TITLE

Spiral

BYLINE

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

SYNONYMS

none

DEFINITION

Spiral is a program generation system (software that generates other software) for linear transforms and an increasing list of other mathematical functions. The goal of Spiral is to automate the development and porting of performance libraries. Linear transforms include the discrete Fourier transform (DFT), discrete cosine transforms, convolution, and the discrete wavelet transform. The input to Spiral consists of a high-level mathematical algorithm specification and selected architectural and microarchitectural parameters. The output is performance-optimized code in a high-level language such as C, possibly augmented with vector intrinsics and threading instructions.

DISCUSSION

Introduction

The advent of computers with multiple cores, SIMD (single-instruction multiple-data) vector instruction sets, and deep memory hierarchies has a dramatic effect on the development of high performance software. The problem is particularly apparent for functions that perform mathematical computations, which form the core of most data or information processing applications. Namely, on a current workstation the performance difference between a straightforward implementation of an optimal (minimizing operations count) algorithm and the fastest possible implementation is typically 10–100 times.

As an example consider Fig. 1 which shows the performance (in gigafloating point operations per second) of four implementations of the discrete Fourier transform for varying input sizes on a quadcore Intel Core i7. Each on uses a fast algorithm with roughly the same operations count. Yet the difference between the slowest and fastest is 12–35 times. The bottom line is the code from Numerical Recipes [20]. The best standard C code is about 5 times faster due to memory hierarchy optimizations and constant precomputation. Proper use of explicit vector intrinsics instructions yields another 3 times. Explicit threading for the four cores, properly done, yields another 3 times for large sizes.

The plot shows that the compiler cannot perform these optimizations as is true for most mathematical functions. The reason is in both the compiler's lack of domain-knowledge needed for the necessary transformations and the large set of optimization choices with uncertain outcome that the compiler cannot assess. Hence the optimization task falls with the programmer and requires considerable skill. Further,

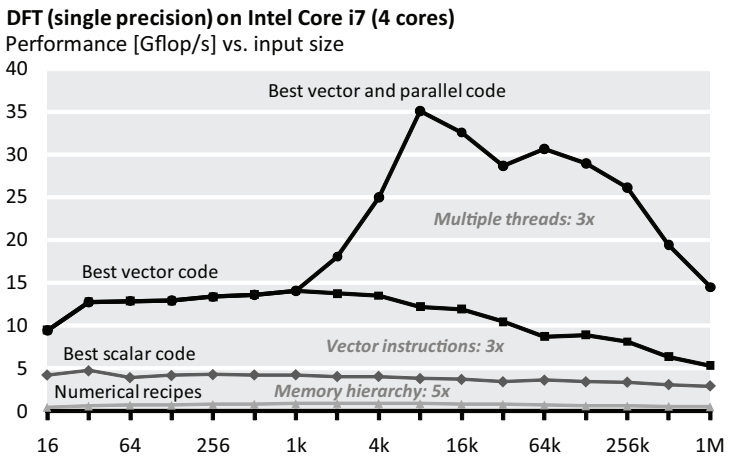


Figure 1: Performance of different implementations of the discrete Fourier transform (DFT) and reason for the performance difference (figure from [10]).

the optimizations are usually platform-specific and hence have to be repeated with every new generation of computers.

Spiral overcomes these problems by completely automating the implementation and optimization process for the functions it supports. Complete automation means that Spiral produces source code for a given function given only a very high-level representation of the algorithms for this function and a high-level platform description. After algorithm and platform knowledge are inserted, Spiral can generate various types of code including for fixed and general input size, threaded or vectorized.

The approach taken by Spiral is based on the following key principles:

- Algorithm knowledge for a given mathematical function is represented in the form of *breakdown rules* in a *domain-specific language*. Each rule represents a divide-and-conquer algorithm. The language is based on mathematics, is declarative, and platform independent. These properties enable the mapping to various forms of parallelism from algorithm knowledge that is inserted only once. It also enables the derivation of the library structure for general input size implementations by computing the so-called *recursion step closure*.

- Platform knowledge is organized into *paradigms*. A paradigm is a feature of a platform that requires structural optimization and possibly source code extensions. Examples include shared-memory parallelism or SIMD vector processing. Each paradigm consists of a set of *parameterized rewrite rules* and *base cases* expressed in the same language as the algorithm knowledge. The base cases constitute a subset of the domain-specific language that maps well to a paradigm. The rewrite rules interact with the breakdown rules to produce algorithms that are base cases, which means they are structurally optimized for the considered paradigm. Examples of parameters include the SIMD vector length or the cacheline size. Paradigms are designed to be composable.

- Spiral uses *empirical search* to automatically explore choices in a feedback loop. This is done by generating candidate implementations and evaluating their performance. Even though theoretically unsatisfying, search enables further optimization for intricate microarchitectural details that may be unknown or are not well understood.

In summary, Spiral integrates techniques from mathematics, programming languages, compilers, automatic performance tuning, and symbolic computation. The entire system Spiral combines aspects of a compiler, generative programming, and an expert system.

The remainder of this section describes the framework underlying Spiral and the inner workings of the actual system. The presentation focuses on linear transforms; extensions of Spiral beyond transforms are briefly discussed in the end.

Algorithm Representation

Linear transforms. A linear transform is the function

$$x \mapsto Mx,$$

where M is a fixed matrix, x is the input vector, and $y = Mx$ the output vector. Different transforms correspond to different matrices M. For simplicity, M is referred to as transform in the following. Most transforms M are square $n \times n$, which implies that x and y are of length n. Most transforms exist for all $n = 1, 2, \ldots$.

The possibly most famous transform is the DFT defined by the $n \times n$ matrix

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

Other examples include the discrete Hartley transform,

$$\mathbf{DHT}_n = \left[\cos(2\pi k\ell/n) + \sin(2\pi k\ell/n)\right]_{0 \le k,\ell < n},$$

the discrete cosine transform (DCT) of type 2,

$$\mathbf{DCT\text{-}2}_n = \left[\cos(k(\ell + \tfrac{1}{2})\pi/n)\right]_{0 \le k,\ell < n},$$

as well as other types of discrete cosine and sine transforms, the Walsh-Hadamard transform, the real DFT, the discrete wavelet transform, the inverses and other variants of the preceding transforms, and finite impulses response filters.

Fast transform algorithms: SPL. If M is $n \times n$ and has few or no zero entries, then a direct computation of $y = Mx$ requires $O(n^2)$ many operations. However, all the transforms mentioned above have fast algorithms that reduce their complexity below that, typically to $O(n \log(n))$. Every algorithm can be expressed as a factorization of the transform matrix M into a product of sparse matrices. As an example assume $M = M_1 M_2 M_3 M_4$, then $y = Mx$ can be computed in four steps as

$$t = M_4 x,\ u = M_3 t,\ v = M_2 u,\ y = M_1 v.$$

If the M_i are sufficiently sparse, this reduces the operations count.

