points}, \tag{11}$$

expresses (potential) ownership change of cache lines. (10) and (11) are shared memory base cases. The shared memory rule set consists of rules like

$$\underbrace{A_m \otimes \mathrm{I}_n}_{\mathrm{smp}(p,\mu)} \quad \rightarrow \quad \underbrace{\big(\mathrm{L}^{mp}_m \otimes \mathrm{I}_{n/p}\big)}_{\mathrm{smp}(p,\mu)} \big(\mathrm{I}_p \otimes_\parallel (A_m \otimes \mathrm{I}_{n/p})\big) \underbrace{\big(\mathrm{L}^{mp}_p \otimes \mathrm{I}_{n/p}\big)}_{\mathrm{smp}(p,\mu)} \tag{12}$$

$$\underbrace{(P \otimes \mathrm{I}_n)}_{\mathrm{smp}(p,\mu)} \quad \rightarrow \quad (P \otimes \mathrm{I}_{n/\mu}) \bar{\otimes} \mathrm{I}_\mu, \; P \text{ permutation} \tag{13}$$

and allows to parallelize even small transforms and to produce efficient multicore code. Further information on Spiral's shared memory and multicore code generation can be found in [7].

Distributed memory. The goal on distributed memory machines is to minimize the number of messages to be sent and maximize their size. In addition, the computational load should be balanced for all processors.

We introduce the tag "mpi(p)" for message passing on p processors. The construct (10) expresses an embarrassingly parallel, load balanced operation on p processors and can be easily translated into a single program multiple data (SPMD) program. The construct

$$\underbrace{P \otimes \mathrm{I}_n}_{\text{all-to-all}}, \quad P \text{ a permutation on } mp \text{ points}, \tag{14}$$

expresses an explicit all-to-all communication between p processors with packets of size n. (10) and (14) are message passing base cases. The distributed memory rule set consists of (5) and rules like

$$\underbrace{\mathrm{L}^{np}_n}_{\mathrm{mpi}(p)} \quad \rightarrow \quad \big(\mathrm{I}_p \otimes_\parallel \mathrm{L}^n_{n/p}\big) \underbrace{\big(\mathrm{L}^{p^2}_p \otimes \mathrm{I}_{n/p}\big)}_{\text{all-to-all}} \tag{15}$$

and enables Spiral to generate efficient MPI code for linear transforms. Further details on Spiral's distributed memory code generation can be found in [3].

Cell processor. The Cell processor features a two-way symmetric multithreaded main processor (PPE) and 8 vector SIMD coprocessors (SPE) with private memory. It has a theoretical single-precision peak performance of 45 Pentium 4 processors running at the same clock speed. To take full advantage of the Cell processor's potential one needs to write vectorized multithreaded and explicit message-passing programs to be run on a single chip. Particularly important is the scheduling of on-chip communication as the bandwidth strongly depends on the units that communicate. To enable Spiral to generate programs for the Cell processor, we are porting and combining Spiral's vector SIMD, SMP and message passing capabilities to the Cell processor.

In addition, we tackle the communication scheduling by means of software pipelining. The construct (2) expresses a loop with m iterations. It ensures that during

computing iteration i, data for iteration $i-1$ can be sent to its consumer and data for iteration $i+1$ can be received from its producer. In addition, this data is contiguous. Together with (14) this allows Spiral to implement a software pipeline and to schedule messages. This is work in progress.

Graphics processors. The latest generation of graphics processors (GPUs) supports programmable pixel shaders, floating-point units, and high bandwidth to the main memory. The theoretical single-precision peak performance of Nvidia's GeForce 7900 GTX is 250 Gflop/s compared to the 15.2 Gflop/s peak performance of a 3.8 GHz Pentium 4. The advent of the C-based graphics language Cg makes it easier to program these powerful and ubiquitous special-purpose processors. Recent research shows that the newest GPUs can be used to perform linear algebra and DFT computations [11].

