# AUTOMATIC GENERATION AND ADAPTATION OF NUMERICAL KERNELS

*Yevgen Voronenko*

Department of Electrical and Computer Engineering
Carnegie Mellon University, Pittsburgh, PA, U.S.A.

**ABSTRACT**

Designing software that achieves peak performance on modern architectures is a difficult, expensive and often highly platform specific task. In this paper we discuss recent automatic adaptive optimization approaches to high-performance programming: ATLAS, FFTW, and SPIRAL. They are designed to eliminate hand-coding and hand-tuning for various numerical kernels. Further, we describe our own work on the redesign of the SPIRAL system, which aims to generalize its code generation framework to support more general and more complicated kernels.

## 1. INTRODUCTION

The runtime of many scientific computations is dominated by a few compute intensive numerical kernels. Examples of such kernels are linear algebra computations, dense and sparse matrix-matrix and matrix-vector multiplication, and variants of the discrete Fourier transform.

Historically, the scientific community has relied on hardware vendors, who provided manually optimized kernel libraries for the given platform. However, as the processor microarchitecture and memory system evolve, the creation of high performance libraries is becoming an increasingly difficult and time-consuming process. Most importantly the optimization goal is shifting. While most of the earlier work on numerical algorithms concentrated on reducing the number of arithmetic operations, today the most important issue is memory locality. For example, a level 2 cache miss on a Pentium 4 may take more than 300 cycles, while a floating point multiplication can be executed every 2 cycles.

In this paper we first describe three systems that attack the problem of creating fast numerical kernels through automatic code generation and platform adaptation: 1) ATLAS [1], code generator for basic linear algebra routines; 2) FFTW [2, 3], an adaptive library for computing the discrete Fourier transform (DFT); and 3) SPIRAL [4], a code generator for linear signal transforms, including, but going beyond the DFT. Then we describe our own work on the redesign of SPIRAL to generalize its code generation and optimization capabilities. An example would be DFTs of large prime (or multiples of large primes) sizes, which are typically not efficiently implemented due to their algorithmic complexity [5].

The paper is organized as follows. Section 2 discusses ATLAS, FFTW, and SPIRAL in detail and compares the three approaches; Section 3 discusses the new SPIRAL design; and Section 4 presents conclusions and future research directions.

## 2. OVERVIEW OF ADAPTIVE LIBRARIES AND CODE GENERATORS

All systems we discuss share a common methodology: they use domain-specific methods to generate and optimize code, and use empirical search in a set of implementation alternatives to perform platform adaptation. Here give a brief overview of ATLAS, FFTW and SPIRAL, focusing on SPIRAL.

**ATLAS.** The domain of ATLAS [1] is Basic Linear Algebra Subroutines (BLAS). BLAS are not only very important as standalone kernels, but also form the basis of practically all linear algebra computations. BLAS routines are subdivided into three groups called “levels.” Level 1 includes vector-vector routines, level 2 matrix-vector routines, and level 3 matrix-matrix routines, such as matrix-matrix multiplication (MMM). Level 1 and level 2 routines have a fairly low computation / data ratio, and therefore are IO bound and fairly easy to optimize by choosing the correct blocking parameters. Level 3 routines are more difficult to optimize, since they perform more computations per data element and have more potential data reuse, which must be efficiently exploited. We focus our attention on MMM, the most important level 3 routine.

ATLAS uses three techniques to adapt MMM code to the underlying platform: parametrization, code generation, and multiple implementation.

A straightforward blocked $O(n^3)$ MMM algorithm is used in ATLAS as the top-level algorithm that reduces MMM to so-called Mini-MMMs, which perform matrix-matrix multiplication of the blocks. Parametrization involves introducing variable implementation parameters; at the top-level MMM these include the block (or tile) size. Code generation is used to automatically generate optimized Mini-

MMM code. ATLAS code generator unrolls the loops of Mini-MMM, to exploit additional low-level optimizations, such as operation scheduling, blocking for registers, scalar replacement of array references, and others. Finally, with multiple implementation, ATLAS chooses between different implementations of MMM and Mini-MMM. At the top level it has the choice to use MMM with block row-major or column-major loop order. At the Mini-MMM level, ATLAS might choose its own generated kernel or a contributed hand-written implementation.

