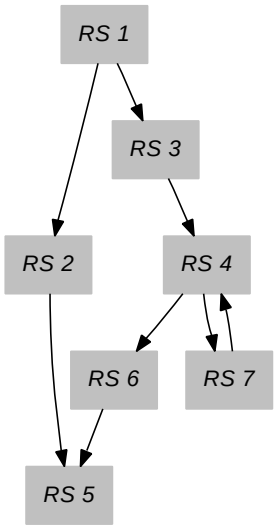
Library Generation For Linear Transforms

Yevgen Voronenko
May 2008



Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of
Doctor of Philosophy in
Electrical and Computer Engineering
Carnegie Institute of Technology
Carnegie Mellon University

Acknowledgements

The roots of this thesis lie in the grand vision of my advisor Markus Püschel. Back in 2000, after listening to one of his talks, I thought it was a neat idea to generate a full library from some very high-level rules that define the algorithm. Yet it was not at all clear that this would be possible.

I thank Markus for providing me with the never-ending source of inspiration and countless weekends proofreading drafts of various papers and this thesis.

This work would not be possible without major contributions made by Franz Franchetti, who developed a number of key mathematical insights, on which I heavily built my work. With Franz we spent long hours arguing about how our rewriting systems should and should not work.

I would like to extend my special thanks to Peter Tang, Dmitry Baksheev, and Victor Pasko whom I had a chance to work with at Intel. Peter provided me with great encouragement to work on new and exciting projects, just to "see what happens". I enjoyed having worked with Dmitry and Victor, and really hope to be able to continue our fruitful collaboration. I bring my sincere apologies for letting you guys suffer through my e-mail lag.

Many of the interesting experimental results of this thesis are possible due to work by Frédéric de Mesmay, who helped me just because it was interesting, and Srinivas Chellappa who maintained the machines used for benchmarking.

I would also like to thank all other members of the Spiral group for lots of interesting discussions and for suggesting many ideas that helped to improve this thesis.

I am very grateful to Jeremy Johnson who helped me enormously throughout my undergraduate years at Drexel University, and also got me interested in symbolic computation, which before I regarded as the most boring topic ever existed.

Many of the enabling ideas of this work go back to my internship at MathStar in Minneapolis with Andrew Carter and Steven Hawkins, with whom we worked on the hypothetical hardware compiler. I am thankful to Andrew and Steve for making my work enjoyable and challenging.

Finally, I would like to thank my wife and children for their tremendous patience and support during my seemingly never-ending time at graduate school.

Abstract

The development of high-performance numeric libraries has become extraordinarily difficult due to multiple processor cores, vector instruction sets, and deep memory hierarchies. To make things worse, often each library has to be re-implemented and re-optimized, whenever a new platform is released.

In this thesis we develop a library generator that completely automates the library development for one important numerical domain: linear transforms, which include the discrete Fourier transform, discrete cosine transforms, filters, and discrete wavelet transforms. The input to our generator is a specification of the transform and a set of recursive algorithms for the transform, represented in a high-level domain-specific language; the output is a C++ library that supports general input size, is vectorized and multithreaded, and provides an optional adaptation mechanism for the memory hierarchy. Further, as we show in extensive benchmarks, the runtime performance of our automatically generated libraries is comparable to and often even higher than the best existing human-written code, including the widely used library FFTW and the commercially developed and maintained Intel Integrated Performance Primitives (IPP) and AMD Performance Library (APL).

Our generator automates all library development steps typically performed manually by programmers, such as analyzing the algorithm and finding the set of required recursive functions and base cases, the appropriate restructuring of the algorithm to parallelize for multiple threads and to vectorize for the available vector instruction set and vector length, and performing code level optimizations such as algebraic simplification and others. The key to achieving full automation as well as excellent performance is a proper set of abstraction layers in the form of domain-specific languages called SPL (Signal Processing Language), index-free $\sum$-SPL, regular $\sum$-SPL, and intermediate code representation, and the use of rewriting systems to perform all difficult optimizations at a suitable, high level of abstraction.

In addition, we demonstrate that our automatic library generation framework enables various forms of customization that would be very costly to perform manually. As examples, we show generated trade-offs between code size and performance, generated Java libraries obtained by modifying the backend, and functional customization for important transform variants.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Many compute intensive applications process large data sets or operate under real-time constraints, which poses the need for very fast software implementations. Important examples can be grouped into scientific computing applications such as weather simulations, computer aided drug discovery, or oil exploration, consumer and multimedia applications such as image and video compression, medical imaging, or speech recognition, and embedded applications such as wireless communication, signal processing, and control.

In many of these applications the bulk of the computation is performed by well-defined mathematical functionality and application developers rely on high-performance numerical libraries that provide this functionality. If these libraries are fast then the applications that use them will run fast too. As computing platforms change, ideally only the library has to be updated to port the application and take advantage of new platform features. Commonly used numeric library domains include dense and sparse linear algebra, linear transforms, optimization, coding, and others.

However, as we will explain, the development of libraries that achieve the best possible performance has become very difficult and costly due to the evolution of computer platforms. This thesis aims at completely automating the library development and library optimization without sacrificing performance for one of the most commonly used domains: linear transforms.

The most prominent example of a linear transform is the discrete Fourier transform (DFT), which is arguably among the most important tools used across disciplines in science and engineering. Other important linear transforms include digital filters, discrete cosine and sine transforms, and wavelet transforms.

To better understand the problem of developing high performance libraries, and hence the motivation for this thesis, we start by analyzing the evolution of computer platforms.

1.1 Platform Evolution and the Difficulty of Library Development

Figure 1.1 shows the evolution of both clock frequency and floating-point peak performance (single and double precision) of Intel platforms over the past 15 years. The growth pattern identifies the key problems in the development of high performance libraries, as explained next.

The plot shows that early on the growth rates of CPU frequency and peak performance were coupled. However, in recent years CPU frequency has stalled (at around 3 GHz), while the peak performance continues to increase (even at a faster rate). This means that hardware vendors have turned to increasing on-chip parallelism to improve peak performance. On the other hand,

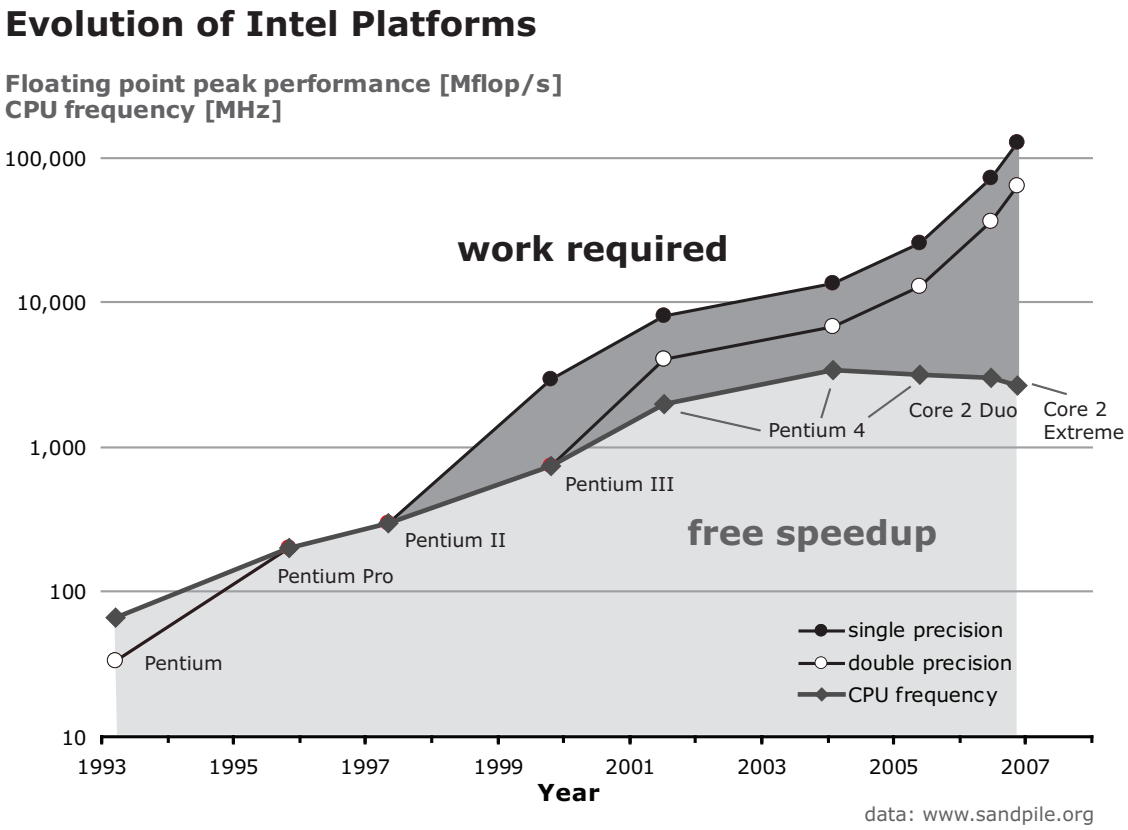


Figure 1.1: Evolution of Intel platforms.

clock frequency cannot keep growing due to physical limitations. This paradigm shift means that any further performance improvement in numerical libraries will largely depend on the effective exploitation of the available on-chip parallelism, which is usually difficult. While clock frequency scaling translates into "free" speedup for numeric libraries across time, exploiting the on-chip parallelism usually requires programming effort.

Modern processors have three types of on-chip parallelism: instruction-level parallelism, vector (SIMD) parallelism, and thread parallelism. Instruction-level parallelism enables the processor to execute multiple instructions concurrently, by means of out-of-order execution. This is the only type of parallelism that requires no software development effort, even though it may require proper instruction scheduling for optimal efficiency.

Vector or SIMD (single instruction multiple data) parallelism is realized by adding vector extensions to the instruction set that enable parallel processing of multiple data elements. The only ways to exploit SIMD parallelism are to use assembly code, to use special C/C++ intrinsics, or to rely on automatic compiler vectorization. The latter tends to be far suboptimal for most numerical problems.

Finally, thread parallelism is the coarse-grain parallelism provided by incorporating multiple processor cores into a single CPU. Exploiting this type of parallelism requires programs with multiple threads of control.

To complicate matters further, the exponential growth of peak performance has not been

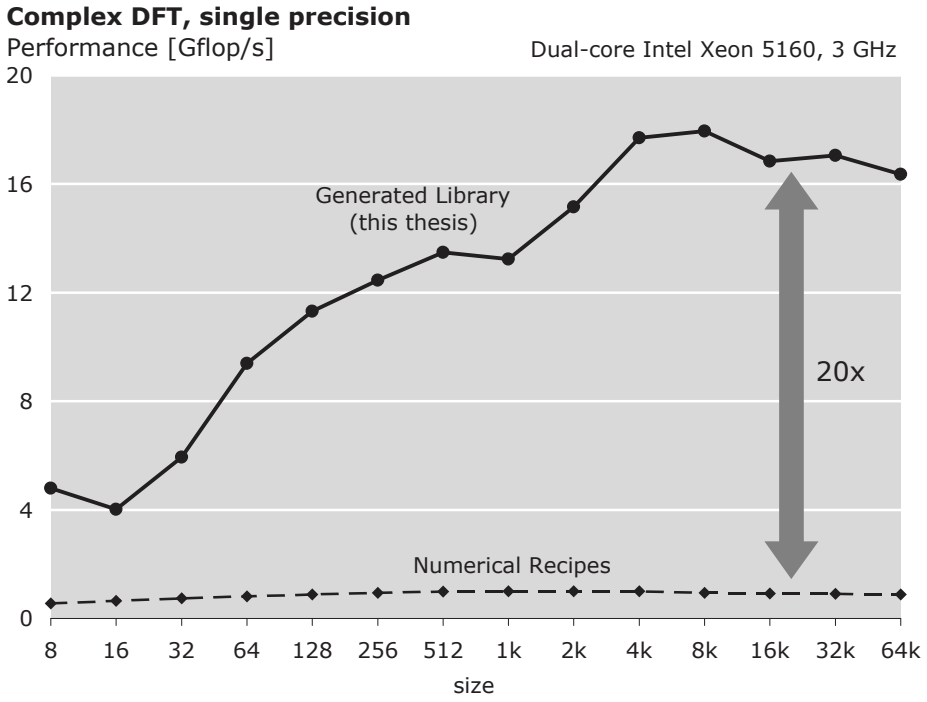


Figure 1.2: Numerical recipes versus the optimized implementation of the DFT .

matched by the increase in memory bandwidth (not shown in Fig. 1.1), leading to the processor–memory bottleneck problem and the emergence of multi-level cache memory hierarchies, multiple memory banks, and prefetch and temporal load/store instructions, designed to alleviate the problem. To achieve the best possible performance a library hence has to also be optimized and "tuned" to the peculiarities of the memory hierarchy.

As a result, library development is very time consuming and expensive, requires expert domain knowledge, expert platform knowledge, and expert programming skills, which means very few people are trained to do it. Further, performance is in general not portable, which means that libraries have to be reoptimized or even reimplemented for every new platform.

To better illustrate these problems consider the plot in Fig. 1.2, which compares the reference implementation of the discrete Fourier transform (DFT) from "Numerical Recipes" [94], a standard book on numerical programming, and a highly platform optimized implementation, obtained automatically using the methods in this thesis. The platform is the latest dual-core workstation with an Intel Xeon 5160 processor. The performance difference is up to a surprising factor of 20, even though both implementations have roughly the same operations count. To obtain the best performance, the optimized library is 4-way vectorized using platform-specific SIMD instructions, uses up to 2 threads, and is optimized for the memory hierarchy.

The "Numerical Recipes" implementation was compiled using the latest (at the time of writing) Intel C Compiler 10.1. Although the compiler provides automatic vectorization and parallelization, it could not parallelize or vectorize any parts of this program. This is typical for non-trivial functionality like the DFT, because these optimizations require domain knowledge in the form of sophisticated high-level algorithm manipulations, which are not feasible with a standard compiler.

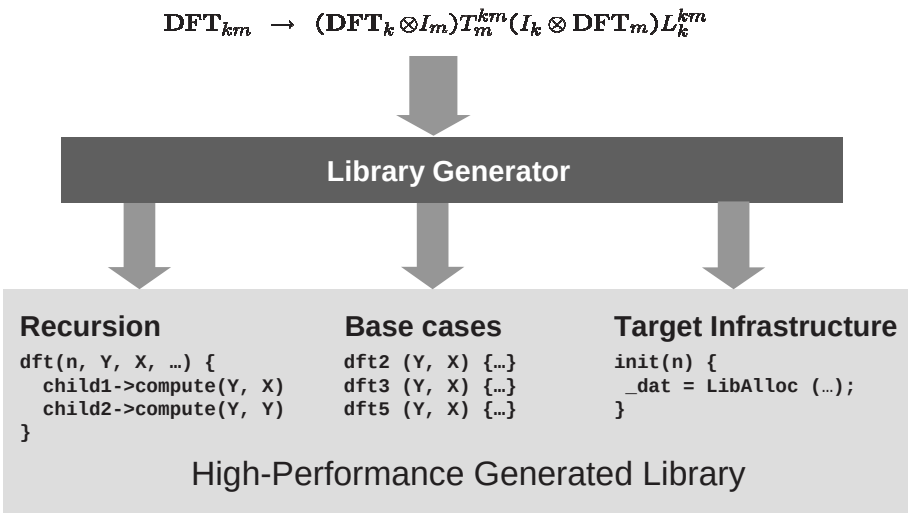


Figure 1.3: Library generator: input and output. The generated high performance libraries consist of three main components: recursive functions, base cases, and additional infrastructure.

1.2 Goal of the Thesis

The goal of this thesis is the computer generation of high performance libraries for the entire domain of linear transforms, given only a high level algorithm specification, request for multithreading, and SIMD vector length.

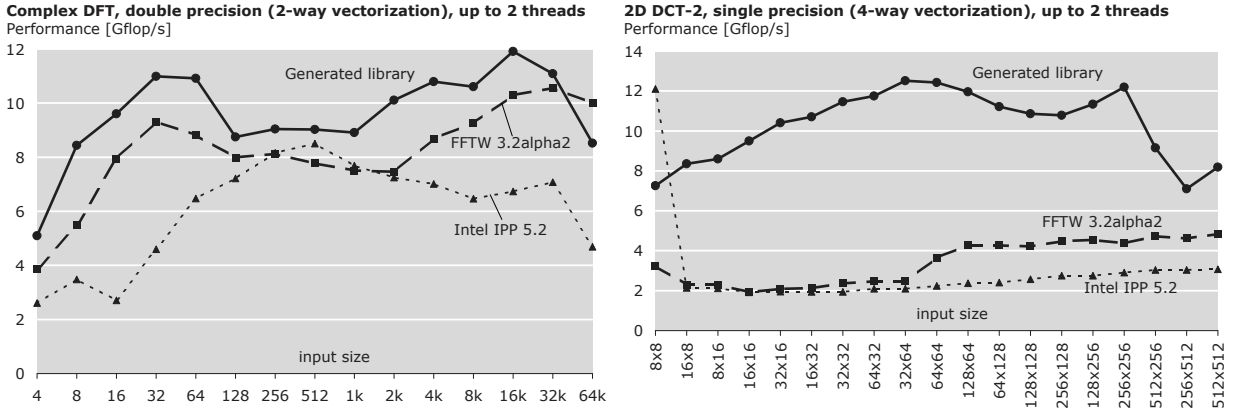
In other words, we want to achieve *complete* automation in library development for an entire domain of structurally complex numerical algorithms. This way the cost of library development and maintenance is dramatically reduced and the very fast porting to new platforms becomes possible. Further, library generation provides various other benefits. For example, it expands the set of functionality for which high performance is readily available, and it enables library customization.

We generate transform libraries given only a high-level specification (in a domain-specific language) of the recursive algorithms that the library should use. For example, a typical input to our library generator is

$$\begin{aligned} \text{Transform: } & \mathbf{DFT}_n, \\ \text{Algorithms: } & \mathbf{DFT}_{km} \rightarrow (\mathbf{DFT}_k \otimes I_m)\, \mathrm{diag}(\Omega_{k,m})(I_k \otimes \mathbf{DFT}_m) L^{km}_k, \\ & \mathbf{DFT}_2 \rightarrow \left[\begin{smallmatrix} 1 & 1 \\ 1 & -1 \end{smallmatrix}\right], \\ \text{Vectorization: } & \text{2-way SSE} \\ \text{Multithreading: } & \text{yes} \end{aligned}$$

The output is a generated library that is

- for general input size;
- vectorized using the available SIMD vector instruction set;
- multithreaded with a fixed or variable number of threads;



(a) DFT library (15,000 lines of code, 0.75 MB)

(b) 2D DCT-2 library (27,000 lines of code, 2.1 MB)

Figure 1.4: Performance of automatically generated DFT and 2D DCT-2 libraries. Platform: Dual-core Intel Xeon 5160, 3 GHz.

- (optionally) equipped with a feedback-driven platform adaptation mechanism to select the best recursion strategy at runtime, and thus to automatically adapt to the underlying memory hierarchy and other platform-specific details;
- performance competitive with the best existing hand-written libraries.

The last point is particularly important, it means that even though we completely automate library development, there is no performance loss compared to an expertly hand-written and hand-tuned library. In fact, as we will show, our generated libraries often outperform the best existing hand-written code.

Fig. 1.3 illustrates the concept of a library generator. The generated libraries consist of three main components whose exact form is automatically derived and implemented by the generator: a) a set of recursive functions needed to compute the transform; b) a set of optimized base case functions that implement small size transforms and terminate the recursion; c) an infrastructure responsible for initialization, such as precomputations, and adaptive search. Not shown in Fig. 1.3 is the efficient use of vector instructions and threading, also obtained automatically.

The structure of the generated libraries is roughly modeled after FFTW [61], a widely used DFT library. We discuss FFTW later together with other related work.

The target platforms we consider are modern workstations, such as systems based on Intel Core 2 Duo, and AMD Opteron processors. These workstations have deep memory hierarchies, SIMD vector extensions, and shared memory multicore processors.

As an example result we show in Figure 1.4 the performance of two automatically generated libraries, for the DFT and for the 2-dimensional DCT-2 (discrete cosine transform of type 2). We compare the performance of our generated library to the best available (hand-written) libraries: FFTW [61] and Intel IPP on an Intel Xeon 5160 (server variant of Core 2 Duo) processor machine. We show performance in pseudo Gflop/s, which is standard practice in the domain of transforms. Pseudo Gflop/s are computed as

$$\frac{\text{normalized arithmetic cost}}{\text{runtime [sec]}} \cdot 10^9,$$

assuming the normalized arithmetic cost (number of additions and multiplications) of $5n \log n$ for the DFT of size n, and $2.5mn \log mn$ for the $m \times n$ 2-dimensional DCT-2. Because of the normalized arithmetic cost, pseudo Gflop/s are proportional to inverse runtime.

In the case of the DFT, where considerable development effort is spent, the performance is comparable, even though our generated library is for most sizes somewhat faster than both competitors. IPP mainly suffers from the lack of threading. For the DCT-2, however, where less development effort is invested, the performance of the generated library is on average a factor of 4 faster than FFTW and IPP. In FFTW (and most likely IPP) the DCT is implemented using a conversion to a real-data DFT. The primary reason is to reuse code and to save on the implementation effort. This indirect method results in performance degradation. In contrast, our generated library is a "native" DCT library that offers the same performance as the DFT.

Note the large size of our generated libraries (15,000 and 27,000 lines of code, respectively). This is typical for high-performance libraries since necessary optimizations include loop and recursion unrolling and also require the introduction and implementation of various versions of functions used for efficient recursive computation. This need for large code size is another argument for automatic library generation.

1.3 Related Work

The high development costs for numerical libraries, the need to reoptimize libraries for every new platform, and in some cases the lack of desired functionality have spawned academic research efforts on automating the library development and optimization process. Most of the automation work is focused on the performance-critical domains of linear algebra and linear transforms, but there are also research efforts in other domains.

Domain: Linear transforms. In the domain of linear transforms, the DFT libraries FFTW [61] and UHFFT [84] partially automate the development process by employing a limited form of code generation and automatic runtime selection of the best DFT algorithm.

Specifically, FFTW and UHFFT use a special "codelet generator" [5, 59] to generate fully unrolled code for small fixed size transform functions, called "codelets". These codelets serve as a building blocks for larger transforms. There are many different types of codelets, and the codelet generators are used to automatically generate a cross-product of the required codelet types and desired transform sizes.

The second form of automation is the runtime selection of the best algorithm. Both libraries provide hand-written recursions for the transform computations, and the base cases of these recursions are the codelets. There are multiple alternative recursions, and at runtime the fastest alternative is chosen using a heuristic feedback-driven search. Since the best choice is platform dependent, this search provides a form of automatic platform tuning.

Compared to our generated libraries in Fig. 1.3 FFTW and UHFFT automate the base case development, but, besides that, are still hand-written libraries based on a sophisticated design. In particular, the design includes vectorization, parallelization, and recursion structure to achieve high performance. As a consequence extending FFTW or developing a new FFTW-like library is a major undertaking. For example, consider the manual steps necessary to add a new DFT algorithm to one of these libraries. First, the general-size algorithm has to be hand-developed to obtain a new transform recursion. The recursion will typically decompose the original DFT "problem" into smaller problems, which could be other transforms, or other DFT types. Next, the code has to be

analyzed to determine if new problem types (i.e. transform variants) are required (which is usually the case), and if yes, then either the new recursions must be implemented for the new problem types, or existing recursions must be modified to support the new problem types. Finally, the code is analyzed to determine the set of required codelet types, and if new codelet types are required, then the codelet generator is modified to be able to generate them.

In addition to the above steps, considerable effort is required for efficient vectorization and parallelization. The parallelization will most likely affect only the top-level recursion. However, vectorization may require new problem types, and thus a waterfall of changes, including new codelet types, and modifications to the codelet generator.

Inspired by the ingenious design of FFTW, our goal is to eliminate *all* manual work and generate complete libraries like FFTW, for a large set of linear transforms.

Another effort in the domain of linear transforms is a program generator Spiral [99], which served as the foundation of our work. We will discuss Spiral later.

Domain: Dense linear algebra. Virtually all dense linear algebra functionality depends on the efficient implementation of basic matrix operations called BLAS (basic linear algebra subroutines). The most prominent dense linear algebra library is the widely used LAPACK [8], The BLAS are typically implemented and reoptimized by the hardware vendors for each new platform. For example, high-performance platform optimized BLAS are provided by the Intel MKL and AMD ACML libraries.

ATLAS [130,131] (Automatically Tuned Linear Algebra Subroutines) is an automatically tuned BLAS library, which uses a code generator together with parameter search mechanism to automatically adapt to the underlying platform. Most of the critical BLAS functionality reduces to several flavors of matrix-matrix multiply (MMM) operation, and thus the focus of ATLAS is to provide a very high-performance MMM implementation. ATLAS uses a code generator to generate several variants of the performance critical inner kernel of the MMM, the so-called micro-MMM. In addition, ATLAS uses feedback-driven search to find the best values of the implementation parameters, for example, matrix block size and the best choice of the inner kernel.

ATLAS provides only partial automation. The (hand-written) ATLAS infrastructure provides threading, but not vectorization, which should be done as optimization in the micro-MMM kernel. The ATLAS code generator, however, does not generate vectorized micro-MMM kernels. If vectorization is desired, as is the case with practically all modern processors, then the micro-MMM kernel has to be written again by hand.

The FLAME [23] (Formal Linear Algebra Method Environment) project enables the semi-automatic derivation of dense linear algebra matrix algorithms (such as the ones in LAPACK) from the concise specification of a *loop invariant* and the so-called *partitioned matrix expression*. The generated algorithms can be translated into code using FLAME APIs [24], and even into parallel code [138] by combining FLAME APIs and task queue parallelism. The code relies on the availability of fast, vectorized BLAS.

Domain: Sparse linear algebra. Most of the sparse linear algebra functionality is built around the sparse matrix-vector product (SpMV), which hence becomes the performance-critical function in this domain. The goal of the OSKI (Optimized Sparse Kernel Interface, earlier called Sparsity) project [44, 68] is to automate the optimization and tuning of the SpMV. The overall design is similar to ATLAS, the library uses a set of generated kernels which perform small dense matrix-vector products of different block sizes, and at runtime the library chooses the best algorithm and the best kernel size, employing a combination of a heuristic model, and a runtime benchmark.

Unlike ATLAS, in the sparse case the best choices depend on the sparsity pattern of the matrix, and thus these decisions must be done dynamically at runtime, rather at code generation time, as in ATLAS.

Other domains. The tensor contraction engine [15, 16] generates code for tensor contractions from a specification in a domain-specific language. The tool performance code reorganization in order to jointly optimize for the minimal arithmetic cost and the minimal amount of temporary storage.

Adaptive sorting libraries [22, 79, 80] include several sorting algorithms, and use a runtime selection mechanism to determine the best algorithm from the statistical properties of the input data.

Generative programming and domain-specific languages. Independent of the above work, the idea of program generation (programs that write other programs), also called generative programming, has recently gained considerable interest in the programming language and, to some extent, the software engineering community [1, 13, 14, 39, 111]. The basic goal is to reduce the development, maintenance, and analysis of software. Among the key tools for achieving these goals are domain-specific languages that raise the level of abstraction for specific problem domains and hence enable the more compact representation and the manipulation of programs [19, 40, 65, 67]. However, this community has to date not considered numerical problems nor performance optimization as goal.

In this thesis we show that domain-specific languages and generative programming techniques can be used for numerical problems and, more importantly, to achieve very high performance. The key is to design the domain-specific language based on the mathematics of the domain; this way, sophisticated optimizations such as parallelization and vectorization can be done at a high level of abstraction using rewriting systems.

Platform Vendor Libraries. A major part of the numerical library development is done by hardware platform vendors, which provide optimized high performance implementations of important numerical algorithms for many domains, including linear transforms. Examples include the Intel Math Kernel Library (MKL) and Intel Integrated Performance Primitives (IPP), the AMD Performance Library (APL) and AMD Core Math Library (ACML), IBM ESSL, and the Sun Performance Library.

For the library user these hardware vendor libraries make it possible to tap into the high performance of available platforms, without delving into low-level implementation details. At the same time they enable the porting to new platforms provided the libraries have been updated by the vendor.

Developing and maintaining these libraries is becoming more and more expensive for the reasons we discussed earlier. This makes automation an important goal with real world impact.

As part of the work on this thesis and as our joint effort with Intel [11] some of our generated code was included in the Intel MKL starting with version 9.0. A much larger set of our generated code will be included in Intel IPP, which, starting with version 6.0 in late 2008, will provide a special domain, which consists exclusively of our automatically generated libraries. The domain is called ippg, where the "g" stands for generated. The introduction of ippg may mark the first time that the development of performance libraries is done by a computer rather than a human.

(a) The first generation of Spiral reported in [99].

Code type	Scalar	Vectorized	Threaded	Threaded & vectorized
Fixed size, straightline	all	**DFT**	n/a	n/a
Fixed size, looped	**DFT**	**DFT**	-	-
Fixed size, looped with reuse	-	-	-	-
General size library, recursive	-	-	-	-

(b) The second generation of Spiral.

Code type	Scalar	Vectorized	Threaded	Threaded & vectorized
Fixed size, straightline	all	**DFT**	n/a	n/a
Fixed size, looped	most	**DFT**	**DFT**	**DFT**
Fixed size, looped with reuse	most	**DFT**	**DFT**	**DFT**
General size library, recursive	**DFT**	-	-	-

(c) The third generation of Spiral: the contribution of this thesis.

Code type	Scalar	Vectorized	Threaded	Threaded & vectorized
Fixed size, straightline	all	all	n/a	n/a
Fixed size, looped	all	all	all	all
Fixed size, looped with reuse	all	all	all	all
General size library, recursive	all	all	all	all

Table 1.1: Generations of Spiral. The columns correspond to the algorithm optimizations and the rows represent different types of code. Each entry indicates for what transforms such code can be generated. The code types are illustrated in Table 1.2. "All" denotes all transforms in Table 2.1 shown in Chapter 2. "Most" excludes the discrete cosine/sine transforms.

1.4 Contribution of the Thesis

Our contribution, to the best of our knowledge, is the design and implementation of the first system that completely automates high performance library development for an entire numerical domain and for state-of-the-art workstation computers. The domain are linear transforms, which have structurally very complex algorithms as we will see later. The word "high performance" is important since our generated libraries match, and often supersede, the performance of the best hand-written libraries. In other words, we achieve complete automation without sacrifices in performance.

Equally important as the abilities of our library generator are the methods used. Namely, the framework underlying our generator is based on domain-specific languages and rewriting systems, tools that to date have barely been used in the area of library development or performance optimization (except, to some extent, for the prior work on Spiral explained next).

To explain our contributions to greater detail, we first discuss prior work on Spiral that is relevant for this thesis.

Prior work. Unlike FFTW and UHFFT discussed earlier, Spiral is not a library, but a program generator, which can generate standalone programs for transforms. In contrast to other work on

(a) Fixed size, unrolled

```
void dft_4(complex *Y, complex *X)
{
  complex 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(complex *Y, complex *X)
{
  complex T[4];
  for(int i = 0; i <= 1; i++) {
    T[2*i] = D[2*i]*(X[i] + X[i+2]);
    T[2*i+1] = D[2*i+1]*(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) Fixed size, looped with reuse

```
void dft_4(complex *Y, complex *X)
{
  for(int i = 0; i <= 1; i++) {
    dft_2_1(T, X, 2*i, D+2*i, i);
  }
  for(int j = 0; j <= 1; j++) {
    dft_2_2(Y, T, j, j);
  }
}
```

(d) General size library, recursive

```
struct dft : public Env {
  dft(int n); // constructor
  void compute(complex *Y, complex *X);

  int _rule, f, n;
  char  *_dat;
  Env  *child1, *child2;
};

