Domain-Specific Library Generation for Parallel Software and Hardware Platforms

Franz Franchetti[1], Yevgen Voronenko[1], Peter A. Milder[1], Srinivas Chellappa[1], Marek R. Telgarsky[1], Hao Shen[2], Paolo D'Alberto[3], Frédéric de Mesmay[1] James C. Hoe[1], José M. F. Moura[1], Markus Püschel[1]
[1]Carnegie Mellon University, [2]Technical University of Denmark, [3]Yahoo Inc.*

Abstract

We overview a library generation framework called Spiral. For the domain of linear transforms, Spiral automatically generates implementations for parallel platforms including SIMD vector extensions, multicore processors, field-programmable gate arrays (FPGAs) and FPGA accelerated processors. The performance of the generated code is competitive with the best available hand-written libraries.

1 Introduction

The development of high performance numerical libraries has become extraordinarily difficult and time consuming. The reason is that recent computing platforms have become increasingly complex and offer different forms and usually multiple levels of parallelism (such as vector processing or multiple cores). To make things worse, hardware platforms based on field-programmable gate arrays (FPGAs) have become mainstream as stand-alone platform or accelerator.

In this paper we show that for the performance critical domain of linear transforms, the library development can be automated including the mapping to different forms of parallelism, while at the same time matching the performance of hand-written code. The key ingredients of the framework are 1) a domain-specific, declarative, mathematical language to describe algorithms; and 2) the use of rewriting to parallelize and optimize algorithms at a high level of abstraction. The library generation system is called Spiral and builds upon an earlier version [9] with the same name.

Related work. Several other efforts have addressed the problem the problem of automating library development or performance optimization. ATLAS [11] 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. FFTW [5] is a library for the discrete Fourier transform (DFT), using a program generator for small codelets and heuristic search to compute larger DFTs based using breakdowns and codelets. Other examples of automatic tuning include [6] for sparse linear algebra and [1] for parallel tensor computations.

Except for [1], which targets clusters, the automation in the above work is restricted to sequential code and in all cases to software implementations. In contrast, our work includes automatic parallelization for different forms of parallelism and the mapping to software and hardware platforms.

2 Spiral

Spiral is a program generator for linear transforms including 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. The input to Spiral is a formally specified transform (e.g., DFT of size 245); the output is a highly optimized C program or Verilog design 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 or Verilog program (for computing the transform) using a special formula compiler. Structural optimization includes parallelization for different forms of parallelism. The performance of the generated code is measured or estimated and fed into a search engine, which decides how to modify the algorithm.

*This work was supported by NSF through awards 0325687, 0702386, and by DARPA through the Department of Interior grant NBCH1050009. This paper appeared in the NSF Next Generation Software Program workshop co-located with IPDPS 2008, Miami, Florida.

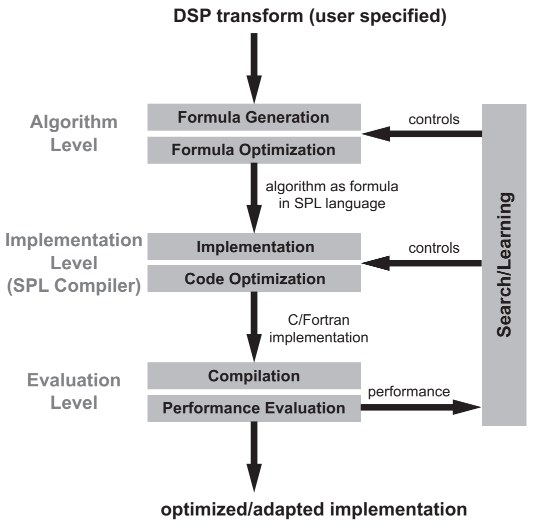

Figure 1. The program generator Spiral.

This means, the search 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. The basic architecture of the system was built in [9]. Here, we explain how we extended the system to parallel platforms including FPGAs. The key is the use of a declarative mathematical language as described next.

Declarative framework. 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

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

Algorithms for transforms are typically divide-and-conquer and can be viewed as factorizations of the transform into sparse, structured matrices [10]. The structure is exhibited through matrix operators like the product $\cdot$, the direct sum $\oplus$, and, most importantly the Kronecker product $\otimes$ defined as

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

Then, for example, the Cooley-Tukey FFT algorithm can be written as the *break-down rule*

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

with the identity matrix I_n, the diagonal matrix $D_{m,n}$ and the stride permutation matrix L_m^{nm}.