General purpose computation on GPUs utilizes their programmable shaders, which must be programmed in a "for all pixels do" SIMD paradigm. Shader programs can load multiple pixels (t-way vectors) from multiple textures (rectangular data arrays), and produce a limited number of output pixels (multi-target rendering). We define a tag "gpu(n,t)" for GPUs supporting n-way multi-texture rendering and t color values per pixel (typically, t=4). The construct

$$A_m \otimes_\times \mathrm{I}_k \,\vec{\otimes}\, \mathrm{I}_t$$

encodes "for all pixel do" loops. We cast the pixel shaders' SIMD programming model as a GPU base case,

$$\big(A_m \otimes_\times \mathrm{I}_k \,\vec{\otimes}\, \mathrm{I}_t\big)\big(P \vec{\otimes} \mathrm{I}_t\big), \quad m \le n,$$

and introduce rewriting rules like

$$\underbrace{A_m \otimes \mathrm{I}_{kt}}_{\mathrm{gpu}(n,t)} \to A_m \otimes_\times \mathrm{I}_k \,\vec{\otimes}\, \mathrm{I}_t\,.$$

Permutations must always be to the right of (i.e., precede) a "for all pixels do" loop. A full computation can contain multiple passes with different shader programs. This is work in progress.

4. EXPERIMENTAL RESULTS

Spiral generates highly optimized transform programs for a large class of advanced architectures. The generated code is consistently competitive with the best available vendor libraries and open source libraries like FFTW. Note that a vendor library like Intel's IPP requires many man-years of coding effort.

In the following we show a few example benchmarks.

SIMD vector code. Spiral is particularly successful in optimizing for vector SIMD extensions like Intel's SSE, and AltiVec/VMX supported by PowerPC G4, G5 and Cell processors [5, 4, 8]. We show the performance of Spiral generated vector code in Fig. 2: Complex 1D DFT on Intel Pentium 4, 3.6 GHz, 8-way SIMD (SSE2), 16-bit fixed point. Spiral is almost twice as fast as Intel's IPP 5.0.

SMP and multicore code. In Fig. 3 we compare sequential and parallel performance of Spiral generated

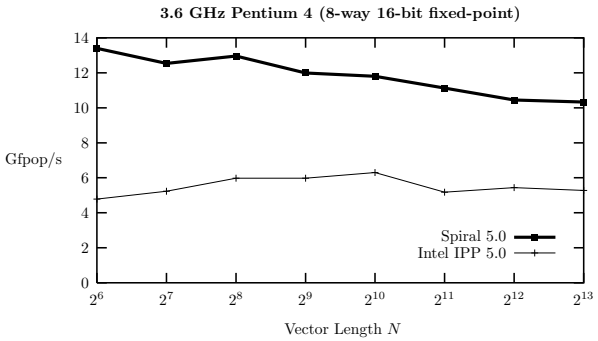

Figure 2: Vector SIMD. Performance of 1D DFTs, Spiral and Intel IPP 5.0. Higher is better.

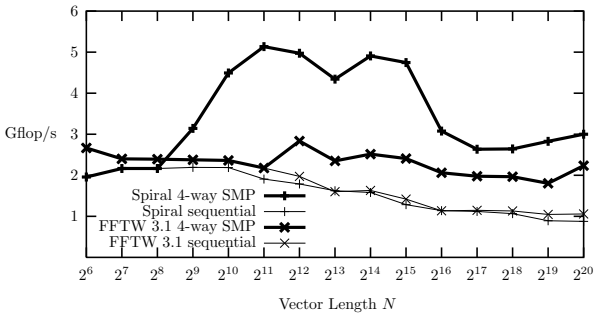

Figure 3: Multicore CPU. Performance of 1D DFTs, Spiral and FFTW 3.1. Higher is better.

