Computer Generation of General Size Linear Transform Libraries

Yevgen Voronenko, Frédéric de Mesmay, and Markus Püschel
Department of Electrical and Computer Engineering
Carnegie Mellon University Pittsburgh, PA, USA {*yvoronen, fdemesma, pueschel*}*@ece.cmu.edu*

Abstract

The development of high-performance libraries has become extraordinarily difficult due to multiple processor cores, vector instruction sets, and deep memory hierarchies. Often, the library has to be reimplemented and reoptimized, when a new platform is released. In this paper we show how to automatically generate general input-size libraries for the domain of linear transforms. The input to our generator is a formal specification of the transform and the recursive algorithms the library should use; the output is a library that supports general input size, is vectorized and multithreaded, provides an adaptation mechanism for the memory hierarchy, and has excellent performance, comparable to or better than the best human-written libraries. Further, we show that our library generator enables various customizations; one example is the generation of Java libraries.

1. Introduction

The development of high performance numerical libraries has become extraordinarily difficult due to deep memory hierarchies, vector instruction sets, and multiple processor cores. Compilers lack domain knowledge and hence cannot optimize numerical code to the extent possible. Hence, it is up to the programmer to perform all necessary optimizations, a very time consuming task that often has to be repeated whenever a new platform is released.

A tantalizing prospect is to automate the numerical library development, which means letting the computer write the software given only a short, high-level description of the algorithms to be implemented as input. Some advances have been made in this direction. For example, ATLAS [1] generates the kernel code for matrix-matrix multiplication (MMM) by searching over different blocking and unrolling strategies. The kernel code multiplies matrices of fixed size and is used inside a general size MMM routine. Similarly, FFTW [2], a library for the discrete Fourier transform (DFT) and related transforms, uses a program generator [3] to obtain efficient base cases (called codelets) for small fixed sizes (see Table 1(a)). Again, these codelets are used in a hand-written general-size library infrastructure, whose design is rather sophisticated: it supports various variants of the DFT needed in the recursive computation, uses an initialization routine for precomputation and runtime adaptation, and provides multi-threading support.

Since codelets consist of straightline code, the codelet generator [3] can use a directed acyclic graph (DAG) representation to mechanically perform optimizations such as constant folding and scheduling, and the generator is restricted to small problem sizes.

The program generation problem for the DFT, and transforms in general, becomes considerably harder if code for the arbitrary (again fixed) input sizes, and hence the looped code, has to be generated (see Table 1(b)). In this case, the domain knowledge is needed to perform all the necessary optimizations. Spiral [4] solves this problem for a large class of transforms by using two domain-specific languages called SPL and $\sum$-SPL and by using rewriting systems to perform loop optimizations [5], vectorization [6], and parallelization [7] at a high level of abstraction.

Spiral pushes the limits of automation, but one problem is still unsolved: the computer generation of *general input size* libraries (see Table 1(c)) for linear transforms. We solve this problem in this paper to achieve, what one may call "complete" automation in the domain of linear transforms. More specifically, we show that given only a high-level description of one or several transform algorithm, we can generate a library that is optimized for the memory hierarchy, vectorized, parallelized, and provides an optional runtime adaptation mechanism. In other words, the generated library could be FFTW-like or hardware vendor library-like.

It turns out that the main challenge in generating such libraries is the identification of the set of recursive functions needed. The reason is that the optimizations that can be inlined in fixed-size code (Table 1(b)), now cross function boundaries and hence create a set of transform *variants* that need to be computed as subroutines. We derive these needed subroutines automatically, including their exact signatures, and derive which parameters are precomputed and which are runtime parameters. The procedure is compatible with the prior work on vectorization and parallelization, which means it enables us to also generate vectorized and multithreaded libraries.

We implemented the complete library generator and used it to generate libraries for a variety of transforms. In most case the performance is competitive with and often is considerably faster than the best existing (partially and fully hand-written) libraries FFTW and Intel Integrated Performance Primitives (IPP), if they support the same functionality. Further, we demonstrate other benefits of our library generator such as the computer generation of libraries with custom

(a) Fixed 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 size, looped

```
void dft_4(cpx *Y, cpx *X) {
  cpx T[4];
  for(int i = 0; i <= 1; i++) {
    cpx d1 = D[2*i], d2 = D[2*i+1];
    T[2*i] = d1*(X[i] + X[i+2]);
    T[2*i+1] = d2*(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 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 1: Code types. Automatic general size library (c) generation is the goal of this paper.

source code size or Java libraries.

Organization of this paper. We provide the background on transforms and a formal specification of the problem solved in this paper in Section 2. We also illustrate the problem in obtaining general size transform libraries: the derivation of the set of recursive functions needed. We show how to solve this problem using $\sum$-SPL and rewriting in Section 3. The complete library generation is discussed in detail in Section 4. The experiments and benchmarks are shown in Section 5. Section 6 is our conclusion.

2. Motivation and Problem Statement

We provide the necessary background on linear transforms and use the DFT as example to explain the main challenge in obtaining a library for general input size: the identification of the set of recursive functions needed to implement the transform. Then we state the library generation problem formally and discuss its relevance.

Background on linear transforms. A linear transform is a matrix-vector product $y = Mx$, where x, y are the input and output vectors, respectively, and M is the transform matrix. For the DFT, M is the matrix

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

Many fast Fourier transform algorithms (FFTs) exist, and can be represented as factorizations of $\mathbf{DFT}_n$ into products of structured sparse matrices [8]. For example, the Cooley-Tukey FFT is a divide-and-conquer algorithm that for $n = km$ can be written as

$$\mathbf{DFT}_n = (\mathbf{DFT}_k \otimes I_m) T^n_m (I_k \otimes \mathbf{DFT}_m) L^n_k. \tag{2}$$

Here, I_n is the $n \times n$ identity matrix, $T^n_m = \mathrm{diag}_{0\le j<n}(d_j)$ is a diagonal matrix, and L^n_k (with $n = mk$) is the $n \times n$ stride permutation matrix defined by the underlying permutation

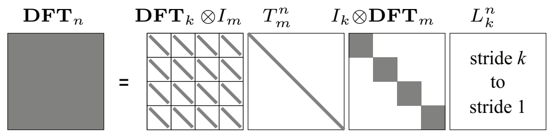
$$\ell^n_k : \; jm + i \mapsto ik + j, \text{ for } 0 \le i < m, 0 \le j < k. \tag{3}$$

In words, L^n_k transposes a $k \times m$ matrix stored in row-major order, or equivalently, reads the input at stride k and writes the output at stride 1.

Most important in this formalism is the tensor (or Kronecker) product $\otimes$ of matrices, defined as

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

(2) is called a *breakdown rule* in Spiral [4] and the matrix formalism is called signal processing language (SPL) [9]. It is best understood by visualizing the nonzero pattern of the matrices in (2), done here for $k = m = 4$. In the leftmost sparse factor $\mathbf{DFT}_k \otimes I_m$, all the 1st, 2nd, $\dots$, mth, entries of the small diagonals constitute one $\mathbf{DFT}_k$, respectively.



(4)

Recursive application of (2) for a two-power $n = 2^t$ yields an $O(n \log(n))$ algorithm, terminated by $\mathbf{DFT}_2$, which is computed by definition:

$$\mathbf{DFT}_2 = \left[\begin{smallmatrix} 1 & 1 \\ 1 & -1 \end{smallmatrix}\right]. \tag{5}$$

For other sizes, other FFT algorithms are needed.

Problem motivation: Recursive functions needed to implement a transform. Implementing a recursive library based on (2) is seemingly straightforward with the pseudo code shown in Implementation 1 that computes $y = \mathbf{DFT}_n\, x$ in four steps corresponding to the four factors in (2) or (4) from right to left.

We observe that even this direct implementation is not self-contained but needs the extra function `dft_str` that reads and writes input at stride `m`. Further, Implementation 1 is far suboptimal since it makes four passes through the data. A far better solution, Implementation 2, close to the one in FFTW 2.x, merges the first two steps and the last two steps, and precomputes the diagonal elements `d(i)`.

Now additional two functions `dft_str` and `dft_scaled` are needed and need to be implemented as well. We call these functions *recursion steps*. Their implementation may produce additional recursion steps. We call the full set of required recursion steps the *recursion step closure*.

Further, the recursion must eventually be terminated. Hence the library needs base cases which are transforms of

$$\begin{aligned}
\mathbf{DFT}_n &= P^\top_{k/2,2m}\left(\mathbf{DFT}_{2m} \oplus \left(I_{k/2-1} \otimes_i C_{2m}\, \mathbf{rDFT}_{2m}(i/k)\right)\right)\left(\mathbf{RDFT}'_k \otimes I_m\right), \quad k \text{ even},\\
\begin{vmatrix}\mathbf{RDFT}_n\\ \mathbf{RDFT}'_n\end{vmatrix} &= (P^\top_{k/2,m} \otimes I_2)\left(\begin{vmatrix}\mathbf{RDFT}_{2m}\\ \mathbf{RDFT}'_{2m}\end{vmatrix} \oplus \left(I_{k/2-1} \otimes_i D_{2m}\begin{vmatrix}\mathbf{rDFT}_{2m}(i/k)\\ \mathbf{rDFT}_{2m}(i/k)\end{vmatrix}\right)\right)\left(\begin{vmatrix}\mathbf{RDFT}'_k\\ \mathbf{RDFT}'_k\end{vmatrix} \otimes I_m\right), \quad k \text{ even},\\
\mathbf{rDFT}_{2n}(u) &= L^{2n}_m\left(I_k \otimes_i \mathbf{rDFT}_{2m}((i+u)/k)\right)\left(\mathbf{rDFT}_{2k}(u) \otimes I_m\right),\\
\mathbf{DCT\text{-}2}_n &= P^\top_{k/2,2m}\left(\mathbf{DCT\text{-}2}_{2m}\, K^{2m}_2 \oplus \left(I_{k/2-1} \otimes N_{2m}\, \mathbf{RDFT\text{-}3}^\top_{2m}\right)\right) B_n (L^{n/2}_{k/2} \otimes I_2)(I_m \otimes \mathbf{RDFT}'_k) Q_{m/2,k},
\end{aligned}$$

Table 2: Breakdown rules for a variety of transforms: complex DFT, DFT for real input (RDFT), and discrete cosine transforms (DCT) of type 2. $\mathbf{rDFT}$ and $\mathbf{RDFT}'$ are auxiliary transforms used in the computation. P, Q are permutation matrices, B, C, D, N are other sparse matrices. The second rule is for two transforms simultaneously.

Implementation 1 (Direct from (2)*)*