At install time the system determines the best top-level blocking parameters, top-level loop order, and scheduling parameters for the code generator using runtime feedback-driven search. In addition, it times all contributed Mini-MMMs to compare against the generated code. If a contributed Mini-MMM outperforms the generated code, the contributed routine is used instead.

Reference [6] developed performance models that enable computing the parameters for MMM and Mini-MMM from architectural information, such as the cache sizes and the number of registers without using search. The authors show that the MMM code generated this way (without search) matches the performance of ATLAS generated code (with search).

An interesting fact pointed out by [6] is that ATLAS performs compiler optimizations that have been known to the research community for years, and most have been implemented in various general purpose high performance C and Fortran compilers. In particular, blocking for cache locality, hand-coded in the parameterized top-level MMM algorithm, and automatically generated by the Mini-MMM code generator, is achieved using a general compiler optimization known as loop tiling [7]. Nevertheless, no general purpose compiler was shown to match the performance of ATLAS generated code.

**FFTW.** While ATLAS uses a single parametrized blocked MMM algorithm, FFTW [2] has a mechanism to enumerate a large space of different recursive algorithms, called "plans," for computing the DFT.

Each step of the plan is a recursion step in the conventional program. However, FFTW has several choices in picking each recursion step, and these choices affect the memory locality of the algorithm and to some extent the operations count, and thus the performance. FFTW thus uses these choices to adapt to the host platform using search.

For smaller size DFTs, FFTW uses a special-purpose compiler, described in [3], to generate fast fully unrolled basic blocks called *codelets* for small DFT sizes, which are used in the plans for large DFT sizes. In addition to standard optimizations, the compiler performs highly efficient strength reduction optimizations that reduce the number of floating point operations, and aggressively schedules for register locality using a DFT-specific scheduling approach.

While the special purpose compiler ensures high performance for small DFT sizes, the search-based planner finds the best recursion strategy for large DFT sizes.

An important difference between MMM and DFT is that MMM has an extremely regular and well understood structure, with blocking and other parameters mapping directly to the microarchitectural parameters, while the degrees of freedom in the DFT do not correspond directly to the parameters of the microarchitecture.

**SPIRAL.** SPIRAL [4] is a standalone code generator for discrete linear signal transforms, such as the DFT. While in both ATLAS and FFTW only the small kernels are generated, and the top-level algorithm is hand-written, SPIRAL does not use any hand-written code at all. Instead, SPIRAL generates code for the entire transform from scratch. In addition to the DFT, the current version can generate code for FIR filters, wavelets, discrete cosine and sine transforms, and many other transforms.

In SPIRAL, the DFT is called a *transform*, and $\mathrm{DFT}_8$ is an example of a *transform instance*, i.e., the DFT of size 8. All linear signal transforms are uniquely defined by some matrix $M$, and transforming an input signal $x$ into the output signal $y$ corresponds to the matrix-vector multiplication $y = Mx$. For example, the DFT is defined by the $n \times n$ matrix

$$\mathrm{DFT}_n = \left[\omega_n^{k\ell}\right]_{0\le k,\ell<n}, \quad \omega_n = e^{-2\pi\sqrt{-1}/n},$$

and computing the DFT of an $n$-dimensional input vector $x$ corresponds to the matrix vector product $y = \mathrm{DFT}_n\, x$.