1D complex double-precision DFT programs to FFTW 3.1 on a dual 2.2 GHz Opteron dualcore (4 processors) system, AMDs newest multicore processor. Spiral generated parallel code outperforms FFTW on this machine and makes the parallelization of very small FFTs possible [7]. Spiral provides consistent speed-up between 3x and 3.5x using the four processors. We see Spiral's sequential/parallel break even for DFT_{256}, which runs for less than 10,000 cycles and fits fully into the L1 cache. FFTW has its break even for DFT_{2048} and starts using all four processors only at $\mathrm{DFT}_{2^{20}}$; before, only two processors are used.

Message passing code. Fig. 4 shows the performance of Spiral-generated 1D complex double-precision DFT MPI code, run on an 8 node cluster of Opteron 2.4 GHz and Infiniband connectivity, compared to FFTW 2.1.5 MPI (FFTW's latest MPI version). In this setup, Spiral-generated code is between 25% and two times faster than FFTW 2.1.5 [3].

5. REFERENCES

[1] G. Baumgartner, A. Auer, D. E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. J. Harrison, S. Hirata, S. Krishanmoorthy, S. Krishnan, C.-C. Lam, Q. Lu, M. Nooijen, R. M. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov. Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. *Proceedings of the IEEE*,

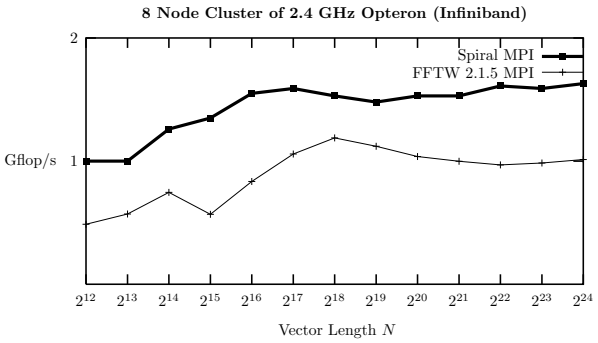

Figure 4: Message passing. Performance of 1D DFTs, Spiral and FFTW 2.1.5 MPI. Higher is better.

93(2):276–292, 2005. Special issue on "Program Generation, Optimization, and Adaptation".

[2] P. Bientinesi, J. A. Gunnels, M. E. Myers, E. Quintana-Orti, and R. van de Geijn. The science of deriving dense linear algebra algorithms. *ACM Transactions on Mathematical Software*, 31(1):1–26, March 2005.

[3] A. Bonelli, F. Franchetti, J. Lorenz, M. Püschel, and C. W. Ueberhuber. Automatic performance optimization of the discrete Fourier transform on distributed memory computers, 2006. Submitted for publication.

[4] F. Franchetti, S. Kral, J. Lorenz, and C. Ueberhuber. Efficient utilization of SIMD extensions. *Proceedings of the IEEE*, 93(2):409–425, 2005. Special issue on "Program Generation, Optimization, and Adaptation".

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

[6] F. Franchetti, Y. Voronenko, and M. Püschel. Formal loop merging for signal transforms. In *Proc. Programming Language Design and Implementation (PLDI)*, pages 315–326, 2005.

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

[8] 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.

[9] M. Frigo. A fast Fourier transform compiler. In *Proc. Programming Language Design and Implementation (PLDI)*, pages 169–180, 1999.

[10] M. Frigo and S. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Adaptation".

[11] N. K. Govindaraju, S. Larsen, J. Gray, and D. Manocha. Efficient memory model for scientific algorithms on graphics processors, 2006. UNC Tech. Report.

[12] J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn. Flame: Formal linear algebra methods environment. *TOMS*, 27(4):422–455, 2001.

[13] E.-J. Im, K. Yelick, and R. Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *Int'l Journal of High Performance Computing Applications*, 18(1), 2004.

[14] J. R. Johnson, R. W. Johnson, D. Rodriguez, and R. Tolimieri. A methodology for designing, modifying, and implementing FFT algorithms on various architectures. *Circuits Systems Signal Processing*, 9:449–500, 1990.

[15] M. Püschel, J. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005. Special issue on "Program Generation, Optimization, and Adaptation".

[16] Spiral website: http://www.spiral.net.

[17] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.