The matrix formula language, called SPL, is declarative: it describes the dataflow of computation and its structure. Spiral contains about 50 transforms and more than 200 breakdown rules.

3 Mapping Formulas to Parallel Platforms

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 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 [3]. However, (2) is ill-suited for vector SIMD architectures and (3) is ill-suited for parallel machines.

Further, formulas can be manipulated using algebraic identities [7] to change their structure. For instance, the identity

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

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 important 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 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, "stream(w)" for streaming on FPGAs, "partition(s, A_n)" for HW/SW partitioning, and "gpu(n,t)" for graphics processors. The meaning of the 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) with $\mathrm{I}_p \otimes_{\|} A_n$, using the tagged operator "$\otimes_{\|}$;" A_n is any $n \times n$ matrix expression.

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

$$\underbrace{A_m \otimes \mathrm{I}_n}_{\mathrm{smp}(p,\mu)} \rightarrow \underbrace{\mathrm{L}_m^{mn}}_{\mathrm{smp}(p,\mu)} \left(\mathrm{I}_p \otimes_{\|} (\mathrm{I}_{n/p} \otimes A_m)\right) \underbrace{\mathrm{L}_n^{mn}}_{\mathrm{smp}(p,\mu)}.$$

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

For every given platform or form of parallelism, we follow the same procedure to obtain the above components: 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, GPUs, FPGAs, and partitioning across embedded CPUs and FPGAs.

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 "$\mathrm{vec}(\nu)$" for vector SIMD operation using ν-way vector instructions. The construct

$$A \vec{\otimes} \mathrm{I}_\nu \tag{5}$$

(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 for every supported vector architecture. The construct (5) and the vectorized permutations constitute some of the base cases for vector architectures.

Vector rewriting rules like

$$\underbrace{A \otimes \mathrm{I}_n}_{\mathrm{vec}(\nu)} \to (A \otimes \mathrm{I}_{n/\nu}) \vec{\otimes} \mathrm{I}_\nu \tag{6}$$

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 [3].

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 "$\mathrm{smp}(p,\mu)$" for shared memory computation on p processors with cache line size μ. The constructs

$$\mathrm{I}_p \otimes_\| A \quad \text{and} \quad P \bar{\otimes} \mathrm{I}_\mu, \quad P \text{ a permutation}, \tag{7}$$

expresses embarrassingly parallel, load balanced operation on p processors and ownership change of cache lines, respectively; both can easily be translated into shared memory code. The shared memory rule set consists of rules like

$$\underbrace{A_m \otimes \mathrm{I}_n}_{\mathrm{smp}(p,\mu)} \to \big(\mathrm{I}_p \otimes_\| (A_m \otimes \mathrm{I}_{n/p})\big)^{\overbrace{\left(\mathrm{L}^{mp}_p \otimes \mathrm{I}_{n/p}\right)}_{\mathrm{smp}(p,\mu)}} \tag{8}$$

and allows to parallelize even small transforms and to produce efficient multicore code. (we use conjugation $A^P = P^{-1}AP$ in (8).) Further information on Spiral's shared memory and multicore code generation can be found in [4].

To enable Spiral to generate parallel programs for multiple SPUs of the Cell processor, we simulate a cache coherency protocol using DMA transfers, exchanging relatively small packets between SPUs whenever they are ready.

Graphics processors. Current graphics processors (GPUs) support programmable pixel shaders, floating-point units, and high bandwidth to the main memory. General purpose computation on GPUs is based on these shaders, which must be programmed in a "for all pixels do" SIMD paradigm using OpenGL or Cg. 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 "$\mathrm{gpu}(n,t)$" for GPUs supporting n-way multi-texture rendering and t color values per pixel (typically, t=4). 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

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

Streaming on FPGAs. The governing property on FPGAs is that data is streamed through the data path at a certain stream rate of w elements per cycle. Moreover, to limit resource usage reuse is required. Spiral supports *sequential reuse* (reuse of a computational block multiple times within the same problem) and *iterative reuse* (reuse within a cascade of identical blocks in the horizontal dimension).

