System Demonstration of Spiral: Generator for High-Performance Linear Transform Libraries

Yevgen Voronenko, Franz Franchetti, Frédéric de Mesmay, and Markus Püschel*

Department of Electrical and Computer Engineering
Carnegie Mellon University, 5000 Forbes Ave, Pittsburgh, PA 15213, USA,
{yvoronen,franzf,fdemesma,pueschel}@ece.cmu.edu

Abstract. We demonstrate Spiral, a domain-specific library generation system. Spiral generates high performance source code for linear transforms (such as the discrete Fourier transform and many others) directly from a problem specification. The key idea underlying Spiral is to perform automatic reasoning and optimizations at a high abstraction level using the mathematical, declarative domain-specific languages SPL and $\sum$-SPL and a rigorous rewriting framework. Optimization includes various forms of parallelization. Even though Spiral provides complete automation, its generated libraries run often faster than any existing hand-written code.

Key words: Linear transform, discrete Fourier transform, FFT, domain-specific language, program generation, rewriting, matrix algebra, automatic performance tuning, multithreading, SIMD vector instructions

1 Introduction

The advent of mainstream parallel platforms has made the development of high performance numerical libraries extremely difficult. Practically every off-the-shelf computer has multiple processor cores, SIMD vector instruction sets, and a deep memory hierarchy. Compilers cannot optimize numerical code efficiently, since the necessary code transformations often require domain knowledge that the compiler does not have. Consequently, the library developer is forced to write multithreaded code, use vector instructions through C language extensions or assembly code, and tune the algorithm to the memory hierarchy. Often, this process is repeated once a new platform is released. Automating high performance library development is a goal at the core of computer science.

Some advances have been made towards this goal, in particular in two performance-critical domains: linear algebra and linear transforms. One example is FFTW [1], a widely used library for the discrete Fourier transform (DFT). FFTW partially automates the development process, by using a special "codelet generator" [2] to generate code for small fixed size transform functions, called "codelets". However, all top-level recursive routines are still hand-developed and vectorization and parallelization are also performed manually.

* This work was supported by NSF through awards 0325687, 0702386, by DARPA through the DOI grant NBCH1050009 and the ARO grant W911NF0710416, and by an Intel grant.

We demonstrate Spiral, a system which takes domain-specific source code generation to the next level, by *completely* automating the library development process. Spiral enables the generation of the *entire* library, similar to FFTW, including the necessary codelet generator, given only a specification (in a domain-specific language) of the recursive algorithms that the library should use. Further, the library is vectorized and parallelized for highest performance. These capabilities extend our earlier work [3].

Even though Spiral achieves complete automation, the runtime performance of its generated libraries is often faster than any existing human-written code.

The framework underlying Spiral is built on two mathematical domain-specific languages, called SPL [4] (Signal Processing Language) and $\sum$-SPL [5]. These languages are derived from matrix algebra and used to represent and manipulate algorithms using rewriting systems. The rewriting is used to generate algorithm variants, to automatically parallelize [6] and vectorize [7] algorithms, and to discover and generate the library structure [8]. The latter includes the set of mutually recursive functions that comprise the library, and the set of required codelets.

2 Background

A linear transform is a matrix-vector multiplication $y = Mx$, where x, y are the input and output vectors, respectively, and M is the fixed transform matrix. For example, the DFT is given by the matrix $M = \mathbf{DFT}_n = \left[\omega_n^{k\ell}\right]_{0 \le k,\ell < n}$, with complex $\omega_n = e^{-2\pi i/n}$.

SPL. Many fast Fourier transform algorithms (FFTs) exist, and can be represented as factorizations of $\mathbf{DFT}_n$ into products of structured sparse matrices [9]. This representation forms the core of Spiral's domain-specific mathematical language SPL [4]. For example, the Cooley-Tukey FFT is a divide-and-conquer algorithm that for $n = km$ can be written as

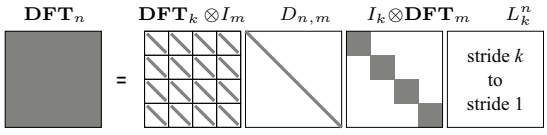
$$\mathbf{DFT}_n = (\mathbf{DFT}_k \otimes I_m) D_{n,m} (I_k \otimes \mathbf{DFT}_m) L^n_k. \tag{1}$$