The sparse matrices occurring in transform algorithms have structure that can be formally expressed using basic matrices and matrix operators such as the direct sum and the tensor or Kronecker product. This notation forms the basis for the language SPL (signal processing language) explained next.

Basic matrices include the $n \times n$ identity matrix I_n, diagonal matrices $D_n = \mathrm{diag}(a_0, \ldots, a_{n-1})$, the 2×2 butterfly matrix

$$F_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix},$$

the stride permutation matrix L^n_k, defined for $n = km$ by the underlying permutation

$$\ell^n_k :\ im + j \mapsto jk + i,\ 0 \le i < k,\ 0 \le j < m, \tag{1}$$

and several others.

⟨spl⟩	::=	⟨generic⟩ | ⟨basic⟩ | ⟨transform⟩ |	
		⟨spl⟩ · · · · ⟨spl⟩ |	(product)
		⟨spl⟩ ⊕ . . . ⊕ ⟨spl⟩ |	(direct sum)
		⟨spl⟩ ⊗ · · · · ⊗⟨spl⟩ |	(tensor product)
		. . .	
⟨generic⟩	::=	$\mathrm{diag}(a_0, \ldots, a_{n-1})$ | . . .	
⟨basic⟩	::=	$I_n \mid L^n_k \mid F_2 \mid \ldots$	
⟨transform⟩	::=	$\mathbf{DFT}_n \mid \mathbf{DHT}_n \mid \mathbf{DCT\text{-}2}_n \mid \ldots$	

Table 1: A subset of SPL in Backus-Naur form; n, k are positive integers, a_i are real or complex numbers.

Matrix operators include the matrix product, the direct sum

$$A \oplus B = \begin{bmatrix} A & \\ & B \end{bmatrix},$$

and the tensor product

$$A \otimes B = [a_{k,\ell}B]_{0 \le k,\ell < n}, \quad \text{for } A = [a_{k,\ell}]_{0 \le k,\ell < n}.$$

Most important are the tensor products where A or B are the identity:

$$I_n \otimes B = \begin{bmatrix} B & & \\ & \ddots & \\ & & B \end{bmatrix},$$

and, for example,

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \otimes I_3 = \begin{bmatrix} aI_3 & bI_3 \\ cI_3 & dI_3 \end{bmatrix} = \begin{bmatrix} a & & & b & & \\ & a & & & b & \\ & & a & & & b \\ c & & & d & & \\ & c & & & d & \\ & & c & & & d \end{bmatrix}$$

A (partial) description of SPL in Backus-Naur form is provided in Table 1.

Algorithms as SPL breakdown rules. Using SPL, the algorithm knowledge in Spiral is captured by *breakdown rules*. A breakdown rule represents a one-step divide-and-conquer algorithm of a transform. This means the transform is factorized into sparse matrices involving other, typically smaller, transforms.

The most famous example is the general-radix Cooley-Tukey fast Fourier transform (FFT):

$$\mathbf{DFT}_n \to (\mathbf{DFT}_k \otimes I_m) T^n_m (I_k \otimes \mathbf{DFT}_m) L^n_k, \quad n = km, \tag{2}$$

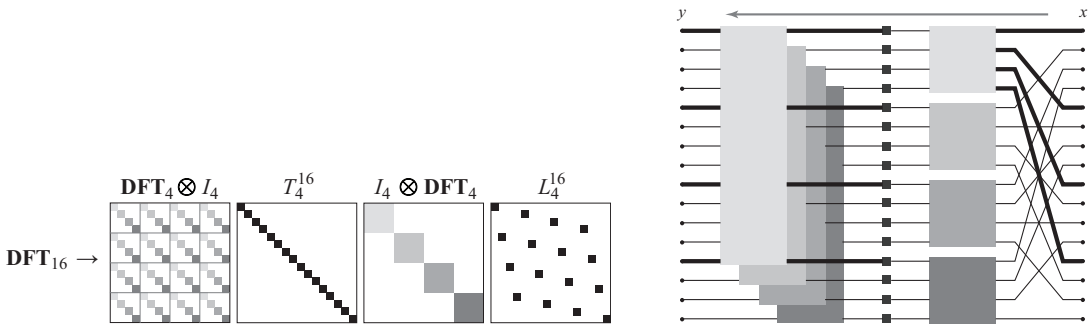
where T^n_m is the diagonal matrix of *twiddle factors*. For $n = 16 = 4 \times 4$, the factorization is visualized in Fig. 2 together with the associated dataflow graph. The smaller $\mathbf{DFT}_4$'s are boxes of equal shades of gray.

To terminate the recursion, base cases are needed. For example, for two-powers n a size two base case is sufficient:

$$\mathbf{DFT}_2 \to F_2. \tag{3}$$

A few things are worth noting about this representation of transform algorithms:

- The representation (2) is *point-free*, i.e., the input vector is not present.
- The representation (2) is declarative.
- Since the rule (2) is a matrix equation, it can be manipulated using matrix identities. For example, both sides can be inverted or transposed, to obtain an inverse or transposed transform algorithm.



(a) Matrix factorization (b) Data-flow graph

Figure 2: Cooley-Tukey FFT (2) for $16 = 4 \times 4$ as SPL rule and as (complex) data-flow graph (from right to left). Some lines are bold to emphasize the strided access of the $\mathbf{DFT}_4$'s (figure from [10]).

$\mathbf{DFT}_n$	$=$	$(\mathbf{DFT}_k \otimes I_m) T^n_m (I_k \otimes \mathbf{DFT}_m) L^n_k,$	(Cooley-Tukey FFT)	$n = km$
$\mathbf{DFT}_n$	$=$	$V_n^{-1} (\mathbf{DFT}_k \otimes I_m)(I_k \otimes \mathbf{DFT}_m) V_n,$	(Prime-factor FFT)	$n = km,\ \gcd(k,m) = 1$
$\mathbf{DFT}_n$	$=$	$W_n^{-1} (I_1 \oplus \mathbf{DFT}_{p-1}) E_n (I_1 \oplus \mathbf{DFT}_{p-1}) W_n,$	(Rader FFT)	n prime
$\mathbf{DFT}_n$	$=$	$B'_n D_m \, \mathbf{DFT}_m \, D'_m \, \mathbf{DFT}_m \, D''_m B_n,$	(Bluestein FFT)	$n > 2m$
$\mathbf{DFT}_n$	$=$	$P^\top_{k,2m} \left(\mathbf{DFT}_{2m} \oplus \left(I_{k-1} \otimes_i C_{2m} \, \mathbf{rDFT}_{2m,i/2k} \right) \right) (\mathbf{RDFT}_{2k} \otimes I_m),$		$n = 2km$
$\mathbf{RDFT}_n$	$=$	$(P^\top_{k,m} \otimes I_2) \left(\mathbf{RDFT}_{2m} \oplus \left(I_{k-1} \otimes_i D_{2m} \, \mathbf{rDFT}_{2m,i/2k} \right) \right) (\mathbf{RDFT}_{2k} \otimes I_m),$		$n = 2km$
$\mathbf{rDFT}_{n,u}$	$=$	$L^{2n}_m \left(I_k \otimes_i \mathbf{rDFT}_{2m,(i+u)/k} \right) (\mathbf{rDFT}_{2k,u} \otimes I_m),$		$n = 2km$

Table 2: A selection of breakdown rules representing algorithm knowledge for the DFT. $\mathbf{rDFT}$ is an auxiliary transform and has two parameters. $\mathbf{RDFT}$ is a version of the real DFT.