```
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 = diag( d(j) )*t2
   for(int i=0; i<n; ++i)
      t3[i] = d(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_str(k, m, y + i, t3 + i);
}
// to be implemented
void dft_str(int n, int str, cpx *Y, cpx *X);
```

Implementation 2 (FFTW 2.x-like implementation of (2)*)*

```
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_str(m, k, 1, t1 + m*i, x + m*i);
   // y = (DFT_k tensor I_m) diag(d(j))
   for(int i=0; i < m; ++i)
      dft_scaled(k, m, precomp_d[i], y + i, t1 + i);
}

// to be implemented
void dft_str(int n, int istr, int ostr, cpx *y, cpx *x);
void dft_scaled(int n, int str, cpx *d, cpx *y, cpx *x);
```

small fixed sizes that are typically unrolled and must exist for all the recursion steps.

FFTW 2.x implements the two recursion steps in Implementation 2 and the base cases are generated codelets for sizes 2–16, 32, 64. `dft_scaled` is always assumed to be a base case called twiddle codelet. The degrees of freedom in recursively choosing k in (2) is used for platform adaptation and fixed in a so-called plan during a precomputation step that runs a search.

Implementation 2 is not yet vectorized or parallelized. This, as one can imagine, can only increase the number of required recursion steps. Indeed, in FFTW 3.2a, there are 4 types of codelets for scalar complex DFT, and an additional 15 types for the vectorized complex DFT for a total of 19 types. The need for these types, and the FFTW library infrastructure in general, are identified and implemented by hand—only the base cases (codelets, Table 1(a)) are generated for each type and input size.

Now we can formulate the automatic library generation problem considered in this paper.

Problem statement (Library generation for transforms). *Given:* A set of transforms and associated breakdown rules. *Generate:* A vectorized, parallelized, adaptive, library of high performance for the transforms. *Tasks:*

1) Find the recursion step closure, i.e., the minimal set of needed recursion steps (functions), including their signatures, based on the given rules, and derive breakdown rules for these recursion steps.
2) Parallelize and vectorize the breakdown rules for each recursion step if needed and generate the corresponding recursive functions.
3) Generate the (vectorized if needed) base cases (codelets) for all recursion steps for a set of small fixed sizes. For best performance, a range of small sizes must be implemented.
4) Combine the recursion steps and codelets into a generated library infrastructure, responsible for initialization, precomputed data management, and the control of the degrees of freedom in the recursion for adaptation.

Discussion. Why does one need a library generator if some well-designed hand-written libraries like FFTW or Intel's IPP/MKL already exist? There are several reasons.

First, these libraries often need to be updated, if there are significant platform changes. A generator reduces the effort considerably.

Second, using a library generator we can generate libraries for other transforms, including the discrete cosine and sine transforms, FIR filters, and wavelets. Table 2 shows several transforms and examples of their breakdown rules. Note the general complexity and that several of them require auxiliary transforms (e.g., $\mathbf{rDFT}$) and their rules to be implemented. We generate and evaluate the associated libraries in Section 5.

Third, it is not clear whether (2) is always the best way to compute the DFT. Indeed, later we will show a DFT library generated from the FFT shown in Table 2, which improves performance over the one based on (2).

Fourth, adding additional breakdown rules for the same transform can considerable increase the number of recursion steps due to the merging of permutations similar as in Implementation 2. The resulting complexity is difficult to manage manually.

Fifth, our automatic framework enables the generation of custom libraries without performance loss. For example, a light-weight library for two-power sizes only or custom interfaces such as a DFT followed by scaling. Further, by changing the backend we can generate libraries in a different target language such as Java.

Lastly, with this paper we show that for an entire domain of numerical algorithms (namely transforms) every major implementation task can be automated starting only from a specification of the problem and its algorithms. The key is a properly designed domain-specific framework that is used to perform all major tasks at a high level of abstraction including the generation of alternative algorithms, adaptation and optimization for the memory hierarchy, parallelization, vectorization, and finally, the creation of general size adaptive libraries (this paper).

3. Identifying Recursion Steps: $\sum$-SPL

Our solution to the automatic library generation problem relies on the domain-specific language $\sum$-SPL as internal algorithm representation. SPL, the matrix formalism used in (2) and Table 2, is not expressive enough. $\sum$-SPL was introduced in [5] to automate loop merging for transforms; here it is the key to automatically derive the set of recursion steps needed to implement a transform and to perform the other necessary optimizations. We start with a brief introduction of $\sum$-SPL and explain how it is used to mechanically derive the recursion steps in Implementation 2.

$\sum$-SPL is similar to SPL in that it represents structured matrix factorizations in a declarative form, but it introduces loops, index mapping functions (such as strides), and explicit I/O. The loops are captured by an iterative matrix sum operator $\sum$. Read (load) and write (store) operations are represented by gather matrices $\mathrm{G}(f)$ and scatter matrices $\mathrm{S}(f)$, respectively, both parametrized by index functions f. Similarly, permutation matrices and diagonal matrices are parametrized by the underlying permutations and scalar functions, respectively. We now explain $\sum$-SPL in detail.

Functions. $\sum$-SPL uses functions with one input and one output, which can be composed in the usual way, e.g., $f \circ g$. A function f with domain A and range B is written as

$$f^{A \to B} : i \mapsto f(i), \quad i \in A,\ f(i) \in B.$$

The domain is always an integer interval $\mathbb{I}_n = \{0 \dots n-1\}$, the range may be $\mathbb{I}_N$ or the real or complex numbers $\mathbb{R}, \mathbb{C}$. For convenience, we abbreviate $f^{n \to N} = f^{\mathbb{I}_n \to \mathbb{I}_N}$, or omit the domain and range, where it is clear from the context. Permutations are bijective functions written as $f^n = f^{n \to n}$.

Our running example uses the stride permutation in (3) and the *stride function* defined as

$$h_{b,s}^{n \to N} : \mathbb{I}_n \to \mathbb{I}_N;\ i \mapsto b + is. \qquad (6)$$

Parametrized matrices. $\sum$-SPL contains four types of matrices parametrized by functions: $\mathrm{G}(f^{n \to N})$, $\mathrm{S}(f^{n \to N})$, $\mathrm{perm}(f^n)$ and $\mathrm{diag}\left(f^{n \to \mathbb{C}}\right)$. They are defined below. Let $e_k^n \in \mathbb{C}^{n \times 1}$ be the column basis vector with the 1 in k-th position and 0 elsewhere, and $(\cdot)^\top$ denote matrix transposition, then

$$\begin{aligned} \mathrm{G}(f^{n \to N}) &:= \left[e_{f(0)}^N \mid e_{f(1)}^N \mid \cdots \mid e_{f(n-1)}^N \right]^\top, \\ \mathrm{perm}(f^n) &:= \mathrm{G}(f) = \left[e_{f(0)}^n \mid e_{f(1)}^n \mid \cdots \mid e_{f(n-1)}^n \right]^\top, \\ \mathrm{S}(f^{n \to N}) &:= \mathrm{G}(f)^\top = \left[e_{f(0)}^N \mid e_{f(1)}^N \mid \cdots \mid e_{f(n-1)}^N \right], \\ \mathrm{diag}\left(f^{n \to \mathbb{C}}\right) &:= \mathrm{diag}\left(f(0), \dots, f(n-1)\right). \end{aligned}$$

From the definition it follows that