Evaluating $y = \mathbf{DFT}_n\, x$ by successively multiplying x with factors of (1) reduces the overall arithmetic cost. Above, I_n is the $n \times n$ identity matrix, $D_{n,m}$ is a diagonal matrix, and L^n_k is a stride permutation matrix, which precise form is irrelevant here. Most important in this formalism is the tensor (or Kronecker) product $\otimes$ of matrices, defined as

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

Tensor products of the form $A \otimes I$ and $I \otimes A$ are special, because they naturally express loops with independent iterations and special data layouts.

(1) is called a *breakdown rule* in Spiral [3], it is best understood by visualizing the nonzero pattern of the factor matrices, done here for $k = m = 4$. In the leftmost factor, all the 1st, 2nd, ..., mth, entries of the small diagonals constitute one $\mathbf{DFT}_k$, respectively.



(2)

Recursive application of (1) for a two-power $n = 2^t$ yields an $O(n \log(n))$ algorithm, terminated by $\mathbf{DFT}_2$, which is computed by definition. For prime sizes other FFT algorithms are needed. Note that SPL is declarative: only the structure of the algorithm is described; not how exactly it is computed.

$\sum$-SPL . In order to generate looped code, we developed a lower-level representation, called $\sum$-SPL [5]. $\sum$-SPL like SPL is a structured sparse matrix factorization, however, it breaks down tensor products into iterative sums of products of smaller, rectangular matrices. Iterative sums serve as explicit representation of loops.

For example, if A is $n \times n$:

$$I_k \otimes A = \begin{bmatrix} A & & \\ & \ddots & \\ & & A \end{bmatrix} = \begin{bmatrix} A & & \\ & & \\ & & \end{bmatrix} + \cdots + \begin{bmatrix} & & \\ & & \\ & & A \end{bmatrix}$$

$$= \mathrm{S}_0 \, A \, \mathrm{G}_0 + \cdots + \mathrm{S}_{k-1} \, A \, \mathrm{G}_{k-1} = \sum_{j=0}^{k-1} \mathrm{S}_j \, A \, \mathrm{G}_j,$$

$$\mathrm{G}_j = [\quad I_n \quad] \quad (I_n \text{ in } j\text{th block}), \quad \mathrm{S}_j = \mathrm{G}_j^\top .$$

$\sum$-SPL admits several optimizations not possible with SPL, in particular it enables the merging of tensor products (loops) with permutations, which converts them into a readdressing of the input data.

3 Library Generation

The library generation process in Spiral is shown in Fig. 1. The input to the system is a set of transforms and associated breakdown rules. For example, it could be just $\mathbf{DFT}_n$ and (1). The process has two stages, library structure and library target, which we explain next. The output is the library implemented in C++.

Library structure. One main goal of the *library structure* stage is to determine the minimum set of mutually recursive functions that computes the given transforms. We call this set the *recursion step closure*, and each function is called a *recursion step*. Each recursion step is represented by a $\sum$-SPL formula. The original transform specification $\mathbf{DFT}_n$ is also a (trivial) $\sum$-SPL formula and a recursion step.

This stage generates formulas and optimizes them using rewrite rules, which among other things perform loop merging, vectorization and parallelization.

When a breakdown rule is applied to a transform it decomposes the transform into smaller transforms. Even if the smaller transforms are still DFTs (as in (1)), the $\sum$-SPL optimizations will merge these DFTs with additional operations (e.g. strided data loads and stores, scaling, etc.) thus changing the interface and creating new types of recursion steps. Breakdown rules applied to these new steps may spawn others. This process is continued until we find a finite set of mutually recursive recursion steps. This set is the recursion step closure.

As an example, Fig. 2 shows the recursion step closure generated for the DFT with breakdown rule (1). Four recursion steps are needed and the arrows capture the associated call graph.