- A breakdown rule may have degrees of freedom. An example is the choice of k in (2).
- A rule like (2) does not specify how to compute the smaller transforms. This implies that rules have to be applied recursively until an algorithm is completely specified. Because of the that, and the availability of different rules for the same transform, there is a large set of choices. In other words, the relatively few existing rules yield a very large space of possible algorithms. This makes rules a very efficient representation of algorithm knowledge. For example, for $n = 2^\ell$, (2) alone yields $\Theta(5^\ell/\ell^{3/2})$ different algorithms, all with roughly the same operations count.

Spiral contains about 200 breakdown rules for about 40 transforms, some of which are auxiliary. The most important rules for the DFT, without complete specification, are shown in Table 2. Note the occurrence of auxiliary transforms.

Spiral Program Generation: Overview

The task performed by Spiral is to translate the algorithm knowledge (represented as in Table 2) for a given transform into optimized source code (we assume C/C++) for a given platform.

The exact approach for generating the code depends on the type of code that has to be generated. The most important distinctions are the following:

- *Fixed input size versus general input size:* If the input size is known (e.g., "DFT of size 4" as shown in Fig. 3(a) and (b)) the algorithm to be used and other decisions can be determined at program generation time and can be inlined. The result is a function containing only loops and basic blocks of straightline code. If the input size is not known, it becomes an additional input and the implementation

(a) Fixed input size, unrolled

```
void dft_4(cpx *Y, cpx *X){
  cpx s, t, t2, t3;
  t = (X[0] + X[2]);
  t2 = (X[0] - X[2]);
  t3 = (X[1] + X[3]);
  s = _I_*(X[1] - X[3]);
  Y[0] = (t + t3);
  Y[2] = (t - t3);
  Y[1] = (t2 + s);
  Y[3] = (t2 - s);
}
```

(b) Fixed input size, looped

```
void dft_4(cpx *Y, cpx *X){
  cpx T[4];
  cpx W[2] = {1, _I_};
  for(int i = 0; i <= 1; i++) {
    cpx w = W[i];
    T[2*i] = (X[i] + X[i+2]);
    T[2*i+1] = w*(X[i] - X[i+2]);
  }
  for(int j = 0; j <= 1; j++) {
    Y[j] = T[j] + T[j+2];
    Y[2+j] = T[j] - T[j+2];
  }
}
```

(c) General input size library, recursive

```
struct dft : public Env{
  dft(int n); // constructor
  void compute(cpx *Y, cpx *X);
  int _rule, f, n;
  char  *_dat;
  Env  *ch1, *ch2;
};

void dft::compute(cpx *Y, cpx *X){
  ch2->compute(Y, X, n, f, n, f);
  ch1->compute(Y, Y, n, f, n, n/f);
}
```

Table 3: Code types.

becomes recursive (Fig. 3(c)). The actual algorithm, i.e., recursive computation is now chosen at runtime once the input size is known.

- *Straightline code versus loop code (fixed input size only):* Straightline code (Fig. 3(a)) is only suitable for small sizes, but can be faster, due to reduced overhead and increased opportunities for algebraic simplifications. Loop code requires additional optimizations that merge redundant loops.
- *Scalar code versus parallel code:* Code that is parallelized for SIMD vector extensions or multiple cores requires specific optimizations and the use of explicit vector intrinsics or threading directives.

The program generation process is explained in the next four sections corresponding to four different code types of increasing difficulty. The order matches the historic development, since for each move to the next code type at least one new idea had to be introduced. The types and main ideas (in parentheses) are

- Fixed input size straightline code (SPL, breakdown rules, feedback loop)
- Fixed input size loop code (Σ-SPL, loop merging)
- Fixed input size parallel code (paradigms, tagged rewriting)
- General input size code (recursion step closure, parametrization)

Spiral generates code for fixed input size transforms (first three bullets) as shown in Fig. 3. The input is the transform symbol (e.g., "DFT") and the size (e.g., "128"). The output is a C function that computes the transform ($y = \mathbf{DFT}_{128}\, x$ in this case). Depending on the code type, not all blocks in Fig. 3 may be used.

The block diagram for general input size is shown later.

Fixed Input Size: Straightline code

Given as input to Spiral is a transform symbol ("DFT") and the input size. The program generation does not need the parallelization and loop optimizations blocks. Further, no Σ-SPL is needed, which means the SPL-to-Σ-SPL block and the Σ-SPL-to-code block are joined to one SPL-to-code block.

Algorithm generation. Spiral uses a rewrite system that recursively applies the breakdown rules (e.g., Table 2) to generate a complete SPL algorithm for the transform. As said before there are many choices due to the choice of rule and the degree of freedom in some rules (e.g., k in (2)).

SPL to C code and optimization. The SPL expression is then compiled into actual C code using the internal SPL compiler, which recursively applies the translation rules sketched in Table 4.

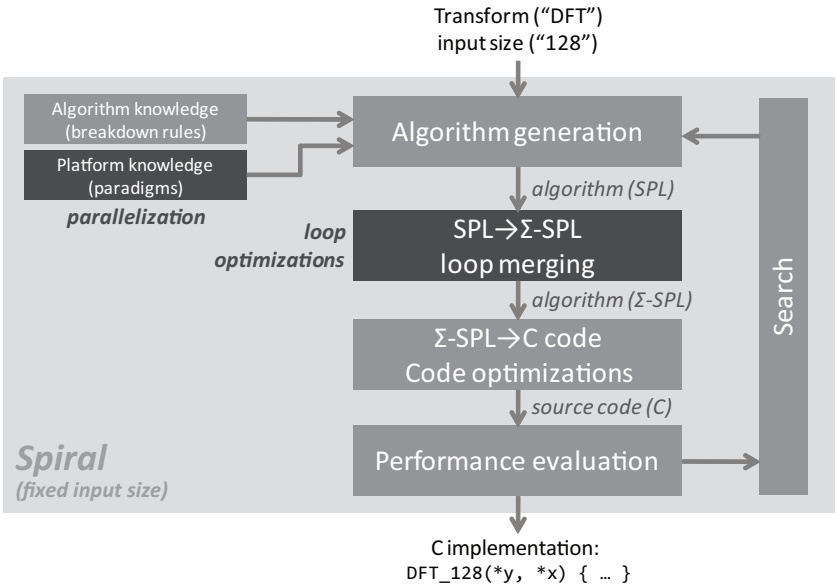


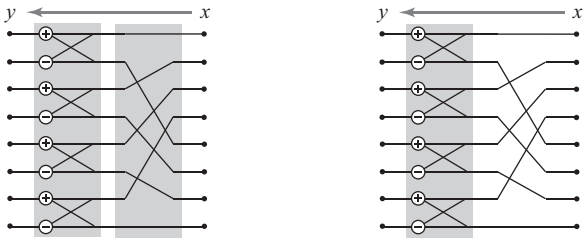
Figure 3: Spiral program generator for fixed input size functions. For straightline code, no Σ-SPL is needed and SPL is translated directly into C code.