$$\begin{aligned} y = \mathrm{G}(f) \cdot x \quad &\Leftrightarrow \quad y_i = x_{f(i)}, \\ y = \mathrm{S}(f) \cdot x \quad &\Leftrightarrow \quad y_{f(i)} = x_i, \quad y_j = \begin{cases} x_i & \text{if } j = f(i) \\ 0 & \text{else} \end{cases}. \end{aligned}$$

The above explains their interpretation as code, given in Table 3(a).

Regular and iterative matrix sum. Using the constructs defined above we can convert the $\otimes$ in SPL into a summation. This is done as follows, assuming that A is $n \times n$:

$$\begin{aligned} I_k \otimes A &= \begin{bmatrix} A & & \\ & \ddots & \\ & & A \end{bmatrix} = \begin{bmatrix} A & & \\ & & \\ & & \end{bmatrix} + \cdots + \begin{bmatrix} & & \\ & & \\ & & A \end{bmatrix} \qquad (7) \\ &= \mathrm{S}_0\, A\, \mathrm{G}_0 + \cdots + \mathrm{S}_{k-1}\, A\, \mathrm{G}_{k-1} = \sum_{j=0}^{k-1} \mathrm{S}_j\, A\, \mathrm{G}_j\,. \end{aligned}$$

In this equation, $\mathrm{G}_j = \mathrm{G}(h_{nj,1})$ and $S_j = G_j^\top$. For example,

$$\mathrm{G}_0 = \mathrm{G}(h_{0,1}) = \begin{bmatrix} 1 & & & & \\ & \ddots & & & \\ & & 1 & & \end{bmatrix},$$

and $\mathrm{S}_0 = \mathrm{S}(h_{0,1}) = \mathrm{G}(h_{0,1})^\top$. Intuitively, the conversion to $\sum$-SPL makes the loop structure explicit. In each iteration j, $\mathrm{G}(\cdot)$ and $\mathrm{S}(\cdot)$ specify how to read and write a portion of the in- and output, respectively, to be processed by A.

We always require that the sum as in (7) does not require actual additions; hence, the sum encodes a loop in which all iterations are independent and each iteration produces a unique part of the final output vector. The code for $y = (\sum_{j=0}^{n-1} M_j)x$ is shown in Table 3(b).

The conversion from SPL to $\sum$-SPL, the merging of iterative algorithm steps, and their simplification is accomplished using the rewrite rules shown in Table 4 [5] . We consider (2) as example and show—crucial for this paper—how the recursion steps in Implementation 2 become explicit.

Cooley-Tukey FFT example. We start with (2) and first

(a) Parametrized matrices	(b) Matrix sums
$\mathbf{Code}(\mathrm{G}(f^{n\to N}), y, x) \to$ `for(j=0..n-1) y[j] = x[f(j)];`	$\mathbf{Code}(M_1 + M_2, y, x) \to \mathbf{Code}(M_1, y, x);\ \mathbf{Code}(M_2, y, x);$
$\mathbf{Code}(\mathrm{S}(f^{n\to N}), y, x) \to$ `for(j=0..n-1) y[f(j)] = x[j];`	$\mathbf{Code}\left(\sum_{j=0}^{k-1} M_j, y, x\right) \to$ `for(j=0..k-1)` $\mathbf{Code}(M_j, y, x);$
$\mathbf{Code}(\mathrm{perm}(p^{n\to n}), y, x) \to$ `for(j=0..n-1) y[j] = x[p(j)];`	
$\mathbf{Code}(\mathrm{diag}\left(f^{n\to\mathbb{C}}\right), y, x) \to$ `for(j=0..n-1) y[j] = f(j)*x[j];`	

Table 3: Translating $\sum$-SPL constructs to code; x and y denote the input and output vectors.

SPL to $\sum$-SPL conversion

$$A \otimes I_k \ \to \sum_{j=0}^{k-1} \mathrm{S}(h_{j,k}) A\, \mathrm{G}(h_{j,k}) \tag{8}$$

$$I_k \otimes A \to \ \sum_{j=0}^{k-1} \mathrm{S}(h_{mj,1}) A\, \mathrm{G}(h_{mj,1}) \tag{9}$$

$$L_k^n \to \ \mathrm{perm}(\ell_k^n) \tag{10}$$

Index mapping function simplification

$$\ell_m^{mk} \circ h_{kj,1}^{k\to mk} \ \to h_{j,m}^{k\to mk} \tag{11}$$

$$\left(\ell_m^{mk}\right)^{-1} \ \to \ell_k^{mk} \tag{12}$$

$$h_{b',s'}^{N\to N'} \circ h_{b,s}^{n\to N} \ \to h_{b'+s'b,s's}^{n\to N'} \tag{13}$$

Loop merging

$$\left(\textstyle\sum_j A_j\right) M \ \to \left(\textstyle\sum_j A_j M\right), \quad M \in \{\mathrm{G}, \mathrm{diag}\} \tag{14}$$

$$M\left(\textstyle\sum_j A_j\right) \ \to \left(\textstyle\sum_j M A_j\right), \quad M \in \{\mathrm{S}, \mathrm{diag}\} \tag{15}$$

$$\mathrm{G}(f)\, \mathrm{G}(g) \ \to \mathrm{G}(g \circ f) \tag{16}$$

$$\mathrm{S}(f)\, \mathrm{S}(g) \ \to \mathrm{S}(f \circ g) \tag{17}$$

$$\mathrm{G}(f)\, \mathrm{perm}(g) \ \to \mathrm{G}(g \circ f) \tag{18}$$

$$\mathrm{perm}(f)\, \mathrm{S}(g) \ \to \mathrm{S}(f^{-1} \circ g) \tag{19}$$

$$\mathrm{G}(f)\, \mathrm{diag}\left(d\right) \ \to \mathrm{diag}\left(d \circ f\right) \mathrm{G}(f) \tag{20}$$

$$\mathrm{diag}\left(d\right) \mathrm{S}(f) \ \to \mathrm{S}(f)\, \mathrm{diag}\left(d \circ f\right) \tag{21}$$

Table 4: $\sum$-SPL rewrite rules.

rewrite $(I_k \otimes \mathbf{DFT}_m) L_k^n$ using (9) and (10):

$$\left(\sum_{j=0}^{k-1} \mathrm{S}(h_{jm,1})\, \mathbf{DFT}_m\, \mathrm{G}(h_{jm,1})\right) \mathrm{perm}(\ell_k^n),$$

then merge the permutation using rules (14) and (18),

$$\left(\sum_{j=0}^{k-1} \mathrm{S}(h_{jm,1})\, \mathbf{DFT}_m\, \mathrm{G}(\ell_k^n \circ h_{jm,1})\right),$$

and finally simplify the index function using rule (11) to get

$$\left(\sum_{j=0}^{k-1} \mathrm{S}(h_{jm,1})\, \mathbf{DFT}_m\, \mathrm{G}(h_{j,k})\right). \tag{22}$$

The above formula represents a loop over DFTs of strided input data and corresponds to the first loop of Implementation 2.

Now we rewrite $(\mathbf{DFT}_k \otimes I_m)\, \mathrm{diag}\left(f\right)$ using rules (8), (14), and (20) into

$$\left(\sum_{j=0}^{m-1} \mathrm{S}(h_{j,m})\, \mathbf{DFT}_k\, \mathrm{diag}\left(f \circ h_{j,k}\right) \mathrm{G}(h_{j,m})\right), \tag{23}$$

which is a loop over scaled DFTs on strided input and equivalent to the second loop in Implementation 2.

Identifying recursion steps. Expressions (22) and (23) directly identify the recursion steps in Implementation 2, namely `dft_str` $\leftrightarrow \mathrm{S}(h_{jm,1})\, \mathbf{DFT}_m\, \mathrm{G}(h_{j,k})$ and `dft_scaled` $\leftrightarrow \mathrm{S}(h_{j,m})\, \mathbf{DFT}_k\, \mathrm{diag}\left(f \circ h_{j,k}\right) \mathrm{G}(h_{j,m})$. To find the recursion steps automatically, we introduce a "function tag" into $\sum$-SPL that identifies those parts of a formula that are to be implemented as separate functions and use rewriting rules that propagate these tags. We denote the tag with a brace. The rewriting is straightforward and looks (sketched) as follows in our example:

$$\underbrace{\mathbf{DFT}}$$

$$= (\underbrace{\mathbf{DFT}} \otimes I)\, \mathrm{diag}\, (I \otimes \underbrace{\mathbf{DFT}}) L \tag{24}$$

$$= \left(\sum \mathrm{S}(h)\, \underbrace{\mathbf{DFT}}\, \mathrm{diag}\ \mathrm{G}(h)\right) \left(\sum \mathrm{S}(h)\, \underbrace{\mathbf{DFT}}\, \mathrm{G}(h)\right)$$

$$= \left(\sum \underbrace{\mathrm{S}(h)\, \mathbf{DFT}\, \mathrm{diag}\ \mathrm{G}(h)}\right) \left(\sum \underbrace{\mathrm{S}(h)\, \mathbf{DFT}\, \mathrm{G}(h)}\right) \tag{25}$$

In essence, the rewriting proceeds as before, but in the final step the function tags are expanded to include adjacent gather, scatter, and diagonal matrices.

Our motivating example shows only the first step. The procedure needs to be repeated separately for each recursion step until closure is reached, if possible. Further, the recursion steps have to be parallelized and vectorized, and implemented, together with the base cases and the library infrastructure. The details are discussed in the remaining paper.

4. Library Generation

In this section we explain the library generation process. As before, we use the DFT and breakdown rule (2) as running example and also include the necessary base case rule (5).

The main steps of the procedure are shown in Figure 1. The input to the library generator is a set of transforms and

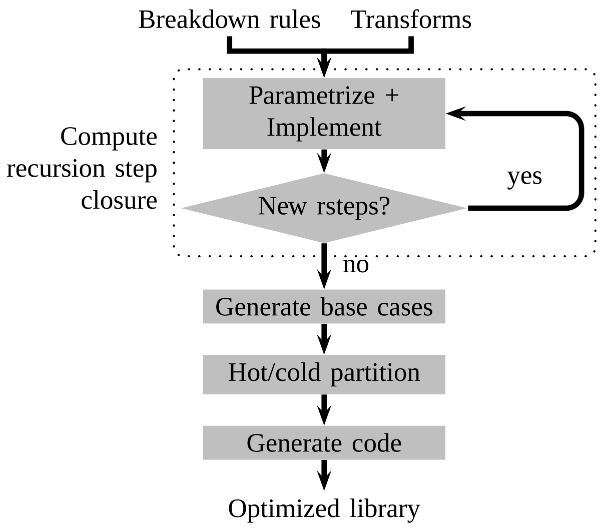


Figure 1: Library generation. Input: transforms and breakdown rules. Output: optimized library.

breakdown rules. The output is an optimized library. Below, we will explain each step in detail, after introducing some terminology.

A *recursion step* is a transform, usually surrounded by scatter and gather constructs; examples are (22) and (23). A recursion step (after parametrization, explained later) specifies the signature of the associated recursive function. For example, (22) defines `dft_str` in Implementation 2. Further, we refer to a recursion step expanded using a breakdown rule and rewritten as explained in Section 3 as a $\sum$*-SPL implementation* of a recursion step. For example, (25) is a $\sum$-SPL implementation of the recursion step $\mathbf{DFT}_n$; indeed, it can be converted into the DFT code Implementation 2.