In addition to the general size recursive implementations of recursion steps (which call other recursion steps), the library structure stage also generates fixed size *base*

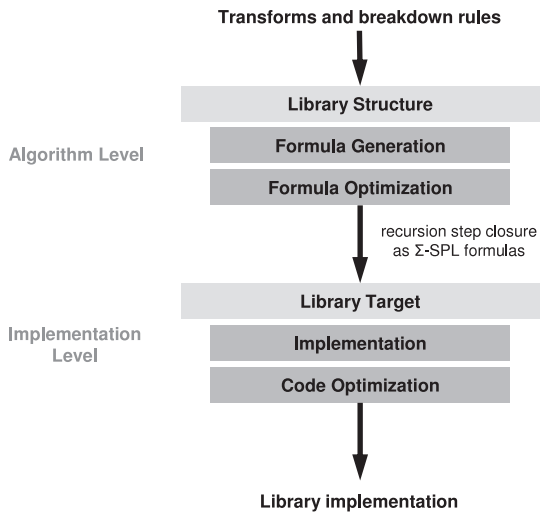


Fig. 1. Library generation process in Spiral

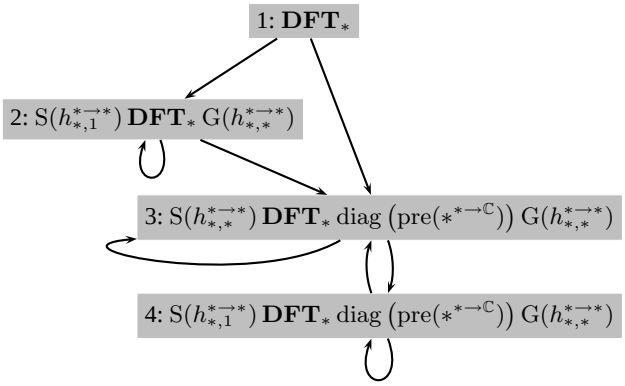


Fig. 2. Recursion step closure for $\mathbf{DFT}_n$, generated from (1), represented as a call graph. For readability, we replace all parameters of $\sum$-SPL formulas by "*".

case implementations. Each such base case is equivalent to a codelet in FFTW. The number of recursion step types with base cases is equivalent to the number of codelet types in FFTW. As Spiral discovers the codelet types automatically, it readily obtains the codelet generator, which becomes a call to the $\sum$-SPL compiler on the appropriate $\sum$-SPL formula with the known transform size inserted.

Library target. In this stage, the recursion step closure and $\sum$-SPL implementations are mapped to the target language C++. This stage must take care of generating auxiliary initialization code, which allocates temporary buffers, precomputes the necessary constants, and more.

The system can be used to generate code which extends an existing library. In this case, the auxiliary code must follow the specific library conventions, for example, for memory management.

After the initial code is generated, it is also optimized using a combination of rewrite rules and traditional compiler optimizations, such as constant propagation, common subexpression elimination, and loop unrolling.

Performance. The performance of two example libraries, generated using Spiral, is shown in Fig. 3 and compared to FFTW and Intel IPP (Integrated Performance Primitives) on a dual-core workstation. All compared libraries are 2-way vectorized and support threading, and the plots show maximum performance between 1 and 2 threads. While both FFTW and IPP achieve excellent performance for the DFT, they are less optimized for the less widely used DCT-2. Library generation, on the other hand, automates the tedious implementation and optimization process, and thus achieves uniform performance across a wide variety of transforms. The generated DFT library achieves a speedup over IPP and FFTW, due to using a specialized variant of (1) which reduces the number of vector shuffles. The generated DCT-2 library uses a "native" DCT-2 algorithm, instead of the suboptimal, but easy to implement, conversion to the DFT, used in FFTW and probably IPP.

4 Demonstration

We will demonstrate several key components of Spiral, including a live run of generating a fully vectorized and parallelized DCT-2 library. In detail, we will show:

- An example of formula generation and formula rewriting;
- Generation of straightline and looped code from a sample $\sum$-SPL formula;
- An example of code rewriting;
- Generation of the recursion step closure;
- Compilation of the recursion step closure into a library implementation.

5 Conclusions

Automating high performance library development is a problem at the core of computer science. We demonstrate a system that achieves this goal for the domain of linear transforms. The system is based on a set of techniques from different disciplines including linear algebra, algorithms, programming languages, generative programming, rewriting systems, and compilers. Properly applied, these techniques makes high-performance library generation feasible, efficient, and rigorous.