void dft::compute(complex *Y, complex *X)
{
  child2->compute(Y, X, n, f, n, f, f);
  child1->compute(Y, Y, n, f, n, n/f, n/f);
}
```

Table 1.2: Code types.

automatic performance tuning, Spiral uses internally a domain-specific language called SPL (signal processing language). In SPL, divide-and-conquer algorithms for transforms are expressed as rules and stored in Spiral. For a user-specified transform and transform size, Spiral applies these rules to generate different algorithms, represented in SPL. An SPL compiler translates these algorithms into C code, which is further optimized. Based on the runtime of the generated code, a search engine explores different choices of algorithms to find the best match to the computing platform.

The first generation of Spiral is described in full detail in [99] based on the papers [50, 51, 100, 108, 110, 137]. We introduce all relevant details as background in Chapter 2.

Table 1.1(a) shows the abilities of this first generation of Spiral, described in detail in [99]. The types of code referred to in the rows of this table are illustrated in Table 1.2. The main limitations of the the first generation of Spiral were the inability to

- generate loop code for transforms other than the DFT based on the Cooley-Tukey FFT algorithm;

- generate vector code for transforms other than the DFT;
- generate multithreaded code;
- generate general input size libraries.

The reasons for these limitations were rooted in the underlying framework. First, it could not support generality across transforms, except for the generation of scalar, straightline code, similar to FFTW. Second, the optimizations for loop code and vector code were, in a sense, "hard-coded" using a mechanism that did not allow for generalization to a large class of transforms. Third, there was no support for multithreading. Fourth, and most importantly, Spiral's approach was limited to generating code only for a given transform of fixed input size, which greatly simplifies the problem as all optimizations can be inlined and there is no control flow in the program. The automatic generation of a complete, general input size library like FFTW or others was out of reach.

Contribution of this thesis. Our contribution is the complete design and complete implementation of what we call the "third generation of Spiral" with the abilities shown in Table 1.1(c). As an intermediate step, and part of our thesis work, we first designed and implemented from scratch the second generation of Spiral with the abilities shown in Table 1.1(b). This second generation already included new concepts and ideas to overcome some of the limitations of the first generation, in particular general framework for loop merging [52], a redesigned and more general vectorization framework [54] and preliminary multithreading [53].

Full generality across transforms, across the types of code, and, in particular, the framework to generate general input size libraries was achieved with the third generation of Spiral, which is the main subject of this thesis.

To achieve the abilities in Table 1.1(c), our library generator includes and integrates a number of novel ideas and techniques from the general areas of programming languages, symbolic computation, and compilers. We list the most important techniques developed in this thesis:

- We introduce a new mathematical domain-specific language, called $\sum$-SPL, to symbolically represent transform algorithms. $\sum$-SPL is an extension to SPL [137], which is at the core of the first generation of Spiral. $\sum$-SPL is of mathematical nature, but makes explicit the descriptions of loops, index mappings, parametrization, and recursive calls. We further develop extensions to $\sum$-SPL necessary for the generation of general-size libraries explained below.

- We design and use rewriting systems for performing difficult optimizations such as parallelization, vectorization, and loop merging automatically. The rewriting operates on the $\sum$-SPL representation of algorithms, i.e., on a high abstraction level. This way these optimizations and become feasible and overcome known compiler limitations.

- We design and use a rewriting framework to automatically derive the *$\sum$-SPL recursion step closure*, a term introduced in this thesis to describe the smallest set of mutually recursive functions (expressed in $\sum$-SPL) sufficient to compute a given set of transforms. This framework requires $\sum$-SPL extensions including the loop non-terminal, ranked functions, and index-free $\sum$-SPL, all introduced in this thesis.

- We developed a compiler that translates the $\sum$-SPL recursion step closure into actual code in an intermediate representation. The compiler further optimizes the code using several

standard compiler techniques. We implemented two target language backends which translate this intermediate code representation into Java or C++, and generates the necessary initialization and glue code to obtain the desired libraries.[1]

- Finally, we integrate all the above components into a completely automatic, "push-button," library generation system that we designed and implemented. The system requires additional software engineering techniques such as conditional term-rewriting and various object-oriented extensions to simplify programming and other tasks.

1.5 Organization of the Thesis

The rest of this thesis is organized as follows. In Chapter 2 we review background on transforms, their fast algorithms, explain the Spiral framework, and discuss $\sum$-SPL and loop merging optimization, which will be an important component for library generation. In Chapter 3 we overview the the general size library generation process, explain the derivation of the library structure, which computes the recursion step closure using $\sum$-SPL, and also explain several restructuring loop optimizations which can be done on the $\sum$-SPL level. In Chapter 4, we describe the parallelism framework, which enables automatic parallelization and vectorization at the SPL and $\sum$-SPL levels. In Chapter 5 we describe the library generation backend, which compiles the recursion step closure into the target language code, in particular, we explain the C++ target. In Chapter 6 we show experimental results, comparing the performance of a variety of generated libraries to the state of the art hand-written libraries. In Chapter 7 we talk about future work and conclude.

[1] Parts of this backend were developed jointly with Frédéric de Mesmay.

Chapter 2

Background

In this chapter, we give background on linear transforms and their fast algorithms. We describe the declarative representation of fast algorithms, called SPL, and show how this representation can be translated into code. Next, we overview Spiral, a program generator for linear transforms, which uses SPL to automatically generate code for linear transforms and underlies our work. Finally, we present $\sum$-SPL, a lower level extension of SPL, which solves the fundamental problem of loop merging in SPL, and enables robust looped code generation, by following the path SPL $\rightarrow$ $\sum$-SPL $\rightarrow$ code.

2.1 Transforms

Any linear signal transform can be defined as a matrix-vector product

$$y = Mx,$$

where M is the fixed transform matrix, and x, y are the input and output vectors, respectively. M can be real or complex and correspondingly x and y are real or complex. There is a large number of well-known linear transforms. We list the most important ones below along with relevant citations.

- Discrete Fourier transform (DFT) [121, 122];
- Walsh Hadamard transform (WHT) [17, 18];
- Real discrete Fourier transform (RDFT, also known as DFT of real data or real-valued DFT) [20, 98, 112, 125];
- Discrete Hartley transform (DHT) [31, 32, 98, 125, 126];
- Discrete cosine and sine transforms of types 1–4 (DCT-t and DST-t, with $t \in \{1, 2, 3, 4\}$) [33, 96, 97, 103];
- Modified discrete cosine transform (MDCT) and its inverse (IMDCT) [82]
- FIR filter [62, 93];
- Downsampled FIR filter (as part of the wavelet transform) [62, 113, 124].

Main transforms

$$\mathbf{DFT}_n = \left[\omega_n^{k\ell}\right]_{0\le k,\ell<n}, \quad \omega_n = e^{-2\pi j/n} \qquad \mathbf{WHT}_n = \begin{bmatrix} \mathbf{WHT}_{n/2} & \mathbf{WHT}_{n/2} \\ \mathbf{WHT}_{n/2} & -\mathbf{WHT}_{n/2} \end{bmatrix}$$

$$\mathbf{RDFT}_n = \begin{bmatrix} I''_n \\ \begin{bmatrix} \cos\frac{2\pi k\ell}{n} \\ -\sin\frac{2\pi k\ell}{n} \end{bmatrix} \end{bmatrix} \qquad \mathbf{DHT}_n = \begin{bmatrix} I''_n \\ \begin{bmatrix} \mathrm{cas}\frac{2\pi k\ell}{n} \\ \mathrm{cms}\frac{2\pi k\ell}{n} \end{bmatrix} \end{bmatrix}$$

$$\mathbf{DCT\text{-}1}_n = \left[\cos\frac{k\ell\pi}{n-1}\right] \qquad \mathbf{DST\text{-}1}_n = \left[\sin\frac{(k+1)(\ell+1)\pi}{n+1}\right]$$

$$\mathbf{DCT\text{-}2}_n = \left[\cos\frac{k(2\ell+1)\pi}{2n}\right] \qquad \mathbf{DST\text{-}2}_n = \left[\sin\frac{(k+1)(2\ell+1)\pi}{2n}\right]$$

$$\mathbf{DCT\text{-}3}_n = \left[\cos\frac{(2k+1)\ell\pi}{2n}\right] \qquad \mathbf{DST\text{-}3}_n = \left[\sin\frac{(2k+1)(\ell+1)\pi}{2n}\right]$$

$$\mathbf{DCT\text{-}4}_n = \left[\cos\frac{(2k+1)(2\ell+1)\pi}{4n}\right] \qquad \mathbf{DST\text{-}4}_n = \left[\sin\frac{(2k+1)(2\ell+1)\pi}{4n}\right]$$

$$\mathbf{MDCT}_n = \left[\cos\frac{(2k+1)(2\ell+1+n)\pi}{4n}\right] \qquad \mathbf{IMDCT}_n = \left[\cos\frac{(2\ell+1)(2k+1+n)\pi}{4n}\right]$$

$$\mathbf{Filt}_n(t) = \begin{bmatrix} t_0 & \dots & t_{k-1} & & \\ & \ddots & \dots & \ddots & \\ & & t_0 & \dots & t_{k-1} \end{bmatrix} \qquad \downarrow 2\,\mathbf{Filt}_n(t) = \begin{bmatrix} t_0 & t_1 & \dots & t_{k-1} & & & \\ & & t_0 & t_1 & \dots & t_{k-1} & \\ & & & \ddots & \dots & \dots & \ddots \\ & & & & t_0 & \dots & t_{k-1} \end{bmatrix}$$

Auxiliary transforms

$$\mathbf{RDFT\text{-}2}_n = \begin{bmatrix} I''_n \\ \begin{bmatrix} \cos\frac{\pi k(2\ell+1)}{n} \\ -\sin\frac{\pi k(2\ell+1)}{n} \end{bmatrix} \end{bmatrix} \qquad \mathbf{DHT\text{-}2}_n = \begin{bmatrix} I''_n \\ \begin{bmatrix} \mathrm{cas}\frac{\pi k(2\ell+1)}{n} \\ \mathrm{cms}\frac{\pi k(2\ell+1)}{n} \end{bmatrix} \end{bmatrix}$$

$$\mathbf{RDFT\text{-}3}_n = \begin{bmatrix} \cos\frac{\pi(2k+1)\ell}{n} \\ -\sin\frac{\pi(2k+1)\ell}{n} \end{bmatrix} \qquad \mathbf{DHT\text{-}3}_n = \begin{bmatrix} \mathrm{cas}\frac{\pi(2k+1)\ell}{n} \\ \mathrm{cms}\frac{2\pi(2k+1)\ell}{n} \end{bmatrix}$$

$$\mathbf{DCT\text{-}3}_n(r) = \left[\cos\frac{r_k\ell\pi}{n}\right] \qquad \mathbf{DST\text{-}3}_n(r) = \left[\sin\frac{r_k(\ell+1)\pi}{n}\right]$$

Table 2.1: Linear transforms and their defining matrices.

In Table 2.1 we show the respective transform matrices. Besides these main transforms, Table 2.1 also shows some *auxiliary* transforms. These are rarely used in practical applications, but arise as building blocks of fast algorithms for the main transforms.

In each case the subscript n specifies the length of the input vector x, i.e., the number of columns of the matrix. In Table 2.1, all transforms are $n \times n$ matrices, except the $\mathbf{IMDCT}_n$ which is $2n \times n$, and the $\mathbf{MDCT}_n$ which is $n \times 2n$. Transforms are always bold-faced in this thesis.

The goal of our work is to enable library generation for these and other linear transforms. In Chapter 6 we show generated libraries for a large subset of transforms in Table 2.1.

2.2 Fast Transform Algorithms: SPL

A generic matrix-vector product for an $n \times n$ matrix requires $\Theta(n^2)$ arithmetic operations [34]. In contrast, for most transforms there exist fast algorithms that exploit the structure of the transform to reduce the complexity to $O(n \log n)$. Every fast algorithm can be viewed as a factorization of the dense transform matrix M into a product of structured sparse matrices. Namely, if $M = M_k M_{k-1} \dots M_1$, and M_i are sparse, the product $y = Mx$ can be computed as a sequence of matrix-vector multiplications:

$$\begin{aligned} t_1 &= M_1 x, \\ t_2 &= M_2 t_1, \\ &\dots \\ t_{k-1} &= M_{k-1} t_{k-2}, \\ y &= M_k t_{k-1}. \end{aligned}$$

We will express such factorizations using the domain-specific language SPL (Signal Processing Language). SPL is derived from matrix algebra and was described in [99, 137]. It is an extension of the Kronecker product formalism introduced by Van Loan in [122] and Johnson, et al. in [70]. We call elements of SPL *formulas*.

SPL. SPL consists of

- predefined sparse and dense matrices (including transforms);
- arbitrary matrices of predefined structure;
- arbitrary unstructured matrices; and
- matrix operators.

Predefined matrices include the transforms, like $\mathbf{DFT}_n$ and $\mathbf{DCT\text{-}2}_n$, and several other matrices that are used as building blocks. These include the $n \times n$ identity matrix I_n and the reversed identity matrix J_n:

$$I_n = \begin{bmatrix} 1 & & \\ & \ddots & \\ & & 1 \end{bmatrix}, \quad J_n = \begin{bmatrix} & & 1 \\ & \iddots & \\ 1 & & \end{bmatrix}.$$

A 2×2 butterfly matrix and a rotation matrix are defined by

$$F_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \quad R_\alpha = \begin{bmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{bmatrix}.$$

Another predefined matrix L^n_m with $n = mk$ is the $n \times n$ stride permutation matrix defined by the underlying permutation $jk + i \mapsto im + j$ for $0 \le i < k, 0 \le j < m$. In words, L^n_m transposes an $m \times k$ matrix stored in row-major order. $L^n_{n/2}$ is also called perfect shuffle. For example,

$$L^8_4 = \begin{pmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 \end{pmatrix}$$

Arbitrary matrices of predefined structure include permutation and diagonal matrices. For example, $\mathrm{diag}(a_0, a_1, \ldots, a_{n-1})$ is an $n \times n$ diagonal matrix with diagonal entries a_i:

$$\mathrm{diag}(a_0, a_1, \ldots, a_{n-1}) = \begin{bmatrix} a_0 & & & \\ & a_1 & & \\ & & \ddots & \\ & & & a_{n-1} \end{bmatrix}.$$

Permutation matrices are defined by their underlying permutations $i \mapsto \pi(i)$. If π is a permutation on $\{0, \ldots, n-1\}$ then the corresponding permutation matrix has 1 at the positions $(i, \pi(i))$ and is zero everywhere else.

Arbitrary unstructured matrices include matrices that cannot be decomposed into smaller building blocks using matrix operators (defined below).

$$\begin{bmatrix} 1 & \sqrt{2} \\ 0 & 1 \end{bmatrix}$$

is an example of such matrix.

The three major *matrix operators* in SPL are the matrix product $A \cdot B = AB$, the matrix direct sum

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

and the tensor product

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

Two important special cases of the tensor product arise when A or B are the identity matrix. In

$$
\begin{array}{rcll}
\langle \text{spl} \rangle & ::= & \langle \text{generic} \rangle \mid \langle \text{symbol} \rangle \mid \langle \text{transform} \rangle \mid & \\
& & \langle \text{spl} \rangle \cdot \cdots \cdot \langle \text{spl} \rangle \mid & \text{(product)} \\
& & \langle \text{spl} \rangle \oplus \ldots \oplus \langle \text{spl} \rangle \mid & \text{(direct sum)} \\
& & \langle \text{spl} \rangle \otimes \cdots \otimes \langle \text{spl} \rangle \mid & \text{(tensor product)} \\
& & I_n \otimes_k \langle \text{spl} \rangle \mid I_n \otimes^k \langle \text{spl} \rangle \mid & \text{(overlapped tensor product)} \\
& & \overline{\langle \text{spl} \rangle} \mid & \text{(conversion to real)} \\
& & \ldots & \\
\langle \text{generic} \rangle & ::= & \text{diag}(a_0, \ldots, a_{n-1}) \mid \ldots & \\
\langle \text{symbol} \rangle & ::= & I_n \mid J_n \mid L^n_k \mid R_\alpha \mid F_2 \mid \ldots & \\
\langle \text{transform} \rangle & ::= & \mathbf{DFT}_n \mid \mathbf{RDFT}_n \mid \mathbf{DCT\text{-}2}_n \mid \mathbf{Filt}_n(h(z)) \mid \ldots &
\end{array}
$$

Table 2.2: Definition of the most important SPL constructs in Backus-Naur form; n, k are positive integers, α, a_i real numbers.

particular

$$I_n \otimes A = \begin{bmatrix} A & & \\ & \ddots & \\ & & A \end{bmatrix} = \underbrace{A \oplus \ldots \oplus A}_{n},$$

expresses the obvious block diagonal stacking (i.e., a direct sum) of n copies of A. Computing $y = (I_n \otimes A)x$, where A is $m \times m$, means that x is divided into n equal subvectors of size m each, and each subvector is independently multiplied with A.

The second special case is

$$A \otimes I_n = \begin{bmatrix} a_{0,0}I_n & \ldots & a_{0,m-1}I_n \\ \vdots & \ddots & \vdots \\ a_{m-1,0}I_n & \ldots & a_{m-1,m-1} \end{bmatrix}, \quad A = [a_{k\ell}]_{k,\ell}.$$

The above can be viewed as either the "coarse-grain" version of A (i.e., A that operates on vectors of size m instead of scalars) or as a tight interleaving of n copies of A. Computing $y = (A \otimes I_n)x$, where A is $m \times m$, means that x is divided into m equal subvectors of size n, and now A is applied to these n subvectors as if they were scalars. Alternatively, y is a tight interleaving of n independent matrix-vector products of stride-m portions of x with A.

All of the SPL matrix operators also have iterative forms:

$$\bigoplus_{i=0}^{n-1} A_i = A_0 \oplus \ldots \oplus A_{n-1} = \begin{bmatrix} A_0 & & \\ & \ddots & \\ & & A_{n-1} \end{bmatrix},$$

$$\prod_{i=0}^{n-1} A_i = A_0 \cdot \cdots \cdot A_{n-1},$$

$$\bigotimes_{i=0}^{n-1} A_i = A_0 \otimes \ldots \otimes A_{n-1}.$$

Table 2.2 provides a grammar for SPL in Backus-Naur form (BNF) [105]. It defines the SPL

as the disjoint union of choices (separated by |) for each symbol marked by $\langle\cdot\rangle$. Such symbols are called *non-terminal* symbols in BNF terminology, and all other symbols are called *terminals*. As said before, we call the elements of SPL *formulas*.

SPL is a domain-specific language, because it can only be used to describe structured matrices. Further, SPL is declarative since it only describes the structure or dataflow of the computation and not the actual implementation.

Divide-and-conquer algorithms as breakdown rules. Every fast transform algorithm is obtained through a succession of divide-and-conquer decompositions. We call these decompositions *breakdown rules* and express them in SPL following [99]. A breakdown rule typically decomposes a transform into several smaller transforms or converts it into a different transform of usually lesser complexity. The left-hand side of a breakdown rule is a transform, and the right-hand side is the corresponding decomposition, expressed as an SPL *formula*. Examples of breakdown rules are shown in Table 2.3.

Instead of equality, we use $\rightarrow$ to indicate that these rules are *applied* by replacing the left-hand side of the rule by the right-hand side.

Breakdown rules (2.1)–(2.4), for example, correspond to the well known Cooley-Tukey [38], prime-factor [64], Rader [102], and Bluestein [27] DFT algorithms, respectively.

Some breakdown rules require auxiliary transforms, and to implement these auxiliary transforms, additional breakdown rules are necessary. For example, in Table 2.3, **rDFT** and **rDHT** are auxiliary transforms, neededed to implement **RDFT** and **DHT** using (2.6).

A few hundred breakdown rules can be found in the literature. Examples include [20,32,38,43, 47,64,89,97,102,103,121,123,125,126].

For a complete description of a transform algorithm, besides breakdown rules, also *base case rules* are needed. For example:

$$\mathbf{DFT}_2 \rightarrow F_2 = \left[\begin{smallmatrix} 1 & 1 \\ 1 & -1 \end{smallmatrix}\right], \tag{2.19}$$

$$\mathbf{DCT\text{-}2}_2 \rightarrow \mathrm{diag}(1, 1/\sqrt{2})F_2, \tag{2.20}$$

$$\mathbf{DCT\text{-}4}_2 \rightarrow J_2 R_{13\pi/8}, \tag{2.21}$$

$$\mathbf{DCT\text{-}4}_3 \rightarrow (I_1 \oplus J_2)(F_2 \oplus I_1)\left(\sqrt{3/8}I_1 \oplus \sqrt{1/2}\begin{bmatrix} 1/2 & 1 \\ 1 & -1 \end{bmatrix}\right)(F_2 \oplus I_1)(I_1 \oplus J_2). \tag{2.22}$$

We will call a set of breakdown and base cases rules *sufficient* for a transform T, if it is possible to fully expand T for a set of sizes using these rules. For example, rules (2.9), (2.13), and (2.20) are sufficient for **DCT-2** and **DCT-4** of 2-power sizes. Rules (2.9) and (2.20) are not sufficient for **DCT-2**, since the rule for **DCT-4** is missing.

Algorithm space. For a given transform, the recursive composition of breakdown rules and the choices of rules at each level yields a very large space of alternative formulas. Each formula corresponds to a fast algorithm. Most of the formulas have approximately the same operations count, but have different data flow.

From formulas to code. SPL formulas can be directly translated into code by applying code generation templates as explained in [137]. We show a few examples of templates in Table 2.4.

In Section 2.4 we explain the limitations of the template based approach, and explain an alternative code generation method.

Breakdown rules for main transforms

$$\mathbf{DFT}_n \longrightarrow (\mathbf{DFT}_k \otimes I_m) D_{k,m} (I_k \otimes \mathbf{DFT}_m) L^n_k \tag{2.1}$$

$$\mathbf{DFT}_n \longrightarrow V^{-1}_{m,k} (\mathbf{DFT}_k \otimes I_m)(I_k \otimes \mathbf{DFT}_m) V_{m,k} \tag{2.2}$$

$$\mathbf{DFT}_n \longrightarrow W_n^{-1} (I_1 \oplus \mathbf{DFT}_{n-1}) E_n (I_1 \oplus \mathbf{DFT}_{n-1}) W_n \tag{2.3}$$

$$\mathbf{DFT}_n \longrightarrow B^\top_{n,m} D_m \, \mathbf{DFT}_m \, D'_m \, \mathbf{DFT}_m \, D''_m B_{n,m}, \quad m \geq 2n-1 \tag{2.4}$$

$$\mathbf{DFT}_n \longrightarrow P^\top_{k/2,2m} \left(\mathbf{DFT}_{2m} \oplus \left(I_{k/2-1} \otimes_i C_{2m} \, \mathbf{rDFT}_{2m}((i+1)/k) \right) \right) \left(\mathbf{RDFT}'_k \otimes I_m \right) \tag{2.5}$$

$$\begin{vmatrix} \mathbf{RDFT}_n \\ \mathbf{RDFT}'_n \\ \mathbf{DHT}_n \\ \mathbf{DHT}'_n \end{vmatrix} \longrightarrow (P^\top_{k/2,m} \otimes I_2) \left(\begin{vmatrix} \mathbf{RDFT}_{2m} \\ \mathbf{RDFT}'_{2m} \\ \mathbf{DHT}_{2m} \\ \mathbf{DHT}'_{2m} \end{vmatrix} \oplus \left(I_{k/2-1} \otimes_i M_{2m} \begin{vmatrix} \mathbf{rDFT}_{2m}((i+1)/k) \\ \mathbf{rDFT}_{2m}((i+1)/k) \\ \mathbf{rDHT}_{2m}((i+1)/k) \\ \mathbf{rDHT}_{2m}((i+1)/k) \end{vmatrix} \right) \right) \cdot \left(\begin{vmatrix} \mathbf{RDFT}'_k \\ \mathbf{RDFT}'_k \\ \mathbf{DHT}'_k \\ \mathbf{DHT}'_k \end{vmatrix} \otimes I_m \right) \tag{2.6}$$

$$\mathbf{RDFT}_n \longrightarrow D_n \cdot \mathbf{DCT\text{-}2}_n \cdot P_n, \quad n \text{ odd} \tag{2.7}$$

$$\mathbf{DCT\text{-}2}_n \longrightarrow P^\top_{k/2,2m} \left(\mathbf{DCT\text{-}2}_{2m} \, K^{2m}_2 \oplus \left(I_{k/2-1} \otimes N_{2m} \, \mathbf{RDFT\text{-}3}^\top_{2m} \right) \right) G_n (L^{n/2}_{k/2} \otimes I_2) \cdot (I_m \otimes \mathbf{RDFT}'_k) Q_{m/2,k} \tag{2.8}$$

$$\mathbf{DCT\text{-}2}_n \longrightarrow L^n_{n/2} \cdot (\mathbf{DCT\text{-}2}_{n/2} \oplus \mathbf{DCT\text{-}4}_{n/2}) \cdot \begin{bmatrix} I_{n/2} & J_{n/2} \\ I_{n/2} & -J_{n/2} \end{bmatrix} \tag{2.9}$$

$$\mathbf{DCT\text{-}2}_n \longrightarrow S_n \cdot \mathbf{RDFT}_n \cdot K^n_2 \tag{2.10}$$

$$\mathbf{DCT\text{-}3}_n \longrightarrow \mathbf{DCT\text{-}2}^\top_n \tag{2.11}$$

$$\mathbf{DCT\text{-}4}_n \longrightarrow Q^\top_{k/2,2m} \left(I_{k/2} \otimes N_{2m} \, \mathbf{RDFT\text{-}3}^\top_{2m} \right) G'_n (L^{n/2}_{k/2} \otimes I_2)(I_m \otimes \mathbf{RDFT\text{-}3}_k) Q_{m/2,k} \tag{2.12}$$

$$\mathbf{DCT\text{-}4}_n \longrightarrow S'_n \cdot \mathbf{DCT\text{-}2}_n \cdot D'''_n \tag{2.13}$$

$$\mathbf{MDCT}_n \longrightarrow \mathbf{DCT\text{-}4}_n \cdot \begin{bmatrix} & & -J_n & I_n \\ I_n & -J_n & & \end{bmatrix} \tag{2.14}$$

$$\mathbf{IMDCT}_n \longrightarrow \begin{bmatrix} & & I_n & \\ & & & -J_n \\ -J_n & & & \\ & I_n & & \end{bmatrix} \cdot \mathbf{DCT\text{-}4}_n \tag{2.15}$$

$$\mathbf{WHT}_n \longrightarrow (\mathbf{WHT}_k \otimes I_m)(I_k \otimes \mathbf{WHT}_m) \tag{2.16}$$

Breakdown rules for auxiliary transforms

$$\begin{vmatrix} \mathbf{rDFT}_{2n}(u) \\ \mathbf{rDHT}_{2n}(u) \end{vmatrix} \longrightarrow L^{2n}_m \left(I_k \otimes_i \begin{vmatrix} \mathbf{rDFT}_{2m}((i+u)/k) \\ \mathbf{rDHT}_{2m}((i+u)/k) \end{vmatrix} \right) \left(\begin{vmatrix} \mathbf{rDFT}_{2k}(u) \\ \mathbf{rDHT}_{2k}(u) \end{vmatrix} \otimes I_m \right) \tag{2.17}$$

$$\mathbf{RDFT\text{-}3}_n \longrightarrow (Q^\top_{k/2,m} \otimes I_2) \, (I_k \otimes_i \mathbf{rDFT}_{2m}(i+1/2)/k)) \, (\mathbf{RDFT\text{-}3}_k \otimes I_m) \tag{2.18}$$

Table 2.3: Examples of breakdown rules written in SPL. Above, Q, P, K, V, W are various permutation matrices, D are diagonal matrices, and B, C, E, G, M, N, S are other sparse matrices, whose precise form is irrelevant.

Formula M	**Code for** $y = Mx$
F_2	`y[0] = x[0] + x[1];` `y[1] = x[0] - x[1];`

L^n_k	`for(i=0; i<k; i++) {`
	`  for(j=0; j<n/k; j++) {`
	`  y[j+i*n/k] = x[j*k+i];`
	`  }`
	`}`

$(A \oplus B)$	`<code for: y[0:1:k-1] = A * x[0:1:k-1]>` `<code for: y[k:1:k+m-1] = B * x[k:1:k+m-1]>`

$(I_n \otimes A)$	`for(j=0; j<n; j++) {`
	`  <code for: y[jk:1:jk+m-1] = A * x[jk:1:jk+m-1]>`
	`}`

$(A \otimes I_n)$	`for(j=0; j<n; j++) {`
	`  <code for: y[j:n:j+n(k-1)] = A * x[j:n:j+n(k-1)]>`
	`}`

Table 2.4: Translating SPL constructs to code: examples.

2.3 Spiral

Spiral [99] is an automatic program generation and optimization system for linear signal transforms, which builds on the framework explained in Sections 2.1–2.2. Spiral uses SPL to represents fast transform algorithms, and automates the process of obtaining fast fixed size transform implementation, starting just from the breakdown rules and base case rules.

The high level structure of the optimization and code generation process in Spiral is shown in Fig. 2.1. The input to Spiral is a formally specified transform of a fixed size (known at generation time), e.g., "$\mathbf{DFT}_{1024}$"; the output is an optimized C program that computes the transform.

The generated program is automatically tuned to the platform on which Spiral is installed using a feedback directed search in the space of alternative algorithms. We now explain this process in more detail.

Program generation in Spiral. After the user specifies a transform to be implemented, the *Search/Learning* module applies different breakdown rules to the transform to generate alternative SPL formulas. This process is guided by a heuristic feedback-driven search, with different possibilities for the search strategies.

The feedback is obtained either by predicting the performance in a certain way, e.g. [109, 110], or by translating the formula into target language (e.g., C) code and measuring the desired performance metric, which usually is the runtime of the code.

Before generating code, each formula undergoes optimization at several abstraction levels.

At the algorithm level, in the *Formula Generation* stage, Spiral applies breakdown rules such as (2.1) to the given transform to generate one out of many possible formulas represented in SPL. In the *Formula Optimization* stage Spiral applies high level algorithmic optimizations using rewriting systems [45]. This block was fully developed in this thesis to handle loop optimizations,

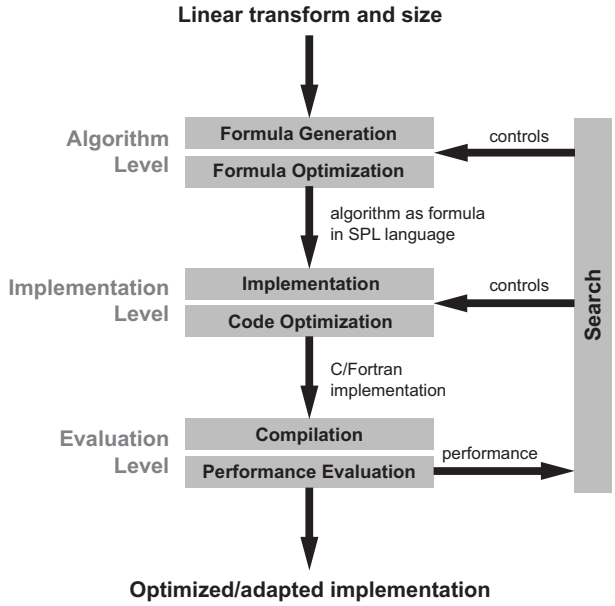


Figure 2.1: Adaptive program generator Spiral.

vectorization, and parallelization for a large class of transforms (Table 1.1(c)). In the original Spiral system [99] (Table 1.1(a)) this block consisted only of a preliminary form of formula vectorization.

At the implementation level, in the *Implementation* stage, Spiral translates the optimized formula into an intermediate code representation. In the *Code Optimization* stage, further optimizations are performed on the intermediate code representation including strength reduction, constant folding, copy propagation, and array scalar replacement [137]. The result is then unparsed to a C program.

At the evaluation level, the final program is compiled using a standard C compiler in the *Compilation* stage and the actual runtime or other performance metric is measured in the *Performance Evaluation* stage.

To date, several search strategies have been implemented in Spiral. The most effective strategies are dynamic programming and evolutionary (genetic) search [108]. The different search strategies are described in detail in [99].

Limitations of Spiral. While describing a large algorithm space for a wide variety of transforms, the original Spiral was limited by the types of code it could produce, as we explained in Table 1.1(a). The major limitations of the original Spiral are listed below.

- **No general loop merging.** Efficient loop code could not be produced for transforms other than the DFT, this problem is discussed in detail in the following section.

- **Fixed size transforms only.** Transform sizes must be known at generation time. The problem of producing general size code is the main focus of this thesis. We explain why this is difficult in Section 3.2.

- **No support for general vectorization and parallelization.** Only vectorized DFTs could be produced by the original Spiral. Multithreaded code for transforms could not be produced. Further, some of our early experiments with parallel code were limited by the requirement

that the number of threads is known at generation time, similarly to how the transform sizes must be fixed and known.

In this thesis, we address all of these limitations. We solve the general loop merging problem in Section 2.4 Chapters 3 and 5 explain how the fixed size limitation can be overcome. Chapter 4 presents the general vectorization and parallelization framework.

2.4 $\sum$-SPL and Loop Merging

Most of the fast transforms algorithms (see Table 2.3) are "divide-and-conquer," which means that a transform is decomposed into smaller transforms. In principle, this produces a structure that is well-suited for achieving good performance on memory hierarchies. However, the conquer step in these algorithms is iterative, i.e., requires extra passes through the data. In particular, some of these steps are complicated permutations, which can deteriorate performance considerably. As a consequence, one of the keys to obtaining high performance is to *merge* these iterative steps to improve data locality and data reuse. For example, permutations can be converted into a reindexing in the subsequent computational loop. The Cooley-Tukey FFT breakdown rule in (2.1), for instance, contains four factors, and thus a straightforward implementation would perform four passes over the data. However, in practice (2.1) is implemented in only two passes, as explained later.

Small example. To illustrate the problem, we show a small example of translating a simple SPL formula to code. Consider, the formula

$$M = (I_4 \otimes F_2)L_4^8. \tag{2.23}$$

To obtain code for $y = Mx$ we use the translation rules from Table 2.4, and get:

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

However, this is known to be suboptimal, since the the stride permutation L_4^8 can be incorporated into the computation loop, instead of being performed explicitly. This eliminates the redundant pass over the array, and leads to the simplified code with only a single loop below:

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

In the original Spiral a special template was used to handle this specific formula, and implement the optimization above.

The goal of the loop merging optimizations is to automate the above optimizations without the use of specialized templates.

2.4.1 Overview

The general problem of loop merging [42, 72] is NP-complete. Further, loop merging requires array dependence information, for which the most general methods, like [95], achieve exact results only if the array indices are affine expressions, and even for this class the analysis has exponential worst-case runtime. Since many transform algorithms use non-affine index mappings, standard array dependence tests do not work in general. We developed a simple domain-specific form of loop merging, which does not face these problems.

In the domain of linear transforms, the problem of loop merging has been completely solved only for the DFT and only for one DFT method: the Cooley-Tukey FFT (see [122]). Examples include the first generation of Spiral [99], and the DFT libraries FFTW [48, 61] and UHFFT [84]. For the prime-factor FFT, which involves a different permutation, [122] describes a partial solution without the explicit permutation, which can only be used for a restrictive set of DFT sizes. This solution is implemented manually in UHFFT. Consequently, these libraries achieve high performance for DFT sizes that factor into small prime numbers, but are suboptimal for other sizes. More importantly, loop merging is not solved for other transforms. For example, the DCTs/DSTs in FFTW, as we already saw from Fig. 1.4 suffer a large performance penalty. For these transforms, in FFTW some limited conquer step merging has been performed manually, and as a result FFTW outperforms Intel IPP, however, overall the performance is still rather suboptimal.

Since our goal is automatic library generation, loop merging has to be performed automatically and across a wide range of transforms and breakdown rules, including the DCT/DST algorithms, the different DFT algorithms, and other breakdown rules in Table 2.3. The solution to this problem is an extension to SPL, called $\sum$-SPL (Sigma-SPL), which makes loops and index mapping functions explicit. Using $\sum$-SPL, loop merging can be performed efficiently through rewriting [52]. In the following chapters $\sum$-SPL will also serve as the main tool for library generation.

2.4.2 Motivation for the General Method

There are two major problems which cannot be solved by using templates, and that the general loop merging framework is expected to solve. First, there exists a large variety of breakdown rules with different permutations (see Table 2.3) and hence different instantiations of the above problem. Second, the breakdown rules are applied recursively, or in other words, inserted into each other to produce several levels of permutations and other simple operations like scaling (i.e. diagonal matrices in SPL) that ideally all would be fused. Both of these lead to a combinatorial explosion of the number of required SPL templates.

We will use the DFT and its most important recursive algorithms as the motivating example throughout this section.

The three most important FFTs shown in (2.1)–(2.4) and restated below are respectively called

Cooley-Tukey, prime-factor (or Good-Thomas), and Rader:

$$\mathbf{DFT}_n = (\mathbf{DFT}_k \otimes I_m) D_{k,m} (I_k \otimes \mathbf{DFT}_m) L^n_k, \tag{2.24}$$

$$\mathbf{DFT}_n = V^{-1}_{n,k} (\mathbf{DFT}_k \otimes I_m) (I_k \otimes \mathbf{DFT}_m) V_{n,k}, \tag{2.25}$$

$$\mathbf{DFT}_n = W^{-1}_n (I_1 \oplus \mathbf{DFT}_{p-1}) E_n (I_1 \oplus \mathbf{DFT}_{p-1}) W_n. \tag{2.26}$$

Each of these FFTs reduces the problem of computing a DFT of size n to computing several DFTs of different and smaller sizes. The applicability of each FFT depends on the size n:

- The Cooley-Tukey FFT requires that $n = km$ factors and reduces the $\mathbf{DFT}_n$ to m $\mathbf{DFT}_k$'s and k $\mathbf{DFT}_m$'s.
- The prime-factor FFT requires that $n = km$ factors and that the factors are coprime: $\gcd(k, m) = 1$. As in the Cooley-Tukey FFT, the $\mathbf{DFT}_n$ is then computed using m $\mathbf{DFT}_k$'s and k $\mathbf{DFT}_m$'s.
- The Rader FFT requires that $n = p$ is prime and reduces the $\mathbf{DFT}_p$ to 2 $\mathbf{DFT}_{p-1}$'s.

In (2.24)–(2.26), L, V, W are permutation matrices, D is a diagonal matrix, and E is "almost" diagonal with 2 additional off-diagonal entries.

The permutations matrices L, V, W in (2.24)–(2.26) correspond to the following permutations:

$$\ell^{mk}_k : i \mapsto \left\lfloor \tfrac{i}{m} \right\rfloor + k \, (i \bmod m), \tag{2.27}$$

$$\mathrm{v}^{m,k} : i \mapsto (m \left\lfloor \tfrac{i}{m} \right\rfloor + k \, (i \bmod m)) \bmod mk, \quad \gcd(m, k) = 1, \tag{2.28}$$

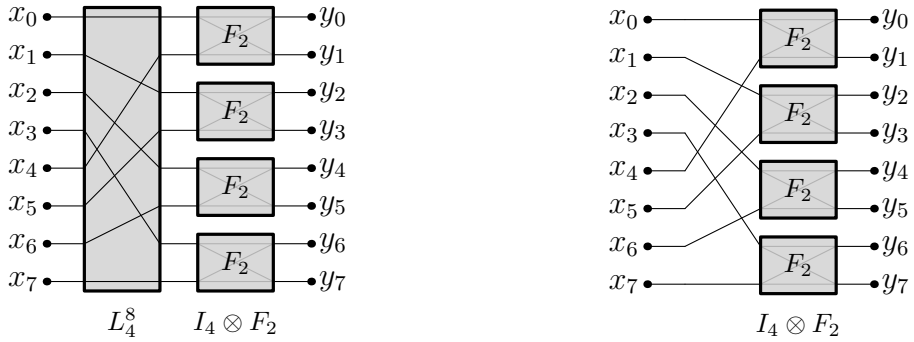
$$w^n_g : i \mapsto \begin{cases} 0, & \text{if } i = 0, \\ g^{i-1} \bmod n, & \text{else.} \end{cases} \tag{2.29}$$

where g is a suitably chosen, fixed integer[1]. Note, that (2.27) is equivalent to our previous definition of $jk + i \mapsto im + j$, but here we uniformly express all permutations as $i \to p(i)$.

For high performance it is crucial to handle these permutations, and also their combinations (since the DFT is computed recursively) efficiently. The "affine" permutation (2.27) has been studied extensively in the compiler literature, whereas the more expensive mappings (2.28) and (2.29) have not received much attention, since they only occur in FFTs. Below we will identify the transformations necessary to optimize the recursive combinations of these permutations. These transformations are not performed on the actual code, where they would be prohibitively expensive, but at a higher level of abstraction provided by $\sum$-SPL introduced in this section.

Implementation in FFTW. FFTW implements only (2.24) and (2.26) as the top-level recursive algorithms. In (2.24) the permutation L is never explicitly performed, instead the stride value is passed as an argument in the recursion. The composition of strides yields a new stride, which is possible, because of special properties of L. Similarly, the scaling by $D_{\cdot,\cdot}$ (the twiddle factors) is not performed as an extra step, but, also is passed down the recursion and finally performed by special "twiddle codelets." As a consequence, each Cooley-Tukey recursion requires only two passes through the data instead of four. Besides (2.24), FFTW supports also (2.26), for which the permutations are performed explicitly.

[1] Specifically, g is a generator of the multiplicative group $(\mathbb{Z}/n\mathbb{Z})^\times$.



(a) Original: $y = (I_4 \otimes F_2)L_4^8 x$ (two passes) (b) After loop merging: $y = \sum_{j=0}^{3} \Big(S(h_{2j,\,1})F_2 G(h_{j,\,2})\Big)x$ (one pass)

Figure 2.2: Simple loop merging example.

In other words, the FFTW implementation faces the same problem as the original Spiral. Namely, the loop merging in the Cooley-Tukey FFT (2.24) is implemented manually as a special case, and efficiently handling alternative recursions (with other permutations) requires additional manual effort.

Implementation in Spiral. As we briefly explained above, for (2.24), the original Spiral (Table 1.1(a)) performed optimizations equivalent to FFTW. Namely, when translating an SPL formula based on (2.24) into code, the SPL compiler fused the twiddle factors and the stride permutation with the adjacent loops using a special purpose templates for SPL expressions of the form $(I_k \otimes A_m)L_k^{km}$ and $(A_k \otimes I_m)T_m^{km}$. In other words, this optimization was also hard-coded specifically for Cooley-Tukey based algorithms.

Summary. In summary, both FFTW and the original Spiral handle (2.24) as a special case, an approach that is neither easily extensible, nor one that gives any insight into how to optimize the other FFT recursions or other transforms. To solve this problem we propose a lower level extension to SPL, called $\sum$-SPL.

Next, after explaining $\sum$-SPL and the loop merging optimizations, we show how to apply them to other breakdown rules, namely the FFTs (2.25) and (2.26). We have also used this framework to generate fast loop code for other transforms, which established the first column in Table 1.1(b).

2.4.3 Loop Merging Example

Before we formally introduce $\sum$-SPL we continue with the example (2.23) from the beginning of Section 2.4 and show the main ingredients of $\sum$-SPL and how the actual loop merging is performed.

The SPL describes transform algorithms as sparse structured matrix factorization built from small matrices, permutation matrices, diagonal matrices, tensor products, and other constructs (Section 2.2). SPL captures the data flow of an algorithm. However, all data reorganization steps are described explicitly as stages that perform passes through the data.

The formula

$$(I_4 \otimes F_2)L_4^8 \tag{2.30}$$

from Section 2.4.1 describes two passes through the data. We show this graphically in Fig. 2.2(a). First, the data vector is shuffled according to L_4^8, and then the loop corresponding to $I_4 \otimes F_2$

applies the computational kernel F_2 to consecutive subvectors. Our goal is to merge these two loops into a single loop that implements the data reorganization as readdressing of the input of the computational kernels F_2. This is done using $\sum$-SPL and a small number of rewrite rules, leading to the result in Fig. 2.2(b).

First, we translate (2.30) into $\sum$-SPL (no loop merging is performed) to get

$$\left(\sum_{j=0}^{3} S(h_{2j,\,1})F_2G(h_{2j,\,1})\right)\mathrm{perm}(\ell_4^8). \tag{2.31}$$

In (2.31) the permutation matrix L_4^8 is expressed by its underlying permutation ℓ_4^8. The sum expresses the four iterations of the loop for $I_4 \otimes F_2$. The *gather* and *scatter* matrices $G()$ and $S()$ express respectively the loading and storing of the data in the loop. The actual loop merging is now performed by rewriting (2.31) twice as

$$(2.31) \;\rightarrow \sum_{j=0}^{3} \Big(S(h_{2j,\,1})F_2G(\ell_4^8 \circ h_{2j,\,1})\Big) \rightarrow \sum_{j=0}^{3} \Big(S(h_{2j,\,1})F_2G(h_{j,\,2})\Big). \tag{2.32}$$

In (2.32), first the permutation was fused into the gather operation leading to a new gather index mapping. Then the composed index mapping function $\ell_4^8 \circ h_{2j,\,1}$ was simplified to $h_{j,\,2}$.

2.4.4 $\sum$-SPL: Definition

In the example in Fig. 2.2 the main computational loop was described by $I_4 \otimes F_2$ and the data shuffling L_4^8 was combined with the adjacent computation. This motivates our definitions of two categories of SPL constructs: *decoration* and *skeleton*.

The main computation loops in an SPL formula are defined by its *skeleton*. Examples of skeleton objects include direct sums and tensor products with identity matrices,

$$A \oplus B, \quad A \otimes I_n, \quad \text{and} \quad I_n \otimes A.$$

When an SPL formula is mapped into $\sum$-SPL, skeleton objects are translated into iterative sums and gather/scatter operations as in (2.31) to make the loop structure and the index mappings explicit. In (2.30) the skeleton is $I_4 \otimes F_2$.

Permutation matrices and diagonal matrices are called *decorations*. Our optimizations merge decorations into the skeleton such that the extra stages disappear. In (2.30) the decoration is $L_4^8 = \mathrm{perm}(\ell_4^8)$ and the gather index mapping resulting from the loop merging in (2.32) is $\ell_4^8 \circ h_{2j,\,1}$, which becomes $h_{j,\,2}$ after simplification.

With the above example as a motivation, we now provide the formal definition of $\sum$-SPL, starting with an overview of its main components.

Main components. $\sum$-SPL consists of four components:

1. index mapping functions,

2. scalar functions,

3. parametrized matrices,

4. iterative sum $\sum$.

These are defined next.

Index mapping functions. An index mapping function is a function mapping an interval into an interval. For example, we saw in (2.31) that gather, scatter, and permutation matrices are parameterized by such functions. We express index mapping functions in terms of primitive functions and function operators to capture their structure. This enables many simplifications, which are exceedingly difficult on the corresponding C code.

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

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

We use the short-hand notation $f^{n \to N}$ to refer to an index mapping function of the form $f : \mathbb{I}_n \to \mathbb{I}_N$. If index mapping functions depend on a parameter, say j, we write f_j. A bijective index mapping function

$$p : \mathbb{I}_n \to \mathbb{I}_n;\ i \mapsto p(i),$$

defines a permutation on n elements and is abbreviated by p^n.

We introduce two primitive functions, the *identity* function and the *stride* function respectively defined as

$$\imath_n :\ \mathbb{I}_n \to \mathbb{I}_n;\ i \mapsto i, \tag{2.33}$$

$$h_{b,\,s} :\ \mathbb{I}_n \to \mathbb{I}_N;\ i \mapsto b + is, \quad \text{for } s|N. \tag{2.34}$$

Structured index mapping functions are built from the above primitives using function composition, and possibly other function operators. For the two index mapping functions $f^{m \to M}$ and $g^{n \to N}$ with $n = M$, we define the *function composition* in the usual way:

$$g \circ f : \mathbb{I}_m \to \mathbb{I}_N;\ i \mapsto g(f(i)).$$

Scalar functions. 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.

Parametrized matrices. $\sum$-SPL contains four types of parameterized matrices, described by their defining functions:

$$G(r^{n \to N}),\ S(w^{n \to N}),\ \text{perm}(p^n),\ \text{and}\ \text{diag}\left(f^{n \to \mathbb{C}}\right).$$

Their interpretation as C code is summarized in Table 2.5(a).

Let $e^n_k \in \mathbb{C}^{n \times 1}$ be the column basis vector with the 1 in k-th position and 0 elsewhere. The gather matrix for the index mapping $f^{n \to N}$ is

$$G(f^{n \to N}) := \left[e^N_{f(0)} \mid e^N_{f(1)} \mid \cdots \mid e^N_{f(n-1)}\right]^\top.$$

Formula M	**Code for** $y = Mx$
$G(f^{n\to N})$	`for(j=0; j<n; j++)` `  y[j] = x[f(j)];`
$S(f^{n\to N})$	`for(j=0; j<n; j++)` `  y[f(j)] = x[j];`
$\mathrm{perm}(p^n)$	`for(j=0; j<n; j++)` `  y[j] = x[p(j)];`
$\mathrm{diag}\left(f^{n\to\mathbb{C}}\right)$	`for(j=0; j<n; j++)` `  y[j] = f(j)*x[j];`

$\left(\sum_{j=0}^{k-1} A_j\right)$	`for(j=0; j<k; j++) {`
	`  <code for: y = A_j * x>`
	`}`

Table 2.5: Translating $\sum$-SPL constructs to code.

Permutation and scatter matrices are defined as follows; $(\cdot)^\top$ is the matrix transposition.

$$\mathrm{perm}(f^n) := G(f) = \left[e^n_{f(0)} \mid e^n_{f(1)} \mid \cdots \mid e^n_{f(n-1)}\right]^\top,$$
$$S(f^{n\to N}) := G(f)^\top = \left[e^N_{f(0)} \mid e^N_{f(1)} \mid \cdots \mid e^N_{f(n-1)}\right].$$

This implies that

$$\begin{aligned} y = G(f)\cdot x \quad &\Leftrightarrow \quad y_i = x_{f(i)}, \\ y = \mathrm{perm}(f)\cdot x \quad &\Leftrightarrow \quad y_i = x_{f(i)}, \\ y = S(f)\cdot x \quad &\Leftrightarrow \quad y_{f(i)} = x_i, \end{aligned}$$

which explains the corresponding code in Table 2.5(a).

Gather matrices are wide and short, and scatter matrices are narrow and tall. For example,

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

Scalar functions like $f^{n\to\mathbb{C}}$ define $n \times n$ diagonal matrices

$$\mathrm{diag}\left(f^{n\to\mathbb{C}}\right) = \mathrm{diag}\left(f(0), \ldots, f(n-1)\right).$$

Iterative sum. The iterative sum

$$\sum_{j=0}^{n-1} A_j.$$

is used to represent loops.

In $\sum$-SPL the summands A_j of an iterative sum are constrained such that the actual additions (except those incurred by the A_j) are never performed, i.e., no two matrices A_{j_1} and A_{j_2} have a non-zero entry at a common position (i, k)

The interpretation of the iterative sum as a loop makes use of the distributivity of the sum,

$$y = \left(\sum_{j=0}^{n-1} A_j \right) x = \sum_{j=0}^{n-1} (A_j x).$$

Due to the constraint that the iterative sum actually does not incur any additional operation, it encodes a loop in which each iteration produces a unique part of the final output vector.

Now, as an example, we show how $\otimes$ can be converted into a sum. We assume that A is $n \times n$ and omit the domain and range in the occurring stride functions for simplicity.

$$\begin{aligned} I_k \otimes A &= \begin{bmatrix} A & & \\ & \ddots & \\ & & A \end{bmatrix} = \begin{bmatrix} A & & \\ & & \\ & & \end{bmatrix} + \cdots + \begin{bmatrix} & & \\ & & \\ & & A \end{bmatrix} \\ &= S_0 A G_0 + \cdots + S_{k-1} A G_{k-1} \\ &= S_{h_{0,1}} A G_{h_{0,1}} + \cdots + S_{(k-1)n,1} A G_{(k-1)n,1} \\ &= \sum_{j=0}^{k-1} S_{h_{jn,1}} A G_{h_{jn,1}} \end{aligned}$$

For example, in this equation,

$$G_0 = G_{h_{0,1}} = \begin{bmatrix} 1 & & & \\ & \ddots & & \\ & & 1 & \end{bmatrix}, \quad S_0 = S(h_{0,1}) = G(h_{0,1})^\top.$$

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

The code for $y = \left(\sum_{j=0}^{n-1} A_j \right) x$ is shown in Table 2.5(b).

$\sum$-SPL example. As explained in the beginning of this section, we use iterative sums to make the loop structure of skeleton objects in SPL explicit. The summands are sparse matrices that depend on the summation index. For example $I_2 \otimes F_2$ becomes in $\sum$-SPL the sum

$$\begin{bmatrix} F_2 & 0_2 \\ 0_2 & F_2 \end{bmatrix} = \begin{bmatrix} F_2 & 0_2 \\ 0_2 & 0_2 \end{bmatrix} + \begin{bmatrix} 0_2 & 0_2 \\ 0_2 & F_2 \end{bmatrix}. \tag{2.35}$$

With the definition of the scatter and gather matrices

$$\begin{aligned} S_0 = \begin{bmatrix} I_2 | \, 0_2 \end{bmatrix}^\top \quad \text{and} \quad S_1 = \begin{bmatrix} 0_2 \, | I_2 \end{bmatrix}^\top, \\ G_0 = \begin{bmatrix} I_2 | \, 0_2 \end{bmatrix} \quad \text{and} \quad G_1 = \begin{bmatrix} 0_2 \, | I_2 \end{bmatrix}, \end{aligned}$$

(2.35) can be written as the iterative sum

$$\sum_{j=0}^{1} S_j F_2 G_j. \tag{2.36}$$

However, in (2.36) the subscripts of S and G are integers and not functions. Using the stride index

mapping function in (2.34) we express the gather and scatter matrices as

$$S_j = S(h_{2j,\,1}) \quad \text{and} \quad G_j = G(h_{2j,\,1}).$$

The matrix-vector product $y = (I_2 \otimes F_2)x$ then becomes in $\sum$-SPL

$$y = \sum_{j=0}^{1} S(h_{2j,\,1}) F_2 G(h_{2j,\,1}) x. \tag{2.37}$$

Using the rules in Table 2.5 we obtain the unoptimized program in Implementation 1.

Implementation 1 (Unoptimized program for (2.37)*)*