SPL expression S	Pseudo code for $y = Sx$
$A_n B_n$	`<code for: t = Bx>` `<code for: y = At>`
$I_m \otimes A_n$	`for (i=0; i<m; i++)` `  <code for: y[i*n:1:i*n+n-1] = A(x[i*n:1:i*n+n-1])>`
$A_m \otimes I_n$	`for (i=0; i<n; i++)` `  <code for: y[i:n:i+m*n-n] = A(x[i:n:i+m*n-n])>`
D_n	`for (i=0; i<n; i++)` `  y[i] = D[i]*x[i];`

L_k^{km}	`for (i=0; i<k; i++)`
	`  for (j=0; j<m; j++)`
	`  y[i*m+j] = x[j*k+i];`

F_2	`y[0] = x[0] + x[1];` `y[1] = x[0] - x[1];`

Table 4: Translation of SPL to code. The subscript of A, B specifies the (square) matrix size. `x[b:s:e]` denotes (Matlab-style) the subvector of x starting at `b`, ending at `e`, extracted at stride `s`. D is a diagonal matrix, whose diagonal elements are stored in an array with the same name.



(a) Two stages (b) One stage

Figure 4: The loop merging problem for $(I_4 \otimes F_2)L^8_4$.

All loops are unrolled and code level optimizations are applied. These include array scalarization, constant propagation, and algebraic simplification.

Performance evaluation. The runtime of the resulting code is measured and fed into the search block that controls the algorithm generation.

Search. The search drives a feedback loop that generates and evaluates different algorithms to find the fastest. Dynamic programming has proven to work best in many cases, but other techniques including evolutionary search or bandit-based Monte Carlo exploration have been studied.

Fixed Input Size: Loop code

The approach to generating straightline code can also be used to generate loop code (Table 4 yields loops), but the code will be inefficient.

The problem: Loop merging. To illustrate the problem consider the SPL expression

$$(I_4 \otimes F_2)L^8_4.$$

Application of Table 4 yields the code visualized in Fig. 4(a):

```
// Input: double x[8], output: y[8]
double t[8];
for(int i=0; i<4; i++) {
   for (int j=0; j<2; j++) {
       t[i*2+j] = x[j*4+i];
   }
}
for (int j=0; j<4; j++) {
   y[2*j]   = t[2*j] + t[2*j+1];
   y[2*j+1] = t[2*j] - t[2*j+1];
}
```

This is known to be suboptimal since the permutation (first loop) can be fused with the subsequent computation loop, thus eliminating one pass through the data (Fig. 4(b)):

```
// Input: double x[8], output: y[8]
for (int j=0; j<4; j++) {
   y[2*j]   = x[j] + x[j+4];
   y[2*j+1] = x[j] - x[j+4];
}
```

This transformation cannot be expressed in SPL and, in the general case, is difficult to perform on C code. To solve this problem, Σ-SPL was developed, an extension of SPL that can express loops. The loop merging is then performed by rewriting Σ-SPL expressions.

Σ-SPL. Σ-SPL adds four basic components to SPL:

1. index mapping functions,

2. scalar functions,

Σ-SPL expression S	Code for $y = Sx$
$G(f^{n\to N})$	`for(i=0; i<n; i++)` `  y[i] = x[f(i)];`
$S(f^{n\to N})$	`for(i=0; i<n; i++)` `  y[f(i)] = x[i];`
$P(f^n)$	`for(i=0; i<n; i++)` `  y[i] = x[f(i)];`
$\mathrm{diag}\left(f^{n\to\mathbb{C}}\right)$	`for(i=0; i<n; i++)` `  y[i] = f(i)*x[i];`
$\sum_{i=0}^{k-1} A_i$	`for(i=0; i<k; i++)` `  <code for: y = A_i * x>`

Table 5: Translation of Σ-SPL to code.

3. parametrized matrices,
4. iterative sum $\sum$.

These are defined next.

An integer interval is denoted with $\mathbb{I}_n = \{0, \ldots, n-1\}$, and an index mapping function f with domain $\mathbb{I}_n$ and range $\mathbb{I}_N$ is denoted with

$$f^{n\to N} : \mathbb{I}_n \to \mathbb{I}_N;\ i \mapsto f(i).$$

An example is the stride function

$$h_{b,s}^{n\to N} : \mathbb{I}_n \to \mathbb{I}_N;\ i \mapsto b + is, \quad \text{for } s|N. \tag{4}$$

Permutations are written as $f^{n\to n} = f^n$ such as the stride permutation in (1).

A scalar function $f : \mathbb{I}_n \to \mathbb{C};\ i \mapsto f(i)$ maps an integer interval to the domain of complex or real numbers, and is abbreviated by $f^{n\to\mathbb{C}}$. Scalar functions are used to describe diagonal matrices.

Σ-SPL adds four types of parameterized matrices to SPL (gather, scatter, permutation, diagonal):

$$G(f^{n\to N}),\ S(f^{n\to N}),\ P(f^n),\ \text{and}\ \mathrm{diag}\left(f^{n\to\mathbb{C}}\right).$$

Their translation into actual code (which also defines the matrices) is shown in Table 5. For example,

$$G(h_{0,1}^{n\to N}) = \begin{bmatrix} 1 & & \\ & \ddots & \\ & & 1 \end{bmatrix}, \quad S(h_{0,1}^{n\to N}) = G(h_{0,1})^\top.$$

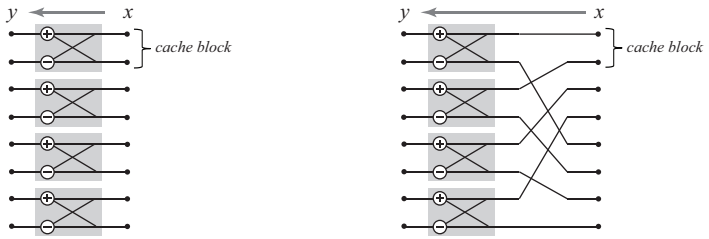
Finally, Σ-SPL adds the iterative matrix sum

$$\sum_{i=0}^{n-1} A_i.$$

to represent loops. The A_i are restricted such that no two A_i have a non-zero entry in the same row.