Compute recursion step closure. The purpose of this step is to iteratively determine the set of recursion steps (i.e., distinct recursive functions) needed to implement the transforms and their breakdown rules.

The *Parametrize and Implement* block takes as input rules and recursion steps (note that the initial transform is a recursion step). First, the recursion steps are *parametrized*. This means expressions of free parameters in the recursion step (for example the loop indices, the degrees of freedom with the breakdown rules, etc.) are replaced by the smallest set of independent parameters. These will become function arguments in the associated function definition. We denote all free parameters with u_i. Besides the generated name, which contains no semantic information, each parameter has a type (e.g. integer, real number, etc.), which is automatically assigned during the parametrization. Parametrization involves three steps: 1) replace every expression with free variables by a new parameter of the same type, 2) determine the equality constraints on the parameters, and 3) find the smallest set of necessary parameters based on the constraints.

After parametrization, the recursion step is implemented. This means it is expanded using a breakdown rule and rewritten as explained in Section 3. Besides obtaining a $\sum$-SPL implementation, this step also identifies new recursion steps arising as children.

The *New rsteps?* block decides whether the current set of parametrized recursion steps contains new elements. If not, the set is the recursion step closure. If yes, the feedback loop iterates on the new recursion steps.

The procedure and further details are best explained using our running example $\mathbf{DFT}_n$ and (2) as input.

$\mathbf{DFT}_n$ has one parameter $n = u_1$ with no constraints, so $\mathbf{DFT}_{u_1}$ is a new recursion step. We expand $\mathbf{DFT}_{u_1}$ using the given rule (2) and rewrite it as explained in Section 3. The result is the implementation (25), which is composed of (22) and (23).

The implementation identifies two new recursion steps, which we have to parametrize. For completeness, we include the domains and ranges of the index mapping functions, which we omitted before. We start with the right recursion step in (25). First, we replace expressions with free variables by fresh parameters:

$$\mathrm{S}(h^{k\to n}_{jk,1})\,\mathbf{DFT}_k\,\mathrm{G}(h^{k\to n}_{j,m}) \to \mathrm{S}(h^{u_1\to u_2}_{u_3,1})\,\mathbf{DFT}_{u_4}\,\mathrm{G}(h^{u_5\to u_6}_{u_7,u_8})$$

Note that the constants, such as 1, survive parametrization. The result above is not yet a valid $\sum$-SPL formula, since the matrix dimensions do not necessarily match. Hence, we determine next the parameter constraints that make the formula valid. In $\sum$-SPL one needs to check matrix products and function compositions. In the example, only the matrix product has to be valid, which means

$$u_1 = u_4, \quad u_4 = u_5.$$

The constraints partition the parameters into groups of interrelated parameters. In each group, the constraints form a system of equations. In the cases we encountered the system was either 1) linear, and thus the number of independent parameters is equal to the rank of the system, or 2) non-linear with a single equation, and thus only one independent parameter is needed. Due to the trivial nature of these constraints, in all libraries that we generated, including the example above, the rank of linear systems was always one. To proceed in this case, one of the parameters is assumed to be known, and the system is solved for the rest of the parameters. The solution is then substituted into the $\sum$-SPL formula. In the above example, we assume u_1 to be known. Solving the trivial linear system gives $u_4 = u_5 = u_1$, which yields the final result:

$$\mathrm{S}(h^{u_1\to u_2}_{u_3,1})\,\mathbf{DFT}_{u_1}\,\mathrm{G}(h^{u_1\to u_6}_{u_7,u_8}). \tag{26}$$

Next, we consider the second recursion step in (25). The diagonal matrix it includes contains a special marker $\mathrm{pre}(\cdot)$

that tells the system that elements of the diagonal are to be precomputed. Therefore it makes sense to abstract away the particular generating function of the diagonal. We do this by allowing parameters to also be functions denoted as before, i.e., $u^{A \to B}$.

With this extension the parametrization follows the same steps as in the previous example, and the final result is

$$\mathrm{S}(h_{u_3,u_4}^{u_1 \to u_2})\, \mathbf{DFT}_{u_1}\, \mathrm{diag}\left(\mathrm{pre}({u_7}^{u_1 \to \mathbb{C}})\right) \mathrm{G}(h_{u_{10},u_{11}}^{u_1 \to u_9}). \quad (27)$$

Now we repeat the procedure with the two new recursion steps (26) and (27) to obtain their implementations. We only discuss (26) in detail.

We expand (26) using (2). Note that in (2), there is one degree of freedom, the value of k, which will become an additional parameter that we will denote with f_1. Inserting (2) into (26) yields

$$\mathrm{S}(h_{u_3,1}^{u_1 \to u_2})(\underbrace{\mathbf{DFT}_{f_1}} \otimes I_{u_1/f_1})\, \mathrm{diag}\left(\mathrm{pre}(d_{u_1/f_1}^{u_1 \to \mathbb{C}})\right) \\ \cdot (I_{f_1} \otimes \underbrace{\mathbf{DFT}_{u_1/f_1}})L_{f_1}^{u_1}\, \mathrm{G}(h_{u_7,u_8}^{u_1 \to u_6}). \quad (29)$$

The potential new recursion steps (DFTs) are marked but are not yet finalized, as we have seen in (24)–(25). Next, $\sum$-SPL conversion and further rewriting is performed using the rewrite rules from Table 4. The final result is given in (28) in Table 5.

The result looks complicated due to the high level of detail, but it is a completely specified $\sum$-SPL implementation of (26) using the Cooley-Tukey rule (2). Before we can generate code for the implementation in (26), the full recursion closure must be constructed.

A similar procedure is used to implement (27). The result is not shown.

After parametrization, the new recursion steps in (28) (Table 5) take the form

$$\mathrm{S}(h_{u_3,1}^{u_1 \to u_2})\, \mathbf{DFT}_{u_1}\, \mathrm{G}(h_{u_7,u_8}^{u_1 \to u_6})$$
$$\mathrm{S}(h_{u_3,u_4}^{u_1 \to u_2})\, \mathbf{DFT}_{u_1}\, \mathrm{diag}\left(\mathrm{pre}({u_7}^{u_1 \to \mathbb{C}})\right) \mathrm{G}(h_{u_{10},u_{11}}^{u_1 \to u_9})$$

Inspection shows that these are equal to (26) and (27). Therefore, no new recursion steps are needed to implement (26).

Similarly, (27) spawns as recursion steps itself and a specialized variant, which itself produces no new recursion steps.

The overall result is a recursion step closure with four recursion steps. The associated call graph is shown in Fig. 2. For readability, we dropped the unknown parameters and replaced them by "*". The first three recursion steps in Fig. 2 correspond to the `dft`, `dft_str`, and `dft_scaled` functions in Implementation 2. The fourth recursion step is a special case of the third, with one parameter of the first stride function h equal to 1. This information is useful, because it can lead to better performance due to reduced index computation.

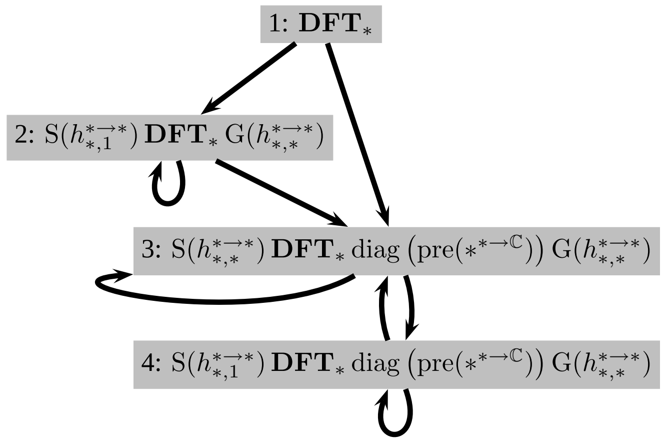

1: $\mathbf{DFT}_*$
2: $\mathrm{S}(h_{*,1}^{*\to*})\, \mathbf{DFT}_*\, \mathrm{G}(h_{*,*}^{*\to*})$
3: $\mathrm{S}(h_{*,*}^{*\to*})\, \mathbf{DFT}_*\, \mathrm{diag}\left(\mathrm{pre}(*^{*\to\mathbb{C}})\right) \mathrm{G}(h_{*,*}^{*\to*})$
4: $\mathrm{S}(h_{*,1}^{*\to*})\, \mathbf{DFT}_*\, \mathrm{diag}\left(\mathrm{pre}(*^{*\to\mathbb{C}})\right) \mathrm{G}(h_{*,*}^{*\to*})$

Figure 2: Graphical representation of the recursion step closure.

After the full closure is constructed, code can be generated for the obtained recursive $\sum$-SPL implementations like (28). This is done by applying rules from Table 3 and replacing the curly braces by calls to other recursion steps. For example, (28) becomes