Transforms supported by SPIRAL have *fast algorithms* that reduce the cost of the matrix-vector product from the general $O(n^2)$ to typically $O(n \log n)$. Fast algorithms are captured in SPIRAL using recursive matrix *decomposition rules*. For example,

$$\mathrm{DFT}_{mk} \Rightarrow (\mathrm{DFT}_m \otimes \mathrm{I}_k) D (\mathrm{I}_m \otimes \mathrm{DFT}_k) P \qquad (1)$$

is a DFT decomposition rule that corresponds to the well known Cooley-Tukey algorithm written in the notation introduced by Van Loan in [8]. In the rule above $D$ denotes a certain diagonal matrix, $P$ a permutation, and $\mathrm{I}_n$ the $n \times n$ identity matrix. The operator $\otimes$ denotes the tensor or Kronecker product of matrices, defined by

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

After applying the rule (1) once, the problem of computing the DFT of an input vector $x$ is reduced to 4 steps corresponding to the 4 factors in (1). First, $x$ is permuted according to $P$, then the $\mathrm{DFT}_k$ is performed on $m$ subvectors, the result is scaled by the diagonal $D$, and finally the $\mathrm{DFT}_m$ is performed on $k$ subvectors.

We call the right hand side of rule (1) a *formula*. Each application of a recursive decomposition rule to the original transform yields a formula with reduced arithmetic cost, and the decomposition is applied until no more transforms appear in the formula. We call such formula a *fully expanded*

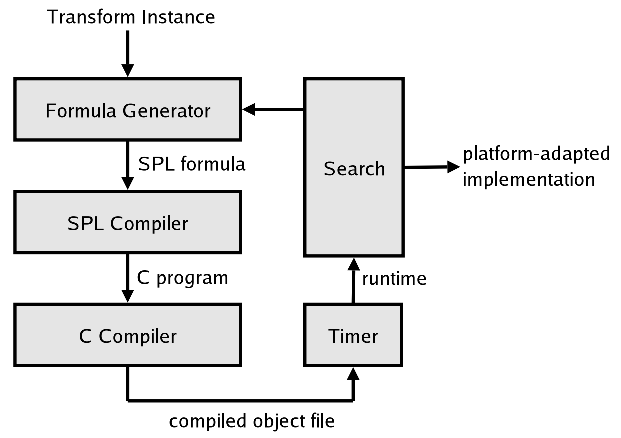

**Fig. 1**. Adaptive code generation in SPIRAL.

formula. Fully expanded formulas are obtained by eventually applying base case rules; the base case rule for the DFT is simple: $\mathrm{DFT}_2 \Rightarrow \left[\begin{smallmatrix} 1 & 1 \\ 1 & -1 \end{smallmatrix}\right]$.

The decomposition rules and the formulas are represented in SPIRAL using a declarative domain-specific language called SPL, which supports operators such as $\otimes$, and also has a mechanism for defining symbols for diagonals, permutations, and other structured matrices.

Figure 1 shows how SPIRAL performs platform adaptation using the formula framework described above. After the user specifies a transform instance to be implemented, the *Search* module starts generating alternative SPL formulas for the transform using the *Formula Generator* and the decomposition rules for the specified transform. Each SPL formula is compiled with the *SPL compiler* into a C program, which is subsequently compiled with a C compiler and timed. The *Search* module uses the runtime feedback to guide the formula generation process. This closes the feedback loop.

To date, several search strategies have been implemented in SPIRAL. The most effective strategies are dynamic programming and evolutionary (genetic) search. These are described in detail in [4].

## 3. REDESIGN OF SPIRAL

The original SPIRAL system as described in [4] has several limitations. First, in many cases only the straightline code achieves high performance, limiting several transforms to small sizes only. Second, it is usually possible to reduce the memory footprint of the generated code by overwriting the input vector with the output data, this type of code is called *inplace*. SPIRAL is not able to generate inplace code. Finally, the system can only generate code for a prespecified transform size, while often it is useful to have a package that can compute a transform of arbitrary size not known a priori. In this section we discuss the transform size limitation in more detail, then outline the redesign of the system, show how it solves the small size limitation, and briefly describe how the new design addresses other limitations.