```
// Input: _Complex double x[4], output: y[4]
_Complex double t0[2], t1[2];
for (int j=0;j<2;j++) {
   for (int i=0; i<2; i++) t0[i] = x[i+2*j]; // G(h_(2j,1))
   t1[0] = t0[0] + t0[1];                    // F_2
   t1[1] = t0[0] - t0[1];                    //
   for (int i=0; i<2; i++) y[i+2*j] = t1[i]; // S(h_(2j,1))
}
```

After standard optimizations (performed by the standard SPL compiler), such as loop unrolling, array scalarization, and copy propagation, we obtain the optimized program in Implementation 2.

Implementation 2 (Optimized program for (2.37)*)*

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

2.4.5 The $\sum$-SPL Rewriting System

In this section we describe the new loop optimization procedure and its implementation. The optimizations are implemented as rewrite rules, which operate in several passes on SPL and $\sum$-SPL expression trees. An overview of the different steps is shown in Figure 2.3. The steps are generic for all transforms and algorithms except the index simplification (marked with (*)), which requires the inclusion of rules specific to the class of algorithms considered. Namely, these are the rules that simplify the composition of index mapping functions, for example (2.33) and (2.34). We start with an overview; then we explain the steps in detail.

1. *Expansion of skeleton.* In the first step we translate an SPL formula (as generated within Spiral) into a corresponding $\sum$-SPL formula. The skeleton (see Section 2.4.4) is expanded into iterative sums and the decorations are expressed in terms of their defining functions.

2. *Loop merging.* A generic set of rules merges the decorations into neighboring iterative sums, thus merging loops. In this process index mapping functions get symbolically composed, and thus become complicated or costly to compute.

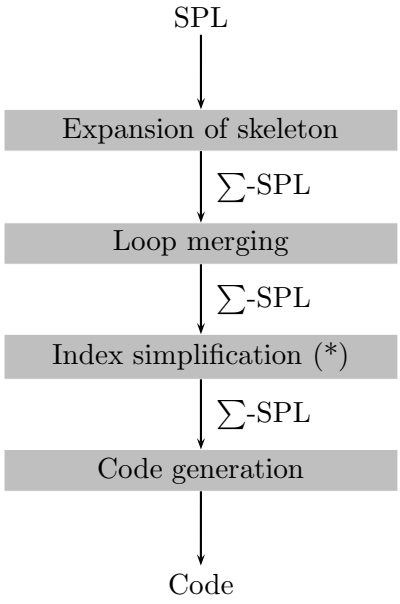


Figure 2.3: Loop merging and code generation for SPL formulas. (*) marks the transform/algorithm specific pass.

3. *Index simplification.* In this stage the index mapping functions are simplified, which is possible due to their symbolic representation and a set of symbolic rules. The rules of this stage are transform dependent and encode the domain specific knowledge how to handle specific permutations. Many identities are based on simple number theoretic properties of the permutations. Identifying these rules for a given transform is usually a research problem.

4. *Code generation.* After the structural optimization the $\sum$-SPL compiler is used to translate the final expression into initial code (see Table 2.5), which is then further optimized as in the original SPL compiler [137].

In the following detailed explanation, we use the following simple formula as a running example:

$$(I_m \otimes \mathbf{DFT}_k)\, L^{mk}_m \quad , \quad L^{mk}_m = \mathrm{perm}(\ell^{mk}_m), \tag{2.38}$$

where ℓ^{mk}_m was defined in (2.27).

The direct translation of (2.38) into code using Table 2.4 for $m = 5$ and $k = 2$ would lead to the code fragment in Implementation 3.

Implementation 3 (Compute $y = (I_5 \otimes F_2)L^{10}_5$ without loop merging)

```
// Input: _Complex double x[10], output: y[10]
_Complex double t[10];
// explicit stride permutation L^10_5
for (int i=0; i<10; i++)
   t[i] = x[(i/5) + 2*(i % 5)];
// kernel loop I_5 x DFT_2
for (int i=0; i<5; i++) {
   y[2*i]   = t[2*i] + t[2*i+1];
   y[2*i+1] = t[2*i] - t[2*i+1];
```

$$A \oplus B \quad \rightarrow \quad S(h_{0,\,1})AG(h_{0,\,1}) + S(h_{m,\,1})BG(h_{n,\,1}) \tag{2.39}$$

$$A \otimes I_k \quad \rightarrow \quad \sum_{j=0}^{k-1} S(h_{j,\,k})AG(h_{j,\,k}) \tag{2.40}$$

$$I_k \otimes A \quad \rightarrow \quad \sum_{j=0}^{k-1} S(h_{mj,\,1})AG(h_{mj,\,1}). \tag{2.41}$$

Table 2.6: Rules to expand the skeleton.

```
}
```

This code contains two loops, originating from the permutation matrix L_m^{mk} and from the computational kernel $I_m \otimes \mathbf{DFT}_k$. The goal in this example is to merge the two loops into one.

Step 1: Expansion of skeleton. This stage translates SPL formulas, generated by Spiral, into $\sum$-SPL formulas. It rewrites all skeleton objects in the SPL formula into iterative sums and makes the defining functions of decorations explicit. Table 2.6 summarizes the rewrite rules used in this step, assuming $A \in \mathbb{C}^{m\times n},\ B \in \mathbb{C}^{m'\times n'}$.

Both tensor products and direct sums of matrices are translated into sums with stride index mapping functions parameterizing the gather and scatter matrices.

In our example (2.38), the rewriting system applies the rule (2.41) to obtain the $\sum$-SPL expression

$$\left(\sum_{j=0}^{m-1} S(h_{nj,\,1})\, \mathbf{DFT}_n\, G(h_{nj,\,1}) \right) \mathrm{perm}(\ell_m^{mn}). \tag{2.42}$$

Step 2: Loop merging. The goal of loop merging is to propagate all index mapping functions for a nested sum into the parameter of one single gather and scatter matrix in the innermost sum, and to propagate all diagonal matrices into the innermost computational kernel. Table 2.7 summarizes the necessary rules. Loop merging consists of two steps: moving matrices into iterative sums, and actually merging or commuting matrices.

First, matrices are moved inside iterative sums by applying rules (2.44) and (2.45) that implement the distributive law. Note, that iterative sums are not moved into other iterative sums.

Second, the system merges gather, scatter, and permutation matrices using rules (2.46)–(2.49) and pulls diagonal matrices into the computational kernel using (2.50) and (2.51). This step simply composes their defining functions, possibly creating complicated terms that must be simplified in the next step.

After loop merging, (2.42) is transformed into

$$\sum_{j=0}^{m-1} \Big(S(h_{kj,\,1})\, \mathbf{DFT}_k\, G(\ell_m^{mk} \circ h_{kj,\,1}) \Big) \tag{2.43}$$

through application of rules (2.44) and (2.46).

Step 3: Index mapping simplification. As said before, this is the only step that depends on the considered transform and its algorithms. We consider the DFT and the FFTs (2.24)–(2.26).

$$\left(\sum_{j=0}^{m-1} A_j\right) M \rightarrow \left(\sum_{j=0}^{m-1} A_j M\right) \tag{2.44}$$

$$M \left(\sum_{j=0}^{m-1} A_j\right) \rightarrow \left(\sum_{j=0}^{m-1} M A_j\right) \tag{2.45}$$

$$G(s^{n\rightarrow N_1})G(r^{N_1\rightarrow N}) \rightarrow G(r \circ s) \tag{2.46}$$

$$S(v^{N_1\rightarrow N})S(w^{n\rightarrow N_1}) \rightarrow S(v \circ w) \tag{2.47}$$

$$G(r^{n\rightarrow N}) \operatorname{perm}(p^N) \rightarrow G(p \circ r) \tag{2.48}$$

$$\operatorname{perm}(p^N)S(w^{n\rightarrow N}) \rightarrow S(p^{-1} \circ w) \tag{2.49}$$

$$G(r^{n\rightarrow N}) \operatorname{diag}\left(f^{N\rightarrow\mathbb{C}}\right) \rightarrow \operatorname{diag}\left(f \circ r\right) G(r) \tag{2.50}$$

$$\operatorname{diag}\left(f^{N\rightarrow\mathbb{C}}\right) S(w^{n\rightarrow N}) \rightarrow S(w) \operatorname{diag}\left(f \circ w\right) \tag{2.51}$$

Table 2.7: Loop merging rules.

These recursions feature permutations that involve integer power computations, modulo operations, and conditionals. In addition to the stride permutation (2.27) in the Cooley-Tukey FFT (2.24), the prime-factor FFT (2.25) uses $V_{m,k} = \operatorname{perm}(\mathrm{v}^{m,k})$ with $\gcd(m,k) = 1$, defined in (2.28), and for prime n the Rader FFT (2.26) requires $W_{r,s} = \operatorname{perm}(w_g^n)$, with g a generator of the multiplicative group $(\mathbb{Z}/n\mathbb{Z})^\times$, defined in (2.29).

If different breakdown rules are applied at different levels of DFT recursion, the rewrite rules (2.46)–(2.49) can yield a composition of many index mapping functions. Since individual index mappings are already rather complicated, the composed mappings become very complex. Thus, our rewriting system must be able to perform powerful index mapping simplifications.

In order to formulate all simplification rules for ℓ, v, w in (2.27)–(2.29) we first have to introduce two helper functions:

$$z_{b,\,s} : i \mapsto b + is \bmod N, \quad N = sn, \tag{2.52}$$

$$\overline{w}_{\varphi,g}^{n\rightarrow N} : i \mapsto \varphi g^i \bmod N, \quad n|N-1, N \text{ prime} \tag{2.53}$$

The power of using symbolic functions and operators becomes apparent next. Namely, we can identify a rather small set of context insensitive simplification rules to simplify all index functions arising from combining the FFT rules (2.24)–(2.26). The required simplifications cannot be done solely using basic integer identities as conflicting identities and special number-theoretical constraints (e.g., a variable is required to be the generator of a cyclic group of order n) would be required. Our function symbols capture these conditions by construction while our rules encode the constraints.

Table 2.8 summarizes the majority of index simplification rules used for the FFTs. These rules were originally identified in [52], although there we used a different notation for some index mapping functions, which required some additional rules. Rules (2.54)–(2.56) are used to simplify Cooley-Tukey FFT decompositions. Rules (2.57)–(2.60) are used for decompositions based on prime factor FFT. Rules (2.61)–(2.64) are used for the Rader FFT.

For Cooley-Tukey FFT:

$$\ell^{mk}_m \circ h^{k\to mk}_{kj,1} \to h^{k\to mk}_{j,m} \tag{2.54}$$
$$\left(\ell^{mk}_m\right)^{-1} \to \ell^{mk}_k \tag{2.55}$$
$$h^{N\to N'}_{b',s'} \circ h^{n\to N}_{b,s} \to h^{n\to N'}_{b'+s'b,s's} \tag{2.56}$$

For prime-factor FFT:

$$\mathrm{v}^{m,k} \circ h^{k\to mk}_{kj,1} \to z^{k\to mk}_{j,m} \tag{2.57}$$
$$\mathrm{v}^{m,k} \circ h^{k\to mk}_{j,m} \to z^{k\to mk}_{j,m} \tag{2.58}$$
$$z^{N\to N'}_{b',s'} \circ h^{n\to N}_{b,s} \to z^{n\to N'}_{b'+s'b,s's}, \quad N = sn \tag{2.59}$$
$$z^{N\to N'}_{b',s'} \circ z^{n\to N}_{b,s} \to z^{n\to N'}_{b'+s'b,s's} \tag{2.60}$$

For Rader FFT:

$$w^N_g \circ h^{1\to N}_{0,1} \to h^{1\to N}_{0,1} \tag{2.61}$$
$$w^N_g \circ h^{N-1\to N}_{N-1,1} \to \overline{w}^{N-1\to N}_{1,g} \tag{2.62}$$
$$\overline{w}^{N'\to N}_{\varphi,g} \circ h^{n\to N'}_{b,s} \to \overline{w}^{n\to N}_{\varphi g^b,g^s} \tag{2.63}$$
$$\overline{w}^{N'\to N}_{\varphi,g} \circ z^{n\to N'}_{b,s} \to \overline{w}^{n\to N}_{\varphi g^b,g^s} \tag{2.64}$$

Table 2.8: Index function simplification rules.

To illustrate the complexity of the algebraic simplifications that some of these rules encode, we consider the rule (2.59) as an example. The underlying algebraic transformation can be obtained by plugging the definitions of h and z from (2.34) and (2.52) into the rule. The right hand side of the rule becomes

$$(b' + s'(b + si)) \bmod N' = (b' + s'b + s'si) \bmod N'$$

The above is equivalent to $z_{b'+s'b,\,s's}$ only if the constraints in (2.52) hold. Namely, we must verify that $N' = s'sn$. We have $N' = s'N$ (constraint for $z_{b',\,s'}$), and $N = sn$ from (2.59), combining these two facts, we obtain $N' = s'sn$ as desired. The constraint in (2.52) is not arbitrary, it is required for (2.60).

Continuing with our running example, the index function simplification applies rule (2.54) to (2.43) to obtain

$$\sum_{j=0}^{m-1} \left(S(h_{nj,\,1})\,\mathbf{DFT}_n\,G(h_{j,\,m})\right). \tag{2.65}$$

Step 4: Code generation. The $\sum$-SPL compiler first generates unoptimized code for an $\sum$-SPL formula using the context insensitive mappings given in Table 2.5. As in the original SPL compiler [137], an unrolling parameter to the $\sum$-SPL compiler controls which loops in a $\sum$-

SPL formula will be fully unrolled and thus become basic blocks in which all decorations and index mappings are inlined. Our approach relies on further basis block level optimizations to produce efficient code. These are a superset of the optimizations implemented in the original SPL compiler, including 1) full loop unrolling of computational kernels, 2) array scalarization, 3) constant folding, 4) algebraic strength reduction, 5) copy propagation, 6) dead code elimination, 7) common subexpression elimination, 8) loop invariant code motion, and 9) induction-variable optimizations.

Translating our running example (2.65) into optimized code for $m = 5, n = 2$ using the $\sum$-SPL compiler leads to Implementation 4.

Implementation 4 (Compute $y = (I_5 \otimes F_2)L^{10}_5$ following (2.65) (with loop merging))

```
// Input: _Complex double x[10], output: y[10]
for (int j=0; j<5; j++) {
   y[2*j]   = x[j] + x[j+5];
   y[2*j+1] = x[j] - x[j+5];
}
```

Compared to Implementation 3 this code only consists of one loop, as desired.

2.4.6 The $\sum$-SPL Rewriting System: Rader and Prime-Factor Example

To demonstrate how our framework applies to more complicated SPL formulas, we show the steps for compiling a part of a 3-level recursion for a non-power-of-2 size DFT that uses all three different recursion rules (2.24)–(2.26), namely a $\mathbf{DFT}_{pq}$ with q prime, and $q - 1 = rs$ (for example, $p = 4$, $q = 7$, $r = 3$, and $s = 2$ is a valid combination). Initially the prime-factor decomposition (2.25) is applied for the size pq, then the Rader decomposition (2.26) decomposes the prime size q into $q - 1$. Finally, a Cooley-Tukey step (2.24) is used to decompose $q - 1$ into rs. We only consider a fragment of the resulting SPL formula

$$(I_p \otimes (I_1 \oplus (I_r \otimes \mathbf{DFT}_s)L^{rs}_r)\, W_q)\, V_{p,q}.$$

This formula has three different permutations, and a naive implementation would require three explicit shuffle operations, leading to three extra passes through the data vector.

Our rewriting system merges these permutation matrices into the innermost loop and then simplifies the index mapping function, effectively reducing the number of necessary mod operations to approximately 3 mods per 2 data points and improving the locality of the computation at the same time. Below we show the intermediate steps.

Steps 1 and 2: Expansion of skeleton and loop merging. Rules (2.41) and (2.39) expand the skeleton. Then, the rules in Table 2.7 are used to merge the loops and thus move the decorations into the innermost sum. The resulting $\sum$-SPL formula is

$$\sum_{j_1=0}^{p-1} \Bigg(S(h_{qj_1,\,1} \circ h_{0,\,1} \circ \imath_1) G(\mathrm{v}^{p,q} \circ h_{qj_1,\,1} \circ w^q_g \circ h_{0,\,1}) \\ + \sum_{j_0=0}^{r-1} S(h_{qj_1,\,1} \circ h_{1,\,1} \circ h_{sj_0,\,1})\, \mathbf{DFT}_s\, G(\mathrm{v}^{p,q} \circ h_{qj_1,\,1} \circ w^q_g \circ h_{1,\,1} \circ \ell^{rs}_r \circ h_{sj_0,\,1}) \Bigg).$$

Note that the occurring index functions are very complicated.

Step 3: Index mapping simplification. Using only the rules in Table 2.8, the gather and scatter index mapping functions are simplified to produce the optimized formula

$$\sum_{\substack{j_1=0 \\ b_1=qj_1}}^{p-1} \left(S(h_{0,\,q} \circ (j_1)_p) G(z_{0,\,q} \circ (j_1)_p) + \sum_{\substack{j_0=0 \\ \phi_1=g^{j_0}}}^{r-1} S(h_{qj_1+sj_0+1,\,1})\, \mathbf{DFT}_s\, G(z_{b_1,\,p} \circ \overline{w}^{s\to q}_{\phi_1,g^s}) \right). \quad (2.66)$$

Step 4: Code generation. Now the code can be generated for some specific values of p, q, r and s using the $\sum$-SPL compiler. Below we show optimized code for $p = 4$, $q = 7$, $r = 3$, $s = 2$.

Implementation 5 (Compute $(I_4 \otimes (I_1 \oplus (I_3 \otimes \mathbf{DFT}_2)L^6_3)W_7)V_{4,7}$ following (2.66).)

```
// Input: _Complex double x[28], output: y[28]
int p1, b1;
for(int j1 = 0; j1 <= 3; j1++) {
   y[7*j1] = x[(7*j1%28)];
   p1 = 1; b1 = 7*j1;
   for(int j0 = 0; j0 <= 2; j0++) {
      y[b1 + 2*j0 + 1] =
         x[(b1 + 4*p1)%28] + x[(b1 + 24*p1)%28];
      y[b1 + 2*j0 + 2] =
         x[(b1 + 4*p1)%28] - x[(b1 + 24*p1)%28];
      p1 = (p1*3%7);
   }
}
```

2.4.7 Final Remarks

We presented an extension of the SPL language and compiler used in Spiral to enable loop optimization for signal transform algorithms at the formula level. The approach concurs with the Spiral philosophy which aims to perform optimizations at the "right level" of abstraction. The "right level" depends on the type of optimization and is usually a research question. For the loop optimizations considered here, we found that the right level is between SPL and the actual code as reflected by $\sum$-SPL, which, unlike SPL, represents loops and index mappings explicitly and compactly. Our general approach still requires us to find the proper set of index function simplification rules, which we did for the FFTs (2.25)–(2.26).

The general loop merging solves one of the three fundamental limitations of Spiral, explained in Section 2.3.

Besides, loop optimizations, the formal $\sum$-SPL framework developed here turns out to be the crucial tool for further research in formula level algorithm transformations. In particular, in the next section we will build upon $\sum$-SPL to enable general size library generation with Spiral.

Chapter 3

Library Generation: Library Structure

In this section we address the main challenge posed in this thesis: the automatic generation of general size library code as sketched in Table 1.2(d). The overall goal of the thesis is to completely fill the fourth row of Table 1.1(c), but here we will talk only about the first entry in the row, namely about scalar library generation. The generation of vectorized and parallel libraries is explained in Chapter 4.

To generate a library for a given transform (e.g., $\mathbf{DFT}_n$) we need to be able to generate code for transforms of symbolic size n, i.e., the size becomes an additional parameter. This is fundamentally different from fixed-size code. For example, fixed-size transforms can be completely decomposed by Spiral (Section 2.3) at code generation time using suitable breakdown rules, but if the size is unknown, the breakdown rules can only be applied at runtime, since different rules have different applicability conditions based on the transform size. Thus, to generate a general size library we will need to "compile" the breakdown rules. As we explain in this chapter our approach is based on compiling a set of breakdown rules into a so-called *recursion step closure*, which corresponds to a set of mutually recursive functions, and can be translated into code as explained in Chapter 5.

3.1 Overview

Our goal is to be able to generate different kinds of libraries, that may be desirable for different applications. Examples include adaptive libraries (similar to FFTW) or fixed libraries (similar to Intel IPP). Depending on the library kind and the library infrastructure already in place, the generated code must look differently. We will collectively call the desired library type, infrastructure, and the target language the *library target* or simply target.

The basic idea of our library generation approach decouples target dependent and target independent tasks and is shown in Fig. 3.1. The library generator consists of two high-level modules. The first module, *Library Structure* (described in this chapter and extended in Chapter 4 to include parallelism framework), operates at the algorithm level, and is independent of the library target. The second module, *Library Implementation* (described in Chapter 5, operates at the code (implementation) level, and is dependent on the library target. The Library Structure module produces the so-called *recursion step closure*. Informally, the closure describes the set of functions (which possibly call each other) that compute the needed transforms. The closure is expressed as a set of

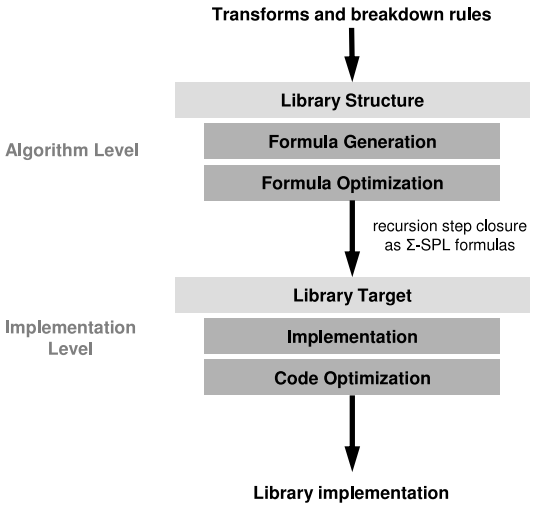


Figure 3.1: Library generation in Spiral.

$\sum$-SPL formulas. The Library Implementation module then uses the closure and the information about the library target to produce the final library implementation.

Our approach separates transform algorithm generation from the library implementation details. For exa mple, if a new transform is requested, and the Library Structure module can generate the closure, the unmodified Library Implementation module can immediately generate several kinds of libraries, adaptive or fixed.

The library generation procedure is completely deterministic, and does not involve any feedback-driven search at *code generation time*, which a lot of previous work on Spiral was centered around. Instead, it moves the platform adaptation to *runtime*, by generating libraries with feedback-driven adaptation mechanisms, similar to FFTW and UHFFT. However, this does not preclude search at code generation time, for example, for the generation of fixed-size library base cases (codelets).

We reuse the transforms, the domain-specific language SPL, and the transform breakdown rules provided with the original Spiral. The rest of the required library generation infrastructure, including $\sum$-SPL, rewrite rules, and the $\sum$-SPL compiler is part of the contribution of this thesis.

In the rest of this chapter we will discuss the *Library Structure* module, based on SPL and $\sum$-SPL. In particular, the $\sum$-SPL framework introduced in the previous section is the crucial tool in the library generation process. First, in Section 3.2 we will introduce the problem of recursive code generation and explain why is it more difficult than generating single size code. Then, in Sections 3.3–3.5 we will walk through a simple example of generating the recursive code for the Cooley-Tukey FFT, and then explain how the process can be automated, and explain all the major steps of the general algorithm that pertain to the *Library Structure* module, leaving out the code-level details to the *Library Implementation* described in Chapter 5.

Next, in Sections 3.6–3.9 we will explain the crucial extension to $\sum$-SPL called the *index-free* $\sum$*-SPL* , which enables a number of important loop optimizations.

3.2 Motivation and Problem Statement

As a motivating example, we consider a simple recursive implementation of the Cooley-Tukey FFT breakdown rule, as applied to $\mathbf{DFT}_n$, with unknown composite n, and divisor $k|n$. We restate the breakdown rule from (2.1), replacing $m = n/k$ and $D_{k,m}$ by $\mathrm{diag}(d)$, where $d^{n\to\mathbb{C}}$ is the scalar function that provides the diagonal entries of $D_{k,m}$.

$$\mathbf{DFT}_n = (\mathbf{DFT}_k \otimes I_{n/k})\,\mathrm{diag}(d)(I_k \otimes \mathbf{DFT}_{n/k})L^n_k. \tag{3.1}$$

Pseudo code for the computation of $y = \mathbf{DFT}_n\, x$ following (3.1) could look as follows.

Implementation 6 (Compute $y = \mathbf{DFT}_n\, x$ following (3.1))

```
void dft(int n, complex *y, complex *x) {
   int k = choose_factor(n);
   complex *t1, *t2, *t3;

   // t1 = L^n_k*x
   t1 = Permute x with L(n,k);

   // t2 = (I_k tensor DFT_n/k)*t1
   for(int i=0; i<k; ++i)
      dft(n/k, t2 + n/k*i, t1 + n/k*i);

   // t3 = diag( d(j) )*t2
   for(int i=0; i<n; ++i)
      t3[i] = d(i) * t2[i];

   // y = (DFT_k tensor I_n/k)*t3
   // cannot call dft() recursively, need strided I/O
   for(int i=0; i<n/k; ++i)
      dft_str(k, n/k, y + i, t3 + i); // stride = s
}

void dft_str(int n, int stride, complex *y, complex *x) {
    // has to be implemented
}
```

The main problem here is that the straightforward implementation of the DFT function is not self-contained. First, it requires the precomputed diagonal elements $d(i)$. Second, and more important, $(\mathbf{DFT}_k \otimes I_{n/k})$ requires the computation of the DFT on *strided* data; this means that we also need a new kind of DFT function, `dft_str`, parametrized by the stride. Thus, even in this straightforward implementation, an additional function that accommodates an extra parameter needs to be implemented.

In a real implementation, the situation can get much more complicated, since optimizations may introduce the need for a larger set of additional functions.

We will call such functions *recursion steps*. The original transform function (here: `dft`) is also a recursion step. Informally, a recursion step implements a transform with some context, which may include, but is not limited to, gather and scatter operations (as in the case of `dft_str` above),

diagonal scaling, and loops. Note, that the name "recursion step" does not necessarily imply recursion. Even an iterative algorithm will require recursion steps in the sense of our definition.

For example, FFTPACK [114–116] internally uses 6 different DFT recursion steps. Each of them corresponds to different stages of a mixed-radix iterative algorithm. The fast iterative split-radix FFT implementation by Takuya Ooura [92] uses 28 recursion steps to implement both the complex DFT and the real-valued DFT (RDFT). FFTW 3.x [61] uses more than 20.

Here we consider the less sophisticated FFTW 2.x [60] for an easier comparison. In FFTW 2.x two different variants of the DFT function are used. The Cooley-Tukey FFT (3.1) is computed in two loops corresponding to the two tensor products and the diagonal and the stride permutation in (3.1) are respectively fused with these loops. Conceptually this implies the following partition of the SPL breakdown rule:

$$\mathbf{DFT}_n = \underbrace{(\mathbf{DFT}_k \otimes I_{n/k})\,\mathrm{diag}(d)}_{\text{loop}}\,\underbrace{(I_k \otimes \mathbf{DFT}_{n/k})L^n_k}_{\text{loop}}.$$

Fusion of the stride permutation L^n_k with the first loop requires a DFT function with different input and output stride, and the fusion of the diagonal $\mathrm{diag}(d)$ with the second loop requires a DFT function with an extra array parameter d, which holds the scaling factors. The corresponding pseudo code is given in Implementation 7. One can think of this implementation, as a recursive equivalent of the $\sum$-SPL loop merging shown in Section 2.4.

Implementation 7 (FFTW 2.x-like implementation of (3.1) *)*

```
void dft(int n, complex *Y, complex *X) { int k = choose_factor(n)

    // y = (I_k tensor DFT_n/k)L(n,k)*x
    for(int i=0; i<k; ++i)
        dft_str(n/k, k, 1, y + (n/k)*i, x + (n/k)*i);

    // y = (DFT_k tensor I_n/k) diag(d(j)) y
    for(int i=0; i<n/k; ++i)
        dft_scaled(k, n/k, precomputed_d[i], y + i, y + i);
}

void dft_str(int n, int in_stride, int out_stride, complex *y, complex *x) {
    // has to be implemented
}

void dft_scaled(int n, int stride, complex *d, complex *y, complex *x) {
    // has to be implemented
}
```

In FFTW 2.x, the function `dft_str` is implemented recursively based on (3.1) similarly to Implementation 7. It calls itself and `dft_scaled`, and does not require any new functions. Eventually, the DFT size becomes sufficiently small, and the recursion terminates using a so-called *codelet* (a function that computes a small fixed-size DFT) for `dft_str`. `dft_scaled` is not recursive in FFTW 2.x, and is always implemented as a codelet, called *twiddle codelet* in this case.

We say that the functions `dft`, `dft_str`, and `dft_scaled` form a so-called *recursion step closure*, which is a minimal set of functions sufficient to compute the transform. The recursion step closure

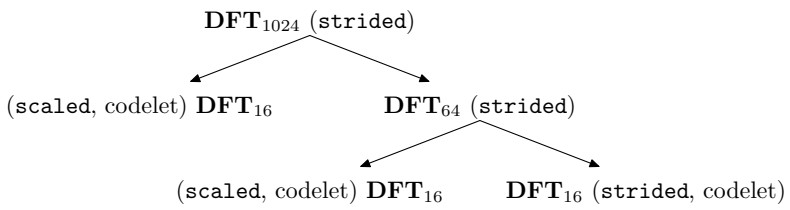


Figure 3.2: Possible call tree for computing $\mathbf{DFT}_{1024}$ in FFTW 2.x using Implementation 7 (functions names do not correspond to actual FFTW functions).

is the central concept in our library generation framework. Later in this chapter we will show how to compute the closure automatically.

The codelets in FFTW are functions that implement `dft_str` and `dft_scaled` for a fixed, small value of n. The library provides codelets for several small values of $n \leq 64$. We can visualize the FFTW recursion graphically, using a call tree. For example, Fig. 3.2 shows a possible recursion for $\mathbf{DFT}_{1024}$. Since `dft_scaled` must be a codelet, only rightmost trees are supported. The array of precomputed diagonal elements `precomputed_d` in `dft_scaled` is created in an initialization stage using the rest of the library infrastructure.

Besides loop merging FFTW 2.x implements also the following optimizations:

- *Buffer reuse.* The second step, `dft_scaled` is performed inplace, reusing the y array, which we call a "buffer". This eliminates the need for extra buffers, beyond the ones provided by the library user. This optimization minimizes memory usage and improves cache performance. In addition, this optimization allows `dft_scaled` to take only a single stride as parameter (in contrast to separate input and output strides as needed for `dft_str`).

- *Looped codelets.* The codelet versions of `dft_scaled` and `dft_str` include the immediate outer loops. This improves performance, since it eliminates extra function call overhead, and enables the C compiler to reuse certain address computations and perform loop invariant code motion.

At the hand of the example above, we can now state the general problem of generating recursive library code for a single breakdown rule.

Problem 1 (Library generation for a single breakdown rule) **Given:** *A transform T and a breakdown rule that decomposes T into transforms of the same type (as in* (3.1)*).* **Tasks:**

1. *Find the recursion step closure, i.e., the minimal set of needed recursion steps (functions).*
2. *Implement each recursion step using again the given breakdown rule.*
3. *Generate the base cases (codelets) for the recursion steps for several small fixed sizes.*
4. *Combine the recursion steps and codelets into a library infrastructure.*

Tasks 1–3 are performed by the "Library Structure" block in Fig. 3.1, and are the focus of the rest of this chapter. Task 4 is dependent on the desired library infrastructure and is handled by the "Library Implementation" block and explained in Chapter 5.

We will proceed by showing a detailed informal example of manually performing the first two tasks in the procedure above. Then, we explain the general procedure, explain how to generate base cases, and how to incorporate buffer reuse and looped codelets.

3.3 Recursion Step Closure

The first step in the library generation process is the computation of the recursion step closure. The closure provides a rough sketch of the library, and is the central object throughout the thesis.

3.3.1 Example

We will walk through the formal derivation of the required recursion steps for the Cooley-Tukey FFT (3.1) using $\sum$-SPL and explain their implementation. These are tasks 1 and 2 of Problem 1. Informally, the procedure consists of several steps:

1. Apply the breakdown rule to the transform T to obtain an SPL formula.
2. Convert the SPL formula into $\sum$-SPL using Table 2.6.
3. Apply loop merging rewriting rules (Table 2.7).
4. Apply index simplification rules (Table 2.8).
5. Extract the required recursion steps from the $\sum$-SPL formula (we will show that this is possible using rewriting).
6. Repeat this procedure for each recursion step until closure is reached, i.e., no new recursion steps appear.

First, we define the *recursion step tag*. Given a formula F, $\left\{F\right\}$ means that F is to be implemented as a recursion step, i.e., a separate function call. We will not tag arbitrary F, but only transforms and transforms with $\sum$-SPL decorations (i.e., gather, scatter, diagonal matrices, as explained in Sec. 2.4.4). Intuitively, the tag is the $\sum$-SPL equivalent of a function call, and the boundaries of such function calls are shown by tagging larger fragments of formulas. This will become clearer below.

Now, we proceed with the example.

Example 1 (Find and implement recursion steps for the Cooley-Tukey FFT) **Given:** $\mathbf{DFT}_n$ *and Cooley-Tukey FFT* (3.1). **Tasks:**

- *Find the recursion step closure, i.e., the minimal set of needed recursion steps (functions).*
- *Implement each recursion step using again the given breakdown rule.*

We start with $\mathbf{DFT}_n$ and follow the six steps explained above.

Step 1: Apply the breakdown rule. First, we apply the breakdown rule to **DFT**. The smaller transforms are tagged as $\left\{\,\mathbf{DFT}\,\right\}$. We obtain

$$\left\{\,\mathbf{DFT}_n\,\right\} = (\left\{\,\mathbf{DFT}_k\,\right\} \otimes I_m)\,\mathrm{diag}(d)(I_k \otimes \left\{\,\mathbf{DFT}_m\,\right\})L^n_k. \tag{3.2}$$

Step 2: Convert to $\sum$-SPL. Using SPL we cannot merge the permutation and the diagonal with the tensor products as explained in Section 2.4. Thus, in the next step we convert (3.2) to $\sum$-SPL. This is done by applying the rules from Table 2.6 and yields

$$\left(\sum_{i=0}^{(k-1)} S(h_{i,\,k})\big\{\,\mathbf{DFT}_{n/k}\,\big\}G(h_{i,\,k})\right)\operatorname{diag}\big(d\big)\left(\sum_{j=0}^{(n/k-1)} S(h_{jk,\,1})\big\{\,\mathbf{DFT}_{k}\,\big\}G(h_{jk,\,1})\right)\operatorname{perm}(\ell^{n}_{n/k}). \tag{3.3}$$

Step 3: Apply loop merging rewrite rules. We apply the rules from Table 2.7 to merge in (3.3) the permutation and the diagonal with the adjacent loops. The result is

$$\left(\sum_{i=0}^{(k-1)} S(h_{i,\,k})\big\{\,\mathbf{DFT}_{n/k}\,\big\}\operatorname{diag}\big(d\circ h_{i,\,k}\big)G(h_{i,\,k})\right)\left(\sum_{j=0}^{(n/k-1)} S(h_{jk,\,1})\big\{\,\mathbf{DFT}_{k}\,\big\}G(\ell^{n}_{n/k}\circ h_{jk,\,1})\right). \tag{3.4}$$

Step 4: Apply index simplification rules. We apply the rewriting rules from Table 2.8 to simplify the index mapping functions in (3.4), and obtain the formula

$$\left(\sum_{i=0}^{(k-1)} S(h_{i,\,k})\big\{\,\mathbf{DFT}_{n/k}\,\big\}\operatorname{diag}\big(d\circ h_{i,\,k}\big)G(h_{i,\,k})\right)\left(\sum_{j=0}^{(n/k-1)} S(h_{jk,\,1})\big\{\,\mathbf{DFT}_{k}\,\big\}G(h_{j,\,n/k})\right). \tag{3.5}$$

The only affected construct is the rightmost gather $G()$, in which the composed index mapping was simplified.

Step 5: Extract the required recursion steps. At this point, we effectively performed loop merging for one step of the Cooley-Tukey FFT. To further merge loops *recursively* for the smaller $\mathbf{DFT}_{n/k}$ and $\mathbf{DFT}_k$ we push the decoration constructs inside these recursion steps, effectively creating more powerful recursion steps. Formally, this is done by expanding the scope of the recursion step tag, i.e., by simply moving all of the adjacent decoration constructs inside the recursion step tag (curly braces):

$$\left(\sum_{i=0}^{(k-1)} \big\{S(h_{i,\,k})\,\mathbf{DFT}_{n/k}\operatorname{diag}\big(d\circ h_{i,\,k}\big)G(h_{i,\,k})\big\}\right)\left(\sum_{j=0}^{(n/k-1)} \big\{S(h_{jk,\,1})\,\mathbf{DFT}_{k}\,G(h_{j,\,n/k})\big\}\right). \tag{3.6}$$

In (3.6) we can already see the dual loop structure, and the following two distinct recursion steps:

$$\big\{S(h_{i,\,k})\,\mathbf{DFT}_{n/k}\operatorname{diag}\big(d\circ h_{i,\,k}\big)G(h_{i,\,k})\big\}\text{ and} \tag{3.7}$$

$$\big\{S(h_{jk,\,1})\,\mathbf{DFT}_{k}\,G(h_{j,\,n/k})\big\}. \tag{3.8}$$

In both recursion steps the input and the output are accessed using the stride function h. In addition, the first recursion step (3.7) contains the diagonal matrix, which scales the input. This formally captures the fact that (3.7) can be implemented using the `dft_scaled` function, and (3.8) can be implemented with the `dft_str` function in Implementation 7.

In summary $\sum$-SPL and the simple application of $\sum$-SPL rewriting rules reveals the two recursion steps necessary for the Cooley-Tukey FFT. The pseudo code for the implementation of (3.6) now looks essentially the same as in FFTW 2.x:

Implementation 8 (Compute $y = \mathbf{DFT}_n x$ following (3.6))