The following example shows how $\otimes$ is converted into a sum. A is assumed to be $n \times n$ and domain and range in the occurring stride functions are omitted for simplicity.

$$\begin{aligned} I_k \otimes A &= \begin{bmatrix} A & & \\ & \ddots & \\ & & A \end{bmatrix} = \begin{bmatrix} A & & \\ & & \\ & & \end{bmatrix} + \cdots + \begin{bmatrix} & & \\ & & \\ & & A \end{bmatrix} \\ &= S(h_{0,1})AG(h_{0,1}) + \cdots + S(h_{(k-1)n,1})AG(h_{(k-1)n,1}) \\ &= \sum_{i=0}^{k-1} S(h_{in,1})AG(h_{in,1}) \end{aligned}$$



(a) $I_4 \otimes F_2$ (b) $(I_4 \otimes F_2)L_4$

Figure 5: Mapping SPL constructs to four threads. Each thread computes one F_2. Both computations are data parallel, but (a) produces no false sharing whereas (b) does.

Intuitively, the conversion to Σ-SPL makes the loop structure of $y = (I_k \otimes A)x$ explicit. In each iteration i, $G(\cdot)$ and $S(\cdot)$ specify how to read and write a portion of the input and output, respectively, to be processed by A.

Loop merging using Σ-SPL and rewriting. Using Σ-SPL, the loop merging problem identified before in the example $(I_4 \otimes F_2)L_4^8$ is solved by the loop optimization block in Fig. 3 as follows:

$$\begin{aligned}
(I_4 \otimes F_2)L_4^8 &\rightarrow \left(\sum_{i=0}^{3} S(h_{2i,\,1})F_2G(h_{2i,\,1})\right) P(\ell_4^8) \\
&\rightarrow \sum_{i=0}^{3} \Big(S(h_{2i,\,1})F_2G(\ell_4^8 \circ h_{2i,\,1})\Big) \\
&\rightarrow \sum_{i=0}^{3} \Big(S(h_{2i,\,1})F_2G(h_{i,\,2})\Big)
\end{aligned}$$

The first step translates SPL into Σ-SPL. The second step performs the loop merging by composing the permutation ℓ_4^8 with the index functions of the subsequent gathers. The third step simplifies the resulting index functions. After that, actual C code is generated using Table 5.

Besides the added loop optimizations block in Fig. 3, the program generation for loop code operates iteratively exactly as for straightline code.

Fixed input size: Parallel code

As was illustrated with Fig. 1, for compute function compilers usually fail to optimally (or at all) exploit the parallelism offered by a platform. Hence the task falls with the programmer, who has to leave the standard C programming model and insert explicit threading or OpenMP loops for shared memory parallelism and so-called intrinsics for vector instruction sets. However, doing so in a straightforward way does not necessarily yield good performance.

The problem: Algorithm structure. To illustrate the problem consider a target platform with four cores that share a cache with a cache block size of two complex numbers.

The first goal is to obtain parallel code with four threads for $I_4 \otimes F_2$ visualized in Fig. 5(a). The computation is data parallel; hence, the loop suggested in Table 4 can be replaced, for example, by an OpenMP parallel loop. Note that each processor "owns" as working set exactly one cache block, hence the parallelization will be efficient.

Now consider again the SPL expression $(I_4 \otimes F_2)L_4^8$ visualized in Fig. 5(b). The computation is again data parallel, but the access pattern has changed such that always two processors access the same cache block. This produces false sharing, which triggers the cache coherency protocol and reduces performance.

$\langle \text{smp} \rangle$	::=	$\langle \text{generic} \rangle \mid \langle \text{basic} \rangle \mid$	
		$\langle \text{smp} \rangle \cdot \cdots \cdot \langle \text{smp} \rangle \mid$	(product)
		$\langle \text{smp} \rangle \oplus \ldots \oplus \langle \text{smp} \rangle \mid$	(direct sum)
		$I_n \otimes \langle \text{smp} \rangle \mid$	(tensor product)
		$\ldots$	
$\langle \text{generic} \rangle$	::=	$\text{diag}(a_0, \ldots, a_{n-1}) \mid \ldots$	
$\langle \text{basic} \rangle$	::=	$I_p \otimes A_n \mid P \otimes I_\mu \mid \ldots$	

Table 6: $\text{smp}(p, \mu)$ base cases in Backus-Naur form; n is a positive integer, a_i are real or complex numbers.

The problem is obviously the permutation L^8_4. Since the rules (e.g., those in Table 2) contain many, and various, permutations, a straightforward mapping to parallel code will yield hugely suboptimal performance. To solve this problem inside Spiral, another rewrite system is introduced to restructure algorithms before mapping to parallel code. The restructuring will be different for different forms of parallelism, called paradigms.

Paradigms and tagged rewriting. A *paradigm* in Spiral is a feature of the target platform that requires structural optimization. Typically, a paradigm is a form of parallelism. Examples include shared memory parallelism (SMP) and SIMD parallelism. A paradigm may be parameterized, e.g., by the vector length ν for SIMD parallelism. In Spiral, a paradigm manifests itself by another rewrite system provided by the additional parallelism block in Fig. 3 (and backend extensions in the Σ-SPL to C code block to produce the actual code).

The goal of the new rewrite system is to structurally optimize a given SPL expression into a form that can be efficiently mapped to a given paradigm. The rewrite system is built from the three main components:

- *Tags* encode the paradigm and relevant parameters. Examples include the tags "$\text{vec}(\nu)$" for SIMD vector extensions and the tag "$\text{smp}(p, \mu)$" for SMP. The meaning of the parameters is explained later.

- *Base cases* are SPL constructs that can be mapped well to a given paradigm. As illustrated above, one example is any $I_p \otimes A_n$ for p-way SMP.

- *Tagged rewrite rules* are mathematical identities that translate general SPL expressions towards base cases. An example is the rule (assuming $p|n$)

$$\underbrace{A_m \otimes I_n}_{\text{smp}(p,\mu)} \rightarrow \underbrace{L^{mn}_m}_{\text{smp}(p,\mu)} \left(I_p \otimes (I_{n/p} \otimes A_m)\right) \underbrace{L^{mn}_n}_{\text{smp}(p,\mu)} .$$

  The rule extracts the p-way parallel loop (base case) $I_p \otimes (I_{n/p} \otimes A_m)$ from $A_m \otimes I_n$. The stride permutations L^{mn}_m and L^{mn}_n are handled by further rewriting.

Example: SMP. For SMP, the tag $\text{smp}(p, \mu)$ contains the number of processors p and the cache block size μ. Base cases include $I_p \otimes A_n$ and $P \otimes I_\mu$, where P is any permutation. $P \otimes I_\mu$ moves data in blocks of size μ, hence false sharing is avoided. From these, other base cases can be built recursively as captured by the sketched grammar in Table 6.

Some SMP rewrite rules are shown in Table 7. Note that the rewriting is not unique, and not every sequence of rules terminates. Once all tags disappear, the rewriting terminates.

Example: SIMD. For SIMD, the tag $\text{vec}(\nu)$ contains only the vector length ν. The most important base case is $A_n \otimes I_\nu$, which can mapped to vector code by generating scalar code for A_n and replacing every operation by its corresponding ν-way vector operation. Other bases cases include $L^{2\nu}_\nu$, $L^{2\nu}_2$, and $L^{\nu^2}_\nu$, which are generated automatically from the instruction set [9]. Similar to Table 6, the entire set of vector base cases is specified by a grammar recursively built from the above special constructs.