**Large transform sizes.** Recall the Cooley-Tukey DFT decomposition rule (1). Straightforward conversion of this formula to code leads to a program with four loops that traverse the entire input vector, as described in Section 2. However, the equivalent of this recursive decomposition handwritten in FFTW uses only two loops, merging the diagonal and the permutation with the computation of the smaller DFTs. Using only two loops enables more data reuse and instruction level parallelism, and reduces memory traffic. The permutation is combined with the computation by readdressing the input, and therefore no data movement is performed. This problem only manifests itself for large transform sizes, because the loops in the code for small transforms are fully unrolled. As the size of the transform increases, more and more recursive steps are required, and since typically each step introduces a permutation, the inefficiency is aggravated.

Original SPIRAL [4] solved this problem by using *templates* to recursively match the common case SPL subformulas and produce optimized code with merged loops. However, these templates had to be handwritten for a large number of possible subformulas, and for many transforms the necessary templates were missing. In the new system we fuse the redundant loops with the computation fully automatically without using hand-written templates.

**$\sum$-SPL[1] and the new design.** In the redesigned architecture we introduce an intermediate formula language called $\sum$-SPL, described in [5]. $\sum$-SPL represents an algorithm at an abstraction level below SPL, but above the actual code. $\sum$-SPL makes the loop structure explicit while still allowing to perform optimization at the formula level. We use a powerful rewriting engine to perform code generation.

Figure 2 shows all of the steps involved in generating code for a given transform instance in the redesigned SPIRAL, i.e., these steps correspond to the Formula Generator and the SPL Compiler blocks in Figure 1. Initially, like in the original system, the decomposition rewrite rules are used to obtain a fast DFT algorithm as a fully expanded SPL formula. Afterwards, the SPL formula is converted to a $\sum$-SPL formula using the rewrite rules shown in [5]. For example, translating the right-hand side of (1) into $\sum$-SPL yields

$$\left(\sum_{i=0}^{k-1} \mathrm{S}_{f_i}\, \mathrm{DFT}_n\, \mathrm{G}_{f_i}\right) \mathrm{Diag}_d \left(\sum_{j=0}^{n-1} \mathrm{S}_{g_j}\, \mathrm{DFT}_k\, \mathrm{G}_{g_j}\right) \mathrm{Perm}_p,$$

where $\mathrm{G}_{f_i}$ denotes an $n \times nk$ *gather* matrix, which loads or gathers $n$ input values out of $nk$ using a certain index mapping function $f_i$, $\mathrm{S}_{f_i}$ denotes an $nk \times n$ *scatter* matrix which stores or scatters $n$ values into an output vector of size

[1]Pronounced as "Sigma-SPL".

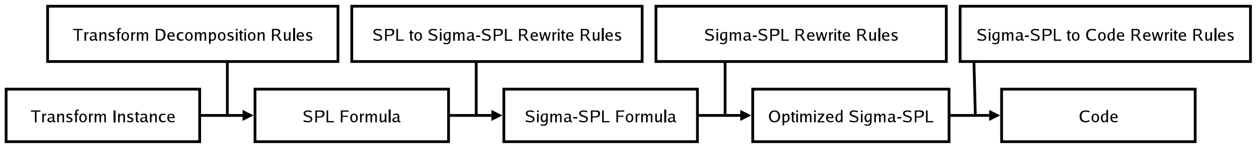

**Fig. 2**. Data flow of the redesigned SPIRAL.

$nk$ using the same index mapping function $f_i$. Further, the diagonal is represented in terms of its generating function $d$, and permutation in terms of the index mapping function $p$.