```
void dft(int n, complex *Y, complex *X) {
    int k = choose_factor(n)
    for(int i=0; i<k; ++i)
        dft_str(k, n/k, 1, X + i, T + i*k)

    for(int i=0; i<n/k; ++i)
        dft_scaled(n/k, k, precomputed_doh + i*k, Y + i, Y + i)
}
```

Note, that the diagonal has to be precomputed according to the scalar function $d \circ h_{i,\,k}$ in (3.7). Different segments of the precomputed diagonal are passed to `dft_scaled` in different iterations.

Step 6: Implement recursion steps. The final step is to implement the recursion steps (3.7) and (3.8), by repeating Steps 1–5. The procedure is analogous, and the recursion step decoration constructs (e.g., $S(h)$ and $G(h)$ in (3.8)) will be automatically merged with the decorations inside the expansion of the **DFT** using loop merging rules in Table 2.7. We will show this only for (3.8), assuming the radix $m \mid k$:

$$\begin{aligned} S(h_{jk,\,1})\,\mathbf{DFT}_k\,G(h_{j,\,n/k}) &= S(h_{jk,\,1})(\{\,\mathbf{DFT}_{k/m}\,\} \otimes I_m)\,\mathrm{diag}\,(d)(I_{k/m} \otimes \{\,\mathbf{DFT}_m\,\})L^k_{k/m}G(h_{j,\,n/k}) \\ &= \left(\sum_{l=0}^{(m-1)} \left\{S(h_{(jk+l),\,m})\,\mathbf{DFT}_{k/m}\,\mathrm{diag}\,\big(d \circ h_{l,\,m}\big)G(h_{l,\,m})\right\}\right) \\ &\quad \cdot \left(\sum_{r=0}^{(k/m-1)} \left\{S(h_{rm,\,1})\,\mathbf{DFT}_m\,G(h_{(j+(nr)/k),\,n/m})\right\}\right). \end{aligned} \tag{3.9}$$

Observe that the final result is similar to (3.6). The transform sizes, the strides, and other parameters differ, but the two recursion steps still correspond to `dft_scaled` and `dft_str`. In other words no new recursion steps are needed.

If we assume that the recursion step (3.7) is always implemented as a codelet (as in FFTW 2.x), then there is no need to recursively implement it. Under this assumption, the recursion steps $\{\,\mathbf{DFT}_n\,\}$, (3.7), and (3.8) form a recursion step *closure*, i.e., a sufficient set of recursion steps to recursively implement the **DFT** using Cooley-Tukey FFT.

3.3.2 Overview of the General Algorithm

Next, we define the terminology we will use and then explain the steps in the previous example in a more complete and rigorous way.

A *recursion step* is a transform with additional "context", e.g., it is usually surrounded by scatter and gather constructs as in (3.7) and (3.8). A recursion step, after applying a procedure that we call *parametrization*, specifies the exact signature of the associated recursive function, such as `dft_str` in Implementation 7.

We refer to the recursion step expanded using a breakdown rule and rewritten as explained in Steps 1–5 in Section 3.3.1 as a $\sum$*-SPL implementation* of a recursion step. The process of obtaining a $\sum$-SPL implementation is called *descending* (into the recursion step). For example, (3.6) is a $\sum$-SPL implementation of $\mathbf{DFT}_n$; indeed, it can be converted into the DFT code in Implementation 8.

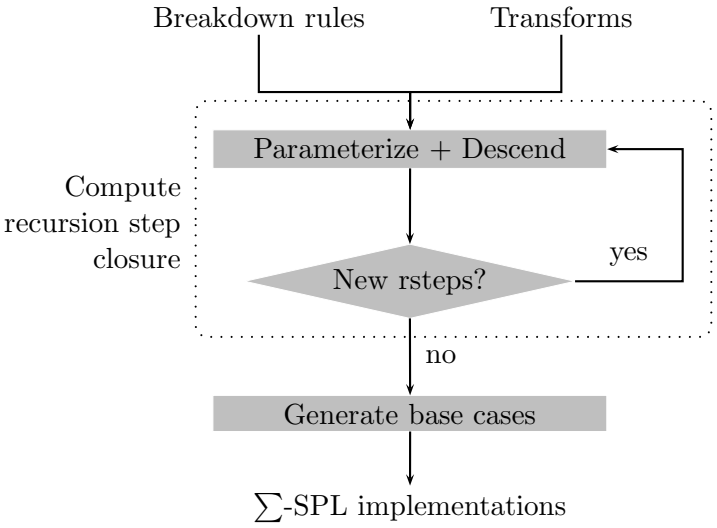


Figure 3.3: Library generation: "Library Structure". Input: transforms and breakdown rules. Output: the recursion step closure (if it exists) and $\sum$-SPL implementations of each recursion step.

A $\sum$-SPL implementation is a $\sum$-SPL formula, which expresses the given recursion step in terms of other (smaller) recursion steps, or in terms of primitive $\sum$-SPL constructs. The former is a recursive $\sum$-SPL implementation obtained by descending, and the latter is non-recursive, implemented by a base case. Two examples of recursive $\sum$-SPL implementations are (3.9) and (3.6).

The main steps performed by the *Library Structure* block are shown in Figure 3.3. The input is a set of transforms (which are the simplest cases of recursion steps) and breakdown rules. The output is the recursion step closure and recursive and non-recursive (base case) $\sum$-SPL implementations of each recursion step.

In this section we discuss the *"Compute recursion step closure"* block, which takes as input breakdown rules and recursion steps (the initial transform is a recursion step) and produces the $\sum$-SPL closure and recursive general-size implementations. This is done by parametrizing and descending into the recursion steps, until no new recursion steps are found. The "Generate base cases" block, discussed in Section 3.4, adds additional, non-recursive fixed-size implementations for several recursion steps.

To make recursion step formulas self-contained functions, we need to replace all expressions with free variables (such as loop indices) by a set of independent parameters. These parameters will become the function arguments. We call this process *parametrization* of the recursion step; it is explained next.

3.3.3 Parametrization

We denote recursion step 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. The 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 these constraints.

The procedure and further details are best explained using our running example, $\mathbf{DFT}_n$.

The parametrization of $\mathbf{DFT}_n$ is trivial. We replace $n = u_1$. There are no constraints, so $\mathbf{DFT}_{u_1}$ is a parametrized recursion step. After descending we obtain (3.6) and the two recursion steps (3.7) and (3.8), 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 (3.8). First, we replace expressions with free variables by new parameters:

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

Note that 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.

Next, we determine 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, one of the parameters is assumed to be known, and the resulting system of linear equations is solved for the rest of the parameters. The solution is then substituted into the $\sum$-SPL formula. In this 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:

$$S(h^{u_1\to u_2}_{u_3,\,1})\,\mathbf{DFT}_{u_1}\,G(h^{u_1\to u_6}_{u_7,\,u_8}). \tag{3.10}$$

Next, we consider (3.7). The actual definition of the diagonal matrix in (3.7) in Spiral, contains a special marker pre$(\cdot)$ (which we did not show) 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. We will denote such parameters using the previous notation, as $u^{A\to B}$. As before, A and B specify the domain and the range of the function u, with the additional caveat that both A and B can be integer parameters themselves, in which case they denote $\mathbb{I}_A$ and $\mathbb{I}_B$.

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

$$S(h_{i,\,k})\,\mathbf{DFT}_{n/k}\,\mathrm{diag}\left(\mathrm{pre}(d\circ h_{i,\,k})\right)G(h_{i,\,k}) \xrightarrow{\text{par}} S(h^{u_1\to u_2}_{u_3,\,u_4})\,\mathbf{DFT}_{u_1}\,\mathrm{diag}\left(\mathrm{pre}({u_7}^{u_1\to\mathbb{C}})\right)G(h^{u_1\to u_9}_{u_{10},\,u_{11}}). \tag{3.11}$$

Each parametrized recursion step can be used to mechanically construct a function declaration. The parameters become function arguments and the function name can be assigned in a variety of ways, including building a name hash or linearizing a $\sum$-SPL formula into text. In the example below, we create function declarations for (3.10) and (3.11) using simple numbering to assign names:

```
void rstep1(int u1, int u2, int u3, int u6, int u7,
    int u8, complex *Y, complex *X);
```

$$S(h^{u_1\to u_2}_{u_3,\,1})\,\mathbf{DFT}_{u_1}\,G(h^{u_1\to u_6}_{u_7,\,u_8}) \xrightarrow{\text{Descend}} \sum_{i=0}^{u_1/f_1-1} \left\{S(h^{f_1\to u_2}_{u_3+i,\,u_1/f_1})\,\mathbf{DFT}_{f_1}\,\mathrm{diag}\left(\mathrm{pre}(\Omega^{u_1}_{u_1/f_1}\circ h^{f_1\to u_1}_{i,\,u_1/f_1})\right)G(h^{f_1\to u_1}_{i,\,u_1/f_1})\right\} \cdot \sum_{j=0}^{f_1-1}\left\{S(h^{u_1/f_1\to u_1}_{u_1 j/f_1,\,1})\,\mathbf{DFT}_{u_1/f_1}\,G(h^{u_1/f_1\to u_6}_{u_7+u_8 j,\,u_8 f_1})\right\} \tag{3.12}$$

$$S(h^{u_1\to u_2}_{u_3,\,1})\,\mathbf{DFT}_{u_1}\,G(h^{u_1\to u_6}_{u_7,\,u_8}) \xrightarrow{\text{RDescend}} \left\{\begin{array}{c} S(h^{u_1\to u_2}_{u_3,\,1})\,\mathbf{DFT}_{u_1}\,G(h^{u_1\to u_6}_{u_7,\,u_8}) \\ S(h^{u_1\to u_2}_{u_3,\,u_4})\,\mathbf{DFT}_{u_1}\,\mathrm{diag}\left(\mathrm{pre}({u_7}^{u_1\to\mathbb{C}})\right)G(h^{u_1\to u_9}_{u_{10},\,u_{11}}) \end{array}\right\} \tag{3.13}$$

$$S(h^{*\to *}_{*,\,1})\,\mathbf{DFT}_{*}\,G(h^{*\to *}_{*,\,*}) \xrightarrow{\text{RDescend}*} \left\{\begin{array}{c} S(h^{*\to *}_{*,\,1})\,\mathbf{DFT}_{*}\,G(h^{*\to *}_{*,\,*}) \\ S(h^{*\to *}_{*,\,*})\,\mathbf{DFT}_{*}\,\mathrm{diag}\left(\mathrm{pre}({*}^{*\to\mathbb{C}})\right)G(h^{*\to *}_{*,\,*}) \end{array}\right\} \tag{3.14}$$

Table 3.1: Descending into the recursion step (3.10) using Cooley-Tukey FFT (3.1). The left-hand side is the original recursion step, and the right-hand sides are as follows: "Descend" is the result of application of the breakdown rule and $\sum$-SPL rewriting, "RDescend" is the list of parametrized recursion steps needed for the implementation, "RDescend*" is same as "RDescend", but uses "*" to denote the parameter slot.

```
void rstep2(int u1, int u2, int u3, int u4,
    complex (*u7)(int), int u9, int u10, int u11,
    complex *Y, complex *X);
```

The order of the parameters is based on the order in the formula. These mechanically derived function declarations should be compared to the declarations of `dft_str` and `dft_scaled` in Implementation 7. The declarations above have more parameters, and also uses a pointer to a generating function (u_7) for the scaling diagonal entries, instead of an array pointer. Extra parameters are an artifact of an automated system, and it might be possible to prove their redundancy under certain circumstances. The function pointers, on the other hand, provide a convenient abstraction. The diagonal entry precomputation is handled by the *Library Implementation* module (Chapter 5), which also replaces function pointers by array pointers, where precomputation was desired.

The parametrization accomplishes an important task: it creates *reusable* recursion steps.

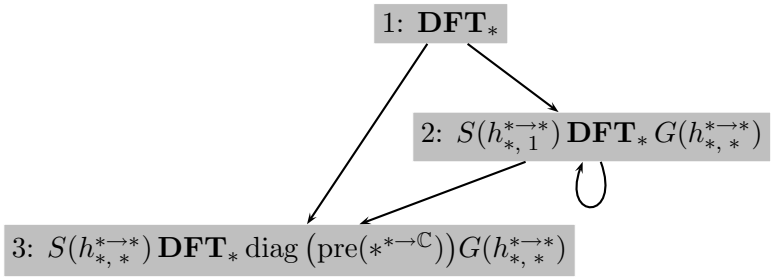
3.3.4 Descend

After parametrization we descend into each recursion step as was outlined earlier. Continuing with the example, we now descend into (3.10) and (3.11) to obtain their implementations. We only discuss (3.11) in detail.

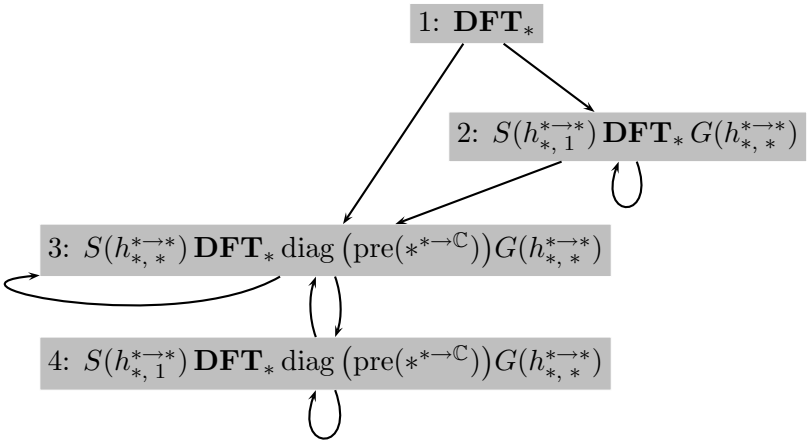
We expand (3.11) using again (3.1). Note that in (3.1), there is one degree of freedom, the value of k, which will become an additional parameter that we will denote with f_i. Inserting (3.1) into (3.11) yields

$$S(h^{u_1\to u_2}_{u_3,\,1})(\left\{\,\mathbf{DFT}_{f_1}\,\right\}\otimes I_{u_1/f_1})\,\mathrm{diag}\left(\mathrm{pre}(d)\right)(I_{f_1}\otimes\left\{\,\mathbf{DFT}_{u_1/f_1}\,\right\})L^{u_1}_{f_1}G(h^{u_1\to u_6}_{u_7,\,u_8}).$$

There are four differences compared to (3.1): the DFT size is u_1, the single degree of freedom k in (3.1) is made explicit and fixed ($k = f_1$), the diagonal is marked with the precompute marker pre($\cdot$),



(a) Restricted recursion



(b) Full recursion

Figure 3.4: Graphical representation of the recursion step closure obtained from the Cooley-Tukey FFT (3.1). The closure in (b) corresponds to (3.15).

and is given in terms of its generating function Ω (same as d before), and finally the potential new recursion steps (DFTs) are marked but are not yet finalized.

Next, $\sum$-SPL conversion and further rewriting is performed using $\sum$-SPL loop merging and index simplification rules. The result is given in (3.12) in Table 3.1.

The result looks complicated due to the high level of detail, but it is a completely specified implementation of (3.11) using the Cooley-Tukey rule (3.1). For the recursion step closure we are only interested in the required children. This is accomplished by RDescend, which extracts the parametrized children from the result of Descend, as shown in (3.13). For readability it makes sense to drop the parameter names, and replace them with "*". This is done by RDescend* in (3.14).

3.3.5 Computing the Closure

Iterating RDescend will produce the desired closure. Inspection shows that the recursion steps in (3.13) are equal to (3.10) and (3.11). Therefore, no new recursion steps are needed.

If we do not allow recursion for (3.10), the recursion step closure is complete with (3.10) and

(3.11). The closure can be visually represented using a call graph. For example, the call graph for the closure with this restricted recursion is shown in Figure 3.4(a). The nodes are the recursion steps (here shown as "*"-ed versions). The edges show the result of RDescend*.

To obtain the full closure without recursion restrictions, we apply RDescend also to (3.10). (3.10) spawns two recursion steps, itself and a new specialized variant, which itself produces no new recursion steps. This yields the result below:

$$\mathbf{DFT}_* \xrightarrow{\text{Closure}^*} \left\{ \begin{array}{c} \mathbf{DFT}_* \\ S(h^{*\to *}_{*,\,1})\,\mathbf{DFT}_*\,G(h^{*\to *}_{*,\,*}) \\ S(h^{*\to *}_{*,\,*})\,\mathbf{DFT}_*\,\mathrm{diag}\left(\mathrm{pre}(*^{*\to \mathbb{C}})\right)G(h^{*\to *}_{*,\,*}) \\ S(h^{*\to *}_{*,\,1})\,\mathbf{DFT}_*\,\mathrm{diag}\left(\mathrm{pre}(*^{*\to \mathbb{C}})\right)G(h^{*\to *}_{*,\,*}) \end{array} \right\} \tag{3.15}$$

The overall recursion step closure has now four recursion steps. The associated call graph is shown in Figure 3.4(b).

The first three recursion steps in Fig. 3.4(a) and Fig. 3.4(b) correspond to the `dft`, `dft_str`, and `dft_scaled` functions in Implementation 7. The fourth recursion step in Fig. 3.4(b) is a special case of the third, with the one parameter (the scatter stride) being equal to 1. This information is useful, because it can lead to better performance due to reduced index computation.

We now state the general algorithm for a single breakdown rule below:

Algorithm 1 (Recursion step closure for a single breakdown rule) **Given:** *A transform T and a breakdown rule B that decomposes T into transforms of the same type.* **Find:** *The recursion step closure R as a set of $\sum$-SPL formulas.*

Closure(T, B)

1: $R \leftarrow \{\}$
2: $W \leftarrow \{T_{u_1}\}$
3: ***while*** $W \neq \{\}$ ***do***
4: $\quad R \leftarrow R \cup W$
5: $\quad W \leftarrow \left(\bigcup_{w \in W} \mathrm{RDescend}(w, B) \right) \backslash R$
6: ***end while***
7: **return** R

The procedure for computing the closure is as follows. Given a transform T, we initialize the closure $R = \{T_{u_1}\}$. Then, we apply RDescend to every element in R, adding the new recursion steps to R. This is repeated until R does not grow any more. The worklist W is used to keep track of new (not yet descended into) recursion steps.

3.3.6 Handling Multiple Breakdown Rules

The extension to multiple breakdown rules is rather straightforward. At every iteration of Algorithm 1, instead of applying a single breakdown rule, we apply all applicable rules to all recursion steps in the worklist W. The algorithm is formally stated below.

Algorithm 2 (Recursion step closure for multiple transforms and multiple breakdown rules) **Given:** *A set of breakdown rules $B = \{B^i\}$ for transforms $T = \{T^i\}$ (not necessarily all different).* **Find:**

The recursion step closure R to compute all T^i using all possible recursive combinations of applicable breakdown rules.

Given a set of parametrized recursion steps W, we denote with $W(T) = \{w \in W \mid T \text{ occurs in } w\}$.
Closure(T, B)

1: $R \leftarrow \{\}$
2: $W \leftarrow \bigcup_i \{T^i_{u_1}\}$
3: ***while*** $W \neq \{\}$ ***do***
4: $\quad R \leftarrow R \cup W$
5: $\quad W \leftarrow \left(\bigcup_i \bigcup_{w \in W(T^i)} \text{RDescend}(w, B^i) \right) \setminus R$
6: ***end while***
7: **return** R

The only difference between Algorithm 2 and Algorithm 1 are lines 2 and 5. The algorithm starts out with the closure being equal to the set of the given transforms. The closure is expanded by applying all applicable breakdown rules to the new recursion steps, until no new recursion steps appear.

3.3.7 Termination

The recursion step closure computation using Algorithm 1 or Algorithm 2 is not guaranteed to terminate; or, in other words, the closure could be infinite. Termination strongly depends on the rewriting rules in Table 2.7 and, most importantly, on the index-mapping function simplification rules in Table 2.8. For example, in the Cooley-Tukey FFT, the three applicable rules are (2.54), (2.55), and (2.56). These form the minimal set of needed rewrite rules. If any one is removed, the closure becomes infinite.

3.3.8 Unification: From $\sum$-SPL Implementations to Function Calls

If we want to compile a complicated $\sum$-SPL expression like (3.12) to code, we need to be able to generate function calls to other recursion steps. This topic is discussed in additional detail in Chapter 5, and here we give the basic intuition.

To generation a function call, we need to determine the arguments for it. To convert the recursion step call $\{F\}$ into a function call, we perform the so-called *unification* [9] of $\{F\}$ with the parametrization of F. This provides the concrete values for the parameters u_i of the invoked recursion step. These values are then used to produce a function call in the code. For example,

$$\text{Unify}(S(h^{k \to n}_{jk,\,1})\,\mathbf{DFT}_k\,G(h^{k \to n}_{j,\,n/k}),\ S(h^{u_1 \to u_2}_{u_3,\,1})\,\mathbf{DFT}_{u_1}\,G(h^{u_1 \to u_6}_{u_7,\,u_8})) \\ = (u_1 \to k, u_2 \to n, u_3 \to jk, u_6 \to n, u_7 \to k, u_8 \to n/k)$$

Using the result above, the pseudo-code for the function call then becomes