Parallelization by rewriting. In Spiral, parallelization adds the new parallelization block in Fig. 3. The parallelization rules are applied interleaved with the breakdown rules to generate SPL algorithms that have the right structure for the desired paradigm. For example, for the DFT it may operate as follows:

$$
\begin{array}{lcl}
\underbrace{AB}_{\text{smp}(p,\mu)} & \rightarrow & \underbrace{A}_{\text{smp}(p,\mu)}\ \underbrace{B}_{\text{smp}(p,\mu)} \\
\underbrace{A_m \otimes I_n}_{\text{smp}(p,\mu)} & \rightarrow & \underbrace{\left(L^{mp}_m \otimes I_{n/p}\right)\left(I_p \otimes (A_m \otimes I_{n/p})\right)\left(L^{mp}_p \otimes I_{n/p}\right)}_{\text{smp}(p,\mu)} \\
\underbrace{L^{mn}_m}_{\text{smp}(p,\mu)} & \rightarrow & \begin{cases} \underbrace{\left(I_p \otimes L^{mn/p}_{m/p}\right)}_{\text{smp}(p,\mu)}\underbrace{\left(L^{pn}_p \otimes I_{m/p}\right)}_{\text{smp}(p,\mu)} \\ \underbrace{\left(L^{pm}_m \otimes I_{n/p}\right)}_{\text{smp}(p,\mu)}\underbrace{\left(I_p \otimes L^{mn/p}_m\right)}_{\text{smp}(p,\mu)} \end{cases} \\
\underbrace{I_m \otimes A_n}_{\text{smp}(p,\mu)} & \rightarrow & I_p \otimes \left(I_{m/p} \otimes A_n\right) \\
\underbrace{\left(P \otimes I_n\right)}_{\text{smp}(p,\mu)} & \rightarrow & \left(P \otimes I_{n/\mu}\right) \otimes I_\mu
\end{array}
$$

Table 7: Examples of smp(p, μ) rewrite rules.

$$
\begin{array}{lcl}
\underbrace{\mathbf{DFT}_{mn}}_{\text{smp}(p,\mu)} & \rightarrow & \underbrace{\left((\mathbf{DFT}_m \otimes I_n) T^{mn}_n (I_m \otimes \mathbf{DFT}_n) L^{mn}_m\right)}_{\text{smp}(p,\mu)} \\
& \dots & \\
& \rightarrow & \underbrace{\left(\mathbf{DFT}_m \otimes I_n\right)}_{\text{smp}(p,\mu)}\ \underbrace{T^{mn}_n}_{\text{smp}(p,\mu)}\ \underbrace{\left(I_m \otimes \mathbf{DFT}_n\right)}_{\text{smp}(p,\mu)}\ \underbrace{L^{nm}_m}_{\text{smp}(p,\mu)} \\
& \dots & \\
& \rightarrow & \left((L^{mp}_m \otimes I_{n/p\mu}) \otimes I_\mu\right)\left(I_p \otimes (\mathbf{DFT}_m \otimes I_{n/p})\right)\left((L^{mp}_p \otimes I_{n/p\mu}) \otimes I_\mu\right) \\
& & T^{mn}_m \Big(I_p \otimes (I_{m/p} \otimes \mathbf{DFT}_n)\Big)\left(I_p \otimes L^{mn/p}_{m/p}\right)\left((L^{pn}_p \otimes I_{m/p\mu}) \otimes I_\mu\right)
\end{array}
$$

First, Spiral applies the breakdown rule (2). Then the parallelization rules transform the resulting SPL expression in several steps. Note how the final expression has only access patterns (permutations) of the form $P \otimes I_\mu$ and all computations are in the form $I_p \otimes A$ (and the diagonal T^{mn}_m). The smaller DFTs can be expanded in different ways, e.g., by rewriting for SIMD. Further choices are used for search.

The remaining operation of Spiral including Σ-SPL conversion and search proceeds as before.

General Input Size

An implementation that can compute a transform for arbitrary input size is fundamentally different from one for fixed input size (compare Fig. 3(b) and (c)). If the input size n is fixed, e.g., $n = 4$, the computation is

```
(x,y) -> dft_4(y,x)
```

and all decisions such as the choice of recursion until base cases are reached can be made at implementation time. In an equivalent implementation (called library) for general input size n,

```
(n,x,y) -> dft(n,y,x)
```

the recursion is fixed only once the input size is known. Formally, the computation now becomes

```
n -> ((x,y) -> dft(n,y,x))
```

which is an example of function currying. A C++ implementation is sketched in Fig. 3(c), where the two steps would take the form

```
dft * f = new dft(n); // initialization
f->compute(y, x);     // computation
```

The first step determines the recursion to be taken using search or heuristics and precomputes the twiddle factors needed for the computation. The second step performs the actual computation. The underlying assumption is that the cost of the first step is amortized by a sufficient number of computations. This model is used by FFTW [15] and the libraries generated by Spiral.

To support the above model, the implementation needs recursive functions. The major problem is that the optimizations introduced before operate in nontrivial ways across function boundaries, thus creating more functions than expected. The challenge is to derive these functions automatically.

The problem: Loop merging across function boundaries. To illustrate the problem consider the Cooley-Tukey FFT (2). A direct recursive implementation would consist of four steps corresponding to the four matrix factors in (2). Two of the steps would call smaller DFTs:

```
void dft(int n, cpx *y, cpx *x) {
   int k = choose_factor(n);
   int m = n/k;
   cpx *t1 = Permute x with L(n,k);
   // t2 = (I_k tensor DFT_m)*t1
   for(int i=0; i<k; ++i)
      dft(m, t2 + m*i, t1 + m*i);
   // t3 = T^n_m*t2, f() computes diagonal entries of T
   for(int i=0; i<n; ++i)
      t3[i] = f(i) * t2[i];
   // y = (DFT_k tensor I_m)*t3, cannot call
   // dft() recursively, need strided I/O
   for(int i=0; i<m; ++i)
      dft_stride(k, m, y + i, t3 + i);
}
// to be implemented
void dft_stride(int n, int stride, cpx *Y, cpx *X);
```

Note how even this simple implementation is not self-contained. A new function `dft_stride` is needed that accesses the input in a stride and produces the output at the same stride (see the data flow in Fig. 2).

However, as explained before, loops should be merged where possible. For fixed size code, Spiral would merge the first loop with the second, and the third loop with the fourth using Σ-SPL rewriting. The same can be done in the general size recursive implementation but the merging crosses function boundaries:

```
void dft(int n, cpx *y, cpx *x) {
   int k = choose_factor(n);
   // t1 = (I_k tensor DFT_m)L(n,k)*x
   for(int i=0; i < k; ++i)
      dft_iostride(m, k, 1, t1 + m*i, x + m*i);
   // y = (DFT_k tensor I_m) T^n_m
   // diagonal entries of T are now precomputed in precomp_f[]
   for(int i=0; i < m; ++i)
      dft_scaled(k, m, precomp_f[i], y + i, t1 + i);
}

// to be implemented
void dft_iostride(int n, int istride, int ostride, cpx *y, cpx *x);
void dft_scaled(int n, int stride, cpx *d, cpx *y, cpx *x);
```

Now there are two additional functions: `dft_iostride` reads at a stride and writes at a different stride, `dft_scaled` first scales the input and then performs a DFT at a stride.