Consequently, we use rewriting rules to merge redundant loops and perform other optimizations of the $\sum$-SPL formula to obtain the optimized $\sum$-SPL formula. For example, the formula above after the optimization becomes

$$\left(\sum_{i=0}^{k-1} \mathrm{S}_{f_i}\,\mathrm{DFT}_n\,\mathrm{Diag}_{d\circ f_i}\,\mathrm{G}_{f_i}\right)\left(\sum_{j=0}^{n-1} \mathrm{S}_{g_j}\,\mathrm{DFT}_k\,\mathrm{G}_{p\circ g_j}\right),$$

which will have only two loops in the code, assuming that both $n$ and $k$ are smaller than the unrolling threshold (otherwise, the optimization can proceed on the formulas for smaller DFTs). Observe, that the permutation was incorporated into the gather operation yielding the composite index mapping function $p \circ g_j$, where $\circ$ denotes function composition. The diagonal is interchanged with the subsequent gather, by permuting its entries. More details and the used rewrite rules can be found in [5]. The final step uses rewrite rules convert the optimized $\sum$-SPL formula to code.

**Results.** Using $\sum$-SPL we can overcome the deficiencies of looped code in the original system, without handwriting a large number of templates. Further, in $\sum$-SPL it is possible to express both inplace and out-of-place computations. As in the case of loop merging, transformations on the $\sum$-SPL level are used to obtain an inplace formula, and subsequently generate inplace code. Finally, the enhanced symbolic computation capabilities enabled with rewriting allow us to generate a recursive routine for a transform with size as a parameter.

Neither FFTW nor ATLAS address the loop code generation problem directly, since both systems can only generate fully unrolled code for smaller subproblems, and use hand-written top-level algorithms. Our approach, however, is general enough to cover the more complicated Rader's algorithm for the prime-size DFT, and the prime-factor DFT algorithm. In FFTW, the complicated permutation used in the Rader's algorithm is not merged with the computation, making it a good candidate for comparison with SPIRAL generated code with automatically merged loops. In this case, using the same algorithm, SPIRAL achieves speedups of a factor of 2-3 over FFTW. Reference [5] gives further runtime results.

## 4. CONCLUSIONS AND FUTURE WORK

Our goal is to evolve SPIRAL into a user-friendly well-designed code generation system that works for all transforms, sizes, and various code types. We believe that $\sum$-SPL is a step in the right direction, as it extends the mathematical SPL language with code-like idioms, and thus enables optimizations, which are prohibitively expensive at the code level, to be performed at the formula level, Further, optimizing the formula directly allows easy verification using the methods described in [4], and also enables us to inspect the optimized formula to understand the generated loop structure, without looking at the potentially very long code sequences.

Efficient code for large sizes of several currently supported transforms still can not be generated, because more general loop merging techniques are required. However, this does not pose a significant challenge, since the optimization is performed at a very high level of abstraction, and therefore is much easier to implement than in a general purpose compiler.

Our long-term goal is to use SPIRAL to generate libraries like FFTW. optionally customized to different subsets of functionality, different memory footprints, or other requirements.

## 5. REFERENCES

[1] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, C. Whaley, and K. Yelick, "Self adapting linear algebra algorithms and software," *Proceedings of the IEEE*, vol. 93, no. 2, 2005.

[2] M. Frigo and S.G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, 2005.

[3] M. Frigo, "A fast fourier transform compiler," in *Proc. PLDI'99*, 1999.

[4] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, "SPIRAL: Code generation for DSP transforms," *Proceedings of the IEEE*, vol. 93, no. 2, 2005.

[5] F. Franchetti, Y. Voronenko, and M. Püschel, "Formal loop merging for signal transforms," 2005, to appear in *PLDI'05*.

[6] K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, and P. Stodghill, "Is search really necessary to generate high-performance BLAS?," *Proceedings of the IEEE*, vol. 93, no. 2, 2005.

[7] M. Wolfe, "More iteration space tiling," in *Supercomputing '89: Proceedings of the 1989 ACM/IEEE conference on Supercomputing*. 1989, pp. 655–664, ACM Press.

[8] C. F. Van Loan, *Computational frameworks for the Fast Fourier Transform*, SIAM, 1992.