```
rstep1(k, n, jk, n, k, n/k);
```

3.4 Generating Base Cases

In Section 3.3 we explained how to compute the recursion step closure and obtain implementations of each recursion step. These implementations are parametrized and, in particular, are for general input size. However, at runtime, the recursion must be terminated with fixed size base cases. In this section we will address the problem of automatically generating such base cases. In FFTW, the base cases are called codelets.

The base cases should be available for a range of fixed small sizes to reduce recursion overhead. A straightforward approach is to generate base cases for each recursion step.

To generate base case implementations we assume the list of desired fixed size transforms is known in advance, e.g., $B = \{\mathbf{DFT}_2, \mathbf{DFT}_3, \mathbf{DFT}_4\}$, and follow the following steps.

1. Generate the set of fixed size recursion step formulas by taking the cross-product of sizes and recursion steps.

2. Find the subset of fixed-size transforms needed and use the standard Spiral system to search for recursion free implementations of these transforms.

3. Form the fixed size implementations of the recursion steps by plugging in transform implementations from step 2 into the fixed size recursion step formulas from step 1.

We will now walk through each step using the closure (3.15) as the running example.

Generate fixed size recursion step formulas. Each recursion step is matched against each fixed size transform in B to determine the constant parameters (including, but not limited to, the size). The parameters are then inserted into the recursion step formula to obtain a fixed size formula.

Consider, for instance, the matching of (3.10) against $\mathbf{DFT}_2$, which yields $u_1 = 2$. Inserting 2 for u_1 in (3.10) leads to

$$S(h^{2\to u_2}_{u_3,\,1})\,\mathbf{DFT}_2\,G(h^{2\to u_6}_{u_7,\,u_8}). \tag{3.16}$$

After matching all elements of B against our closure we obtain the following set of formulas:

$$\left\{\begin{array}{c}
\mathbf{DFT}_2\\
\mathbf{DFT}_3\\
\mathbf{DFT}_4\\
S(h^{2\to u_2}_{u_3,\,1})\,\mathbf{DFT}_2\,G(h^{2\to u_6}_{u_7,\,u_8})\\
S(h^{3\to u_2}_{u_3,\,1})\,\mathbf{DFT}_3\,G(h^{3\to u_6}_{u_7,\,u_8})\\
S(h^{4\to u_2}_{u_3,\,1})\,\mathbf{DFT}_4\,G(h^{4\to u_6}_{u_7,\,u_8})\\
S(h^{2\to u_2}_{u_3,\,u_4})\,\mathbf{DFT}_2\,\mathrm{diag}\left(\mathrm{pre}({u_7}^{2\to\mathbb{C}})\right)G(h^{2\to u_9}_{u_{10},\,u_{11}})\\
S(h^{3\to u_2}_{u_3,\,u_4})\,\mathbf{DFT}_3\,\mathrm{diag}\left(\mathrm{pre}({u_7}^{3\to\mathbb{C}})\right)G(h^{3\to u_9}_{u_{10},\,u_{11}})\\
S(h^{4\to u_2}_{u_3,\,u_4})\,\mathbf{DFT}_4\,\mathrm{diag}\left(\mathrm{pre}({u_7}^{4\to\mathbb{C}})\right)G(h^{4\to u_9}_{u_{10},\,u_{11}})\\
S(h^{2\to u_2}_{u_3,\,1})\,\mathbf{DFT}_2\,\mathrm{diag}\left(\mathrm{pre}({u_6}^{2\to\mathbb{C}})\right)G(h^{2\to u_8}_{u_9,\,u_{10}})\\
S(h^{3\to u_2}_{u_3,\,1})\,\mathbf{DFT}_3\,\mathrm{diag}\left(\mathrm{pre}({u_6}^{3\to\mathbb{C}})\right)G(h^{3\to u_8}_{u_9,\,u_{10}})\\
S(h^{4\to u_2}_{u_3,\,1})\,\mathbf{DFT}_4\,\mathrm{diag}\left(\mathrm{pre}({u_6}^{4\to\mathbb{C}})\right)G(h^{4\to u_8}_{u_9,\,u_{10}})
\end{array}\right\}. \tag{3.17}$$

Find and implement needed transforms. In our example, this step is trivial. Searching for transforms in (3.17) reveals that we need $\{\mathbf{DFT}_2, \mathbf{DFT}_3, \mathbf{DFT}_4\}$. In the general case, however, this step can be more involved, due to possible additional auxiliary transforms. This happens, for

example, for breakdown rules like (2.6). Due to the large number of transforms, the set B lists base cases for all possible transforms, and this step chooses the right subset.

For each transform, we use the standard Spiral system [99] to search for the best implementation and generate a fixed size $\sum$-SPL formula.

For example, for $\mathbf{DFT}_2$ there is only a single implementation

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

Form the fixed size recursion step formulas. The results of the previous step are plugged into the formulas of (3.17). This yields:

$$\left\{ \begin{array}{c} \left[\begin{smallmatrix} 1 & 1 \\ 1 & -1 \end{smallmatrix}\right] \\ \dots \\ S(h^{2\to u_2}_{u_3,\,1})\left[\begin{smallmatrix} 1 & 1 \\ 1 & -1 \end{smallmatrix}\right] G(h^{2\to u_6}_{u_7,\,u_8}) \\ \dots \\ S(h^{2\to u_2}_{u_3,\,u_4})\left[\begin{smallmatrix} 1 & 1 \\ 1 & -1 \end{smallmatrix}\right] \operatorname{diag}\big(\operatorname{pre}({u_7}^{2\to\mathbb{C}})\big) G(h^{2\to u_9}_{u_{10},\,u_{11}}) \\ \dots \\ S(h^{2\to u_2}_{u_3,\,1})\left[\begin{smallmatrix} 1 & 1 \\ 1 & -1 \end{smallmatrix}\right] \operatorname{diag}\big(\operatorname{pre}({u_6}^{2\to\mathbb{C}})\big) G(h^{2\to u_8}_{u_9,\,u_{10}}) \\ \dots \end{array} \right\} \tag{3.18}$$

Note that these $\sum$-SPL implementations in (3.18) are recursion free, fixed size, but are still parametrized.

3.5 Representing Recursion: Descent Trees

We can conveniently visualize the recursive computation of transforms using the so-called *descent trees*. Besides being an indispensable visualization aid in the following sections, descent trees are useful for performance modeling, as briefly explained below.

The nodes of the descent trees are the recursion steps, and the edges connect to the child recursion steps as a result of the application of the descend operation.

An example of a descent tree for the computation of $\mathbf{DFT}_{32}$ using (3.1) is shown in Fig. 3.5. Every time a descend is applied to the recursion steps, several "children" recursion steps are needed, and thus we obtain a tree.

Since we consider the generation of general-size libraries, unlike ruletrees in standard Spiral, these descent trees are unfolded at runtime (instead of at code generation time).

A descent tree is similar to a *ruletree* used in Spiral and described in [99], but has several important differences, highlighted in Table 3.2

Perhaps the most important difference is that descent trees capture the context of the computation better than the ruletree. For example, the input and output strides are easily available in the descent tree, while they are difficult to obtain from the ruletree. The strides come from the $\sum$-SPL description of each recursion step, namely from the index mapping functions.

We use descent trees as a visual aid in explaining various breakdown rule combinations in the following sections.

Another useful application of descent trees is for performance modeling and machine learning applications. For example, the work by Singer [109, 110] attempts to constructs optimal ruletrees

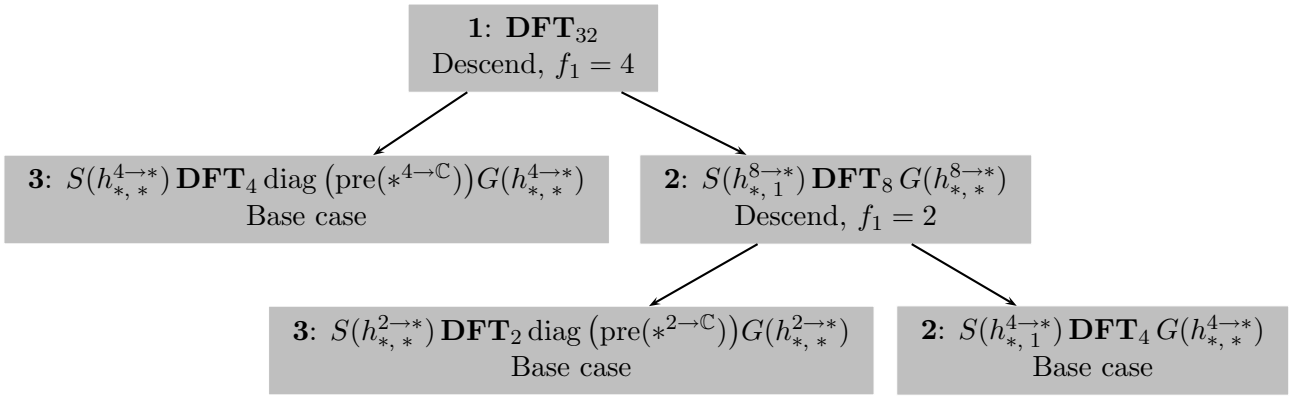


Figure 3.5: Descent tree for $\mathbf{DFT}_{32}$.

	Descent tree	**Ruletree**
Nodes	recursion steps	transforms
Edges	RDescend	breakdown rule application
Leaf size	any (user setting)	fixed by breakdown rules
Context	yes	no

Table 3.2: Comparison of descent trees and ruletrees.

using automated machine learning. Part of the problems raised in the papers is the need for more context information. The authors obtain this information by computing it directly from the ruletree, but this analysis has to explicitly "understand" different breakdown rules to compute this context. In other words, support for the breakdown rules had to be hard-coded in their infrastructure. Using the descent trees, on the other hand, this information is readily available.

3.6 Enabling Looped Recursion Steps: Index-Free $\sum$-SPL

All of our previous examples involve recursion steps with a single instance of a transform and additional "context" constructs, including gather, scatter and diagonals. In this section we discuss the *looped* recursion steps, which include iterative sums, and thus an even larger context. We will first explain why this is desirable, and then explain why this is not achievable with the standard $\sum$-SPL and parametrization.

To solve this problem, we develop an extension to $\sum$-SPL called the *index-free* $\sum$-SPL. It enables the parametrization and further manipulation of looped recursion steps, and was inspired by the concept of I/O tensors in FFTW 3.x [61]. They are used to represent "DFT problems", which are equivalent to recursion steps in our terminology. In fact, one can use index-free $\sum$-SPL as a rigorous mathematical description of the "DFT problems" in FFTW. The transformation from $\sum$-SPL to the index-free $\sum$-SPL is closely related to λ-lifting [71] used in compilers for programming languages that support higher-order functions.

3.6.1 Motivation

There are three motivations for supporting the looped recursion steps. First, they immediately enable the generation of looped base cases. In practical terms, this means that, for example, the application of the Cooley-Tukey FFT (3.1) breakdown rule to the DFT, instead of yielding a pair of smaller DFTs, yields a pair of *loops* over these smaller DFTs. In the former case, only the smaller DFTs can be implemented in a base case, and in the latter case, the base case can also contain these outer loops over the small DFTs. This considerably improves the performance of the generated code, since the target language compiler (C++ compiler in our case) can better optimize the function implementing the base case.

Second, they enable various loop optimizations by making it possible to create breakdown rules also for *loops*, rather than only for transforms. In Section 3.7 we describe a number of these optimizations. Further, we implement vectorization and parallelization (Chapter 4) in the same way.

Third, we believe that the combination of breakdown rules for loops and transforms provides a very elegant way to describe the implementation space.

However, as we show next, directly applying the parametrization to looped $\sum$-SPL expressions is not possible, and this leads us to the index-free $\sum$-SPL, which solves the problem.

Small example. To demonstrate the problems arising with parametrization of looped recursion steps, we show a small example. Consider, the parametrization of $\left\{ \mathbf{DFT}_n \otimes I_k \right\}$. First, we convert the expression to $\sum$-SPL to obtain:

$$\sum_{j=0}^{k-1} S(h_{j,\,k}^{n \to kn})\, \mathbf{DFT}_n\, G(h_{j,\,k}^{n \to kn}). \tag{3.19}$$

The index of the iterative sum j controls the base address of the index mapping functions. If we parametrize (3.19) by applying the parametrization algorithm from Section 3.3.3, first, all of the scalar expressions are replaced by parameters, to obtain

$$\sum_{j=0}^{u_{10}} S(h_{u_3,\,u_4}^{u_1 \to u_2})\, \mathbf{DFT}_{u_5}\, G(h_{u_8,\,u_9}^{u_6 \to u_7}). \tag{3.20}$$

Note that the loop bound is replaced by u_{10}. Next, constraints are applied, to yield $u_1 = u_5 = u_6$, and after back-substitution of u_5 and u_6, we get

$$\sum_{j=0}^{u_{10}} S(h_{u_3,\,u_4}^{u_1 \to u_2})\, \mathbf{DFT}_{u_1}\, G(h_{u_8,\,u_9}^{u_1 \to u_7}). \tag{3.21}$$

Unfortunately, (3.21) is not a valid parametrization, because all references to the index j have disappeared, and instead we obtained the parameters u_3 and u_8. One cannot have parameters in these slots, because the actual values are functions of the loop variable, and are computed within the recursion step itself.

3.6.2 Index-Free $\sum$-SPL: Basic Idea

Instead of trying to modify the parametrization algorithm explained before we designed a special form of $\sum$-SPL which does not use loop indices, and thus does not incur the problem described above.

The problem with loop indices is that they are, in some sense, a departure from a purely declarative language. The use of explicit loop indices establishes a link between the declaration of the index (i.e., in the iterative sum) and all uses of the index in various arithmetic expressions. When these expressions are replaced by the parameters, the link is lost. So our solution is to eliminate the loop indices, which in addition makes $\sum$-SPL purely declarative.

We designed the index-free $\sum$-SPL based on an important insight in FFTW 3 [61]. Namely, the use of the so-called *vector strides* to describe loops over transforms. For example, (3.19) becomes in index-free $\sum$-SPL

$$\sum_k S(h^{n\to kn}_{0,\,k,1})\,\mathbf{DFT}_n\,G(h^{n\to kn}_{0,\,k,1}). \tag{3.22}$$

In (3.22) the iterative sum does not provide a loop variable; instead, k denotes the number of iterations. The index mapping functions now give a pair of strides: the standard stride = k, and the vector stride = 1. They can be interpreted as 2-variable functions, the extra variable is the loop index. Compare the two functions below:

$$\begin{aligned} \text{Original: } & h^{n\to kn}_{j,\,k} : i \mapsto j + ki, \\ \text{Index-free: } & h^{n\to kn}_{0,\,k,1} : (j, i) \mapsto 0 + 1j + ki = j + ki. \end{aligned}$$

We call the 2-variable version of h a *ranked function*. The rank of h is 1, which means that it has 1 extra variable corresponding to 1 loop index.

Our transformation is equivalent to λ-lifting [71], where the goal is to implement nested functions. 1-variable nested functions are transformed into 2-variable global functions.

However, in addition to the pure λ-lifting we hide the extra arguments by using predefined ranked functions (like h), which makes the notation declarative.

3.6.3 Ranked Functions

In order to eliminate loop indices, we capture the inter-iteration behavior in an alternative way. First, we need to identify the objects that are affected by loop indices. In the examples, we see that the most important such objects are gathers, scatters, and diagonals parametrized by functions. Thus, the crucial step is to eliminate the loop index dependence of functions. By doing so we make a large class of $\sum$-SPL formulas index-free.

This is accomplished by using multivariate functions, as we have already shown above in (3.22). The loop index dependence is captured inside the function definition in a structured way. We call these special multivariate functions *ranked* functions. The rank of the function denotes the number of enclosing loops ($\sum$ operators) that affect it. A regular function (without loops) $f^{A\to B}$ has rank 0.

A rank-1 function must be in the scope of at least one loop, $\sum$. The loop index of the sum becomes an extra parameter. For example, below, g is a rank-1 function that lives inside a loop with n iterations:

$$g : \mathbb{I}_n \times A \to B; (j, i) \mapsto g(j, i).$$

$$h^{n\to N}_{b,\ s_0,s_1,\ldots,s_k} : \mathbb{Z}^k \times \mathbb{I}_n \to \mathbb{I}_N; (j_k,\ldots,j_1,i) \mapsto b + is_0 + \sum_{l=1\ldots k} s_k j_k, \tag{3.23}$$

$$z^{n\to N}_{b,\ s_0,s_1,\ldots,s_k} : \mathbb{Z}^k \times \mathbb{I}_n \to \mathbb{I}_N; (j_k,\ldots,j_1,i) \mapsto b + is_0 + \sum_{l=1\ldots k} s_k j_k \bmod N, \tag{3.24}$$

$$r^{n\to N}_{b,\ q,\ s_0,s_1,\ldots,s_k} : \mathbb{Z}^k \times \mathbb{I}_n \to \mathbb{I}_N; (j_k,\ldots,j_1,i) \mapsto \begin{cases} b + is_0 + \sum_{l=1\ldots k} s_k j_k, & i < \left\lfloor \frac{n}{2} \right\rfloor \\ q - b - is_0 - \sum_{l=1\ldots k} s_k j_k, & \text{else.} \end{cases} \tag{3.25}$$

Table 3.3: Rank-k index mapping functions.

The first parameter of the function above is the loop index. If we have a rank-0 function ${f_j}^{A\to B}$ parametrized by the loop index j, we remove the explicit reference to j by using a rank-1 function g above, as $f_j(i) = g(j,i)$. If the transformation is applied to all functions, the explicit loop index is no longer necessary. For example, we obtain

$$\sum_{j=0}^{n-1} S(f_j)AG(f_j) = \sum_{n} S(g)AG(g).$$

Since we don't need the loop index in the right-hand side above, we just write the number of iterations under the sum. We also assume that ranked functions can appear anywhere in the $\sum$-SPL syntax tree, and, that the loop indices (when they are introduced) are automatically bound to the right slots of ranked function parameters. Therefore, for all purposes ranked functions can be treated (e.g., by gather and scatter matrices) as the single argument functions that we had before.

In general, we don't need to keep track of the interval sizes (i.e., loop bounds) for loop index variable slots in each ranked function. After all, we do have the loops available in the formula. So for convenience, we relax the definition slightly, so that a rank-k function is defined as

$$g : \underbrace{\mathbb{Z} \times \cdots \times \mathbb{Z}}_{\mathbb{Z}^k} \times A \to B : (j_k,\ldots,j_1,i) \mapsto g(j_k,\ldots,j_1,i).$$

We order loop index function arguments in "outer-loop first" order. The outermost loop index j_k is first, and the innermost j_1 is last, before i, which is the regular function argument of a rank-0 function. In other words the subscript of j denotes how far away is the associated loop, with 1 being the closest (innermost).

In Table 3.3 we generalize three important index mapping functions to ranked functions. h and z both are defined in Section 2.4 and are used in the Cooley-Tukey (2.24) and the prime-factor FFT (2.25), respectively, r is used in RDFT and DCT algorithms.

3.6.4 Ranked Function Operators

Now we define several operators necessary to manipulate ranked functions. We start by defining an appropriate version of composition, and proceed with other operations specific to ranked functions.

Composition. We define the composition of ranked functions in a special way to be compatible with regular composition. For example, for a pair of (non-ranked) functions f_j and g_j that depend on the loop index $(f_j \circ g_j)(i) = f_j(g_j(i))$, and the index j does not affect the composition. Our goal is to define the ranked function composition in a similar way.

To accomplish this, we define the composition for ranked functions, such that the loop index arguments are not passed along through the composition chain. For example, for a pair of rank-1 functions, composition is defined by:

$$f^{\mathbb{Z}\times B\to C} \circ g^{\mathbb{Z}\times A\to B} : \mathbb{Z}\times A \to C; (j_1, i) \mapsto f(j_1, g(j_1, i)).$$

Composable functions do not have to be of the same rank. The rank of the result is equal to the maximal rank of the composed functions. Formally, given a rank-k function f and a rank-n function g, their composition is defined as:

$$f^{\mathbb{Z}^k\times B\to C} \circ g^{\mathbb{Z}^n\times A\to B} : \mathbb{Z}^r \times A \to C; (j_r, \ldots, j_1, i) \mapsto f(j_k, \ldots, j_1, g(j_n, \ldots, j_1, i)), \quad r = \max(k, n).$$

Downrank. Ranked functions are tied to the loops (iterative sums) in their immediate formula context. If the function needs to be taken out of context, the implicit loop dependencies must be made explicit by introducing the explicit loop variable, and hence decreasing the rank of the function. We call this operation *downranking*. We say that a function is fully downranked if its rank is reduced to 0. Fully downranking a function is equivalent to converting a formula from index-free $\sum$-SPL to regular $\sum$-SPL. Downranking is equivalent to function currying [120]. Formally, for a rank-1 function f, the downrank operation is defined as follows:

$$\text{down}(f^{\mathbb{Z}\times A\to B}, j, 1) : A \to B; i \mapsto f(j, i), \quad \text{rank}(f) = 1.$$

The parameter 1 specifies that we are downranking the first loop (the only choice). Thus the following identity holds:

$$\sum_n S(f)AG(f) = \sum_{j=0}^{n-1} S(\text{down}(f, j, 1))AG(\text{down}(f, j, 1)).$$

For a general rank-k function we can downrank with respect to any of the k loops, specified by l:

$$\text{down}(f^{\mathbb{Z}^k\times A\to B}, j, l) : \mathbb{Z}^{k-1} \times A \to B;$$
$$(j_k, \ldots, j_{l-1}, j_{l+1}, \ldots, j_1, i) \mapsto f(j_k, \ldots, j_{l-1}, j, j_{l+1}, \ldots, j_1, i), \quad \text{rank}(f) = k.$$

Uprank. Another operator we will need is called *uprank*. It is not the inverse of downrank. Before we define uprank, we motivate the need for it. Consider the following restatement of rewrite rule (2.44), which becomes invalid in the presence of ranked functions:

$$\left(\sum A\right) M \to \left(\sum_n AM\right).$$

The rule is invalid, when applied as follows and as we explain next:

$$\left(\sum S(f)\left(\sum A\right)G(f)\right) \to \left(\sum\sum S(f)AG(f)\right), \quad \text{rank}(f)=1. \tag{3.26}$$

The left-hand and right-hand sides of (3.26) are not equal. This can be shown by downranking f on both sides (and thus introducing explicit loop indices):

$$\left(\sum S(f)\left(\sum A\right)G(f)\right) \xrightarrow{\text{downrank}} \left(\sum_j S(\text{down}(f,j,1))\left(\sum_k A\right)G(\text{down}(f,j,1))\right) \tag{3.27}$$

$$\left(\sum\sum S(f)AG(f)\right) \xrightarrow{\text{downrank}} \left(\sum_j\sum_k S(\text{down}(f,k,1))AG(\text{down}(f,k,1))\right) \tag{3.28}$$

Above, (3.27) is correct, and (3.28) is invalid, because f is downranked with k (instead of j) as a loop variable. Originally, f is a rank-1 function tied to its immediate parent loop, which happens to be the outer loop over j. If a rewrite rule is applied, the context changes, and the immediate parent loop of f becomes the inner loop over k. This leads to the invalid result in (3.28).

To fix this problem, we need to increase the rank of f to 2, when applying the rewrite rule (2.44). This is accomplished by the uprank operator, defined below for a rank-1 f.

$$\text{up}(f^{\mathbb{Z}\times A\to B}) : \mathbb{Z}^2 \times A \to B; (j_2, j_1, i) \mapsto f(j_2, i), \quad \text{rank}(f)=1.$$

Since there is no dependence on the new inner loop, the corresponding loop index j_1 is simply ignored. The correct way to transform (3.26) uses upranking as below:

$$\left(\sum S(f)\left(\sum A\right)G(f)\right) \to \left(\sum\sum S(\text{up}(f))AG(\text{up}(f))\right). \tag{3.29}$$

The above is correct for an f of arbitrary rank. As a special case, when $\text{rank}(f)=0$, there is no need to uprank, as reflected in the original rule (2.44).

In the general case of a rank-k function f, upranking is defined as

$$\text{up}(f^{\mathbb{Z}^k\times A\to B}) : \mathbb{Z}^{k+1} \times A \to B; (j_{k+1}, \ldots, j_1, i) \mapsto f(j_{k+1}, \ldots, j_2, i), \quad \text{rank}(f)=k.$$

3.6.5 General λ-Lifting

Using only ranked functions we can convert a very large class of $\sum$-SPL formulas to an index-free representation. The formulas that contain references to loop indices outside of functions must be handled using the general λ-lifting scheme, explained in this section.

Consider the following SPL formula fragment from the RDFT breakdown rule (2.6):

$$\bigoplus_{j=0}^{k/2-2} \mathbf{rDFT}_{2m}((j+1)/k). \tag{3.30}$$

Above, $(j+1)/k$ is a parameter of the auxiliary transform **rDFT**. It is not possible to convert it to an index-free $\sum$-SPL expression using the rewrite rules from Table 3.4, because the reference to

the loop index inside the **rDFT** parameter will stay.

The first step is to represent a scalar expression with j by a dummy single-argument function, to make it consistent with other functions like h. We obtain:

$$\bigoplus_{j=0}^{k/2-2} \mathbf{rDFT}_{2m}(\lambda_j(0)), \quad \lambda_j : i \mapsto (j+1)/k$$

$$\lambda_j : \mathbb{I}_1 \to \mathbb{R}$$

Above, λ_j represents the original expression. Since it is a function we have to apply it to an argument, and the only possible argument is $0 \in \mathbb{I}_1$.

Next, we introduce a dummy function evaluation operator λ-wrap. The intuitive meaning of λ-wrap is "treat this function as a scalar":

$$\lambda\text{-wrap}(f^{1\to A}) = f(0).$$

To convert a function to a scalar, we evaluate it at 0. The result of λ-wrap is a scalar, and it enables to use single-argument functions where scalars are expected. Using λ-wrap, we rewrite (3.30) as

$$\bigoplus_{j=0}^{k/2-2} \mathbf{rDFT}_{2m}(\lambda\text{-wrap}(\lambda_j)).$$

Finally, we convert λ_j to a rank-1 function λ, and eliminate j from the iterative sum, leaving only the number of iterations:

$$\bigoplus_{k/2-1} \mathbf{rDFT}_{2m}(\lambda\text{-wrap}(\lambda)), \quad \lambda : (j,i) \mapsto (j+1)/k.$$

Even though the above is not $\sum$-SPL (the direct sum operator is not converted into $\sum$ with gather and scatter), we show its parametrization to clarify what happens with λ without cluttering with unnecessary details:

$$\bigoplus_{k/2-1} \mathbf{rDFT}_{2m}(\lambda\text{-wrap}(\lambda)) \xrightarrow{\text{par}} \bigoplus_{u_1} \mathbf{rDFT}_{u_2}(\lambda\text{-wrap}({u_3}^{\mathbb{Z}\times 1\to\mathbb{R}})).$$

Basically, the above is parametrized by substituting a parameter function u_3 for λ.

3.6.6 Library Generation with Index-Free $\sum$-SPL

The library generation can use the index-free $\sum$-SPL instead of the regular $\sum$-SPL without any major changes. However, the index-free representation makes it possible to expand the scope of the recursion step tag to include loops, and this will no longer cause problems with parametrization.

Generating code for index-free formulas can be done using the standard $\sum$-SPL code generation rules from Table 2.5, if the index-free formula is converted to a regular one by downranking all index-free functions.

All other formula manipulations, including parametrization, descending and other library related manipulations, work with index-free formulas.

$$(I_k \otimes A_n) \to \sum_k S(h_{0,\,1,n}) A_n G(h_{0,\,1,n}), \tag{3.31}$$

$$(A_n \otimes I_k) \to \sum_k S(h_{0,\,k,1}) A_n G(h_{0,\,k,1}), \tag{3.32}$$

$$(I_m \otimes A_n \otimes I_k) \to \sum_m \sum_k S(h_{0,\,k,1,nk}) A_n G(h_{0,\,k,1,nk}), \text{ or} \tag{3.33}$$

$$\to \sum_k \sum_m S(h_{0,\,k,nk,1}) A_n G(h_{0,\,k,nk,1}) \tag{3.34}$$

Table 3.4: Rewrite rules for converting SPL to ranked $\sum$-SPL.

$$\left(\sum A\right) B(f) \to \left(\sum AB(\mathrm{up}(f))\right), \quad B \in \{G, \mathrm{diag}\}, \tag{3.35}$$

$$B(f) \left(\sum A\right) \to \left(\sum B(\mathrm{up}(f))A\right), \quad B \in \{S, \mathrm{diag}\}, \tag{3.36}$$

$$\mathrm{up}(f \circ g) \to \mathrm{up}(f) \circ \mathrm{up}(g), \tag{3.37}$$

$$\mathrm{down}(f \circ g, j, l) \to \mathrm{down}(f, j, l) \circ \mathrm{down}(g, j, l), \tag{3.38}$$

$$\mathrm{up}(h_{b,\,s,s_1,\ldots}) \to h_{b,\,s,0,s_1,\ldots}, \tag{3.39}$$

$$\mathrm{up}(z_{b,\,s,s_1,\ldots}) \to z_{b,\,s,0,s_1,\ldots}, \tag{3.40}$$

$$\mathrm{up}(r_{b,\,s,s_1,\ldots,\,}) \to r_{b,\,q,\,s,0,s_1,\ldots}, \tag{3.41}$$

$$\mathrm{down}(h_{b,\,s,s_1,\ldots}, j, l) \to h_{b+s_l j,\,s,s_1,\ldots,s_{l-1},s_l,\ldots}, \tag{3.42}$$

$$\mathrm{down}(z_{b,\,s,s_1,\ldots}, j, l) \to z_{b+s_l j,\,s,s_1,\ldots,s_{l-1},s_l,\ldots}, \tag{3.43}$$

$$\mathrm{down}(r_{b,\,q,\,s,s_1,\ldots}, j, l) \to r_{b+s_l j,\,q,\,s,s_1,\ldots,s_{l-1},s_l,\ldots}, \tag{3.44}$$

$$h_{b,\,s,s_1,\ldots} \circ h_{b',\,t,t_1,\ldots} \to h_{b+sb',\,st,s_1+st_1,s_2+st_2,\ldots}, \quad \text{use } s_i, t_i = 0, \text{ for } i > \mathrm{rank}(\cdot) \tag{3.45}$$

$$z_{b,\,s,s_1,\ldots} \circ h_{b',\,t,t_1,\ldots} \to z_{b+sb',\,st,s_1+st_1,s_2+st_2,\ldots}, \quad \text{use } s_i, t_i = 0, \text{ for } i > \mathrm{rank}(\cdot) \tag{3.46}$$

Table 3.5: Rewrite rules for ranked functions.

Table 3.4 gives the rules for conversion from SPL to ranked $\sum$-SPL; this table is an index-free equivalent of Table 2.6 in Section 2.4.

As we explained in the previous section, some of the rewrite rules need to be modified to correctly support ranked functions. Table 3.5 summarizes these updated rewrite rules for ranked functions, and also gives additional rules for upranking and downranking the common index mapping functions from Table 3.3.

Our library generator does not use plain index-free $\sum$-SPL to generate libraries. Instead, we extend this concept even further, by introducing a special *loop non-terminal*, or *loop transform*, which allows us to treat every loop as a transform and apply breakdown rules to it. This allows to introduce degrees of freedom in loop manipulations, and open up an additional search space for

optimization. This is the topic of the next section.

3.7 Advanced Loop Transformations

In this section we explain how the index-free loop representation can be used to implement more complex loop transformations, namely, loop interchange, loop distribution, and strip mining. These transformations are useful as memory hierarchy optimizations, but are also essential for vectorization and parallelization.

These and many other loop transformations, with certain limitations, can be performed by modern general purpose optimizing compilers. However, one of the unsolved challenges in a general purpose compiler is in deciding when and with what parameters to apply these transformations.

To overcome this problem, we can reuse the feedback-driven search to guide the optimization process. In order to use the Spiral search machinery, we need to define a special non-terminal (called "GT") that represents a single or multiple nested loops, and also define breakdown rules for it.

As the index-free $\sum$-SPL, our loop optimization breakdown rules were inspired by the new design of FFTW 3.x [61], which provides special solvers for "vectors" of transforms (i.e. loops over transforms). Our aim is to formalize the essense of these optimizations on the $\sum$-SPL level, and thus enable generation of FFTW 3.x like libraries for a wide variety of transforms.

We proceed by explaining the loop non-terminal GT Section 3.7.1 and explain how several important loop transformations can be captured using GT breakdown rules. We discuss loop interchange in Section 3.7.2, loop distribution in Section 3.7.3, and strip-mining in Section 3.7.4.

3.7.1 GT: The Loop Non-Terminal

The standard pattern of most loops in $\sum$-SPL is an iterative sum with gather and scatter, so for a single loop we define the rank-1 GT as

$$\mathrm{GT}(A,\ g,\ s,\ \{v\}) = \sum_v S(s)AG(g), \quad \mathrm{rank}(g) = \mathrm{rank}(s) = 1.$$

We capture the case of multiple nested loops in the same construct using functions of higher rank. The rank-k GT is given by

$$\mathrm{GT}(A,\ g,\ s,\ \{v_1, \ldots, v_k\}) = \sum_{v_k} \cdots \sum_{v_1} S(s)AG(g), \quad \mathrm{rank}(g) = \mathrm{rank}(s) = k.$$

Above, v_1 corresponds to the number of iterations in the innermost loop, and v_k in the outermost. The ordering of the v_i's corresponds to the ordering of the vector strides in ranked functions.

It is easy to see how familiar SPL constructs map to the GT non-terminal, by comparing GT against the index-free $\sum$-SPL notation. We show several examples in Table 3.6.

In the case of the double tensor product, the representation is not unique, because the two loops are interchangeable, and thus the loop order can be chosen arbitrarily. For a rank-k GT, the number of possible loop orderings is $k!$.

Example: Descent tree for the $\mathbf{DFT}_{32}$ based on GT. Previously, we showed a descent tree for the $\mathbf{DFT}_{32}$ implementation using standard $\sum$-SPL in Fig. 3.5. The former descent tree

SPL	Index-free $\sum$-SPL	GT
$(I_k \otimes A_n)$	$\sum_k S(h_{0,\,1,n}) A_n G(h_{0,\,1,n})$	$\mathrm{GT}(A,\ h_{0,\,1,n},\ h_{0,\,1,n},\ \{k\})$
$(A_n \otimes I_k)$	$\sum_k S(h_{0,\,1,n}) A_n G(h_{0,\,1,n})$	$\mathrm{GT}(A,\ h_{0,\,k,1},\ h_{0,\,k,1},\ \{k\})$
$(I_m \otimes A_n \otimes I_k)$	$\sum_m \sum_k S(h_{0,\,k,1,nk}) A_n G(h_{0,\,k,1,nk})$	$\mathrm{GT}(A,\ h_{0,\,k,1,nk},\ h_{0,\,k,1,nk},\ \{k,m\})$
	$\sum_k \sum_m S(h_{0,\,k,nk,1}) A_n G(h_{0,\,k,nk,1})$	$\mathrm{GT}(A,\ h_{0,\,k,nk,1},\ h_{0,\,k,nk,1},\ \{m,k\})$

Table 3.6: Converting SPL to ranked $\sum$-SPL and GT.

has nodes that are smaller **DFT**s decorated with extra constructs. Treating loops as non-terminals makes the tree larger, since it adds intermediate nodes.

In particular, the single descend step into $\mathbf{DFT}_{32}$ using the Cooley-Tukey rule (3.1) produces a pair of GT recursion steps (i.e., a pair of loops), instead of decorated DFTs.

$$\begin{aligned}\left\{\,\mathbf{DFT}_n\,\right\} &= (\left\{\,\mathbf{DFT}_k\,\right\} \otimes I_m)\,\mathrm{diag}(d)(I_k \otimes \left\{\,\mathbf{DFT}_m\,\right\})L^n_k \\ &= \mathrm{GT}(\mathbf{DFT}_k\,\mathrm{diag}(d \circ h_{0,\,m,1}),\ h_{0,\,m,1},\ h_{0,\,m,1},\ \{m\}) \cdot \mathrm{GT}(\mathbf{DFT}_m,\ h_{0,\,k,1},\ h_{0,\,1,m},\ \{k\}).\end{aligned}$$

To further deal with the new non-terminals we need breakdown rules. To obtain the equivalent of the previous descent tree, we define a downrank breakdown rule, which eliminates GT by producing a regular (non-index-free) $\sum$. For rank-1 GT it is defined as follows:

$$\mathrm{GT}(A,\ g,\ s,\ \{v\}) \xrightarrow[f]{\text{GT-DownRank}} \sum_{j=0}^{v-1} S(\mathrm{down}(s,1,j))\,\mathrm{down}(A,1,j)G(\mathrm{down}(g,1,j)). \tag{3.47}$$

The new descent tree interleaves the Cooley-Tukey (DFT-CTA) and the downrank (GT-DownRank) breakdown rules and is shown in Fig. 3.6.

.

3.7.2 Loop Interchange

The goal of loop interchange optimization is to change the order of loops to optimize for memory locality [134, 139].

Table 3.8 shows an example of loop interchange for a trivial program. In the example, interchanging the loops leads to better memory locality.

Original code	After loop interchange

`for(int i=0; i<100; ++i) {`
`  for(int j=0; j<15; ++j) {`
`  y[100*j + i] = x[100*j + i];`
`  }`
`}`
	`for(int j=0; j<15; ++j) {`
	`  for(int i=0; i<100; ++i) {`
	`  y[100*j + i] = x[100*j + i];`
	`  }`
	`}`

Table 3.7: Loop interchange example.

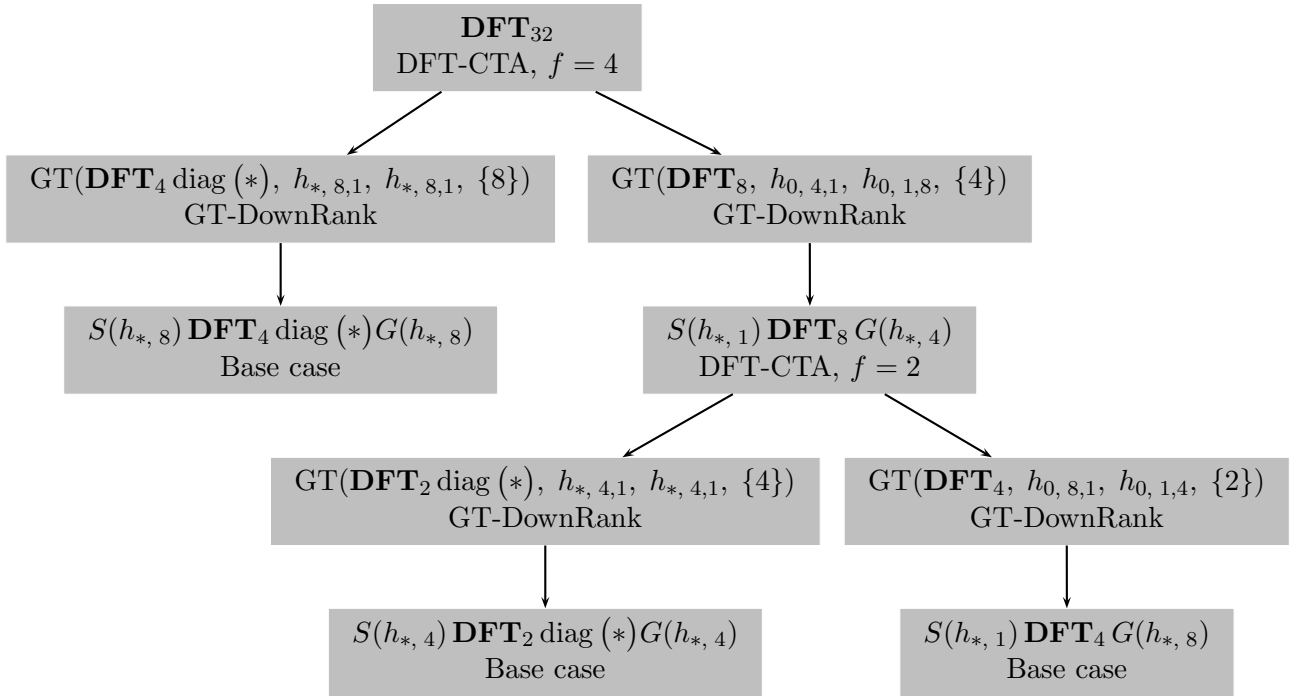


Figure 3.6: Descent tree for $\mathbf{DFT}_{32}$.

Implementation in Spiral. Given a loop nest expressed as rank-k GT, we introduce a degree of freedom $1 \leq f \leq k$ in the downrank breakdown rule, which selects the loop to downrank next. The loop that is downranked first becomes the outer loop. The revised breakdown rule is

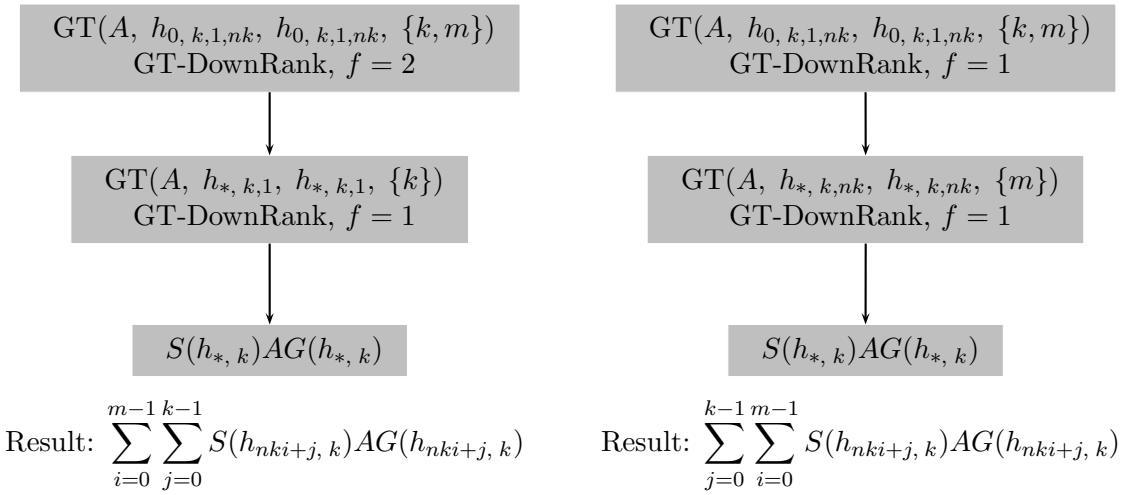
$$\mathrm{GT}(A,\ g,\ s,\ \{v_1,\ldots,v_k\}) \xrightarrow[f]{\text{GT-DownRank}} \sum_{j=0}^{v_f-1} \mathrm{down}(\mathrm{GT}(A,\ g,\ s,\ \{v_1,\ldots,v_k\}), f, j). \tag{3.48}$$

The actual downrank operation on the GT is defined as

$$\begin{aligned}&\mathrm{down}(\mathrm{GT}(A,\ g,\ s,\ \{v_1,\ldots,v_k\}), f, j)\\ &\qquad = \mathrm{GT}(\mathrm{down}(A, f, j),\ \mathrm{down}(g, f, j),\ \mathrm{down}(s, f, j),\ \{v_1,\ldots,v_{f-1},v_{f+1},\ldots,v_k\}).\end{aligned}$$

The degrees of freedom within the breakdown rule are exposed to the search engine. For example, for the rank-2 GT, two descent trees are possible, as illustrated in Fig. 3.7.

The final $\sum$-SPL expressions that result from descending differently are almost identical, except that the loops (iterative sums) are reordered, as expected.



(a) Outer loop first

(b) Inner loop first

Figure 3.7: Descent trees corresponding to different downranking orderings of GT non-terminal associated with $I_m \otimes A \otimes I_k$.

3.7.3 Loop Distribution

Loop distribution splits a single loop into two consecutive loops, each executing a half of the loop body. It can potentially improve cache utilization, by using cache lines more effectively [72].

Table 3.8 shows an example of loop distribution for a trivial program.

Original code	After loop distribution

`for(int i=0; i<15; ++i) {`
`  y[i]  = x[i];`
`  y[100+i] = x[100+i];`
`}`
	`for(int i=0; i<15; ++i) {`
	`  y[i]  = x[i];`
	`}`
	`for(int i=0; i<15; ++i) {`
	`  y[100+i] = x[100+i];`
	`}`

Table 3.8: Loop distribution example.

The transformation shown in Table 3.8 could speedup the code if the target machine had a cache that could hold a single cache line, consisting of at least 2 data elements. The original code loads $x[i]$, but since the entire cache line is loaded, $x[i+1]$ is loaded also. However, due to limited capacity, $x[i+1]$ is evicted by loading $x[100+i]$. The transformed code uses entire cache lines, avoiding the unnecessary eviction.

Mathematical preliminaries. Loop distribution optimization is enabled by the distributivity property of the tensor product, shown below:

$$(I \otimes AB) = (I \otimes A)(I \otimes B).$$

The above can be generalized to allow arbitrary input, output and intermediate data formats, given

by the permutations P, Q and R:

$$Q(I \otimes AB)P = Q(I \otimes A)R^{-1} \cdot R(I \otimes B)P.$$

The two important special cases are $R = P^{-1}$ and $R = Q$, which means that one of the stages can be performed inplace:

$$Q(I \otimes AB)P = Q(I \otimes A)P \cdot \underbrace{P^{-1}(I \otimes B)P}_{\text{inplace}},$$

or

$$Q(I \otimes AB)P = \underbrace{Q(I \otimes A)Q^{-1}}_{\text{inplace}} \cdot Q(I \otimes B)P,$$

This tensor product property translates to the following GT property:

$$\mathrm{GT}(A \cdot B,\ g,\ s,\ \{v\}) = \mathrm{GT}(A,\ f,\ s,\ \{v\}) \cdot \mathrm{GT}(B,\ g,\ f,\ \{v\}). \tag{3.49}$$

Above, the function f describes the intermediate data format. The two special cases are $f = g$ ($\mathrm{GT}(B,\ g,\ f,\ \{v\})$ can be done inplace), and $f = s$ ($\mathrm{GT}(A,\ f,\ s,\ \{v\})$ can be done inplace).

This property holds for arbitrary rank GTs.

Implementation in Spiral. To be able to use (3.49) as a breakdown rule in Spiral, there must exist descent trees with nodes of the form $\mathrm{GT}(A \cdot B,\ g,\ s,\ \{v\})$. However, even the GT-based descent trees do not contain such nodes. See, for instance, Figure 3.6.

In order to obtain the desired nodes we add an extra degree of freedom for descending into GT nodes. The original way of descending is to apply a GT breakdown rule, such as GT-DownRank. The extra degree of freedom also allows descending into the *child* of the GT. In the case of the DFT, it would like as follows:

$$\begin{aligned}\mathrm{GT}(\mathbf{DFT},\ g,\ s,\ \{v\}) &\xrightarrow{\text{DFT-CT}} \mathrm{GT}((\mathbf{DFT} \otimes I)\,\mathrm{diag}(I \otimes \mathbf{DFT})L,\ g,\ s,\ \{v\}) \\ &= \mathrm{GT}(\mathrm{GT}(\mathbf{DFT}\,\mathrm{diag},\ h,\ h,\ \{\cdot\}) \cdot \mathrm{GT}(\mathbf{DFT},\ h,\ h,\ \{\cdot\}),\ g,\ s,\ \{v\}).\end{aligned}$$

Above, after applying the Cooley-Tukey breakdown rule, we can downrank the outer GT. However, this is no different than performing the downrank operation first, and applying the Cooley-Tukey breakdown rule afterward. If the two steps are done in reverse order, the descent tree is slightly different, but the result is the same as in Figure 3.6.

Therefore the only motivation to apply the breakdown (Cooley-Tukey rule) to the child of the GT first (instead of the GT itself) is to obtain the structure suitable for loop distribution. With this in mind, we made the loop distribution rule (3.49) into a rewrite rule, applied unconditionally. The decision to perform loop distribution lies in the fact that we descended into the child of the GT, instead of descending into the GT itself. The more flexible option is to make it into a breakdown rule, due to the degree of freedom in the choice of f.

The final loop distribution breakdown (or rewrite, depending on implementation) rule is stated below:

$$\mathrm{GT}(\mathrm{GT}_A \cdot \mathrm{GT}_B,\ g,\ s,\ \{\boldsymbol{v}\}) \xrightarrow{\text{GT-Distr}} \mathrm{GT}(\mathrm{GT}_A,\ f,\ s,\ \{\boldsymbol{v}\}) \cdot \mathrm{GT}(\mathrm{GT}_B,\ g,\ f,\ \{\boldsymbol{v}\}), \quad f \in \{s, g\}.$$

To obtain a rewrite rule we must decide a priori the strategy for choosing f.

3.7.4 Strip-Mining

Both parallelization and vectorization of loops rely on a transformation called *strip-mining* [134, 139]. Strip-mining transforms a loop into a doubly nested loop, where the inner loop performs a "strip" of the iterations of the original loop. Because this transformation is not beneficial by itself, we only use specialized variants tailored for parallelization and vectorization, as explained in Chapter 4. However, for consistency, we explain strip-mining here.

Strip-mining is a degenerate form of loop tiling. Table 3.9 shows an example of strip-mining.

Original code	After strip-mining

`for(int i=0; i<16; ++i) {`
`  y[i] = x[i];`
`}`
	`for(int i=0; i<4; ++i) {`
	`  for(int j=0; j<4; ++j) {`
	`  y[4*i+j] = x[4*i+j];`
	`  }`
	`}`

Table 3.9: Strip-mining example.

Mathematical preliminaries. The underlying tensor product property is

$$(I_{mn} \otimes A_k) = (I_m \otimes I_n \otimes A_k),$$

and with the most general dataflow pattern (given by permutations P and Q) it is

$$P(I_{mn} \otimes A_k)Q = P(I_m \otimes I_n \otimes A_k)Q,$$

Converting the above to the GT representation yields

$$\mathrm{GT}(A_k,\ q \circ h_{0,\ 1,k},\ p \circ h_{0,\ 1,k},\ \{mn\}) = \mathrm{GT}(A_k,\ q \circ h_{0,\ 1,k,km},\ p \circ h_{0,\ 1,k,km},\ \{m, n\}).$$

Formulating the transformation in the general case, i.e., with general index mapping functions f and g becomes non-trivial because the functions' must be split across the partitioned iteration space. To solve this problem, we introduce a rank splitting operator for functions.

$$\begin{aligned}
\mathrm{split}(h_{0,\ 1,k}, 1, m) &= h_{0,\ 1,k,km}, \\
\mathrm{split}(q \circ h_{0,\ 1,k}, 1, m) &= q \circ h_{0,\ 1,k,km}, \\
\mathrm{split}((j_1, i) \mapsto f(j_1, i), 1, m) &= (j_2, j_1, i) \mapsto f(mj_2 + j_1, i).
\end{aligned}$$

The splitting operator above increases the rank of the function through the implied splitting of the original loop into two nested loops with m inner iterations. The second parameter of split denotes which loop is being split. For rank-1 functions the only possible value is 1.

Splitting a rank-k function f yields a rank-$(k+1)$ function, as shown below:

$$\begin{aligned}
\mathrm{split}(f^{A \to B}, t, m) &: \mathbb{Z}^{k+1} \times A \to B \\
&: (j_k, \ldots, j_{t+1}, j'_t, j''_t, j_{t-1}, \ldots, j_1, i) \mapsto f(j_k, \ldots, j_{t+1}, mj'_t + j''_t, j_{t-1}, \ldots, j_1, i).
\end{aligned}$$

Using the split operator we can formulate the general strip-mining transformation for GT (which

we also call "split") as below:

$$\mathrm{GT}(A,\ g,\ s,\ \{mn\}) \xrightarrow{\mathrm{GT-Split}} \mathrm{GT}(\mathrm{split}(A,1,m),\ \mathrm{split}(g,1,m),\ \mathrm{split}(s,1,m),\ \{m,n\}) \tag{3.50}$$

Implementation in Spiral. Strip-mining is only required by vectorization and parallelization, thus we only implement specialized variants, as described in Chapter 4.

3.8 Inplaceness

Mathematical preliminaries. The fundamental inplace loop originates from the tensor product $I_n \otimes A_k$, where A_k is a square $k \times k$ matrix. Suppose we want to perform the inplace computation:

$$x \leftarrow (I_n \otimes A_k)x,$$
$$\begin{bmatrix} x_0 \\ \vdots \\ x_{kn-1} \end{bmatrix} \leftarrow \begin{bmatrix} A_k & & \\ & \ddots & \\ & & A_k \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ \vdots \\ x_{kn-1} \end{bmatrix}.$$

The goal is to perform n matrix vector products of the (square) A_k with different sections of x, and locally overwrite the corresponding sections with the results. Thus we can perform the distinct matrix vector products independently.

To express this property formally, we introduce the inplace tag $\mathrm{Inplace}(\cdot)$, which denotes that the formula must be transformed, such that input and output vectors can be safely aliased. The above observation can now be formally restated in the formula language as

$$\mathrm{Inplace}(I_n \otimes A_k) = \big(I_n \otimes \mathrm{Inplace}(A_k)\big).$$

The above can be generalized to handle an arbitrary data layout given by a permutation P, as

$$\mathrm{Inplace}(P^{-1}(I_n \otimes A_k)P) = P^{-1}(I_n \otimes \mathrm{Inplace}(A_k))P. \tag{3.51}$$

Note that the initial P and final P^{-1} express the fact that input and output array are permuted equally. Above the tensor product representation already becomes ambiguous and inadequate, since in the right-hand side of (3.51) P is outside the tensor product, and does not seem related to any inplaceness property at all.

In the GT domain, the equivalent of (3.51) takes a particularly clean and unambiguous form:

$$\mathrm{Inplace}(\mathrm{GT}(A_k,\ f,\ f,\ \{n\})) = \mathrm{GT}(\mathrm{Inplace}(A_k),\ f,\ f,\ \{n\}). \tag{3.52}$$

In other words, the data must be read and written using the same index mapping function f. If $P = \mathrm{perm}(p)$, then $f = p \circ h_{0,\ 1k}$. We will call (3.52) an inplace GT.

Inplace GT. Due to our simple parametrization algorithm, an inplace GT as in (3.52) is "parametrized away" after parametrization. Consider the following example:

$$\mathrm{GT}(A,\ h^{k\to km}_{0,\ 1,k},\ h^{k\to km}_{0,\ 1,k},\ \{n\}) \xrightarrow{\mathrm{Par}} \mathrm{GT}(A,\ h^{u_1\to u_2}_{0,\ 1,u_3},\ h^{u_1\to u_4}_{0,\ 1,u_5},\ \{u_6\}) \tag{3.53}$$

In the right-hand side of the above, $u_2 \neq u_4$ and $u_3 \neq u_5$, and thus it is no longer an inplace GT.

To solve this problem, we could introduce additional constraints on the index mapping functions. However, the root of the problem is in the redundant representation of an inplace GT, namely the index mapping function is stored twice, as gather and scatter function. Thus, a cleaner and more efficient solution is to introduce a special non-redundant notation for an inplace GT.

Below we will denote $\boldsymbol{v} = \{v_1, \ldots, v_m\}$ in rank-m GTs for brevity. For square A, a rank-m inplace GT is represented as

$$\mathrm{GTI}(A,\ f,\ \boldsymbol{v}) = \sum_{v_m} \cdots \sum_{v_1} S(f) A G(f), \quad |\boldsymbol{v}| = m.$$

Using GTI, the parametrization that did not work as expected in (3.53) now takes the form:

$$\mathrm{GTI}(A,\ h^{k \to kn}_{0,\ 1,k},\ \{n\})) \xrightarrow{\mathrm{Par}} \mathrm{GTI}(A,\ h^{u_1 \to u_2}_{0,\ 1,u_3},\ \{u_4\})$$

Not only the inplaceness property is preserved in the parametrized form, but also the parametrized formula requires less parameters, which means that we can expect less overhead in the implementation (e.g., less register pressure).

Rewriting with GTI. To be able to rewrite complicated $\sum$-SPL expressions with GTI, we need equivalents of the rules in Table 2.7. These equivalents take a somewhat unexpected form, and were first formulated by Franz Franchetti [55].

The origin of the rules is the simple tensor product transformation

$$P(I \otimes A) \to \big(P(I \otimes A)P^{-1}\big)P \to \mathrm{Inplace}(P(I \otimes A)P^{-1})P. \tag{3.54}$$

With one-stage formulas, there is no clear advantage. However, with two-stage formulas as in many DFT and other trigonometric transform algorithms, moving P to the right-hand side of the second stage allows it to be fused with the first stage. The most dramatic effect is achieved by applying (3.54) to the prime-factor FFT (2.2). In the original form (2.2) has two out-of-place stages, and after applying (3.54) both stages become inplace:

$$\begin{aligned}
\mathbf{DFT}_n &\to V^{-1}_{m,k}(\mathbf{DFT}_k \otimes I_m)(I_k \otimes \mathbf{DFT}_m)V_{m,k} \\
&\to V^{-1}_{m,k}(\mathbf{DFT}_k \otimes I_m)V_{m,k}V^{-1}_{m,k}(I_k \otimes \mathbf{DFT}_m)V_{m,k} \\
&\to \mathrm{Inplace}(V^{-1}_{m,k}(\mathbf{DFT}_k \otimes I_m)V_{m,k}) \cdot \mathrm{Inplace}(V^{-1}_{m,k}(I_k \otimes \mathbf{DFT}_m)V_{m,k}).
\end{aligned}$$

This transformation is extensively used in [121, 122] to manipulate different FFT algorithms.

Converting (3.54) into the GT representation yields

$$\mathrm{perm}(p) \cdot \mathrm{GTI}(A,\ f,\ \boldsymbol{v}) \to \mathrm{GTI}(A,\ p^{-1} \circ f,\ \boldsymbol{v}) \cdot \mathrm{perm}(p). \tag{3.55}$$

Obviously, there exists a dual rule (obtained by transposing (3.55)):

$$\mathrm{GTI}(A,\ f,\ \boldsymbol{v}) \cdot \mathrm{perm}(p) \to \mathrm{perm}(p) \cdot \mathrm{GTI}(A,\ p \circ f,\ \boldsymbol{v}). \tag{3.56}$$

Clearly, its impossible to have both (3.55) and (3.56) as rewrite rules, because rewriting would not terminate. Choosing only one of these rules, on the other hand, breaks symmetry, and disallows transposing breakdown rules, a standard mechanism in Spiral. Our solution is to include neither

of the rules, but use the alternative directional rules below:

$$S(p) \cdot \mathrm{GTI}(A,\ f,\ \boldsymbol{v}) \rightarrow \mathrm{GTI}(A,\ p \circ f,\ \boldsymbol{v}) \cdot S(p), \tag{3.57}$$
$$\mathrm{GTI}(A,\ f,\ \boldsymbol{v}) \cdot G(p) \rightarrow G(p) \cdot \mathrm{GTI}(A,\ p \circ f,\ \boldsymbol{v}). \tag{3.58}$$

These rules are somewhat counter intuitive, but they are a generalization of (3.55), and trivially follow from the fact that $G(p)S(p) = I_n,\ n = \mathrm{domain}(p)$ and GTI is necessarily a square matrix.

Even though neither scatter nor gather constructs are eliminated by (3.57) and (3.58), they are propagated and can be fused with the subsequent formula stage, as sketched in the following example.

Example 2 (Descending into the recursion step (3.10) *with Cooley-Tukey FFT* (3.1) *using GTI)*

$$\begin{aligned}
& S(h_{*,\,1})\,\mathbf{DFT}\,G(h_{*,\,*}) \\
\langle \textit{apply}\ (3.1)\rangle \rightarrow\ & S(h_{*,\,1}) \cdot (\mathbf{DFT} \otimes I)\,\mathrm{diag}(\Omega)(I \otimes \mathbf{DFT})L \cdot G(h_{*,\,*}) \\
\langle \textit{convert to GT}\rangle \rightarrow\ & S(h_{*,\,1}) \cdot \mathrm{GTI}(\mathbf{DFT}\,\mathrm{diag}(*),\ h_{0,\,*,1},\ \{*\}) \\
& \cdot \mathrm{GT}(\mathbf{DFT},\ h_{0,\,*,1},\ h_{0,\,1,*},\ \{*\}) \cdot G(h_{*,\,*}) \\
\langle \textit{apply}\ (3.57)\rangle \rightarrow\ & \mathrm{GTI}(\mathbf{DFT}\,\mathrm{diag}(*),\ h_{*,\,*,1},\ \{*\}) \\
& \cdot S(h_{*,\,1}) \cdot \mathrm{GT}(\mathbf{DFT},\ h_{0,\,*,1},\ h_{0,\,1,*},\ \{*\}) \cdot G(h_{*,\,*}) \\
\langle \textit{apply}\ (3.35)\text{–}(3.36)\rangle \rightarrow\ & \mathrm{GTI}(\mathbf{DFT}\,\mathrm{diag}(*),\ h_{*,\,*,1},\ \{*\}) \cdot \mathrm{GT}(\mathbf{DFT},\ h_{*,\,*,*},\ h_{*,\,1,*},\ \{*\})
\end{aligned}$$

The above example also demonstrates that the recursive implementation of (3.10) can be done with no intermediate storage, if non-aliased input and output vectors x and y are given. The right GT works from x into y, and the GTI works inplace in y.

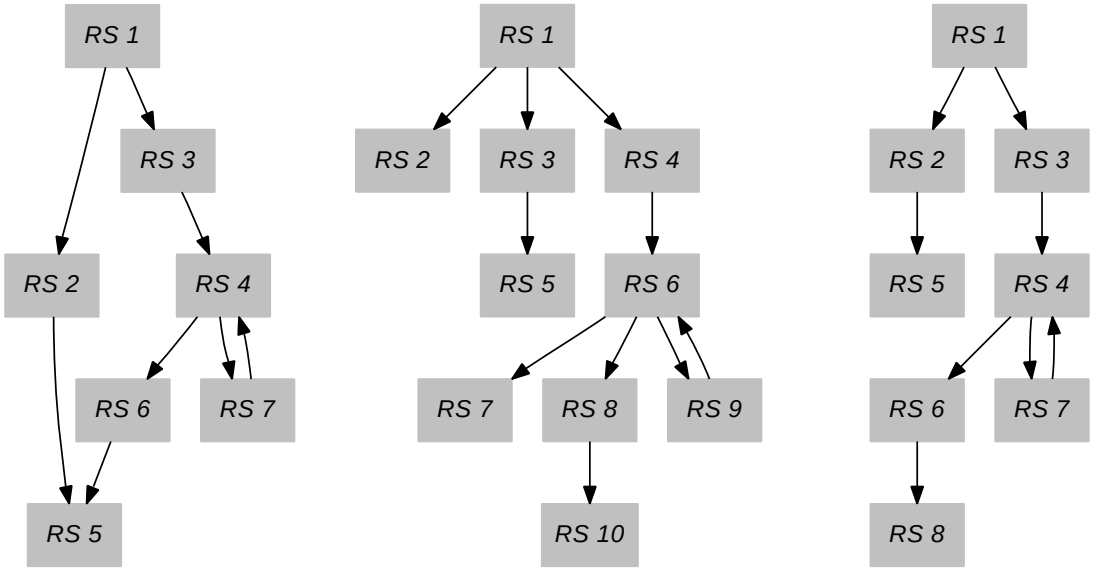
In FFTW, the above is the rationale for forcing the recursion step (3.11) to always be a base case, and thus allowing only right-expanded descent trees.

3.9 Examples of Complicated Closures

Tables 3.10–3.11 show the examples of closures generated for three different transforms (DFT, RDFT and DCT-4). Table 3.10 shows the closures using standard $\sum$-SPL notation, and Table 3.11 shows the same closures using the GT notation, which also explicitly shows how the inplaceness is exploited (via GTI) in the case of DFT. The closures were generated using the following breakdown rules:

- **DFT** – Cooley-Tukey FFT (2.1);
- **RDFT** – real Cooley-Tukey FFT equivalent (2.6), and (2.17) for the auxiliary transform;
- **DCT-4** – "Cooley-Tukey"-type algorithm (2.12), and (2.17), (2.18) for the auxiliary transforms.

In Fig. 3.8 we show the call graphs that correspond to these closures. Note, that we have restricted the recursion (and thus the closure generation) to a subset of all possibilities. For example, the Cooley-Tukey FFT (used in Fig. 3.8(a)) reexpresses the DFT as two smaller DFTs. We restrict the further application of the Cooley-Tukey FFT to one of these smaller DFTs. The same is done



(a) DFT (b) RDFT (c) DCT-4

Figure 3.8: Call graphs for the generated libraries with looped recursion steps (corresponding to Table 3.11).

for RDFT and DCT-4. This means that some nodes in the call graphs are "dead-end", i.e., they have no outgoing edges. Since recursion is not possible for these nodes, they must be implemented as base cases.

Example of a parameter equality constraints. For the more complicated closures the linear system of equations that arises from the parameter equality constraints during parametrization is no longer completely trivial, but still very simple. For example, for the recursion step 3 in the RDFT closure in the parameter equality constraints were

$$\begin{aligned} 2u_8 &= 2u_1, \\ u_2 &= u_3, \\ 2u_1 &= u_2. \end{aligned}$$

There is only one connected component. There are four unknowns and three equations. Fixing u_1 as a constant, and solving for all other parameters gives:

$$\begin{aligned} u_2 &= 2u_1, \\ u_3 &= 2u_1, \\ u_8 &= u_1 \end{aligned}$$

After these constraints were incorporated into the parametrized recursion step we got the results shown in Tables 3.10 and 3.11. The equations above also explain how the parametrized recursion steps can have expressions such as $2u_1$, rather than plain parameters u_i.

Redundancy in the closure. The automatically obtained closures are not minimal. For example, consider the DFT closure in Fig. 3.8(a). The nodes RS 4–7 form a so-called *subclosure*. A subclosure is any smaller closure contained in a larger one. Any recursion step in the subclosure

DFT

$$
\begin{aligned}
1:&\ \mathbf{DFT}_{u_1}\\
2:&\ \sum_{u_7} S(h^{u_1\to u_5}_{0,\ u_6,1})\mathbf{DFT}_{u_1}\operatorname{diag}\left(\operatorname{pre}({u_3}^{\mathbb{Z}\times u_1\to\mathbb{C}})\right)G(h^{u_1\to u_5}_{0,\ u_6,1})\\
3:&\ \sum_{u_8} S(h^{u_1\to u_6}_{0,\ 1,u_7})\mathbf{DFT}_{u_1}G(h^{u_1\to u_3}_{0,\ u_4,1})\\
4:&\ S(h^{u_1\to u_2}_{u_3,\ 1})\mathbf{DFT}_{u_1}G(h^{u_1\to u_6}_{u_7,\ u_8})\\
5:&\ S(h^{u_1\to u_5}_{u_6,\ u_7})\mathbf{DFT}_{u_1}\operatorname{diag}\left(\operatorname{pre}({u_3}^{u_1\to\mathbb{C}})\right)G(h^{u_1\to u_5}_{u_6,\ u_7})\\
6:&\ \sum_{u_8} S(h^{u_1\to u_5}_{u_6,\ u_7,1})\mathbf{DFT}_{u_1}\operatorname{diag}\left(\operatorname{pre}({u_3}^{\mathbb{Z}\times u_1\to\mathbb{C}})\right)G(h^{u_1\to u_5}_{u_6,\ u_7,1})\\
7:&\ \sum_{u_{11}} S(h^{u_1\to u_8}_{u_9,\ 1,u_{10}})\mathbf{DFT}_{u_1}G(h^{u_1\to u_3}_{u_4,\ u_5,u_6})
\end{aligned}
$$

RDFT

$$
\begin{aligned}
1:&\ \mathbf{RDFT}_{u_1}\\
2:&\ S(h^{u_1\to u_2}_{0,\ u_3}\otimes \imath_2)\,\mathbf{RDFT}_{2u_1}\,G(h^{2u_1\to u_6}_{0,\ 1})\\
3:&\ \sum_{u_{11}} S(r^{u_1\to u_7}_{1,\ u_9,1,\ u_{10}}\otimes \imath_2)\operatorname{diag}\left(C_{u_1}\right)\mathbf{rDFT}_{2u_1}(\lambda\text{-wrap}({\lambda_1}^{\mathbb{Z}\times\mathbb{Z}\to\mathbb{R}}))G(h^{2u_1\to u_4}_{u_5,\ 1,u_6})\\
4:&\ \sum_{u_8} S(h^{u_1\to u_6}_{0,\ u_7,1})\,\mathbf{URDFT}_{u_1}\,G(h^{u_1\to u_3}_{0,\ u_4,1})\\
5:&\ S(r^{u_2\to u_1}_{u_3,\ u_4,\ u_5}\otimes \imath_2)\operatorname{diag}\left(C_{u_2}\right)\mathbf{rDFT}_{2u_2}(\lambda\text{-wrap}({\lambda_1}^{\mathbb{Z}\to\mathbb{R}}))G(h^{2u_2\to u_9}_{u_{10},\ 1})\\
6:&\ S(h^{u_1\to u_2}_{u_3,\ u_4})\,\mathbf{URDFT}_{u_1}\,G(h^{u_1\to u_7}_{u_8,\ u_9})\\
7:&\ S(h^{2u_6\to u_2}_{u_3,\ u_4}\circ h^{u_5\to u_6}_{0,\ u_7}\otimes \imath_2)\,\mathbf{URDFT}_{2u_5}\,G(h^{2u_5\to u_{10}}_{0,\ 1})\\
8:&\ \sum_{u_{15}} S(h^{2u_{11}\to u_8}_{u_9,\ u_{10}}\circ r^{u_1\to u_{11}}_{1,\ u_{13},1,\ u_{14}}\otimes \imath_2)\operatorname{diag}\left(C_{u_1}\right)\mathbf{rDFT}_{2u_1}(\lambda\text{-wrap}({\lambda_1}^{\mathbb{Z}\times\mathbb{Z}\to\mathbb{R}}))G(h^{2u_1\to u_4}_{u_5,\ 1,u_6})\\
9:&\ \sum_{u_{10}} S(h^{u_1\to u_8}_{0,\ u_9,1})\,\mathbf{URDFT}_{u_1}\,G(h^{u_1\to u_3}_{u_4,\ u_5,u_6})\\
10:&\ S(h^{2u_5\to u_2}_{u_3,\ u_4}\circ r^{u_6\to u_5}_{u_7,\ u_8,\ u_9}\otimes \imath_2)\operatorname{diag}\left(C_{u_6}\right)\mathbf{rDFT}_{2u_6}(\lambda\text{-wrap}({\lambda_1}^{\mathbb{Z}\to\mathbb{R}}))G(h^{2u_6\to u_{13}}_{u_{14},\ 1})
\end{aligned}
$$

DCT-4

$$
\begin{aligned}
1:&\ \mathbf{DCT\text{-}4}_{u_1}\\
2:&\ \sum_{u_{13}} S(r^{2u_8\to u_9}_{0,\ u_{11},1,\ u_{12}})\operatorname{diag}\left(N_{2u_8}\right)\mathbf{RDFT\text{-}3}^{\top}_{2u_8}\operatorname{rcdiag}(\operatorname{pre}({u_4}^{\mathbb{Z}\times 2u_8\to\mathbb{R}}))G(h^{2u_8\to u_6}_{0,\ 1,u_7}\circ \ell^{2u_8}_{u_8})\\
3:&\ \sum_{u_{10}} S(h^{u_1\to u_8}_{0,\ u_9,1})\mathbf{RDFT\text{-}3}_{u_1}\operatorname{diag}\left(N_{u_1}\right)G(r^{u_1\to u_3}_{0,\ u_5,1,\ u_6})\\
4:&\ S(h^{u_1\to u_2}_{u_3,\ u_4})\,\mathbf{RDFT\text{-}3}_{u_1}\operatorname{diag}\left(N_{u_1}\right)G(r^{u_1\to u_7}_{u_9,\ u_{10},\ u_{11}})\\
5:&\ S(r^{2u_{13}\to u_1}_{u_3,\ u_4,\ u_5})\operatorname{diag}\left(N_{2u_{13}}\right)\mathbf{RDFT\text{-}3}^{\top}_{2u_{13}}\operatorname{rcdiag}(\operatorname{pre}({u_9}^{2u_{13}\to\mathbb{R}}))G(h^{2u_{13}\to u_{11}}_{u_{12},\ 1}\circ \ell^{2u_{13}}_{u_{13}})\\
6:&\ \sum_{u_{14}} S(h^{2u_{10}\to u_7}_{u_8,\ u_9}\circ r^{u_1\to u_{10}}_{0,\ u_{12},1,\ u_{13}}\otimes \imath_2)\operatorname{diag}\left(C_{u_1}\right)\mathbf{rDFT}_{2u_1}(1/4)G(h^{2u_1\to u_4}_{0,\ 1,u_5})\\
7:&\ \sum_{u_{12}} S(h^{u_1\to u_{10}}_{0,\ u_{11},1})\mathbf{RDFT\text{-}3}_{u_1}\operatorname{diag}\left(N_{u_1}\right)G(r^{u_1\to u_3}_{u_5,\ u_6,u_7,\ u_8})\\
8:&\ S(h^{2u_5\to u_2}_{u_3,\ u_4}\circ r^{u_6\to u_5}_{u_7,\ u_8,\ u_9}\otimes \imath_2)\operatorname{diag}\left(C_{u_6}\right)\mathbf{rDFT}_{2u_6}(1/4)G(h^{2u_6\to u_{13}}_{u_{14},\ 1})
\end{aligned}
$$

Table 3.10: Generated recursion step closures for **DFT**, **RDFT**, and **DCT-4** with looped recursion steps in index-free $\sum$-SPL.

DFT

1 : $\mathbf{DFT}_{u_1}$
2 : $\mathrm{GTI}(\mathbf{DFT}_{u_1} \mathrm{diag}\left(\mathrm{pre}({u_3}^{\mathbb{Z}\times u_1\to\mathbb{C}})\right),\ h^{u_1\to u_5}_{0,\ u_6,1},\ \{u_7\})$
3 : $\mathrm{GT}(\mathbf{DFT}_{u_1},\ h^{u_1\to u_3}_{0,\ u_4,1},\ h^{u_1\to u_6}_{0,\ 1,u_7},\ \{u_8\})$
4 : $S(h^{u_1\to u_2}_{u_3,\ 1})\mathbf{DFT}_{u_1} G(h^{u_1\to u_6}_{u_7,\ u_8})$
5 : $\mathrm{GTI}(\mathbf{DFT}_{u_1} \mathrm{diag}\left(\mathrm{pre}({u_3}^{u_1\to\mathbb{C}})\right),\ h^{u_1\to u_5}_{u_6,\ u_7},\ \{\})$
6 : $\mathrm{GTI}(\mathbf{DFT}_{u_1} \mathrm{diag}\left(\mathrm{pre}({u_3}^{\mathbb{Z}\times u_1\to\mathbb{C}})\right),\ h^{u_1\to u_5}_{u_6,\ u_7,1},\ \{u_8\})$
7 : $\mathrm{GT}(\mathbf{DFT}_{u_1},\ h^{u_1\to u_3}_{u_4,\ u_5,u_6},\ h^{u_1\to u_8}_{u_9,\ 1,u_{10}},\ \{u_{11}\})$

RDFT

1 : $\mathbf{RDFT}_{u_1}$
2 : $S((h^{u_1\to u_2}_{0,\ u_3}\otimes \imath_2))\,\mathbf{RDFT}_{2u_1}\, G(h^{2u_1\to u_6}_{0,\ 1})$
3 : $\mathrm{GT}(\mathrm{diag}\left(C_{u_1}\right)\mathbf{rDFT}_{2u_1}(\lambda\text{-wrap}({\lambda_1}^{\mathbb{Z}\times\mathbb{Z}\to\mathbb{R}})),\ h^{2u_1\to u_4}_{u_5,\ 1,u_6},\ (r^{u_1\to u_7}_{1,\ u_9,1,\ u_{10}}\otimes \imath_2),\ \{u_{11}\})$
4 : $\mathrm{GT}(\mathbf{URDFT}_{u_1},\ h^{u_1\to u_3}_{0,\ u_4,1},\ h^{u_1\to u_6}_{0,\ u_7,1},\ \{u_8\})$
5 : $S((r^{u_2\to u_1}_{u_3,\ u_4,\ u_5}\otimes \imath_2))\mathrm{diag}\left(C_{u_2}\right)\mathbf{rDFT}_{2u_2}(\lambda\text{-wrap}({\lambda_1}^{\mathbb{Z}\to\mathbb{R}}))G(h^{2u_2\to u_9}_{u_{10},\ 1})$
6 : $S(h^{u_1\to u_2}_{u_3,\ u_4})\,\mathbf{URDFT}_{u_1}\, G(h^{u_1\to u_7}_{u_8,\ u_9})$
7 : $S(h^{2u_6\to u_2}_{u_3,\ u_4}\circ(h^{u_5\to u_6}_{0,\ u_7}\otimes \imath_2))\,\mathbf{URDFT}_{2u_5}\, G(h^{2u_5\to u_{10}}_{0,\ 1})$
8 : $\mathrm{GT}(\mathrm{diag}\left(C_{u_1}\right)\mathbf{rDFT}_{2u_1}(\lambda\text{-wrap}({\lambda_1}^{\mathbb{Z}\times\mathbb{Z}\to\mathbb{R}})),\ h^{2u_1\to u_4}_{u_5,\ 1,u_6},\ h^{2u_{11}\to u_8}_{u_9,\ u_{10}}\circ(r^{u_1\to u_{11}}_{1,\ u_{13},1,\ u_{14}}\otimes \imath_2),\ \{u_{15}\})$
9 : $\mathrm{GT}(\mathbf{URDFT}_{u_1},\ h^{u_1\to u_3}_{u_4,\ u_5,u_6},\ h^{u_1\to u_8}_{0,\ u_9,1},\ \{u_{10}\})$
10 : $S(h^{2u_5\to u_2}_{u_3,\ u_4}\circ(r^{u_6\to u_5}_{u_7,\ u_8,\ u_9}\otimes \imath_2))\mathrm{diag}\left(C_{u_6}\right)\mathbf{rDFT}_{2u_6}(\lambda\text{-wrap}({\lambda_1}^{\mathbb{Z}\to\mathbb{R}}))G(h^{2u_6\to u_{13}}_{u_{14},\ 1})$

DCT-4

1 : $\mathbf{DCT\text{-}4}_{u_1}$
2 : $\mathrm{GT}(\mathrm{diag}\left(N_{2u_8}\right)\mathbf{RDFT\text{-}3}^{\top}_{2u_8}\,\mathrm{rcdiag}(\mathrm{pre}({u_4}^{\mathbb{Z}\times 2u_8\to\mathbb{R}})),\ h^{2u_8\to u_6}_{0,\ 1,u_7}\circ\ell^{2u_8}_{u_8},\ r^{2u_8\to u_9}_{0,\ u_{11},1,\ u_{12}},\ \{u_{13}\})$
3 : $\mathrm{GT}(\mathbf{RDFT\text{-}3}_{u_1}\,\mathrm{diag}\left(N_{u_1}\right),\ r^{u_1\to u_3}_{0,\ u_5,1,\ u_6},\ h^{u_1\to u_8}_{0,\ u_9,1},\ \{u_{10}\})$
4 : $S(h^{u_1\to u_2}_{u_3,\ u_4})\,\mathbf{RDFT\text{-}3}_{u_1}\,\mathrm{diag}\left(N_{u_1}\right)G(r^{u_1\to u_7}_{u_9,\ u_{10},\ u_{11}})$
5 : $S(r^{2u_{13}\to u_1}_{u_3,\ u_4,\ u_5})\mathrm{diag}\left(N_{2u_{13}}\right)\mathbf{RDFT\text{-}3}^{\top}_{2u_{13}}\,\mathrm{rcdiag}(\mathrm{pre}({u_9}^{2u_{13}\to\mathbb{R}}))G(h^{2u_{13}\to u_{11}}_{u_{12},\ 1}\circ\ell^{2u_{13}}_{u_{13}})$
6 : $\mathrm{GT}(\mathrm{diag}\left(C_{u_1}\right)\mathbf{rDFT}_{2u_1}(\lambda\text{-wrap}({\lambda_1}^{\mathbb{Z}\times\mathbb{Z}\to\mathbb{R}})),\ h^{2u_1\to u_4}_{0,\ 1,u_5},\ h^{2u_{10}\to u_7}_{u_8,\ u_9}\circ(r^{u_1\to u_{10}}_{0,\ u_{12},1,\ u_{13}}\otimes \imath_2),\ \{u_{14}\})$
7 : $\mathrm{GT}(\mathbf{RDFT\text{-}3}_{u_1}\,\mathrm{diag}\left(N_{u_1}\right),\ r^{u_1\to u_3}_{u_5,\ u_6,u_7,\ u_8},\ h^{u_1\to u_{10}}_{0,\ u_{11},1},\ \{u_{12}\})$
8 : $S(h^{2u_5\to u_2}_{u_3,\ u_4}\circ(r^{u_6\to u_5}_{u_7,\ u_8,\ u_9}\otimes \imath_2))\mathrm{diag}\left(C_{u_6}\right)\mathbf{rDFT}_{2u_6}(\lambda\text{-wrap}({\lambda_1}^{\mathbb{Z}\to\mathbb{R}}))G(h^{2u_6\to u_{13}}_{u_{14},\ 1})$

Table 3.11: Generated recursion step closures for **DFT**, **RDFT**, and **DCT-4** (of Table 3.10) with looped recursion steps using GT.

can be implemented using only other recursion steps in the same subclosure. On the call graph the subclosure is a subgraph which has no outgoing edges that connect to nodes outside of the subgraph. Indeed, starting from RS 4 we can only reach nodes RS 4–7.

It is very often the case that one of the elements of the subclosure can be used to implement the root of the original closure. If we look up the definition of RS 4 in Table 3.10, we find it to be

a DFT with a strided scatter and gather. Indeed, RS 1 can be easily implemented using RS 4 by setting $u_3 = u_7 = 0$ (gather/scatter bases) and $u_8 = 1$ (gather stride) in RS 4.

The same kind of closure reduction can be done for the RDFT. Starting from RS 6, the only reachable nodes are RS 6–10. Thus, we have a smaller subclosure with only 5 recursion steps (instead of 10). Strictly speaking, RS 6 is not a more general version of RS 1 (see Table 3.10), but it can still be used to implement RS 1, using a close relationship between **RDFT** and **URDFT**.

To summarize the above method, the size of the closure can be reduced by 1) finding the smallest closed subgraph in the original call graph (i.e., a subclosure); 2) determining how to implement RS 1 using one of the nodes in the subclosure; 3) using the subclosure instead of the original closure. In the case of the DFT, this procedure is equivalent to eliminating the more specialized variants recursion steps in favors of less specialized. In the case of the RDFT, a conversion between **RDFT** and **URDFT** is necessary to apply the method. In the case of the DCT-4, this method does not work, since no other recursion steps contains the DCT-4 or related transform.

An alternative way to reduce the closure size is by fusing pairs of recursion steps. A pair of recursion steps can be fused into a single one, if either of the recursion steps can be used to implement the other. For example, RS 1 and RS 4 in the DFT closure can be fused, because RS 4 can implement RS 1 as we already explained. After fusing RS 1 and RS 4, we obtain the new closure consisting of RS 4–7 only, which is the same as using the subclosure method.

However, fusion is more generally applicable, because it does not require the existence of the subclosure. For example, it can be applied to the DCT-4, when the subclosure method fails. For example, RS 3 and RS 7 for the DCT-4 can be fused.

Eliminating the specialized variants to reduce the closure size using either of the methods can lead to performance degradation. For example, we see this for small transform sizes in FFTW as explained in Section 6.2.4 (Chapter 6, see Fig. 6.7).

To date, we have not implemented any of these reduction schemes, however, we believe that both methods can be successfully automated. If code size is important, the closure reduction methods become a crucial tool in reducing the code size.

Chapter 4

Library Generation: Parallelism

To obtain the highest possible performance on the current computers the programmer must exploit two levels of parallelism: vector parallelism and thread parallelism. Fig. 4.1 graphically shows the

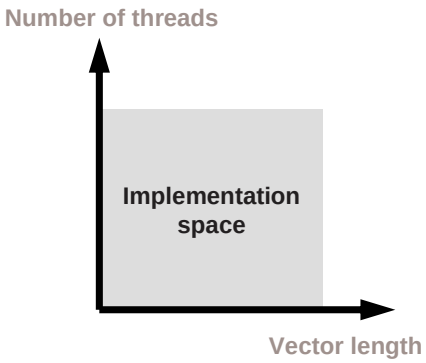


Figure 4.1: The space of implementations including both vector and thread parallelism.

implementation space within both types of parallelism. Our goal is to be able to span this entire space with our generator.

For example, a common computer available today might have 4 processor cores, and support 4-way SIMD parallelism. In this case the maximum possible speedup from correctly exploiting parallelism is 16x.

For typical numeric code, exploiting vector parallelism involves focusing on inner loops and parallelism within operations. Exploiting thread parallelism means focusing on the outer loops.

With $\sum$-SPL framework, however, we have a unique situation: using loop interchange and loop distribution outer loops can become inner loops, and thus we are very flexible. In fact the same loop can be used for both vector and thread parallelism.

In the following two sections we explain how we enable both kinds of parallelism in the library generator. We follow the same outline. First, we explain how this can be done at the SPL level, perhaps with some restrictions, and then explain the more general method at the $\sum$-SPL level. In the case of vectorization, we have not yet translated all SPL vectorization techniques to the $\sum$-SPL, and only a subset is available, which results in suboptimal, but very acceptable performance.

4.1 Vectorization

4.1.1 Background

Many current general purpose and DSP architectures provide short vector SIMD (single instruction, multiple data) instructions, which operate in parallel on the elements of a vector register, typically of length 2 to 16. Introduction of SIMD vector instructions is a relatively straightforward way to add parallelism to an existing CPU data-path, and as the clock frequency scaling is becoming less viable, SIMD extensions are becoming ubiquitous.

For instance, Intel and AMD defined over the years MMX, SSE, SSE2, SSE3, SSSE, SSE4, SSE5, 3DNow!, Extended 3DNow!, and 3DNow! Professional as x86 SIMD vector extensions. Modern PowerPC architectures provide AltiVec SIMD instruction set, and Cell SPU and BlueGene/L have their own custom vector extensions. Additional extensions are defined by PA-RISC (MAX), MIPS (MIPS-3D) and ARM (Wireless MMX) processor architectures, and also many VLIW DSP processors.

Although exploiting SIMD instructions does not require a new programming model (unlike threads), it requires either programming in assembly or using low-level compiler intrinsics available in C and C++. Another alternative is using the compiler vectorization, although it remains rather limited, as manifested by the fact that most high performance libraries rely on human-vectorized code.

Previous work. There is a large body of work on vectorizing compilers for the vector supercomputers of 1980s. For example, [6,139]. With the advent of SIMD in mainstream platforms, the topic has received renewed attention. Current SIMD architectures have shorter vectors, strict data alignment requirements, and other differences

The most commonly used technique for vectorizing programs for modern SIMD platforms is vectorization of the inner loops. For example, see [75,90,91]. Vectorization in the popular GNU C Compiler (GCC) is described in [85]. Intel C/C++ Compiler also provides a vectorizer, described in [25]. An excellent overview of the difficulties of vectorization for several relevant multimedia programs is given in [104].

Another technique for vectorization called superword-level parallelization (SLP) was proposed by Larsen, et. al. in [76, 77, 107] and independently in the context of the DFT by Kral, et al. in [49,56,74]. SLP attempts to group isomorphic statements in order to obtain a short vector. Yet another form of SLP, which exploits the 2-way parallelism within the complex number arithmetic, was proposed in FFTW 3.x [61].

Interestingly, SLP can be combined with traditional vectorization to obtain longer vectors, as also shown in [61]. Similarly, one can vectorize several loops at once to produce longer vectors [136] .

Modern SIMD processors impose strict alignment constraints. An approach to vectorization with unaligned data is discussed in [46,135]. Vectorizing with disjoint data sets is discussed in [86].

Data alignment handling and vectorization of disjoint data are useful optimizations which are orthogonal to the general vectorization. We expect these methods to be also expressible on the SPL and/or $\sum$-SPL level and be potentially beneficial to the performance of our generated code, since we did not fully address these issues.

The scope of our work. In this thesis we propose a form of domain specific vectorization, which vectorizes the loops implied from the SPL or $\sum$-SPL constructs. Both inner and outer loops can be vectorized, and unless there exist a transform specific vectorization rule, the outer loops are

vectorized to minimize the vector shuffle overhead.

We worked with Intel's SSE, SSE2 and SSE3 extensions. These extensions are available on most current Intel and AMD processors. Collectively, they define six 128-bit *modes*: 4-way single-precision and 2-way double-precision floating point vectors; and 2-way 64-bit, 4-way 32-bit, 8-way 16-bit, and 16-way 8-bit integer vectors. We denote the vector length of a SIMD extension mode with ν.

We only consider the 2- and 4-way floating point modes, since these are by far the most widely used and relevant for linear transforms. Our approach is not limited to $\nu = 4$, and in our earlier work [54] we already show results with 8-way and 16-way vectors.

For the code examples with vector instructions we will use the Intel C compiler's intrinsics interface (also supported by the GNU C compiler). The interface includes *vector data types* that correspond vector registers or aligned memory locations, and *vector operations* that correspond to SSE instructions.

As an example, below we show a short instruction sequence using the two SSE intrinsic operations `_mm_add_ps` and `_mm_sub_ps`, which correspond to vector addition and subtraction:

```
/* Intel C SSE intrinsics, 4-way 32-bit floating-point data-type */
__m128 x[2], y[2];
y[0] = _mm_add_ps(x[0], x[1]);
y[1] = _mm_sub_ps(x[0], x[1]);
```

In the rest of this section we first explain at a high level how the vectorization is performed at the SPL level of abstraction using a combination of specialized breakdown and rewrite rules. This is the result of our previous work described in [53]. Next, we explain how the vectorization can be adapted for generating libraries. In order to work with libraries, the SPL vectorization rules had to be "ported" to the $\sum$-SPL level.

4.1.2 Vectorization by Rewriting SPL Formulas.

In Spiral, vectorized and parallelized implementations are generated by first obtaining "fully optimized" SPL formulas, which means they can be mapped to efficient vector or parallel code. This is possible because certain SPL constructs express parallelism. For example, $I \otimes A$ corresponds to a parallelizable loop with no loop carried dependencies.

In this section, our goal is to take formulas obtained by the recursive application of breakdown rules like (2.1) and automatically manipulate them into a form that enables a direct mapping into SIMD vector code. Further, we also want to explore different vectorizations for the same formula. The solution is a suitably designed rewriting system that implements our previous ideas for formula-based vectorization in [50, 51].

To produce SIMD vector code, Spiral needs the following four components:

- *Vectorization tags* which introduce the vector length, and mark the formula "to be vectorized".
- *Vector formula constructs* which denote the subformulas can be perfectly mapped to SIMD code.
- *Rewriting rules* which transform general formulas into vector formulas.
- *Vector backend* that emits SIMD code from vector formulas.

Vectorization tags. We introduce a set of tags to propagate vectorization information through the formulas and to perform algebraic simplification of permutations. Note that all objects remain matrices.

We tag a formula construct A to be translated into vector code for vector length ν by

$$\mathbf{Vec}_\nu\,(A)\,.$$

Vector formula constructs. The central formula construct that can be implemented on all ν-way short vector extensions is

$$A \otimes I_\nu, \tag{4.1}$$

where A is an arbitrary real matrix. Vector code is obtained by generating scalar code for A (i.e., for $x \mapsto Ax$) and replacing all scalar operations by their respective ν-way vector operations. For example, `c=a+b` is replaced by `c=vadd(a,b)`.

We write

$$A \overline{\otimes} I_\nu \tag{4.2}$$

to stipulate that the tensor product is to be mapped into vector code as explained above.

Of course, most formulas do not match (4.2). In these cases we manipulate the formula using rewriting rules to consist of components that either match (4.2) or are among a small set of base cases. It turns out that for a large class of formulas the only base cases needed are

$$\underbrace{L^{2\nu}_2}_{\text{base}},\ \underbrace{L^{2\nu}_\nu}_{\text{base}},\ \underbrace{L^{\nu^2}_\nu}_{\text{base}},\ \underbrace{D_\nu}_{\text{base}} \tag{4.3}$$

where D_ν is a real $\nu \times \nu$ diagonal matrix. Above, we used a tag "base" to mark the base cases.

Constructs marked with $\overline{\otimes}$ and "base" are final, i.e., they will not be changed by rewriting rules.

For vectorizing complex formulas, additional base case are needed, we do not show them here, but refer the reader to [53] for details.

Definition 1 We call a formula ν-way vectorized if it is either of the form (4.2) *or one of the forms in* (4.3)*, or of the form*

$$I_m \otimes A \text{ or } AB, \tag{4.4}$$

where A and B are ν-way vectorized.

Rewriting rules. In our framework the rewriting starts from the tagged formula $\mathbf{Vec}_\nu\,(A)$ and proceeds by applying rules that vectorize the formula in the sense of Definition 1. Most of our rewriting rules are shown in Tables 4.1–4.2.

As an example of applying the rewriting rules we explain how $I_m \otimes A$ is handled by our rewriting system. We start with the tagged formula

$$\mathbf{Vec}_\nu\,(I_m \otimes A)\,,$$

which means "$I_m \otimes A$ is to be vectorized." The system can only apply one of the alternatives of rule (4.9). Suppose it chooses the first alternative, which yields

$$I_{m/\nu} \otimes (\mathbf{Vec}_\nu\,(I_\nu \otimes A))$$

$$\mathbf{Vec}_\nu\left(L^{n\nu}_n\right) \rightarrow \left(I_{n/\nu} \otimes \underbrace{L^{\nu^2}_\nu}_{\text{base}}\right)\left(L^n_{n/\nu}\overline{\otimes} I_\nu\right) \tag{4.5}$$

$$\mathbf{Vec}_\nu\left(L^{n\nu}_\nu\right) \rightarrow \left(L^n_\nu\overline{\otimes} I_\nu\right)\left(I_{n/\nu} \otimes \underbrace{L^{\nu^2}_\nu}_{\text{base}}\right) \tag{4.6}$$

$$\mathbf{Vec}_\nu\left(L^{mn}_m\right) \rightarrow \left(L^{mn/\nu}_m\overline{\otimes} I_\nu\right)\left(I_{mn/\nu^2} \otimes \underbrace{L^{\nu^2}_\nu}_{\text{base}}\right)\left((I_{n/\nu} \otimes L^m_{m/\nu})\overline{\otimes} I_\nu\right) \tag{4.7}$$

Table 4.1: SPL vectorization rules for the stride permutation.

$$\mathbf{Vec}_\nu\left(A \otimes I_m\right) \rightarrow \left(A \otimes I_{m/\nu}\right)\overline{\otimes} I_\nu \tag{4.8}$$

$$\mathbf{Vec}_\nu\left(I_m \otimes A\right) \rightarrow \begin{cases} I_{m/\nu} \otimes \mathbf{Vec}_\nu\left(\left(I_\nu \otimes A\right)\right) \\ \mathbf{Vec}_\nu\left(L^{mn}_m\right)\mathbf{Vec}_\nu\left(A \otimes I_m\right)\mathbf{Vec}_\nu\left(L^{mn}_n\right) \end{cases} \tag{4.9}$$

$$\mathbf{Vec}_\nu\left(\left(I_m \otimes A\right)L^{mn}_m\right) \rightarrow \begin{cases} \mathbf{Vec}_\nu\left(L^{mn}_m\right)\left(\mathbf{Vec}_\nu\left(A \otimes I_m\right)\right) \\ \left(I_{m/\nu} \otimes \mathbf{Vec}_\nu\left(L^{n\nu}_\nu\right)\left(A\overline{\otimes} I_\nu\right)\right)\left(L^{mn/\nu}_{m/\nu}\overline{\otimes} I_\nu\right) \end{cases} \tag{4.10}$$

$$\mathbf{Vec}_\nu\left(\left(I_k \otimes \left(I_m \otimes A^{n\times n}\right)L^{mn}_m\right)L^{kmn}_k\right) \rightarrow \mathbf{Vec}_\nu\left(L^{km}_k \otimes I_n\right)\left(I_m \otimes \mathbf{Vec}_\nu\left(\left(I_k \otimes A^{n\times n}\right)L^{kn}_k\right)\right) \mathbf{Vec}_\nu\left(L^{mn}_m \otimes I_k\right) \tag{4.11}$$

Table 4.2: SPL vectorization rules for tensor products. A is an $n \times n$ matrix.

and then applies the second alternative of (4.9) to $(I_\nu \otimes A)$, which leads to

$$I_{m/\nu} \otimes \left(\mathbf{Vec}_\nu\left(L^{n\nu}_\nu\right)\left(A\overline{\otimes} I_\nu\right)\mathbf{Vec}_\nu\left(L^{n\nu}_n\right)\right).$$

Next, only rules (4.5) and (4.6) match, which yields

$$I_{m/\nu} \otimes \Big((L^n_\nu\overline{\otimes} I_\nu)(I_{n/\nu} \otimes \underbrace{L^{\nu^2}_\nu}_{\text{base}})(A\overline{\otimes} I_\nu)(I_{n/\nu} \otimes \underbrace{L^{\nu^2}_\nu}_{\text{base}})(L^n_{n/\nu}\overline{\otimes} I_\nu)\Big).$$

This is the final vectorized formula.

4.1.3 Vectorization by Rewriting $\sum$-SPL Formulas

To make this approach compatible with library generation we convert the tagged SPL formulas to tagged $\sum$-SPL formulas and convert vectorization/parallelization rules from Table 4.2 to GT breakdown rules.

Index-free $\sum$-SPL and GT representation exposes the data access pattern of loops. Most importantly, we can easily recognize tensor products at such representation, by looking at the index mapping functions. Moreover since the constants (such as vector stride) survive parametrization, the tensor products can still be recognized after the parametrization. Thus, the original SPL

$$
\begin{aligned}
G(f^{n\to N}) &= \sum_j S(h_{j,\,1}) I_1 G(f \circ h_{j,\,1}) && = \mathrm{GT}(I_1,\ h_{0,\,1,1},\ f \circ h_{0,\,1,1},\ \{n\}),\\
\mathrm{perm}(f) &= G(f) && = \mathrm{GT}(I_1,\ h_{0,\,1,1},\ f \circ h_{0,\,1,1},\ \{n\}),\\
S(f) &= G(f)^\top && = \mathrm{GT}(I_1,\ f \circ h_{0,\,1,1},\ h_{0,\,1,1},\ \{n\})\\
\mathrm{diag}\left(d^{n\to\mathbb{C}}\right) &= \sum_j S(h_{j,\,1})\,\mathrm{diag}\left(f \circ h_{j,\,1}\right) G(h_{j,\,1}) && = \mathrm{GT}(\mathrm{diag}\left(f \circ h_{0,\,1,1}\right),\ h_{0,\,1,1},\ h_{0,\,1,1},\ \{n\}).
\end{aligned}
$$

Table 4.3: Converting other constructs to GT.

vectorization of Spiral is guaranteed to be portable to the parametrized index-free $\sum$-SPL with GTs.

Since GT can describe a larger formula space using only few orthogonal constructs, we can vectorize a larger variety of formulas. From another perspective, since GT represents a loop, standard loop vectorization techniques for short vectors [25, 75, 85] apply, and we can recast our vectorization method using standard compiler terminology.

The GT based vectorization for libraries has two separate phases:

- finding vectorizable loops (and marking them for vectorization);
- vectorizing marked loops.

This is similar to a general purpose vectorizing compiler [25,85], however there are important differences. Unlike in a general purpose compiler our vectorization approach is a global transformation, which can vectorize across multiple recursive function calls. As a consequence, we are not limited to vectorizing inner loops, but in fact most of the time, vectorize outermost loops. Finally, we are not limited to unit-stride loops.

Finding vectorizable loops. In a general purpose compiler, a vectorizable loop is the inner loop without loop-carried dependencies that satisfies some additional constraints, e.g., all computations are of the same data type, all data accesses are unit-stride, and there are no function calls.

Interestingly, in $\sum$-SPL loop-carried dependencies do not exist, and we impose no other constraints on vectorizable loops. Thus, surprisingly, *all* loops are vectorizable. Clearly, the best choices are the loops with largest bodies, i.e., the outer loops.

To find maximal vectorizable loops in $\sum$-SPL formula F we start by tagging it as $\mathbf{Vec}_\nu(F)$, and then propagate the tag down using only a pair of rewrite rules below:

$$
\begin{aligned}
\mathbf{Vec}_\nu(A \cdot B) &\to \mathbf{Vec}_\nu(A) \cdot \mathbf{Vec}_\nu(B),\\
\mathbf{Vec}_\nu(A + B) &\to \mathbf{Vec}_\nu(A) + \mathbf{Vec}_\nu(B).
\end{aligned}
$$

Quickly, the rewriting will yield tagged GTs, and in few cases, other standalone constructs, namely, G, S, perm, diag and unstructured sparse matrices. We do not handle vectorizing unstructured sparse matrices, these only occur in formulas for small transforms, obtained using less structured algorithms, which are not practical for large transforms. G, S, *perm* and diag can be trivially converted into iterative sums and thus GT, using rules in Table 4.3,

Vectorizing marked loops. In the previous step we find maximal vectorizable loops (GTs). If it happens that other constructs are tagged, they are also converted to GT. Thus the remaining step is to vectorize these GTs. The standard loop vectorization procedure consists of three substeps:

1. strip-mining;
2. maximal loop distribution (and loop interchange in our case);
3. replacement by vector operations.

We perform the first step using a new GT breakdown rule GT-Vec, a specialized variant of strip-mining (3.50). Second step is performed by the loop transformation infrastructure described in Section (3.7), namely by rules GT-DownRank and GT-Distr ((3.48) and (3.49)). The final step is performed by the Spiral backend, which maps $\sum$-SPL to code.

GT-Vec: strip-mining for vectorization. Strip-mining splits a single loop into two nested loops. For vectorization, we make the inner loop consists of ν iterations (vector length), and tag the inner loop with the so-called it *sticky* vector tag.

$$\mathbf{Vec}_\nu\left(\mathrm{GT}(A,\ g,\ s,\ \{n\})\right) \xrightarrow{\mathrm{GT-Vec}} \mathrm{GT}(\mathrm{split}(A,1,\nu),\ \mathrm{split}(g,1,\nu),\ \mathrm{split}(s,1,\nu),\ \{\overline{\nu}, n/\nu\})$$

We call the tagged inner loop on the right-hand side of above a sticky vector loop. The goal of the rewrite system is to maximally distribute the sticky loop. The goal of the backend is to implement sticky vector loops using vector instructions. However, since these loops are maximally distributed, the backend will only have to handle few orthogonal $\sum$-SPL constructs.

In general purpose vectorizing compilers [25, 85] the scalar operations are replaced by vector equivalents immediately after strip mining, which precludes vectorization to be applied to outer loops. Our method is one possible solution to this problem. The analysis required by the general purpose compiler to vectorize an outer loop in such scenario might be too prohibitive. The nature of $\sum$-SPL, however, guarantees the legality of loop distribution and exchange in all cases.

Maximal loop distribution and loop interchange. In this step we maximally distribute the sticky loop into multiple loops with a single "statement" as a body of each loop. The meaning of "single statement" is context dependent, but here we mean a single $\sum$-SPL construct.

To accomplish the maximal distribution, we unconditionally apply the loop distribution rule GT-Distr to the sticky loop, and disallow downranking of the tagged sticky loop. Proceeding this way, iterating downranking and distribution will push the sticky loop all the way down. For example:

$$GT(GT \cdot GT) \to GT(GT) \cdot GT(GT) \to GT^2 \cdot GT^2$$

Above we denoted rank-2 GTs with GT^2. The rank-2 GTs can be downranked to the rank-1 GT with a sticky vector loops, and the process can be continued as we recurse deeper and deeper.

Replacement by vector operations. Eventually the recursion has to be terminated with a base case, and because the sticky vector loop cannot be downranked the base cases will necessarily be of the form

$$\mathrm{GT}(A,\ g,\ s,\ \{\overline{\nu}\}).$$

This plays the role of $A \otimes I_\nu$ in the SPL vectorization, and means that A must be implemented using vector arithmetic instead of the regular scalar operations. However, the important difference in the $\sum$-SPL case, is that the data required for different vector slots might not be contiguous in

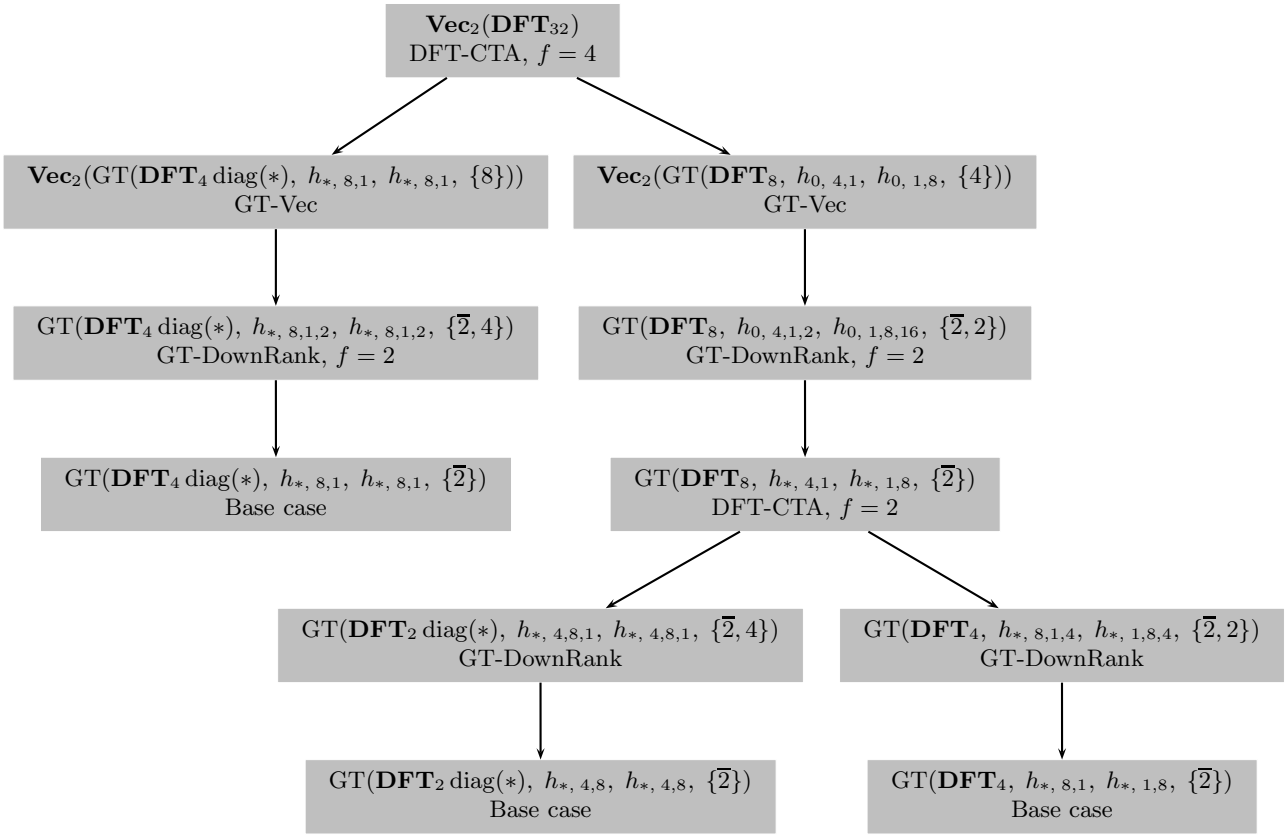


Figure 4.2: Descent tree for $\mathbf{Vec}_2(\mathbf{DFT}_{32})$.

memory. If this is the case, then subvector loads and stores can be used to populate the vectors element-by-element. Although this is generally slower, this enables the $\sum$-SPL vectorization to work for a wider class of formulas than the SPL vectorization.

The gather and scatter functions g and s must be analyzed to determine whether subvector access is required. The simplest case occurs when both g and s are of the form $h_{0,\,s,1}$ and $\nu|s$, which means that the vectors are contiguous in memory, and can be loaded and stored using standard vector load and store instructions.

4.1.4 Vectorized Closure Example

In Fig. 4.2 we show a descent tree for a vectorization of $\mathbf{DFT}_{32}$ implemented using (3.1). The descent tree should be compared to the non-vectorized descent tree in Fig. 3.5

In Table 4.4 we show the recursion step closure and the corresponding call graph for the 2-way vectorized DCT-4. The closure has 16 recursion steps, many of which are rank-1 or rank-2 GTs with sticky vector loops of 2 iterations. These loops are translated to vector instructions in the

generated code.

1 : $\mathrm{Vec}_2(\mathbf{DCT\text{-}4}_{u_1})$
2 : $\mathrm{Vec}_2(\mathrm{GT}(\mathrm{diag}\left(N_{2u_8}\right)\mathbf{RDFT\text{-}3}^{\top}_{2u_8}\,\mathrm{rcdiag}(\mathrm{pre}({u_4}^{\mathbb{Z}\times 2u_8\to\mathbb{R}})),\ h^{2u_8\to u_6}_{0,\ 1,u_7}\circ\ell^{2u_8}_{u_8},\ r^{2u_8\to u_9}_{0,\ u_{11},1,\ u_{12}},\ \{u_{13}\}))$
3 : $\mathrm{Vec}_2(\mathrm{GT}(\mathbf{RDFT\text{-}3}_{u_1}\,\mathrm{diag}\left(N_{u_1}\right),\ r^{u_1\to u_3}_{0,\ u_5,1,\ u_6},\ h^{u_1\to u_8}_{0,\ u_9,1},\ \{u_{10}\}))$
4 : $\mathrm{GT}(\mathrm{diag}\left(N_{2u_9}\right)\mathbf{RDFT\text{-}3}^{\top}_{2u_9}\,\mathrm{rcdiag}(\mathrm{pre}({u_4}^{\mathbb{Z}\times\mathbb{Z}\times 2u_9\to\mathbb{R}})),\ h^{2u_9\to u_6}_{0,\ 1,u_7,u_8}\circ\ell^{2u_9}_{u_9},\ r^{2u_9\to u_{10}}_{0,\ u_{12},1,2,\ u_{13}},\ \{\overline{2},u_{14}\})$
5 : $\mathrm{GT}(\mathrm{diag}\left(N_{2u_9}\right)\mathbf{RDFT\text{-}3}^{\top}_{2u_9}\,\mathrm{rcdiag}(\mathrm{pre}({u_4}^{\mathbb{Z}\times 2u_9\to\mathbb{R}})),\ h^{2u_9\to u_6}_{u_7,\ 1,u_8}\circ\ell^{2u_9}_{u_9},\ r^{2u_9\to u_{10}}_{u_{12},\ u_{13},1,\ u_{14}},\ \{u_{15}\})$
6 : $\mathrm{GT}(\mathbf{RDFT\text{-}3}_{u_1}\,\mathrm{diag}\left(N_{u_1}\right),\ r^{u_1\to u_3}_{0,\ u_5,1,2,\ u_6},\ h^{u_1\to u_8}_{0,\ u_9,1,2},\ \{\overline{2},u_{10}\})$
7 : $\mathrm{GT}(\mathbf{RDFT\text{-}3}_{u_1}\,\mathrm{diag}\left(N_{u_1}\right),\ r^{u_1\to u_3}_{u_5,\ u_6,1,\ u_7},\ h^{u_1\to u_9}_{u_{10},\ u_{11},1},\ \{u_{12}\})$
8 : $S(h^{u_1\to u_2}_{u_3,\ u_4})\,\mathbf{RDFT\text{-}3}_{u_1}\,\mathrm{diag}\left(N_{u_1}\right)G(r^{u_1\to u_7}_{u_9,\ u_{10},\ u_{11}})$
9 : $S(r^{2u_{13}\to u_1}_{u_3,\ u_4,\ u_5})\mathrm{diag}\left(N_{2u_{13}}\right)\mathbf{RDFT\text{-}3}^{\top}_{2u_{13}}\,\mathrm{rcdiag}(\mathrm{pre}({u_9}^{2u_{13}\to\mathbb{R}}))G(h^{2u_{13}\to u_{11}}_{u_{12},\ 1}\circ\ell^{2u_{13}}_{u_{13}})$
10 : $\mathrm{GT}(\mathrm{diag}\left(N_{2u_9}\right)\mathbf{RDFT\text{-}3}^{\top}_{2u_9}\,\mathrm{rcdiag}(\mathrm{pre}({u_4}^{\mathbb{Z}\times 2u_9\to\mathbb{R}})),\ h^{2u_9\to u_6}_{u_7,\ 1,u_8}\circ\ell^{2u_9}_{u_9},\ r^{2u_9\to u_{10}}_{u_{12},\ u_{13},1,\ u_{14}},\ \{\overline{2}\})$
11 : $\mathrm{GT}(\mathbf{RDFT\text{-}3}_{u_1}\,\mathrm{diag}\left(N_{u_1}\right),\ r^{u_1\to u_3}_{u_5,\ u_6,1,\ u_7},\ h^{u_1\to u_9}_{u_{10},\ u_{11},1},\ \{\overline{2}\})$
12 : $\mathrm{GT}(\mathrm{diag}\left(C_{u_1}\right)\mathbf{rDFT}_{2u_1}(\lambda\text{-}\mathrm{wrap}({\lambda_1}^{\mathbb{Z}\times\mathbb{Z}\to\mathbb{R}})),\ h^{2u_1\to u_4}_{0,\ 1,u_5},\ h^{2u_{10}\to u_7}_{u_8,\ u_9}\circ(r^{u_1\to u_{10}}_{0,\ u_{12},1,\ u_{13}}\otimes\imath_2),\ \{u_{14}\})$
13 : $\mathrm{GT}(\mathrm{diag}\left(C_{u_1}\right)\mathbf{rDFT}_{2u_1}(\lambda\text{-}\mathrm{wrap}({\lambda_1}^{\mathbb{Z}\times\mathbb{Z}\times\mathbb{Z}\to\mathbb{R}})),\ h^{2u_1\to u_4}_{u_5,\ u_6,1,u_7},\ h^{2u_{12}\to u_9}_{u_{10},\ u_{11},1}\circ(r^{u_1\to u_{12}}_{0,\ u_{14},0,1,\ u_{15}}\otimes\imath_2),\ \{\overline{2},u_{16}\})$
14 : $\mathrm{GT}(\mathbf{RDFT\text{-}3}_{u_1}\,\mathrm{diag}\left(N_{u_1}\right),\ r^{u_1\to u_3}_{u_5,\ u_6,1,u_7,\ u_8},\ h^{u_1\to u_{10}}_{u_{11},\ u_{12},1,u_{13}},\ \{\overline{2},u_{14}\})$
15 : $\mathrm{GT}(\mathbf{RDFT\text{-}3}_{u_1}\,\mathrm{diag}\left(N_{u_1}\right),\ r^{u_1\to u_3}_{u_5,\ u_6,u_7,\ u_8},\ h^{u_1\to u_{10}}_{0,\ u_{11},1},\ \{u_{12}\})$
16 : $S(h^{2u_5\to u_2}_{u_3,\ u_4}\circ(r^{u_6\to u_5}_{u_7,\ u_8,\ u_9}\otimes\imath_2))\mathrm{diag}\left(C_{u_6}\right)\mathbf{rDFT}_{2u_6}(\lambda\text{-}\mathrm{wrap}({\lambda_1}^{\mathbb{Z}\to\mathbb{R}}))G(h^{2u_6\to u_{13}}_{u_{14},\ 1})$
17 : $\mathrm{GT}(\mathrm{diag}\left(C_{u_1}\right)\mathbf{rDFT}_{2u_1}(\lambda\text{-}\mathrm{wrap}({\lambda_1}^{\mathbb{Z}\times\mathbb{Z}\to\mathbb{R}})),\ h^{2u_1\to u_4}_{u_5,\ u_6,1},\ h^{2u_{11}\to u_8}_{u_9,\ u_{10},1}\circ(r^{u_1\to u_{11}}_{u_{13},\ u_{14},\ u_{15}}\otimes\imath_2),\ \{\overline{2}\})$

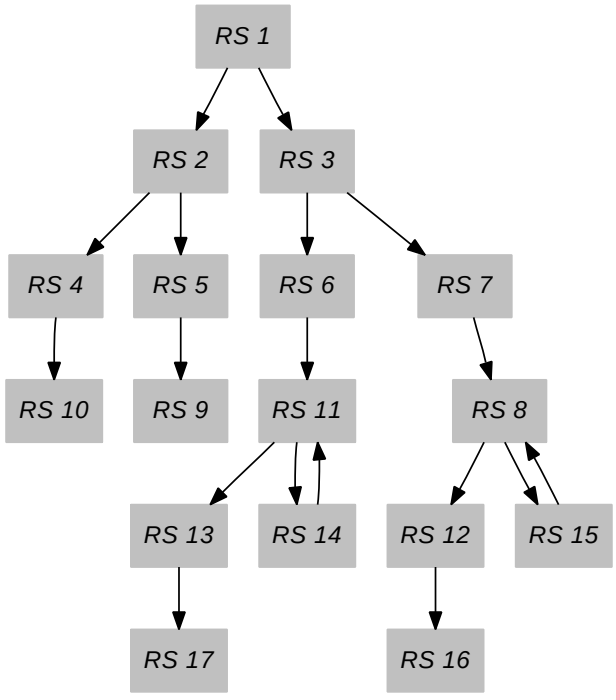


Table 4.4: Recursion step closure and call graph for the 2-way vectorized DCT-4.

4.2 Parallelization

4.2.1 Background

The need for threads. After years of exponential growth, the CPU frequencies of recent generations of microprocessors have practically stalled, a consequence of the physical limits imposed by their power density. To keep Moore's Law on track, chip makers have started to follow a different route: multicore processors, also called chip multiprocessors (CMPs), that integrate multiple processor cores onto one chip. Dual- and quad-core processors are currently sold by Intel, IBM, and AMD. IBM's Cell processor has eight special-purpose cores on one chip. The latest generation of multicore processors already targets consumer desktop and laptop computers, making multicore a mainstream technology.

With this in mind, threading support becomes a defacto requirement of any high-performance library. Programmers in charge of developing high performance libraries are already confronted with the difficult task of optimizing for deep memory hierarchies and extracting the fine-grain parallelism for vector instruction sets. Now, this challenge is compounded with extracting coarse-grain parallelism and multithreaded programming.

Most of the multicore processors are built with parallel *shared memory* architecture, which means that several processor cores have a single shared main memory, but can possibly have private caches. Shared memory architectures impose the shared memory programming model, which means that no explicit communication is required in software, and the threads communicate by accessing the same memory locations. In this thesis we only consider shared memory architectures.

Compilers. Today's parallelizing compilers grew out of a vast body of research in in the late 1980's – early 90's [12, 66, 134, 139]. The resulting compilers are quite successful and provide good performance scaling for relative simple programs. However, any highly optimized library supporting vector instructions and exploiting the memory hierarchy is far from being simple. Unfortunately, parallelizing a large multifunction program into coarse grain threads is still beyond the reach of any parallelizing compiler.

Nevertheless, the compilers can greatly help with parallel programming by providing a simple and portable way to create and control threads. In this thesis we used the OpenMP language extension [35], which is a great example of this. OpenMP extends C/C++ (or Fortran) by directives inlined into the source code as preprocessor pragmas (C/C++) or comments (Fortran) to pass parallelization information to the compiler and also includes a supporting runtime library.

Our solution. Building on our previous work we extend the library generator to automatically generate threaded code. We follow the same general approach as with vectorization, starting with the formula tags and tagged SPL constructs, and finally obtaining parallel code.

As with vectorization, we first show how to formally derive threaded transform algorithms suitable for shared memory parallelism by rewriting the SPL formulas. Next we extend this approach to $\sum$-SPL, which makes it more generally applicable, and covers a larger set of transforms.

The first approach, operating at the SPL level, is applicable only to a subset of transforms and algorithms, but enables us to reason about desirable properties; in particular, we can prove that the algorithms offer perfect load-balancing and avoid false sharing. The second approach is more generally applicable, but the makes the analysis more difficult.

We will sketch the SPL parallelization to give the point of reference, but mostly focus on $\sum$-SPL parallelization, which is used for generating libraries.

Challenges of programming for shared memory multiprocessors. Writing fast parallel

programs is considerably more challenging than writing fast sequential programs. For problems that are data parallel (but not embarrassingly parallel) the programmer (or the library generator) must address the following issues:

1) *Load balancing:* All processors should have an equal amount of work assigned. In particular, sequential parts should be avoided since they limit the achievable speed-up due to Amdahl's law.

2) *Synchronization Overhead:* Synchronization should involve as little overhead as possible and threads should not wait unnecessarily at synchronization points.

3) *Avoiding false sharing:* Private data of different processors should not be in the same cache line, since this leads to cache thrashing and thus severe performance degradation.

Usually, in addition to the above, special care must be taken to avoid excessive locking of shared variables, deadlocks and race conditions. However, Spiral generated libraries do not require locks, and are free of deadlocks and race conditions by construction, because only loops with fully independent iterations are parallelized.

Parallel transform algorithms. Most of the previous work on parallel transform algorithms concentrates on parallel DFTs. [70] shows how to design parallel DFT algorithms for various architecture constraints using the Kronecker product formalism and is a major influence on our work. [122] gives a good overview of sequential and parallel DFT algorithms. The major problem with using the standard Cooley-Tukey FFT algorithm (2.1) on shared memory machines is its memory access pattern: Large strides, and consecutive loop iterations touch the same cache lines, which leads to false sharing.

The governing idea of many parallel algorithms [10,87,106] is to reorder the data in explicit steps to remove false sharing introduced by strided memory access. For example, the six-step algorithm,

$$\mathbf{DFT}_{mn} \rightarrow L^{mn}_m (I_n \otimes \mathbf{DFT}_m) L^{mn}_n D_{m,n} (I_m \otimes \mathbf{DFT}_n) L^{mn}_m \tag{4.12}$$

has embarrassingly parallel computation stages of the form $I_r \otimes \mathbf{DFT}_s$. The three stride permutations in (4.12) are executed separately as explicit matrix transpositions, i.e., data permutations. These transpositions are further optimized, e.g., through blocking [3], and partially folded into the adjacent computation stages [117, 118]. A different optimization approach reduces communication by increasing the computation by using $O(n^2)$ algorithms instead of fast $O(n \log n)$ algorithms to remove dependencies on small subproblems [4].

FFTW 3 [61] offers a state-of-the-art multithreading implementation of the DFT and related transforms. FFTW parallelizes one- and multi-dimensional DFTs by allowing its search mechanism to parallelize the loops that occur inside the algorithms. It uses advanced loop optimizations to avoid cache problems (loop distribution and loop exchange) and supports thread pooling to minimize the startup cost of parallel computation. Hand-written implementation of threading in FFTW is roughly equivalent to our parallelization on the $\sum$-SPL level, which also works by parallelizing loops.

4.2.2 Parallelization by Rewriting SPL Formulas

In this section we explain our first approach, which parallelizes SPL formulas. The parallelization of SPL formulas is performed similarly to vectorization. We notice that SPL constructs have a direct

interpretation in terms of parallel code. For example, the tensor products in (2.1) are essentially loops with fully independent iterations (no loop-carried dependencies) and known memory access patterns. Next, we automatically rewrite a Spiral generated formula to obtain a structure suitable for mapping into efficient multi-threaded code.

A formula fully determines the memory access of the final program (as a function of the loop variables), and thus using rewriting we can statically schedule the loop iterations across p processors to ensure load balancing and eliminate false sharing. For general programs, proving the independence of loop iterations and determining such a schedule is a hard problem requiring expensive analysis [12]. The implied loops in SPL formulas, on the other hand, have by definition independent iterations, and known memory access patterns. Thus, as we show, we can find such a desired schedule efficiently.

We first explain the parallelizing rewriting system, and then show the application to the DFT, effectively deriving a novel variant of the Cooley-Tukey FFT (2.1) different from (4.12) and well-suited for multicore systems.

The extension of Spiral to support shared memory parallelism requires four components:

- *Parallelism tags* which introduce the relevant hardware parameters into the rewriting system.
- *Parallel formula constructs* which denote the subformulas that can be perfectly mapped to shared memory parallel platforms.
- *Rewriting rules* which transform general formulas into parallel formulas.
- *Parallel backend* that maps parallel formulas into parallel C or C++ code (in our case using OpenMP extensions).

Parallelism tags. The two most relevant parameters of shared memory parallel machines are the number of processors p and the cache line length μ of the most important cache level. We measure μ in the elements of the input signal. For instance, for a cache line length of 64 bytes (= 512 bits) and complex double precision data type (= $2\times64 = 128$ bits), $\mu = 4$. These parameters are introduced into the rewriting system with a parallel tag, for example,

$$\mathbf{Par}_{p,\mu}(A).$$

In addition, we assume that all shared data vectors are aligned at cache line boundaries in the final program.

Parallel formula constructs. For arbitrary A and A_i the expressions

$$y = \big(I_p \otimes A\big)x, \quad \text{and} \quad y = \left(\bigoplus_{i=0}^{p-1} A_i\right) x, \quad \text{with} \quad A, A_i \in \mathbb{C}^{m\mu\times m\mu}$$

express embarrassingly parallel computation on p processors as they express block diagonal matrices with p blocks. Requiring the matrix dimensions to be multiples of μ ensures that during computation each cache line is owned by exactly one processor, preventing false sharing. If all A_i have the same computational cost, programs implementing these constructs become load balanced.

Data shuffling of the form

$$y = \big(P \otimes I_\mu\big)x, \quad \text{with } P \text{ permutation},$$

$$\mathbf{Par}_{p,\mu}(AB) \to \mathbf{Par}_{p,\mu}(A)\,\mathbf{Par}_{p,\mu}(B) \tag{4.15}$$

$$\mathbf{Par}_{p,\mu}(A_m \otimes I_n) \to \mathbf{Par}_{p,\mu}\left(\left(L^{mp}_m \otimes I_{n/p}\right)\left(I_p \otimes (A_m \otimes I_{n/p})\right)\left(L^{mp}_p \otimes I_{n/p}\right)\right) \tag{4.16}$$

$$\mathbf{Par}_{p,\mu}(L^{mn}_m) \to \begin{cases} \mathbf{Par}_{p,\mu}\left(I_p \otimes L^{mn/p}_{m/p}\right)\mathbf{Par}_{p,\mu}\left(L^{pn}_p \otimes I_{m/p}\right) \\ \mathbf{Par}_{p,\mu}\left(L^{pm}_m \otimes I_{n/p}\right)\mathbf{Par}_{p,\mu}\left(I_p \otimes L^{mn/p}_m\right) \end{cases} \tag{4.17}$$

$$\mathbf{Par}_{p,\mu}(I_m \otimes A_n) \to I_p \otimes_{\parallel} \left(I_{m/p} \otimes A_n\right) \tag{4.18}$$

$$\mathbf{Par}_{p,\mu}(P \otimes I_n) \to \left(P \otimes I_{n/\mu}\right)\overline{\otimes} I_\mu, \tag{4.19}$$

Table 4.5: SPL shared memory parallelization rules. P is any permutation.

reorders blocks of μ consecutive elements and thus whole cache lines are reordered. On shared memory machines this means that such reordering passes make the threads exchange ownership of entire cache lines, avoiding false sharing. Recall also, that at $\sum$-SPL level, these permutations are combined with adjacent computation blocks.

We introduce tagged versions of the tensor product and direct sum operators in Spiral:

$$I_p \otimes_{\parallel} A, \quad \bigoplus_{i=0}^{p-1}{}_{\parallel} A_i, \quad P\overline{\otimes} I_\mu, \quad \text{with } A, A_i \in \mathbb{C}^{m\mu \times m\mu}. \tag{4.13}$$

These are the same matrix operators as their untagged counterparts, but declare that a construct is fully optimized for shared memory machines and does not require further rewriting. By fully optimized we mean that the formula is load-balanced for p processors (provided the A_i have equal computational cost) and avoids false sharing. This property is preserved for products of these constructs.

Definition 2 We say that a formula is load-balanced *(*avoids false sharing*) if it is of the form* (4.13) *or of the form*

$$I_m \otimes A \quad or \quad AB, \tag{4.14}$$

where A and B are load-balanced formulas (formulas that avoid false sharing). A formula is fully optimized (for shared memory) if it is load-balanced and avoids false sharing.

The goal of the rewriting system (explained next) is to transform formulas into fully optimized formulas.

Rewriting rules. Table 4.5 summarizes the rewriting rules sufficient for parallelizing a subset of SPL formulas, including the Cooley-Tukey FFT (2.1). These rules form the core of the parallelization. All parameters in the rules are integers, and thus an expression n/p on the right-hand side of a rule implies that the precondition $p|n$ must hold for the rule to be applicable.

As an example, consider an example of applying rule (4.16) that encodes a form of loop strip-mining and scheduling.

$$F_2 \otimes I_n \xrightarrow{\text{apply (4.16)}} \left(L^{2p}_2 \otimes I_{n/p}\right)\left(I_p \otimes (F_2 \otimes I_{n/p})\right)\left(L^{2p}_p \otimes I_{n/p}\right). \tag{4.20}$$

The left-hand side, represents a loop with unit stride between iterations. On the right-hand side,

the construct $I_p \otimes (F_2 \otimes I_{n/p})$ corresponds to a double loop with p iterations in the outermost, and n/p in the innermost loop; the permutations $L^{2p}_2 \otimes I_{n/p}$ and $L^{2p}_p \otimes I_{n/p}$ represent data reindexing which is merged with the tensor product by Spiral's loop merging stage (Section 2.4).

To produce the final code, Spiral further applies rules (4.15) and (4.17)–(4.19) and performs loop merging. The resulting pseudo-code for the is shown below.