So at least three functions are needed with different signatures. However, the two additional functions are also implemented recursively, possibly spawning new functions. Calling these functions *recursion steps*, the main challenge is to automatically derive the complete set of recursion steps needed, called the "recursion step closure." Further, for each recursion step in the closure, the signature has to be derived.

Recursion step closure by Σ-SPL rewriting. Spiral derives the recursion step closure using Σ-SPL and the same rewriting system that is used for loop merging. For example, the two additional recursion steps in the optimized implementation above are automatically obtained from (2) as follows. Recursion steps are

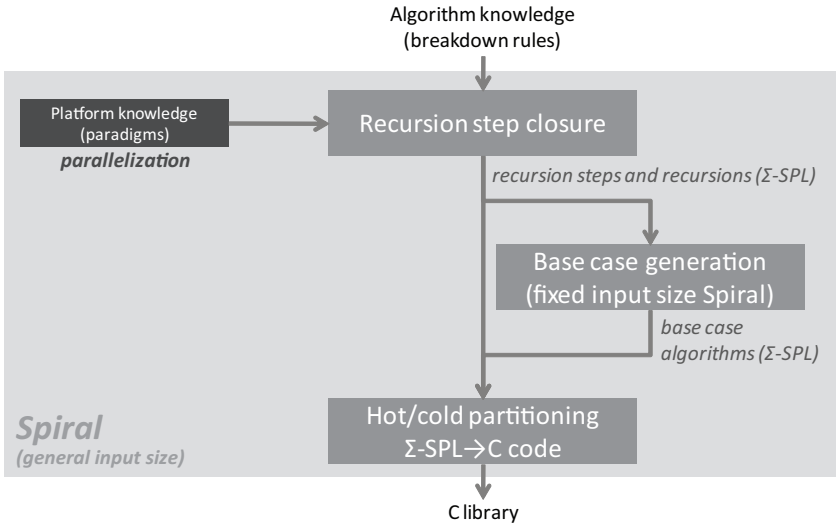


Figure 6: Spiral program generator for general input size libraries.

marked by overbraces.

$$
\begin{aligned}
\overbrace{\mathbf{DFT}_n} &\rightarrow (\overbrace{\mathbf{DFT}_{n/k}} \otimes I_k) T^n_k (I_{n/k} \otimes \overbrace{\mathbf{DFT}_k}) L^n_{n/k} \\
&\rightarrow \left(\sum_{i=0}^{k-1} S(h_{i,k}) \overbrace{\mathbf{DFT}_{n/k}} G(h_{i,k}) \right) \operatorname{diag}(f) \left(\sum_{j=0}^{n/k-1} S(h_{jk,1}) \overbrace{\mathbf{DFT}_k} G(h_{jk,1}) \right) P(\ell^n_{n/k}) \\
&\rightarrow \sum_{i=0}^{k-1} S(h_{i,k}) \overbrace{\mathbf{DFT}_{n/k}} \operatorname{diag}(f \circ h_{i,k}) G(h_{i,k}) \sum_{j=0}^{n/k-1} S(h_{jk,1}) \overbrace{\mathbf{DFT}_k} G(h_{j,n/k}) \\
&\rightarrow \sum_{i=0}^{k-1} \overbrace{S(h_{i,k}) \mathbf{DFT}_{n/k} \operatorname{diag}(f \circ h_{i,k}) G(h_{i,k})} \sum_{j=0}^{n/k-1} \overbrace{S(h_{jk,1}) \mathbf{DFT}_k G(h_{j,n/k})}
\end{aligned} \tag{5}
$$

The first step applies the breakdown rule (2). The second step converts to Σ-SPL. The third step performs loop merging as explained before. The fourth step expands the braces to include the context. The two expression under the braces correspond to the two functions `dft_iostride` and `dft_scaled`. The process is now repeated for the expression under the braces until closure is reached. In this example, only one additional function is needed, i.e., the recursion step closure consists of four mutually recursive functions. The derivation of the recursion steps also yields a Σ-SPL specification of the actual recursion, i.e., their implementation by a recursive function (e.g., (5) for $\mathbf{DFT}_n$).

For the best performance the braces may be extended to also include the loop represented by the iterative sum. Moving the loop into the function enables better C/C++ compiler optimizations.

If the implementation is vectorized or parallelized, the initial breakdown rules are first rewritten as explained before and then the closure is computed. The size of the closure is typically increased in this case.

Program generation for general input size: Overview. The overall process is visualized in Fig. 6. The input to Spiral is now a (sufficient) set of breakdown rules for a given transform or transforms. The rules are parallelized if desired using the appropriate paradigms; then the recursion step closure is computed, which also yields the actual recursions.

The resulting recursion steps need base cases for termination. These are generated using the algorithm generation block from the fixed input size Spiral (Fig. 3) for a range of small sizes (e.g., $n \leq 32$) to improve performance. These, the recursion steps, and the recursions are fed into the final block to generate the final

library. Among other things the block performs the *hot/cold partitioning* that determines which parameters in a recursion step are precomputed during initialization and which become parameters of the actual compute function. Finally, the actual code is generated (which now includes recursive functions) and integrated into a common infrastructure to obtain a complete library.

Many details are omitted in this description and are provided in [30, 29].

Extensions

A major question is whether the approach taken by Spiral can be extended beyond the domain of linear transforms, while maintaining both the basic principles outlined in the introduction and the ability to automatically perform the necessary transformations and reasoning. First progress in this direction was made in [7] with the introduction of the operator language (OL). OL generalizes SPL by considering operators that may be nonlinear and may have more then one vector input or output. Important constructs such as the tensor product are generalized to operators. First results on program generation for functions such as radar imaging, Viterbi decoding, matrix-multiplication, and the physical layer functions of wireless communication protocols have already been developed.

RELATED ENTRIES

FFTW
ATLAS
FFT

BIBLIOGRAPHIC NOTES AND FURTHER READING

Spiral is based on early ideas on using tensor products to map FFT algorithms to parallel supercomputers [16]. The first paper describing SPL and the SPL compiler is [32]. See also [14] for basic block optimizations for transforms. The first complete basic Spiral system including SPL algorithm generation and search was presented in [23] with a more extensive treatment in [24] and the probably best overview paper [22], which fully develops SPL for a variety of transforms. The path to complete automation in the transform domain continued with Σ-SPL and loop merging [11], the introduction of rewriting systems for SIMD vectorization [8, 13] and base case generation [9], SMP parallelization [12], and distributed memory parallelization in [2, 3]. The final step to generating general size, parallel, adaptive libraries was made in [30, 29]. The generated libraries are modeled after FFTW [15], which is written by hand but uses generated basic blocks [14].