```
cpx *T = (cpx*) malloc(sizeof(cpx) * u1);
for(int j=0; j < f1; ++j) {
  RS2(T, X, u1/f1, u1, u1*j/f1, u6, u8*j + u7, u8*f1);
}
for(int i=0; i < (u_1/f_1); ++i) {
  TFunc_Int_Cpx *F = new Func1(u1, u1/f1, f1, u1, i, u1/f1);
  RS3(Y, T, f1, u2, i + u3, u1/f1, F, u1, i, u1/f1);
}
free(T);
```

Above RS2 and RS3 are the 2nd and 3rd recursion steps from Fig. 2. This code is not yet final, because as explained later, in order to handle precomputation, it must be separated into an initialization and computation phase. In addition, a generator function from integers to complex numbers for the diagonal matrix in RS3 is created on the fly, saved in F, and passed into RS2. In many languages (C, Fortran in particular) this is not directly supported.

Generate base cases. The $\sum$-SPL implementations of the recursion steps obtained in the previous step are for general input size, such as u_1 in (28). To terminate the recursion at runtime, fixed size implementations, i.e. the recursion base cases are needed. Due to their fixed size, the base cases can be implemented as fully unrolled code, which reduces the overhead associated with loops and recursive calls. Ideally, the base cases should be available for a range of small sizes to maximally reduce the overheads. In FFTW, the base cases are called codelets.

This step can be automated as follows. We assume the list of desired sizes (e.g., 2–16) to be known in advance. This is currently a configuration setting.

For each obtained general size recursion step and each fixed size transform, the following steps are performed. The fixed size transform is matched against the transform in the recursion step to determine the parameters (including but not limited to the size). These are then inserted into the recursion step to obtain a fixed size recursion step. Code

$$\underbrace{\mathrm{S}(h^{u_1\to u_2}_{u_3,1})\,\mathbf{DFT}_{u_1}\,\mathrm{G}(h^{u_1\to u_6}_{u_7,u_8})} \longrightarrow \sum_{i=0}^{u_1/f_1-1} \underbrace{\mathrm{S}(h^{f_1\to u_2}_{u_3+i,u_1/f_1})\,\mathbf{DFT}_{f_1}\,\mathrm{diag}\left(\mathrm{pre}(\Omega^{u_1}_{u_1/f_1}\circ h^{f_1\to u_1}_{i,u_1/f_1})\right)\mathrm{G}(h^{f_1\to u_1}_{i,u_1/f_1})} \cdot \sum_{j=0}^{f_1-1} \underbrace{\mathrm{S}(h^{u_1/f_1\to u_1}_{u_1 j/f_1,1})\,\mathbf{DFT}_{u_1/f_1}\,\mathrm{G}(h^{u_1/f_1\to u_6}_{u_7+u_8 j,u_8 f_1})} \tag{28}$$

Table 5: Implementation of recursion step (26) using (2).

can be generated for the resulting $\sum$-SPL formula using the rules from Table 3 and applying the code optimizations from the standard Spiral system [4]. The end result is a recursion free $\sum$-SPL implementation and code.

As an example, consider (26) and $\mathbf{DFT}_2$. Matching yields $u_1 = 2$. Inserting 2 for u_1 in (26) there is only one possible $\sum$-SPL implementation based on $\mathbf{DFT}_2 = \left[\begin{smallmatrix}1 & 1\\ 1 & -1\end{smallmatrix}\right]$:

$$\mathrm{S}(h^{2\to u_2}_{u_3,1})\left[\begin{smallmatrix}1 & 1\\ 1 & -1\end{smallmatrix}\right]\mathrm{G}(h^{2\to u_6}_{u_7,u_8}). \tag{30}$$

Note that even though this $\sum$-SPL formula is recursion free, it is still parametrized. After applying the rules from Table 3 and standard compiler optimizations (loop unrolling, array scalarization, copy propagation, and strength reduction) Spiral obtains the code