```
for(int tid = 0; tid < p; ++tid) { /* parallel loop, thread id = tid */
    for(int j = 0; j < n/p; ++j) {
        y[tid*n/p + j]     = x[tid*n/p + j] + x[tid*n/p + j + n];
        y[tid*n/p + j + n] = x[tid*n/p + j] - x[tid*n/p + j + n];
```

Above the p iterations of outer loop are distributed among p threads. Each processor executes n/p consecutive iterations of the original loop given by the left-hand side of (4.20) and touches 2 contiguous memory areas of n/p complex numbers. If $\mu|(n/p)$, then each processor "owns" $2n/(p\mu)$ cache lines.

Rule (4.15) expresses that in products of matrices each factor will be rewritten separately. (4.16) and (4.18) handle tensor products with identity matrices. Both rules distribute the computational load evenly among the p processors and execute as many consecutive iterations as possible on the same processor (as shown above). Rule (4.17) breaks stride permutations into two stages: one performs stride permutations locally for each processor, the other permutes consecutive chunks of data. (4.16) and (4.17) require the subsequent application of (4.15), (4.18), and (4.19) to fully break down to parallel formula constructs (4.13). Tensor products of a permutation and a sufficiently large identity matrix are broken into cache line resolution by (4.19).

The rules in Table 4.5 are based on known formula identities summarized in [51, 53, 70]. They replace the usually expensive analysis required for the associated loop transformations by cheap pattern matching and also encode the actual transformation.

Parallel backend. Extending Spiral's implementation level to support shared memory parallel code is straightforward. The only thing we have to add is the translation of the constructs

$$I_p \otimes_{\parallel} A \quad \text{and} \quad \bigoplus_{i=0}^{p-1}{}_{\parallel} A_i$$

into parallel code for p threads. There is no need for special support for $P\overline{\otimes}I_\mu$ as these permutations are simply merged with the adjacent loops by $\sum$-SPL optimizations.

We use OpenMP to generate parallel C code. The variables defined outside of the parallel constructs must be shared between the threads, and variables defined inside must be thread local. Since this is the default behavior of OpenMP, no special flags are needed. For example, the loop corresponding to $y = (I_p \otimes_{\parallel} F_2)x$ is implemented in OpenMP as:

```
/* one iteration per thread */
#pragma omp parallel num_threads(p)
{
    int tid = omp_get_thread_num();

    /* apply F_2 to subvectors of x */
    y[2*tid]     = x[2*tid] + x[2*tid + 1];
    y[2*tid + 1] = x[2*tid] - x[2*tid + 1];

    /* wait until all threads finish work */
```

```
    #pragma omp barrier
}
```

Note, that we do not use the "parallel for" construct provided by OpenMP, because implementation using barriers turns out to be faster, and more flexible.

Example: parallelizing the Cooley-Tukey FFT. Now, we apply the rewrite rules from Table 4.5 to derive a parallel version of the Cooley-Tukey FFT (2.1). In Spiral these steps are performed automatically. The result is a version of the Cooley-Tukey FFT that is fully optimized for shared memory in the sense of Definition 2.

We start by specifying that we want to compute $y = \mathbf{DFT}_N\, x$ on p processors with cache line size μ,

$$\mathbf{Par}_{p,\mu}\left(\mathbf{DFT}_N\right).$$

In the first step rule (2.1) chooses a factorization of $N = mn$ and breaks $\mathbf{DFT}_N$ into $\mathbf{DFT}_m$ and $\mathbf{DFT}_n$. Next, rule (4.15) propagates the parallelization tag to all factors:

$$\begin{aligned}\mathbf{Par}_{p,\mu}\left(\mathbf{DFT}_{mn}\right) &\rightarrow \mathbf{Par}_{p,\mu}\left(\left(\mathbf{DFT}_m \otimes I_n\right) D_{m,n} \left(I_m \otimes \mathbf{DFT}_n\right) L^{nm}_m\right) &(4.21)\\ &\rightarrow \mathbf{Par}_{p,\mu}\left(\mathbf{DFT}_m \otimes I_n\right) \mathbf{Par}_{p,\mu}\left(D_{m,n}\right) \mathbf{Par}_{p,\mu}\left(I_m \otimes \mathbf{DFT}_n\right) \mathbf{Par}_{p,\mu}\left(L^{nm}_m\right). &(4.22)\end{aligned}$$

We now consider each of the four factors on the right-hand side of (4.22) separately. The first factor is rewritten by applying (4.16) (requiring $p|n$),

$$\mathbf{Par}_{p,\mu}\left(\mathbf{DFT}_m \otimes I_n\right) \rightarrow \mathbf{Par}_{p,\mu}\left(L^{mp}_m \otimes I_{n/p}\right) \mathbf{Par}_{p,\mu}\left(I_p \otimes \left(\mathbf{DFT}_m \otimes I_{n/p}\right)\right) \mathbf{Par}_{p,\mu}\left(L^{mp}_p \otimes I_{n/p}\right),$$

followed by (4.15), (4.18), and (4.19) (requiring $\mu|n/p$),

$$\mathbf{Par}_{p,\mu}\left(\mathbf{DFT}_m \otimes I_n\right) \rightarrow \left(\left(L^{mp}_m \otimes I_{n/p\mu}\right)\overline{\otimes} I_\mu\right)\left(I_p \otimes_\parallel \left(\mathbf{DFT}_m \otimes I_{n/p}\right)\right)\left(\left(L^{mp}_p \otimes I_{n/p\mu}\right)\overline{\otimes} I_\mu\right). \quad (4.23)$$

The second factor in (4.22) is a diagonal which will be merged into the tensor product, thus the tag is simply dropped, and the third factor is handled using (4.18) (requiring $p|m$),

$$\mathbf{Par}_{p,\mu}\left(I_m \otimes \mathbf{DFT}_n\right) \rightarrow I_p \otimes_\parallel \left(I_{m/p} \otimes \mathbf{DFT}_n\right). \quad (4.24)$$

The remaining fourth factor in(4.22) is parallelized by the first choice of rule (4.17) (requiring $p|m$) followed by (4.15), (4.18), and (4.19) (requiring $\mu|m/p$),

$$\begin{aligned}\mathbf{Par}_{p,\mu}\left(L^{mn}_m\right) &\rightarrow \mathbf{Par}_{p,\mu}\left(I_p \otimes L^{mn/p}_{m/p}\right) \mathbf{Par}_{p,\mu}\left(L^{pn}_p \otimes I_{m/p}\right) &(4.25)\\ &\rightarrow \left(I_p \otimes_\parallel L^{mn/p}_{m/p}\right)\left(\left(L^{pn}_p \otimes I_{m/p\mu}\right)\overline{\otimes} I_\mu\right). &(4.26)\end{aligned}$$

Collecting (4.23)–(4.26) and the constraints required for applying the rules leads to the final expression output by our rewriting system, (4.27) displayed in Figure 4.3, with the requirement $p\mu|m$ and $p\mu|n$. Inspection shows that (4.27) is fully optimized for shared memory in the sense of Definition 2.

As a small example, Figure 4.4 shows the C99 OpenMP program generated by Spiral from (4.27) with $m = 4$, $n = 2$, $p = 2$, and $\mu = 2$ after loop merging and fully unrolling the code for $\mathbf{DFT}_2$ and $\mathbf{DFT}_4$.

Discussion. Parallelization on the SPL level only works for a subset of SPL formulas. In

$$\mathbf{Par}_{p,\mu}(\mathbf{DFT}_{mn}) \rightarrow \big((L^{mp}_m \otimes I_{n/p\mu})\overline{\otimes} I_\mu\big)\big(I_p \otimes_\parallel (\mathbf{DFT}_m \otimes I_{n/p})\big)\big((L^{mp}_p \otimes I_{n/p\mu})\overline{\otimes} I_\mu\big)$$
$$D_{m,n}\Big(I_p \otimes_\parallel (I_{m/p} \otimes \mathbf{DFT}_n)\Big)\big(I_p \otimes_\parallel L^{mn/p}_{m/p}\big)\big((L^{pn}_p \otimes I_{m/p\mu})\overline{\otimes} I_\mu\big) \quad (4.27)$$

Figure 4.3: Multicore Cooley-Tukey FFT for p processors and cache line length μ.