The most important extensions of Spiral are the following. Extensions to generate Verilog for field-programmable gate-arrays (FPGAs) are in [19, 18]. Search techniques other than dynamic programming are developed in [26, 5] The use of learning to avoid search was studied in [25, 6]. Finally, [7, 4, 17] make the first steps towards extending Spiral beyond the transform domain including the first OL description. The Spiral project website with more information and all publications is located at [1].

A good introduction to FFTs using tensor products are the books [27, 28]. A comprehensive overview of algorithms for Fourier/cosine/sine transforms is given in [21, 31]. A good introduction to mapping FFTs to multicore platforms is [10].

References

[1] Spiral project website. `www.spiral.net`.

[2] Andreas Bonelli, Franz Franchetti, Juergen Lorenz, Markus Püschel, and Christoph W. Ueberhuber. Automatic performance optimization of the discrete Fourier transform on distributed memory computers. In *International Symposium on Parallel and Distributed Processing and Application (ISPA)*, volume 4330 of *Lecture Notes In Computer Science*, pages 818–832. Springer, 2006.

[3] Srinivas Chellappa, Franz Franchetti, and Markus Püschel. High performance linear transform program generation for the Cell BE. In *High Performance Embedded Computing (HPEC)*, 2009.

[4] Frédéric de Mesmay, Srinivas Chellappa, Franz Franchetti, and Markus Püschel. Computer generation of efficient software Viterbi decoders. In *International Conference on High Performance Embedded Architectures and Compilers (HiPEAC)*, volume 5952 of *Lecture Notes in Computer Science*, pages 353–368. Springer, 2010.

[5] Frédéric de Mesmay, Arpad Rimmel, Yevgen Voronenko, and Markus Püschel. Bandit-based optimization on graphs with application to library performance tuning. In *International Conference on Machine Learning (ICML)*, pages 729–736, 2009.

[6] Frédéric de Mesmay, Yevgen Voronenko, and Markus Püschel. Offline library adaptation using automatically generated heuristics. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2010.

[7] Franz Franchetti, Frédéric de Mesmay, Daniel McFarlin, and Markus Püschel. Operator language: A program generation framework for fast kernels. In *IFIP Working Conference on Domain Specific Languages (DSL WC)*, volume 5658 of *Lecture Notes in Computer Science*, pages 385–410. Springer, 2009.

[8] Franz Franchetti and Markus Püschel. A SIMD vectorizing compiler for digital signal processing algorithms. In *International Parallel and Distributed Processing Symposium (IPDPS)*, pages 20–26, 2002.

[9] Franz Franchetti and Markus Püschel. Generating SIMD vectorized permutations. In *International Conference on Compiler Construction (CC)*, volume 4959 of *Lecture Notes in Computer Science*, pages 116–131. Springer, 2008.

[10] Franz Franchetti, Markus Püschel, Yevgen Voronenko, Srinivas Chellappa, and José M. F. Moura. Discrete Fourier transform on multicore. *IEEE Signal Processing Magazine, special issue on "Signal Processing on Platforms with Multiple Cores"*, 26(6):90–102, 2009.

[11] Franz Franchetti, Yevgen Voronenko, and Markus Püschel. Formal loop merging for signal transforms. In *Programming Languages Design and Implementation (PLDI)*, pages 315–326, 2005.

[12] Franz Franchetti, Yevgen Voronenko, and Markus Püschel. FFT program generation for shared memory: SMP and multicore. In *Supercomputing (SC)*, 2006.

[13] Franz Franchetti, Yevgen Voronenko, and Markus Püschel. A rewriting system for the vectorization of signal transforms. In *High Performance Computing for Computational Science (VECPAR)*, volume 4395 of *Lecture Notes in Computer Science*, pages 363–377. Springer, 2006.

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

[15] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2), 2005. special issue on "Program Generation, Optimization, and Adaptation".

[16] J. Johnson, R. W. Johnson, D. Rodriguez, and R. Tolimieri. A methodology for designing, modifying, and implementing Fourier transform algorithms on various architectures. *IEEE Trans. Circuits and Systems*, 9:449–500, 1990.

[17] Daniel McFarlin, Franz Franchetti, José M. F. Moura, and Markus Püschel. High performance synthetic aperture radar image formation on commodity architectures. In *SPIE Conference on Defense, Security, and Sensing*, volume 7337, page 733708. Proceedings of SPIE, 2009.

[18] Peter A. Milder, Franz Franchetti, James C. Hoe, and Markus Püschel. Formal datapath representation and manipulation for implementing DSP transforms. In *Design Automation Conference (DAC)*, pages 385–390, 2008.

[19] Grace Nordin, Peter A. Milder, James C. Hoe, and Markus Püschel. Automatic generation of customized discrete Fourier transform IPs. In *Design Automation Conference (DAC)*, pages 471–474, 2005.

[20] W. H. Press, B. P. Flannery, Teukolsky S. A., and Vetterling W. T. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 2nd edition, 1992.

[21] Markus Püschel and José M. F. Moura. Algebraic signal processing theory: Cooley-Tukey type algorithms for DCTs and DSTs. *IEEE Transactions on Signal Processing*, 56(4):1502–1521, 2008.

[22] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[23] Markus Püschel, Bryan Singer, Manuela Veloso, and José M. F. Moura. Fast automatic generation of DSP algorithms. In *International Conference on Computational Science (ICCS)*, volume 2073 of *Lecture Notes In Computer Science*, pages 97–106. Springer, 2001.

[24] Markus Püschel, Bryan Singer, Jianxin Xiong, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, and Robert W. Johnson. SPIRAL: A generator for platform-adapted libraries of signal processing algorithms. *Journal of High Performance Computing and Applications, special issue on "Automatic Performance Tuning"*, 18(1):21–45, 2004.

[25] Bryan Singer and Manuela Veloso. Learning to generate fast signal processing implementations. In *International Conference on Machine Learning (ICML)*, pages 529–536, 2001.

[26] Bryan Singer and Manuela Veloso. Stochastic search for signal processing algorithm optimization. In *Supercomputing (SC)*, page 22, 2001.

[27] R. Tolimieri, M. An, and C. Lu. *Algorithms for discrete Fourier transforms and convolution*. Springer, 2nd edition, 1997.

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

[29] Yevgen Voronenko. *Library Generation for Linear Transforms*. PhD thesis, Electrical and Computer Engineering, Carnegie Mellon University, 2008.

[30] Yevgen Voronenko, Frédéric de Mesmay, and Markus Püschel. Computer generation of general size linear transform libraries. In *International Symposium on Code Generation and Optimization (CGO)*, pages 102–113, 2009.

[31] Yevgen Voronenko and Markus Püschel. Algebraic signal processing theory: Cooley-Tukey type algorithms for real DFTs. *IEEE Transactions on Signal Processing*, 57(1):205–222, 2009.

[32] Jianxin Xiong, Jeremy Johnson, Robert W. Johnson, and David Padua. SPL: A language and compiler for DSP algorithms. In *Programming Languages Design and Implementation (PLDI)*, pages 298–308, 2001.