We use a a tagged tensor product $\otimes_s$ to indicate streaming reuse. (9) shows a rule used to extract streaming reuse. The tag "stream(w)" instructs the system to restructure the formula to have w input and output ports.

$$\underbrace{\mathrm{I}_m \otimes A_n}_{\text{stream}(w)} \rightarrow \mathrm{I}_{m/w} \otimes_s (\mathrm{I}_{w/n} \otimes A_n), \quad n \leq w \tag{9}$$

In (9), w/n instances of A_n are built in parallel and reused over m/w cycles. More details on Spiral-generated hardware designs can be found in [8].

Accelerating software by FPGA. Current FPGAs contain embedded processors and thus are a target for FPGA-accelerated software. Spiral generates FPGA-accelerated software, using its hardware design and software tool chain. The hardware and software communicate through Spiral-generated glue code, utilizing the vendor-supplied interface between the embedded CPUs and the FPGA fabric.

(10) shows a rule used to perform the HW/SW partitioning. The tag "partition(s, A_m)" instructs the rule system that constructs A_m are supported in hardware and that the FPGA accelerator block A_m supports streaming of s independent calls.

$$\underbrace{\mathrm{I}_m \otimes A_m}_{\text{partition}(s,\, A_m)} \rightarrow \underbrace{\mathrm{I}_{m/s} \otimes}_{\text{SW}} \underbrace{\left(\mathrm{I}_s \tilde{\otimes} A_m\right)}_{\text{HW}} \tag{10}$$

(10) breaks a tensor product with too many independent calls into a software and hardware loop. More details on Spiral-generated software accelerated by hardware can be found in [2].

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 and MKL or Xilinx LogiCore requires many man-years of coding effort.

In the following we show a few example benchmarks focussing on the DFT due to space limitations.

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 [3]. We show the performance of Spiral generated vector code in Fig. 2 (a): Complex 1D DFT_n for all sizes $n = 2, 3, \ldots, 64$ on Intel Core2 Duo (single core), 2.66 GHz, 4-way SIMD (SSE2), single-precision. Spiral is much faster than Intel's MKL for all sizes.

SMP and multicore code. In Fig. 2 (b) we compare sequential and parallel performance of Spiral generated 1D complex single-precision DFT programs to FFTW 3.2 alpha and Intel's IPP 5.0 on a 3 GHz Core2 Extreme (4 cores). Spiral generated sequential code is on par with FFTW and IPP. Spiral-generated parallel code outperforms FFTW and IPP and makes the parallelization of very small FFTs ($n = 1024$) possible [4].

In Fig. 2 (c) we show the performance of single-precision WHTs on a 3.2 GHz Cell processor. We compare the performance on a single SPU and 4 SPUs, both are 4-way vectorized codes. Starting at $n = 2^{11}$ using 4 SPUs provides speed-up, reaching up to 3 times over a single SPU.

GPU. In Fig. 2 (d) we show the performance of single-precision WHTs on a 3.6 GHz Pentium 4 with a Nvidia 7900 GTX GPU. For small sizes 4-way-vectorized SSE code outperforms the GPU code, but from $n = 2^{16}$ on the CPU code suffers from memory bandwidth problems and the GPU code runs at its full potential. Combining both codes leads to high performance across all sizes.

FPGA. In Fig. 2 (e), we examine performance versus area for DFT 1024 cores generated for the Xilinx Virtex-5 FPGA platform. Cores are generated across a wide range of architectural options; the Pareto front consisting of the most efficient designs is presented here. Xilinx LogiCore designs offer similar cost and performance to our smaller cores, but provide very limited scalability.

FPGA-accelerated software. In Fig. 2 (f) we show the performance of FPGA-accelerated software on a Xilinx Virtex-II Pro FPGA with embedded PowerPC 405 core. We accelerate a whole DFT library of sizes $n = 2, 4, \ldots, 8192$ with two hardware cores. While the cores provide the strongest performance boost at their native size, other sizes are sped up considerably as well.

5 Conclusions

We showed that for an entire domain of structurally complex numerical kernels, the implementation for a wide range of platforms can be automated while still achieving highest performance. Even though our framework is domain-specific, we believe that its underlying principles can be applied to other numerical domains. Doing so is a main direction in our current research.

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*, 93(2), 2005. special issue on "Program Generation, Optimization, and Adaptation".

[2] P. D'Alberto, P. A. Milder, A. Sandryhaila, F. Franchetti, J. C. Hoe, J. M. F. Moura, M. Püschel, and J. Johnson.