```
cpx s1, s2;
s1 = X1[u_7];
s2 = X1[u_7 + u_8];
Y1[u_3] = (s1 + s2);
Y1[1 + u_3] = (s1 - s2);
```

Hot/cold partitioning of parameters. The goal of this step is to determine which parameters of a recursion step should be precomputed during an initialization step and which become function parameters. This initialization step has to be performed at runtime, once per transform size, and the cost of initialization is amortized over multiple uses of the same transform size.

As we explained earlier, the diagonal elements of T^n_m inside the Cooley-Tukey FFT (2) are usually precomputed. Similarly, a few other parameters might need to be precomputed. For example, those that are needed for the diagonal elements such as n and m in T^n_m, and parameters that capture degrees of freedom in the recursion strategy such as f_1 in (29) and their dependent parameters.

Automatic partitioning of the parameter set is important, because it can be rather large, even in the simplest cases. For example, in Fig. 2, even though there is only one parameter (DFT size) for the recursion step 1, there is a total of 21 other parameters across the recursion steps 2–4. These parameters are not seen by the user, because these recursion steps are only internally invoked by the algorithm. For larger libraries, the number of recursion steps is typically around a dozen (e.g., see Table 6 later) and the total number of parameters across all recursion steps is in the hundreds.

To better understand how this works, we give an example of calling Intel MKL using the DFTI interface to compute a complex 1-dimensional DFT of size 32:

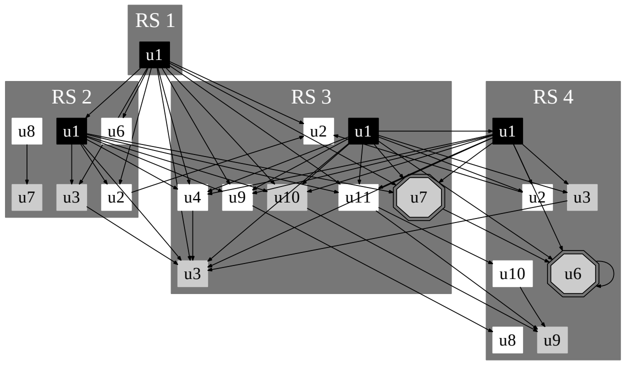


Figure 3: Parameter flow graph corresponding to the closure in Fig. 2 after hot/cold partitioning. The nodes denote parameters, the edges denote parameter dependencies. Black nodes are “cold”, white nodes are “none”, gray square nodes are “hot”, and gray octagon nodes are “reinit” parameters.