References

1. Frigo, M., Johnson, S.G.: The design and implementation of FFTW3. Proceedings of the IEEE **93**(2) (2005) 216–231 special issue on "Program Generation, Optimization, and Adaptation".
2. Frigo, M.: A fast Fourier transform compiler. In: Proc. PLDI. (1999) 169–180
3. 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., Rizzolo, N.: SPIRAL: Code generation for DSP transforms. Proceedings of the IEEE **93**(2) (2005) 232–275
4. Xiong, J., Johnson, J., Johnson, R., Padua, D.: SPL: A language and compiler for DSP algorithms. In: Proc. PLDI. (2001) 298–308

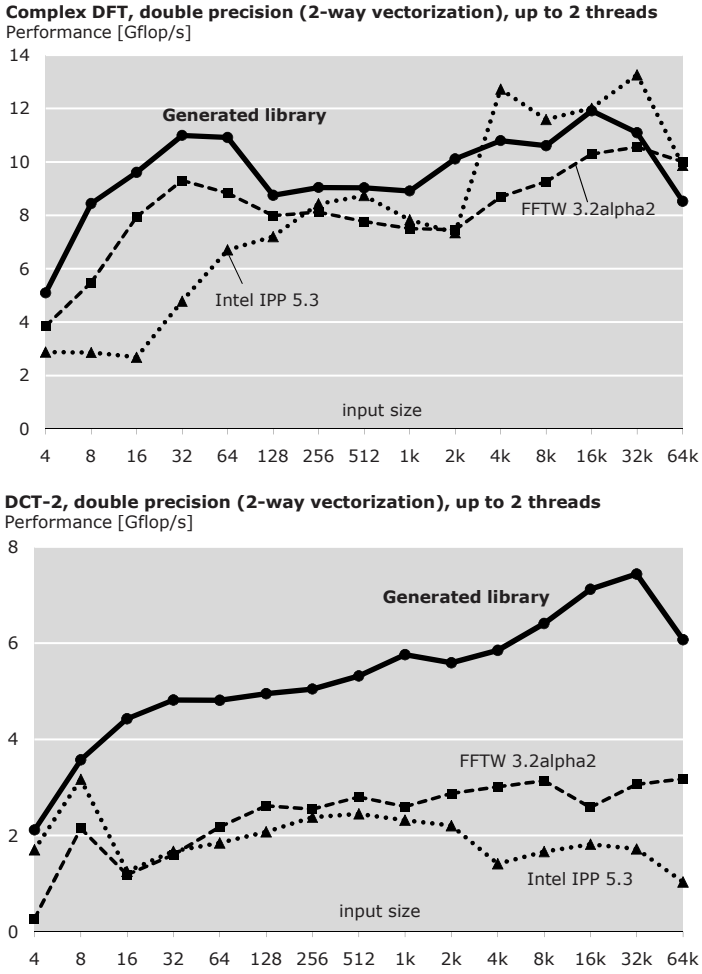


Fig. 3. Performance of automatically generated libraries compared to hand-written libraries (FFTW uses generated code for small fixed size transforms). Double precision, using SSE2 and up to 2 threads. Platform: dual-core 3 GHz Intel Xeon 5160 with 4 MB of L2 cache running Linux. Generated libraries are in C++ and are compiled with Intel C/C++ Compiler 10.1.

5. Franchetti, F., Voronenko, Y., Püschel, M.: Loop merging for signal transforms. In: Proc. PLDI. (2005) 315–326
6. Franchetti, F., Voronenko, Y., Püschel, M.: FFT program generation for shared memory: SMP and multicore. In: Proc. Supercomputing. (2006)
7. Franchetti, F., Voronenko, Y., Püschel, M.: A rewriting system for the vectorization of signal transforms. In: Proc. High Perf. Computing for Computational Science (VECPAR). (2006)
8. Voronenko, Y.: Library Generation for Linear Transforms. PhD thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University (2008)
9. Van Loan, C.: Computational Framework of the Fast Fourier Transform. SIAM (1992)