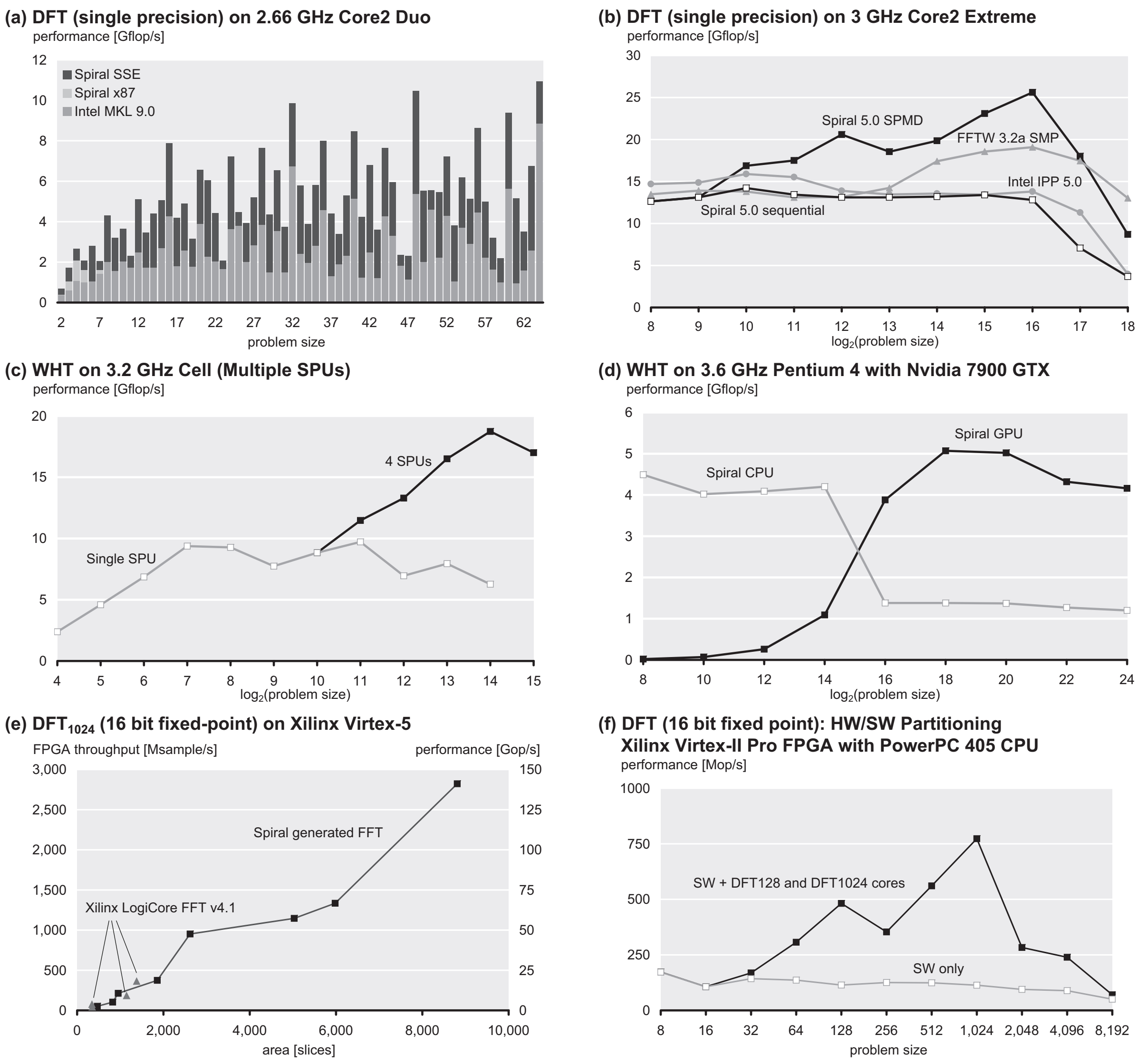

Figure 2. Selection of performance results for DFT and WHT. Higher is better.

Generating fpga accelerated dft libraries. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2007.

[3] 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)*, pages 58–67, 2003.

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

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

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

[7] 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.

[8] G. Nordin, P. A. Milder, J. C. Hoe, and M. Püschel. Automatic generation of customized discrete Fourier transform IPs. In *Design Automation Conference (DAC)*, pages 471–474, 2005.

[9] 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*, 93(2):232–275, 2005. Special issue on *Program Generation, Optimization, and Adaptation*.

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

[11] 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.