```
DFTI_DESCRIPTOR *d;
DftiCreateDescriptor(&d, DFTI_SINGLE, DFTI_COMPLEX, 1, 32);
...
DftiComputeForward(d, X, Y);
```

Above, `DftiCreateDescriptor` is an initialization function which precomputes the necessary constants. The parameters include the precision (`DFTI_SINGLE`), the data type (`DFTI_COMPLEX`), and the size (here: 32). A similar initialization function exists in FFTW.

We call parameters for the initialization *cold*, and parameters provided later *hot*. Changing a cold parameter incurs the overhead of reinitialization, while a hot parameter can be changed with no cost. The goal of the hot/cold partitioning is to determine which parameters are cold and which are hot for each recursion step. The procedure is explained next.

First, a parameter flow graph is constructed with shows the dependencies between the parameters. If one recursion step invokes another one, we create edges between the caller parameters and the corresponding callee parameters computed from them. For example, the partitioned parameter flow graph that corresponds to the closure in Fig. 2 is shown in Fig. 3. The partitioning algorithm initializes each node either to the “none” state (which means that no decision has been made), or to the “cold” state if it is a degree of freedom, or affects the applicability of any breakdown rules for the recursion steps (e.g., transform size), or appears inside the $\mathrm{pre}(\cdot)$ marker.

Next, two iterative dataflow analysis (IDA) [10] phases are applied. The first phase is a backward IDA that marks all mandatory cold parameters, by applying one simple rule. Namely, for each dependence edge $p \rightarrow q$, if q is marked "cold," then p is marked as "cold."

The second phase is a forward IDA that marks all mandatory hot parameters, and if a parameter must be cold and hot at the same time, puts it in a special "reinit" state, explained below. The forward IDA applies the following three rules:

1) If parameter p is marked "none," and depends on a loop index, mark p "hot".
2) If parameter p is marked "cold," and depends on a loop index, mark p "reinit".
3) For each dependence edge $p \rightarrow q$, if p is marked "hot," and q is marked "none", then mark q as "hot".

An example of a "reinit" parameter is u_7 in recursion step 3 from (27). It is the generator function for the diagonal elements. It depends on the loop index and at the same time appears inside the precompute marker. The "reinit" status means that the caller of recursion step 3 must create several copies of its descriptor with different values of u_7. The number of copies will be finite and equal to the number of iterations of the loop from which the recursion step is invoked.

Finally, when the second IDA phase is complete, we can mark all unmarked parameters (i.e., those in "none" state) as either hot or cold depending on the default policy.

Generate code. This step is responsible for generating the final library. For each recursion step several implementations generated in the previous steps must be put together to form a single function, and also the descriptor must be created.

The MKL/DFTI or FFTW descriptor mentioned above is an example of a higher order function. Namely, the descriptor takes an integer and returns a function (computing the DFT), from complex vectors to complex vectors.

Clearly, many target languages, including C and C++, do not directly support higher-order functions, and we eliminate them using the process known as *closure conversion*[1] or *lambda lifting* [11], [12]. As the result, the descriptor system used in Intel IPP and FFTW naturally arises. It is widely known that in object-oriented languages closures can be naturally expressed with objects and vice versa, which makes C++ a good choice for a target language. This conversion is also discussed in detail in [12], [13]. In our library generator, we create a class for each recursion step; the class attributes are the cold and reinit parameters of the recursion step.

For example, RS 2 becomes the following class:

```
struct RS2 : public TimeableEnv {
  char  *_dat;        /* precomputed data mem area */
  AutoFreeEnvList _garbage; /* allocated memory tracker */
  EnvList child1, child2;   /* initizalized callees */
  int _rule, f1, u1, u2, u6, u8;   /* parameters */

  RS2();
  RS2(int u1, int u2, int u6, int u8); /* constructor */
  virtual ~RS2();
  void compute(cpx *Y1, cpx *X1, int u3, int u7);
};
```

1. "Closure" here refers to the standard computer science term that describes a function together with an environment (a set of variable bindings) that must be used for its evaluation. The recursion step closure, in contrast, is a mathematical set closure.

Since its cold and reinit parameters are supplied in the constructor, it can initialize the cold and reinit parameter of its callees (in our case itself and RS3) by calling their constructors, and saving the result in `child1` and `child2`. The actual computation is performed by the `compute` class method, which takes hot parameters as arguments, and already has cold and reinit parameters readily available. For RS2, for instance, the `compute` method is synthesized from the implementations of (28) and (30). The final result is shown below:

```
void RS2::compute(cpx *Y1, cpx *X1, int u3, int u7){
  if (_rule == 1) {
    cpx s1, s2;
    s1 = X1[u_7];
    s2 = X1[u_7 + u_8];
    Y1[u_3] = (s1 + s2);
    Y1[1 + u_3] = (s1 - s2);
  }
  else if (_rule == 2) {
    cpx  *T;
    T = (cpx *) malloc(sizeof(cpx) * u1);
    for(int i = 0; i < f1; i++) {
      (RS2* child2[0])->compute(T, X1, u1*i/f1, u8*i+u7);
    }
    for(int j = 0; j < (u1/f1); j++) {
      (RS3* child1[j])->compute(Y1, T, j+u3, j);
    }
    free(T);
  }
}
```

All of the precomputed data, cold parameters, degrees of freedom and the `_rule` variable, which controls the dispatch, are initialized in the class constructors, which is not shown. The degrees of freedom and the dispatch variable are good candidates for autotuning, which is supported by the generated libraries. As we previously mentioned, the generator function for the diagonal is created on the fly and passed into RS3. This is another example of a higher order function, which needs to be converted into an object, similarly to the way recursion steps are.

Parallel and vector code. We generate a vectorized and/or parallelized library following the same steps in Fig. 1. The difference is that now additional $\sum$-SPL rewriting rules, responsible for vectorization and parallelization, are used during the Descend step. These rules are cognizant of a few architecture parameters (e.g., vector length, number of processors, and cache line size) introduced as tags. The procedure, operating on SPL expressions, is described in [6], [7]. To make it interoperate with the library generation, we ported it to $\sum$-SPL. We omit further details due to lack of space and refer the reader to [14], but mention that the number of recursion step increases compared to the generation of standard scalar libraries.

Table 6 shows the number of recursion steps in a sample of few generated libraries. All libraries use recursion steps

	number of recursion steps		
Transform	scalar	vectorized	vectorized + parallelized
DFT	4 / 3	4 / 7	8 / 8
RDFT	4 / 6	10 / 10	12 / 10
WHT	4 / 3	6 / 4	7 / 4
DCT-2	5 / 9	11 / 13	13 / 13
2D DCT-2	10 / 14	12 / 13	14 / 13
FIR Filter	4 / 4	4 / 5	4 / 4

Table 6: Number of recursion steps m/n in our generated libraries. m is the number of steps with loops; n is the number without loops and a close approximation of the number of base cases (codelets) needed for each small input size.

that may include loops, unlike in the running example we used. This ensures best performance and enables parallelization. However, it increases the number of recursion steps, since for each recursion step with loop there is a corresponding unlooped recursion step (the loop body). The base case can be generated for either variant, but generating a base case for the looped recursion step tends to lead to better performance.

In Table 6, the scalar $\mathbf{DFT}$ library uses the Cooley-Tukey FFT breakdown rule (2), to match our running example. Vectorized DFT libraries use the different breakdown rule shown in Table 2.

5. Experiments

Platform. Due to limited space, here we report the performance data on only one platform, however, extensive experimental data can be found in [14].

Our benchmark platform is has two dual core 3 GHz Intel Xeon 5160 processors (server version of Core 2 Duo) with 4 MB of shared L2 cache per processor, running Linux in 64-bit mode. The generated C++ libraries were compiled using the Intel C/C++ Compiler 10.1, and the generated Java libraries were compiled and run using the Sun JDK 1.6.0. Vectorized code was emitted using intrinsics, and threading is implemented with OpenMP pragmas. For some reason, using OpenMP `parallel for` constructs led to the noticeable performance degradation, so we emulated them using barriers.

We compared against FFTW 3.2 alpha 2 and Intel IPP 5.3. FFTW was compiled with pthreads, and not with OpenMP, since this resulted in some performance degradation.

Generated libraries. We generated libraries for the six transforms shown in Table 6 using the breakdown rules shown in Table 2 and also others. The generated libraries were run in double precision (2-way vectorization) and allowed to use up to 4 threads. For the complex DFT, we benchmarked sizes 2^2–2^{20}, which spans all types of cache residency, including non-L2 resident sizes ($\geq 2^{16}$). These larger sizes, require extra memory hierarchy optimization (loop interchange, loop distribution, data copying to eliminate large striding, and online twiddle factor computation) as explained in [2]. We implemented these optimization as "meta" breakdown rules that operate on loops instead of transforms, and extra details are available in [14]. For other transforms we benchmarked 2-powers sizes up to $2^{16} = 65536$, i.e., the sizes that are L2 resident, because the extra optimizations, did not yet work with these transforms.

Because all of the considered transform algorithms are numerically stable, the accuracy of the generated libraries is roughly the same as the accuracy of FFTW and IPP.

We use pseudo-GFlops to show performance, which is standard for transforms. The complex DFT operations count is assumed to be $5n \log_2 n$; for k-tap length-n FIR filter it is assumed to be $(2k-1)n$; and for the other, real transforms it is assumed to be $2.5n \log_2 n$.

Search. As far as we know, Intel IPP does not use search. FFTW uses runtime performance adaptation by searching for the best recursion strategy. The generated library provides a way to select the recursion strategy by exposing the available degrees of freedom. We have manually implemented a generic dynamic programming search mechanism that can be inserted into each of our generated libraries to find the best recursion strategy, similarly to FFTW.

Scalar code. We benchmarked three scalar libraries: for the complex DFT generated from breakdown rule (2) and for the real DFT (RDFT) and the DCT of type 2 from the breakdown rules in Table 2. We benchmarked against FFTW only, since it is not possible to turn off vectorization in IPP. The results are shown in Fig. 4.

For the real and complex DFT the performance of small sizes (up to 64) is faster than FFTW, and starting at 128, it is the same as FFTW. Both libraries have base cases for sizes up to 64, and the recursion starts at 128. We conjecture that the generated library is faster for small sizes, because it also uses base cases for a plain DFT recursion step (1 in Fig. 2), while FFTW reuses the base cases for strided DFTs (2 in Fig. 2), which degrades performance due to unnecessary index computation.

For larger sizes, the performance is identical. This is not entirely surprising, since both libraries use the same FFT algorithm and similar optimizations.

For DCT-2 the generated library is considerably faster because we generate a library from a general radix DCT-2 algorithm given in Table 2. Implementing such a library manually would require a considerable effort, and since the DCT-2 is less commonly used, it was not done in FFTW and it is translated into a DFT instead.

Vector and parallel code. Next, we benchmarked the generated parallelized and vectorized libraries. Only complex DFT was able to utilize all 4 cores (starting at size 65536). We only show one line per library. For complex DFT it shows the best performance between 1, 2 and 4 threads, and for other transforms between 1 and 2 threads. For all

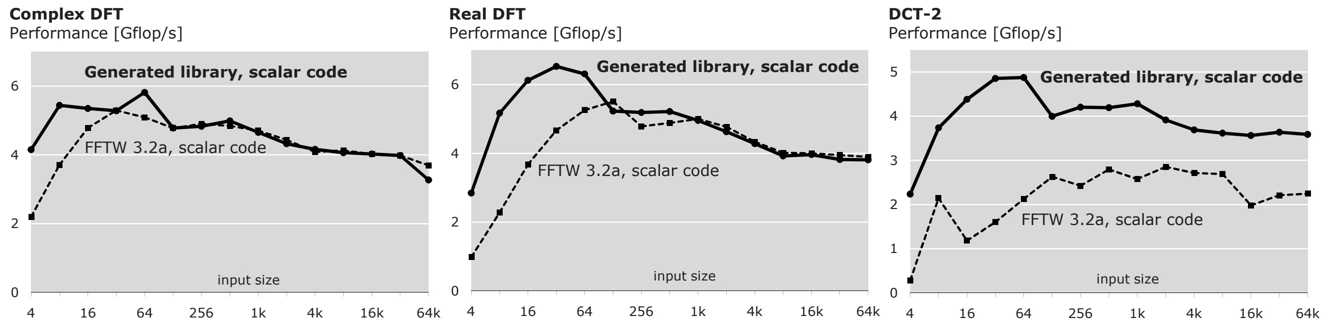


Figure 4: Generated scalar libraries versus scalar FFTW.

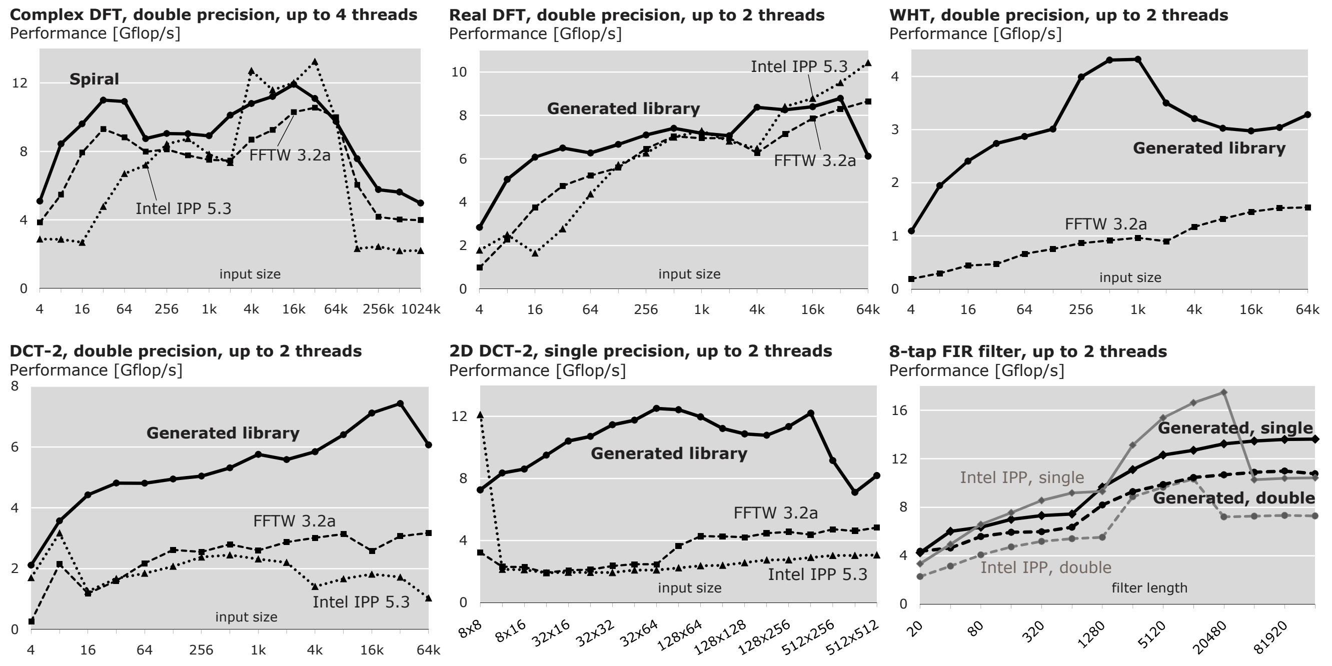


Figure 5: Performance of generated C++ vectorized (2-way for double precision, 4-way for single precision) and threaded libraries.

transforms the generated libraries start to benefit from using 2 threads as early as at sizes 1024 or 2048.

For the complex DFT, instead of the Cooley-Tukey FFT (2) we use the different FFT based on the RDFT from Table 2. It provides better performance for vector code, and provides another motivation for the use of a library generator.

For the DHT and DCTs, FFTW does not implement a native recursive algorithm with merged steps, but implements a conversion to the RDFT. The conversion works by preprocessing the input and postprocessing the output of the RDFT. The pre/post-processing are extra passes that result in performance degradation. Judging by the graphs IPP probably uses a similar conversion.

Finally, we note that the library generation required between 10 and 60 minutes in all cases.

Generator customization. One of the big advantages of library generation is the flexibility that it provides by enabling the generation of different *custom* libraries. This flexibility is not available in fixed hand-written libraries. Here we present just two examples out of many possibilities.

First, we retargeted the generator to produce native Java code. There is no support for vector instructions in Java, and we did not port the threading backend to Java threads, so only scalar code was produced. We show the performance of a few generated Java libraries in Fig. 6. Less effort is spent on Java numeric libraries and thus fewer optimized libraries exist that implement linear transforms. We found only one optimized implementations: open-source JTransforms [15] library for DFTs and DCTs, which provides one- and multi-dimensional DFTs, RDFTs, and DCTs. We did not find an optimized FIR filter Java library, so we benchmarked against the naive double-loop implementation.

Second, as an example of the qualitative customization, we show in Fig. 7 the tradeoff between the code size and performance. The different lines in the plot are obtained by changing the number of generated base case sizes for each recursion step type. The absolute numbers from Fig. 7

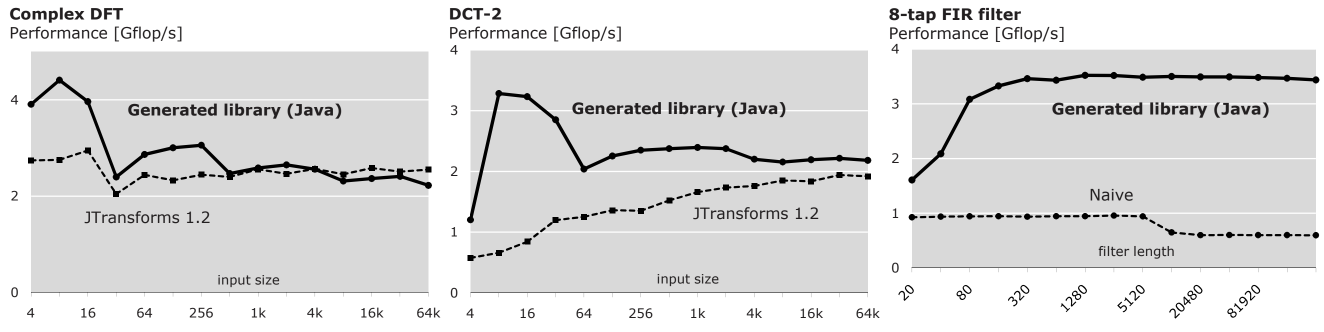


Figure 6: Performance of generated scalar Java libraries.

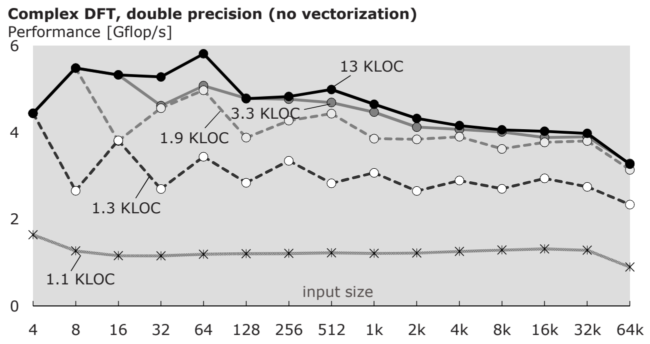


Figure 7: Code size / performance trade-off in scalar C++ libraries.

should not be directly compared to size-optimized human-written code, because the code size is still rather suboptimal and can be further reduced (e.g., by reducing the size of the recursion step closure). The relative reduction, however, manifests the customizability of the generator and is orthogonal to the other size optimizations which are possible.

6. Conclusions

Automating high performance library development is a problem at the core of computer science. In this paper we have shown that for an entire domain of structurally complex algorithms, complete high performance libraries can be generated automatically directly from the mathematical algorithm specification (Table 2), arguably a first in any domain. The key is a properly designed domain-specific language ($\sum$-SPL in our case) that makes it possible to perform all the difficult tasks (derivation of the library infrastructure including functions needed, parallelization, vectorization, loop optimizations, and others) at a high level of abstraction using rewriting and other techniques. Using these techniques, two seemingly conflicting goals can be achieved: complete automation and performance competitiveness with the best human-written code.

Note that Table 2 contains only a few of the available recursive algorithms, so more work is needed to find out which ones yield the fastest libraries. Another major goal is to extend this work to other domains.

Acknowledgments

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

References

[1] R. C. Whaley, A. Petitet, and J. Dongarra, “Automated empirical optimization of software and the ATLAS project,” *Parallel Computing*, vol. 27, no. 1–2, pp. 3–35, 2001.

[2] M. Frigo and S. G. Johnson, “The design and implementation of FFTW3,” *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005, special issue on ”Program Generation, Optimization, and Adaptation”.

[3] M. Frigo, “A fast Fourier transform compiler,” in *Proc. ACM PLDI*, 1999, pp. 169–180.

[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, pp. 232–275, 2005, special issue on ”Program Generation, Optimization, and Adaptation”.

[5] F. Franchetti, Y. Voronenko, and M. Püschel, “Loop merging for signal transforms,” in *Proc. ACM PLDI*, 2005, pp. 315–326.

[6] ——, “A rewriting system for the vectorization of signal transforms,” in *High Performance Computing for Computational Science (VECPAR)*, ser. Lecture Notes in Computer Science, vol. 4395. Springer, 2006, pp. 363–377.

[7] ——, “FFT program generation for shared memory: SMP and multicore,” in *Proc. Supercomputing*, 2006.

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

[9] J. Xiong, J. Johnson, R. Johnson, and D. Padua, “SPL: A language and compiler for DSP algorithms,” in *Proc. ACM PLDI*, 2001, pp. 298–308.

[10] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Addison-Wesley, 2006.

[11] T. Johnsson, “Lambda lifting: transforming programs to recursive equations,” in *Proc. Conf. on Functional Programming Languages and Computer Architecture*. Springer-Verlag, 1985, pp. 190–203.

[12] N. Glew, “Object closure conversion,” *Electronic Notes in Theoretical Computer Science*, vol. 26, 1999.

[13] O. Kiselyov, “A USENET article that discusses implementation of objects as functions (closures) in a non-pure and pure functional languages,” online: http://okmij.org/ftp/Scheme/oop-in-fp.txt.

[14] Y. Voronenko, “Library generation for linear transforms,” Ph.D. dissertation, Electrical and Computer Engineering, Carnegie Mellon University, 2008.

[15] P. Wendykier, “JTransforms 1.2,” April 2008, http://piotr.wendykier.googlepages.com/jtransforms.