```
// C99 OpenMP DFT_8, call by a sequential function
#include <omp.h>

static _Complex double D[8] = {
  1, 1, 1, 0.70710678118654+__I__*0.70710678118654,
  1, 1, 1, -0.70710678118654+__I__*0.70710678118654
};

void DFT_8(_Complex double *Y, _Complex double *X) {
  static _Complex double T[8];
  #pragma omp parallel for
  for(int i1 = 0; i1 <= 1; i1++) {
    _Complex double s1, s2;
    for(int i2 = 0; i2 <= 1; i2++) {
      s1 = X[2*i1 + i2];
      s2 = X[4 + 2*i1 + i2];
      T[4*i1 + 2*i2] = s1 + s2;
      T[4*i1 + 2*i2 + 1] = s1 - s2;
    }
  }
  #pragma omp parallel for
  for(int i3 = 0; i3 <= 1; i3++) {
    _Complex double s3, s4, s5, s6, s7, s8, s9, s10;
    s10 = D[i3]*T[i3];
    s9 = D[4 + i3]*T[4 + i3];
    s8 = s10 + s9;
    s4 = D[6 + i3]*T[6 + i3];
    s7 = D[2 + i3]*T[2 + i3];
    s6 = s7 + s4;
    s5 = s10 - s9;
    s3 = __I__*(s7 - s4);
    Y[i3] = s8 + s6;
    Y[4 + i3] = s8 - s6;
    Y[2 + i3] = s5 + s3;
    Y[6 + i3] = s5 - s3;
  }
}
```

Figure 4.4: Multithreaded C99 OpenMP function computing $y = \mathbf{DFT}_8\, x$ using 2 processors, called by a sequential program.

particular, it is possible to parallelize Cooley-Tukey FFT, multi-dimensional transforms, WHT, and Haar wavelet (which uses 2-tap filters).

4.2.3 Parallelization by Rewriting $\sum$-SPL Formulas

To port SPL parallelization to $\sum$-SPL we need to convert the tensor product parallelization breakdown rules to GT breakdown rules. Since GT can describe a larger formula space, the GT breakdown rules will need to be more general.

Similarly to tensor products, the GT loops are, in parallel Fortran terminology, "DOALL" loops, which means that they have independent iterations and are trivially parallelizable. So the more important questions are 1) which loops to parallelize and 2) how to avoid false sharing and ensure load balancing.

Currently, our strategy is to parallelize the top-level loops only. This is not necessarily always the best strategy. Sometimes it may make sense to parallelize deeper in the loop nest to effectively exploit the shared caches. This can be rather easily accomodated by adding additional breakdown rules, and having the library runtime adaptation mechanism select the best strategy.

False sharing is avoided in the same way as parallelizing SPL formulas, namely by statically scheduling the threads in the special way. In some cases, this is does not avoid false sharing entirely, but provides a good heuristic.

We now discuss how this works in more detail. In order to parallellize by rewriting $\sum$-SPL formulas, we will need the following components, which we explain next.

1. *Parallelization tags* (these are the same as before).
2. *Parallel $\sum$-SPL formula constructs*, which denote formulas with parallel interpretation, in this case we also need few more low-level constructs, such as a barrier.
3. *Rewrite rules* that manipulate general $\sum$-SPL formulas into parallel ones.
4. *Parallel backend* that emits parallel OpenMP code from parallel $\sum$-SPL constructs (it is a straightforward extension of the backend explained in Section 4.2.2, and we do not discuss it here).

Parallelization tags. We will use the same parallelization tag as before, to denote parallelization:

$$\mathbf{Par}_{p,\mu}(A).$$

Parallel $\sum$-SPL constructs. The following parallel $\sum$-SPL constructs are needed

- Parallel sum, expresses a loop which iterations must be executed in parallel:

$$\sum_{\|,i=0}^{p-1} A_i$$

- Barrier, threads wait until all peers finish computing A:

$$\text{barrier}(A)$$

Rewrite rules:

$$\mathbf{Par}_{p,\mu}\left(AB\right) \rightarrow \text{barrier}\left(A\right)\text{barrier}\left(B\right), \tag{4.28}$$

$$\mathbf{Par}_{p,\mu}\left(A+T\right) \rightarrow \text{tid}_{p-1}\left(A\right) + \mathbf{Par}_{p,\mu}\left(T\right). \tag{4.29}$$

Breakdown rules:

$$\mathbf{Par}_{p,\mu}\left(T\right) \xrightarrow[p|n]{\text{GT-Par}} \sum_{\|,j=0}^{p-1} d(T,j), \tag{4.30}$$

$$\mathbf{Par}_{p,\mu}\left(T\right) \xrightarrow[p\nmid n]{\text{GT-ParND}} \sum_{\|,j=0}^{p-2} d(T,j) + \text{tid}_{p-1}\left(\text{GT}(d(A,p-1),\ d(f,p-1),\ d(g,p-1),\ \{n \bmod p\})\right), \tag{4.31}$$

$$\mathbf{Par}_{p,\mu}\left(S\right) \xrightarrow[p|n]{\text{GTI-Par}} \sum_{\|,j=0}^{p-1} d(S,j), \tag{4.32}$$

$$\mathbf{Par}_{p,\mu}\left(S\right) \xrightarrow[p\nmid n]{\text{GTI-ParND}} \sum_{\|,j=0}^{p-2} d(S,j) + \text{tid}_{p-1}\left(\text{GTI}(d(A,p-1),\ d(f,p-1),\ \{n \bmod p\})\right) \tag{4.33}$$

$$d(a,x) = \text{down}(\text{split}(a,1,\lceil n/p\rceil),x,2) \tag{4.34}$$

$$\tag{4.35}$$

Table 4.6: $\sum$-SPL shared memory parallelization rules. $T = \text{GT}(A,\ f,\ g,\ \{n\})$, $S = \text{GTI}(A,\ f,\ \{n\})$.

- Thread ID assignment, A must execute on thread i only:

$$\text{tid}_i(A)$$

Rewrite and breakdown rules. Parallelization is a achieved via a mixture of breakdown and rewrite rules. As earlier, breakdown rules are used when alternatives exist, and provide the possibility to search over the alternatives.

Table 4.6 shows the rewrite and breakdown rules to parallelize at the $\sum$-SPL level with GT nonterminals. We provide parallelization for the cases when number of processors p does not divide the number of loop iterations n.

The rules that we show assume that parallelized GTs are rank 1. In Spiral we implemented a general version of these rules, that always parallelizes the outer loop.

The rationale of all GT loop parallelization rules is to assign to each thread an equal size chunk of consecutive iterations. This coincides with the strategy that OpenMP uses to parallelize 'parallel for' loops, and also with the guaranteed false sharing free rules of SPL parallelization. The iterations space partition is shown in Fig. 4.5. When parallelizing GT loops, the false sharing free

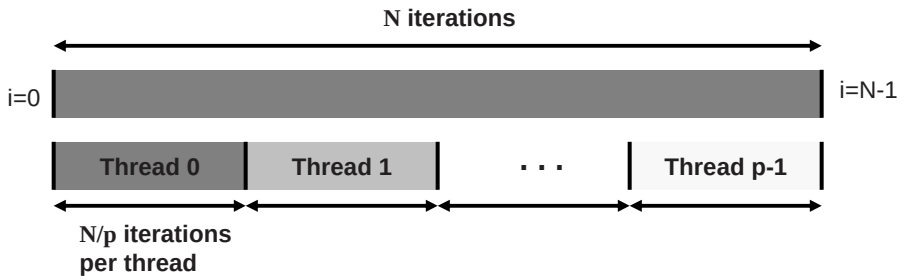


Figure 4.5: GT iteration space partition implied by GT parallelization rules (4.30)–(4.33).

guarantee only holds for specific index mapping functions, for other index mapping functions it is just a heuristic, which tends to work very well.

To find the loops to parallelize the first pair of rewrite rules are used. Note, that there is no general rule for $\mathbf{Par}_{p,\mu}(A+B)$, but only for the case when B is a GT nonterminal (i.e. rule (4.29)). The reason is twofold. First, we did not need such a rule for the transform breakdown rules that were used to generate libraries (Table 6.1), and a special variant (4.29) was sufficient. Second, it is no longer clear how to statically schedule more than 2 threads with summation like $A+B$. The best strategy will most likely be dispatching portion of threads to compute A and another portion to compute B and parallelizing within A and B. The only way to find the best schedule would be to use multiple breakdown rules (instead of rewrite rules) to enable the search at runtime.

4.2.4 Parallelized Closure Example

Table 4.7 shows the closure generated for $\mathbf{Par}_{p,\mu}(\mathbf{DFT})$ implemented via the Cooley-Tukey breakdown rule (DFT-CT). The following breakdown rules were used to construct the closure: DFT-CT (2.1), GT-DownRank (3.47), GT-Par (4.30), GT-ParND (4.31). The non-parallelized equivalent is the closure in Table 3.11 and Fig. 3.8(a).

The shown parallel closure has 8 recursion step, while the non-parallelized closure in Table 3.11 has 7.

$1:\ \mathrm{Par}_2(\mathbf{DFT}_{u_1})$

$2:\ \mathrm{Par}_2(\mathrm{GT}(\mathbf{DFT}_{u_1},\ h^{u_1\to u_3}_{0,\ u_4,1},\ h^{u_1\to u_6}_{0,\ 1,u_7},\ \{u_8\}))$

$3:\ \mathrm{Par}_2(\mathrm{GTI}(\mathbf{DFT}_{u_1}\mathrm{diag}\left(\mathrm{pre}({u_3}^{\mathbb{Z}\times u_1\to\mathbb{C}})\right),\ h^{u_1\to u_5}_{0,\ u_6,1},\ \{u_7\}))$

$4:\ \mathrm{GT}(\mathbf{DFT}_{u_1},\ h^{u_1\to u_3}_{u_4,\ u_5,1},\ h^{u_1\to u_7}_{u_8,\ 1,u_9},\ \{u_{10}\})$

$5:\ \mathrm{GTI}(\mathbf{DFT}_{u_1}\mathrm{diag}\left(\mathrm{pre}({u_3}^{\mathbb{Z}\times u_1\to\mathbb{C}})\right),\ h^{u_1\to u_5}_{u_6,\ u_7,1},\ \{u_8\})$

$6:\ \mathrm{GTI}(\mathbf{DFT}_{u_1}\mathrm{diag}\left(\mathrm{pre}({u_3}^{u_1\to\mathbb{C}})\right),\ h^{u_1\to u_5}_{u_6,\ u_7},\ \{\})$

$7:\ S(h^{u_1\to u_2}_{u_3,\ 1})\mathbf{DFT}_{u_1}G(h^{u_1\to u_6}_{u_7,\ u_8})$

$8:\ \mathrm{GT}(\mathbf{DFT}_{u_1},\ h^{u_1\to u_3}_{u_4,\ u_5,u_6},\ h^{u_1\to u_8}_{u_9,\ 1,u_{10}},\ \{u_{11}\})$

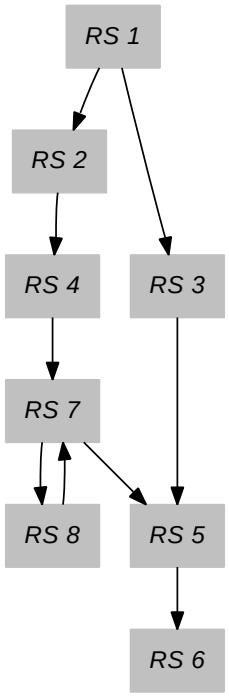


Table 4.7: Recursion step closure and call graph for the 2 thread parallelized DFT. Compare to the DFT closure in Table 3.11 and call graph in Fig. 3.8(a).

Chapter 5

Library Generation: Library Implementation

In Chapter 3 we described the *Library Structure* module of the library generation framework (see Fig. 3.1), which was responsible for generating the recursion step closure, and performing large scale structural optimizations for parallelization and vectorization. The generated recursion step closure defines the final structure of the library in an abstract way. It can also be viewed as a program for the hypothetical $\sum$-SPL virtual machine, which can interpret formulas.

The goal of the *Library Implementation* module is to translate or compile the program for the hypothetical $\sum$-SPL virtual machine into a program in some lower level target language. However, given the declarative and very high-level nature of $\sum$-SPL there are many different ways in which a recursion step closure can be compiled into a library.

In particular, there are different kinds of libraries, for example, adaptive and fixed, floating-point and fixed-point, etc. Our goal is to be able to generate all of these kinds of libraries. We will call the particular set of requirements for the generated library together with the target language the *library target*.

As part of this thesis we implemented both adaptive and fixed C++ floating-point libraries. Other potentially interesting library targets are extensions of existing libraries with generated code; fixed-point arithmetic based libraries; and libraries in C or Java.

It is also useful to choose an existing library, such as Intel IPP, as the library target. In this case the generated code is meant to extend the existing library with new functionality. Generating new code for an existing library requires the generated code to follow certain established coding and naming conventions, and satisfy other requirements. These requirements must be incorporated into the library target specification.

Because the *Library Implementation* module does not require any transform specific code, if support is added for a new library target, one can immediately generate a high-performance library for the new target. At the same time the formula level transformations can proceed independently of the chosen library target.

5.1 Overview

At the high level the task of this module is to combine the transform recursion and base cases within the library infrastructure of the chosen library target. This boils down to generating the

"glue code", which will address the following problems:

- *Degree of freedom in choosing the breakdown rules.* If more than one breakdown rule is applicable to the recursion step, there needs to be a dispatch mechanism which can select which method is invoked at runtime.

- *Integration of base cases.* In order for the recursion to eventually terminate, the code for the base cases needs to be integrated into the dispatch mechanism.

- *Degrees of freedom within the breakdown rule.* The dispatch mechanism must be able to choose the appropriate values for degrees of freedom, such as the value of $k|n$ in (3.1).

- *Initialization code.* Each transform requires some initialization code. The tasks of this code include memory allocation for the temporary buffer, and scaling factors, precomputation of scaling factors, and so forth.

We would like to enable program generation for different library targets. Each library target handles the above issues differently. Possible library targets include:

1. A standalone non-adaptive library.

2. A standalone adaptive library.

3. An extension of an existing non-adaptive (e.g., Intel MKL) or adaptive library (e.g., FFTW).

In the case of standalone libraries, some infrastructure must be provided (i.e. memory management, error handling, and, for adaptive libraries, search capabilities) to be used along with the generated transform and glue code.

Discussion. If we compare our implementation goal above with existing FFT libraries, we get a hint towards the solution.

In virtually all FFT libraries the degrees of freedom within a breakdown rule are chosen according to some static strategy. However, the strategy itself is inseparable from the implementation. The choice of breakdown rules is typically handled by cascading IF statements, which encode the static strategy.

Notable exceptions are FFTW [61] and UHFFT [84]. These libraries make the choices for the degrees of freedom based on runtime feedback. Both libraries build up a *plan* before computing the transform. Among other things, the plan stores the necessary buffers and precomputed constants.

The handling of initialization code depends on whether the library is adaptive or not. In adaptive libraries, initialization uses the plan to figure out what to precompute and how many buffers are needed, and also for storing initialization data. In adaptive libraries the initialization is most conveniently expressed recursively. In non-adaptive libraries there is no universal standard.

Our approach. We will unify the adaptive and non-adaptive library targets by using a plan-like recursive data structure in both cases, and keeping the choice of values for the degrees of freedom orthogonal to other issues. In adaptive libraries the plan can built up dynamically, and in fixed libraries the plans may be fixed, or use the fixed hard-coded strategy. Besides the recursion strategy, the plan also serves as a convenient place to store initialization data and allocate temporary memory buffers used in the computation.

5.2 Recursion Step Semantics as Higher-Order Functions

The descriptor system. Ideally, the recursion step closure should translate into a set of mutually recursive functions. In reality, however, things are more complicated.

Mathematically, the semantics parametrized recursion step is a function of the form

$$(u_1 \times \cdots \times u_n \times X) \to Y.$$

For example, the semantics of the simple recursion step $\mathbf{DFT}_{u_1}$ is a function

$$(u_1 \times X) \to Y, \quad u_1 \in \mathbb{N},\ X, Y \in \mathbb{C}^{u_1}$$

However, it is desired that some of the parameters, for example the transform size, are provided earlier in order to precompute some constants used in the algorithm. Precomputation allows to compensate expensive trigonometric computations if the transform of the same length is computed several times. This is standard practice, used in Intel IPP library, FFTW, and others. Computing the 32-point DFT using IPP, for example, looks as follows:

```
IppsDFTSpec_C_64fc *f;
ippsDFTInitAlloc_C_64fc(f, 32, IPP_FFT_NODIV_BY_ANY, ippAlgHintNone);
ippsDFTFwd_CToC_64fc(X, Y, f, NULL);
```

The first two lines initialize the so-called "DFTSpec" structure `f`, setting the transform size to 32 and selecting unscaled transform via `IPP_FFT_NODIV_BY_ANY`. The third line computes the transform of `X` and writes the result into `Y`. Casting this back into the recursion step framework, the parameters of the recursion step become available at different times.

This temporal discontinuity can be accommodated by currying the semantics function, as in

$$u_1 \to (X \to Y).$$

In the general case, both inner and outer functions have several parameters. We will call the parameters of the outer function "cold" (above u_1 is cold) and the parameters of the inner function "hot" (above X is hot). Thus we have:

$$\mathit{coldparams} \to (\mathit{hotparams} \to Y), \quad \text{params} = \{\mathit{coldparams}\} \cup \{\mathit{hotparams}\}.$$

It is the second occasion on which the higher-order functions (functions returning functions) naturally come into play. The first use of higher-order functions is in parametrizing the generating functions of $\sum$-SPL diag$(\cdot)$ construct, as described in Section 3.3.3.

Clearly, many target languages, such as C or C++, do not directly support higher-order functions. Our strategy is to generate intermediate code that uses higher-order functions, and then eliminate them using the process known as *closure conversion*[1] or *lambda lifting* [63, 71]. As the result, the descriptor system used in Intel IPP and FFTW naturally arises.

In object-oriented languages closures can be expressed with objects (and vice versa [63, 73]). For example, the function $f : u_1 \to (X \to Y)$ can be modeled as the object of the following C++ class:

[1] "Closure" in this context 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 [132]. It is in no way related to the recursion step closure, which is a mathematical set closure [133].

```
class func {
    int u1;
    func(int u1) {
        this->u1 = u1;
    }
    void compute(double *X, double *Y) {
        ...u1 can be used...
    }
};
```

Calling such function is a two step process, identical to the descriptor system of many transform libraries:

```
func f(4);
f.compute(X, Y);
```

Summary. As we now explained how the higher-order functions naturally come into play, we briefly what this means for the compilation:

- Since $\sum$-SPL recursion steps are mutually recursive, we have a set of mutually recursive functions.
- Due to desired temporal discontinuity all of the functions are actually higher-order.
- If the target language does not directly support higher-order functions, they must be implemented with the help of *lexical closures*.
- The process is called *closure conversion* or λ-lifting [63, 71]. We have already used λ-lifting in Chapter 3 for a different purpose, namely to design index-free $\sum$-SPL.
- Our target is C++, it is known that closures and objects are equivalent [73], thus thus closures will be represented as objects in C++.

Overall flow. We show overall flow of the *Library Implementation* block in Fig. 5.1, and explain the individual steps next.

5.3 Library Plan

The first step in the library implementation process is the building of a so-called *library plan*. Library plan is a data structure that stores a more detailed representation of the recursion step closure, and provides full and clear description of the library structure.

Recall the recursion step closure (3.15) for complex DFT based on (3.1), which we repeat below (without using $*$'s):

$$\mathbf{DFT}_{u_1} \xrightarrow{\text{Closure}} \left\{ \begin{array}{c} \mathbf{DFT}_{u_1} \\ S(h^{u_1 \to u_2}_{u_3,\,1})\,\mathbf{DFT}_{u_1}\,G(h^{u_1 \to u_6}_{u_7,\,u_8}) \\ S(h^{u_1 \to u_2}_{u_3,\,u_4})\,\mathbf{DFT}_{u_1}\,\mathrm{diag}\left(\mathrm{pre}({u_7}^{u_1 \to \mathbb{C}})\right) G(h^{u_1 \to u_9}_{u_{10},\,u_{11}}) \\ S(h^{u_1 \to u_2}_{u_3,\,1})\,\mathbf{DFT}_{u_1}\,\mathrm{diag}\left(\mathrm{pre}({u_6}^{u_1 \to \mathbb{C}})\right) G(h^{u_1 \to u_8}_{u_9,\,u_{10}}) \end{array} \right\} \tag{5.1}$$

After processing (5.1) we obtain a library plan shown in Table 5.1.

RS 1

Formula: $\mathbf{DFT}_{u_1}$
Parameters: u_1
Children: RS 2, RS 3
Breakdown 1: DFT-LibBase
* *Applicable*: $u_1 = 2$
* *Descend*: $\mathbf{DFT}_2$

Breakdown 2: DFT-CTA
* *Applicable*: $\text{isPrime}(u_1) = 0$
* *Freedoms*: $f_1 \in \text{divisors}(u_1)$
* *Descend*:

$$\sum_{i=0}^{u_1/f_1-1} \text{RS3}\left((u_2, u_1, u_3, u_4, {u_7}^{\mathbb{Z}\to\mathbb{C}}, u_9, u_{10}, u_{11}) \leftarrow (u_1, f_1, i, u_1/f_1, \Omega^1_{u_1} \circ dt \circ h^{f_1\to u_1}_{i,\ u_1/f_1}, u_1, i, u_1/f_1)\right)$$

$$\sum_{j=0}^{f_1-1} \text{RS2}\left((u_2, u_1, u_3, u_6, u_7, u_8) \leftarrow (u_1, u_1/f_1, u_1 j/f_1, u_1, j, f_1)\right)$$

RS 2

Formula: $S(h^{u_1\to u_2}_{u_3,\ 1})\mathbf{DFT}_{u_1} G(h^{u_1\to u_6}_{u_7,\ u_8})$
Parameters: $u_1, u_2, u_3, u_6, u_7, u_8$
Children: RS 2, RS 3
Breakdown 1: DFT-LibBase
* *Applicable*: $u_1 = 2$
* *Descend*: $S(h^{2\to u_2}_{u_3,\ 1})\mathbf{DFT}_2 G(h^{2\to u_6}_{u_7,\ u_8})$

Breakdown 2: DFT-CTA
* *Applicable*: $\text{isPrime}(u_1) = 0$
* *Freedoms*: $f_1 \in \text{divisors}(u_1)$
* *Descend*:

$$\sum_{i=0}^{u_1/f_1-1} \text{RS3}\left((u_2, u_1, u_3, u_4, {u_7}^{\mathbb{Z}\to\mathbb{C}}, u_9, u_{10}, u_{11}) \leftarrow (u_2, f_1, u_3 + i, u_1/f_1, \Omega^1_{u_1} \circ dt \circ h^{f_1\to u_1}_{i,\ u_1/f_1}, u_1, i, u_1/f_1)\right)$$

$$\sum_{j=0}^{f_1-1} \text{RS2}\left((u_2, u_1, u_3, u_6, u_7, u_8) \leftarrow (u_1, u_1/f_1, u_1 j/f_1, u_6, u_7 + u_8 j, u_8 f_1)\right)$$

RS 3

Formula: $S(h^{u_1\to u_2}_{u_3,\ u_4})\mathbf{DFT}_{u_1} \text{diag}\left(\text{pre}({u_7}^{u_1\to\mathbb{C}})\right) G(h^{u_1\to u_9}_{u_{10},\ u_{11}})$
Parameters: $u_1, u_2, u_3, u_4, u_9, u_{10}, u_{11}, {u_7}^{\mathbb{Z}\to\mathbb{C}}$
Children: RS 3, RS 4
Breakdown 1: DFT-LibBase
* *Applicable*: $u_1 = 2$
* *Descend*: $S(h^{2\to u_2}_{u_3,\ u_4})\mathbf{DFT}_2 \text{diag}\left(\text{pre}({u_7}^{2\to\mathbb{C}})\right) G(h^{2\to u_9}_{u_{10},\ u_{11}})$

Breakdown 2: DFT-CTA
* *Applicable*: $\text{isPrime}(u_1) = 0$
* *Freedoms*: $f_1 \in \text{divisors}(u_1)$
* *Descend*: not shown

RS 4

Formula: $S(h^{u_1\to u_2}_{u_3,\ 1})\mathbf{DFT}_{u_1} \text{diag}\left(\text{pre}({u_6}^{u_1\to\mathbb{C}})\right) G(h^{u_1\to u_8}_{u_9,\ u_{10}})$
Parameters: $u_1, u_2, u_3, u_8, u_9, u_{10}, {u_6}^{\mathbb{Z}\to\mathbb{C}}$
Children: RS 3, RS 4
Breakdown 1: DFT-LibBase
* *Applicable*: $u_1 = 2$
* *Descend*: $S(h^{2\to u_2}_{u_3,\ 1})\mathbf{DFT}_2 \text{diag}\left(\text{pre}({u_6}^{2\to\mathbb{C}})\right) G(h^{2\to u_8}_{u_9,\ u_{10}})$

Breakdown 2: DFT-CTA
* *Applicable*: $\text{isPrime}(u_1) = 0$
* *Freedoms*: $f_1 \in \text{divisors}(u_1)$
* *Descend*: not shown

Table 5.1: Library plan constructed from the recursion step closure in (5.1).

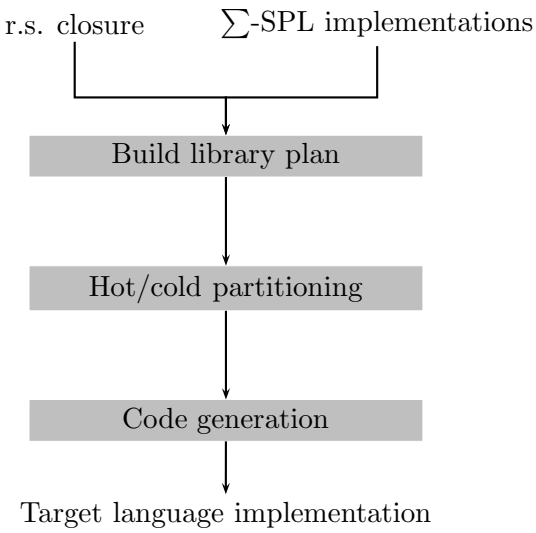


Figure 5.1: Library generation: "Library Implementation". Input: recursion step closure and $\sum$-SPL implementations. Output: library implementation in target language.

Library plan contains an entry for each recursion step in the closure. Each recursion step is given a name, in our implementations the names are simply RS 1 – RS n. Each entry contains the fields below.

- **Formula** gives a parametrized $\sum$-SPL formula for the recursion step;
- **Parameters** lists all parameters of the recursion step formula;
- **Children** is a list of all other recursion steps that are invoked in the breakdown rule implementations of the current recursion step;
- **Parents** is a list of all other recursion that invoke the current recursion step;
- **Breakdown rules** is a list of breakdown rules used for implementing (descending into) the current recursion step.

For each breakdown rule, the library plan stores the applicability condition, as a function of the recursion step parameters, the degrees of freedom, and the $\sum$-SPL implementation, obtained by applying the breakdown rule (descending). Breakdown rule implementation can invoke other recursion steps, in the library plan such calls are made explicit, by providing the name of the recursion step being called and the list of parameter substitutions, as explained further below.

The recursion step call RS$i((u_1, u_2) \leftarrow (a, b))$ is semantically equivalent to the $\sum$-SPL formulas for RSi with a substituted for u_1, and b for u_2.

A recursion step invocation from a $\sum$-SPL implementation is not a simple function call, because the cold parameters must be provided in advance as discussed earlier. Intuitively, invocation works as follows. The cold parameters are provided for the root recursion step (RS 1). For each recursion step call in RS 1, all cold parameters of the callees are initialized, and the callees initialize their own callees. This results in a recursive initialization process.

5.4 Hot/Cold Partitioning

At the first glance, it seems that choosing whether the parameters are cold or hot should be left to the user. This is indeed possible for the simplest recursion steps which have very few parameters, for example $\mathbf{DFT}_{u_1}$ with a single size parameter, where we clearly want it to be cold to enable precomputation. However, libraries consist of many recursion steps, and even with a single library plan as in Table 5.1, the choice of hot/cold for other recursion steps (e.g. RS 2, RS 3 and RS 4) is non-trivial.

In this section we explain how the parameters can be automatically partitioned into hot and cold using by taking into account the library requirements and a user-defined policy.

We define the following requirements:

- *All precompute markers must be honored.* This means that parameters that affect the precomputed data are required to be cold. For example, u_1 and u_7 in RS 3, and u_1 and u_6 in RS 4 of Table 5.1.
- *The choice of a breakdown rule and any degrees of freedom must be resolved at the initialization time.* This requires the parameters that affect breakdown rule applicability and the degrees of freedom are also required to be cold. For example, u_1 in RS 1–4 of Table 5.1.
- *Parameters that change in a loop must be hot.* If a recursion step is called from within a loop, some of the parameters of the recursion step will change with the loop counter. These parameters of the callee must be hot, because the loop can not be executed at the initialization time. For example, u_3, u_7, u_{10} in RS 3 in Table 5.1.

The parameters that were not affected by any of the requirements above can be either hot or cold, with the constraint that the cold parameters can't depend on hot parameters. We propose two different policies "as hot as possible" (AHAP) and "as cold as possible" (ACAP) for dealing with such parameters. Later, we will discuss the advantages and drawbacks of these policies.

The process of partitioning is equivalent to an iterative data flow analysis (IDA) [2] performed on a special form of a control flow graph, which we call a *parameter flow graph*. The nodes of the parameter flow graph are the individual parameters of each recursion steps, and the edges are the dependencies between the parameters. Parameter flow graph is obtained from the library plan, annotated with the data flow information. To start the IDA we need the initial state of each node, the data flow information, and few simple propagation rules. Now, we explain how this works in detail.

Parameter flow graph. To propagate the cold/hot information we need the equivalent of a control flow graph. Instead of using the recursion steps as the nodes of the control flow graph, we make the individual parameters of recursion steps nodes. Hence, we call the graph the parameter flow graph.

If one recursion step (caller) calls another recursion step (callee), then the parameters of the callee are computed from the parameters of the caller. For each such caller/callee parameter pair we add a dependence edge to the graph. The dependence pairs can be easily read off recursion step invocations in the "descend" fields of the library plan records. For example, inspecting the DFT-CTA breakdown rule descend in RS 1 in Table 5.1, shows the following variable bindings in the RS 3 invocation:

$$(u_2, u_1, u_3, u_4, u_7{}^{\mathbb{Z}\to\mathbb{C}}, u_9, u_{10}, u_{11}) \leftarrow (u_1, f_1, i, u_1/f_1, \Omega^1_{u_1} \circ dt \circ h^{f_1\to u_1}_{i,\, u_1/f_1}, u_1, i, u_1/f_1).$$

Which leads to the following dependence edges:

$$u_1 \rightarrow u_2,$$
$$u_1 \rightarrow u_3,$$
$$u_1 \rightarrow u_4,$$
$$u_1 \rightarrow u_7,$$
$$u_1 \rightarrow u_9,$$
$$u_1 \rightarrow u_{10},$$
$$u_1 \rightarrow u_{11}.$$

Above, the left-hand side refers to the parameter u_1 of the caller (RS 1) and the right-hand to the parameters of the callee (RS 3). Combining all of the dependence edges leads to the parameter flow graph shown in Fig. 5.2.

The edges $u_1 \rightarrow u_3$ and $u_1 \rightarrow u_{10}$ are not obvious. From the variable bindings we see that both u_3 and u_{10} are computed from the loop index variable i , and since i's range as the loop variable is bounded by u_1/f_1 we consider it dependent on u_1.

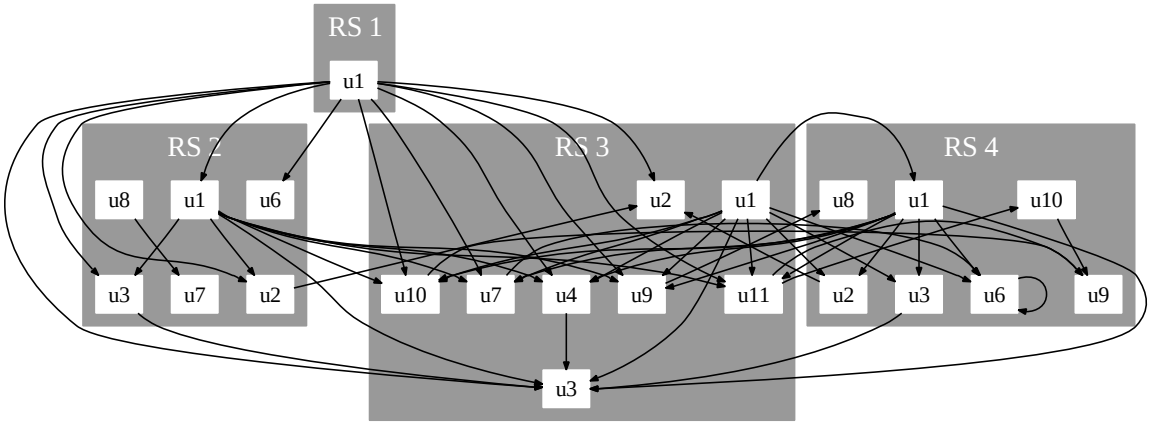
Algorithm. The automatic hot/cold partitioning is a two phase IDA algorithm on the parameter flow graph. The input is the parameter flow graph, and the output is the assignment of a state to each parameter (node). The nodes of the graph can be in four states: "none", "cold", "hot" and "reinit". The algorithm initializes each node to the "none" state, which means that no decision has been made. Then we apply two IDA phases. The first phase is a backward IDA that marks all mandatory cold parameters. The second phase is a forward IDA that marks all mandatory hot parameters, and in case the case that a parameter must be cold and hot at the same time, puts it in special "reinit" state, explained below. Finally, when the second IDA phase is complete, we mark all unmarked parameters (i.e., in "none" state) as either hot or cold depending correspondingly on whether AHAP or ACAP policy is used.

It can happen that a parameter depends on the loop index, and thus must change as the loop index runs, and at the same time the parameter is inside the precompute marker. An example of such parameter is u_7 in RS 3. It is the generator function for the twiddle diagonal. In this case we assign this parameter "reinit" status, which means that the caller of the environment RS 3, will create several copies of RS 3 descriptor with different values of the u_7. The number of copies will be equal to the number of iterations of the loop, from which RS 3 is invoked.

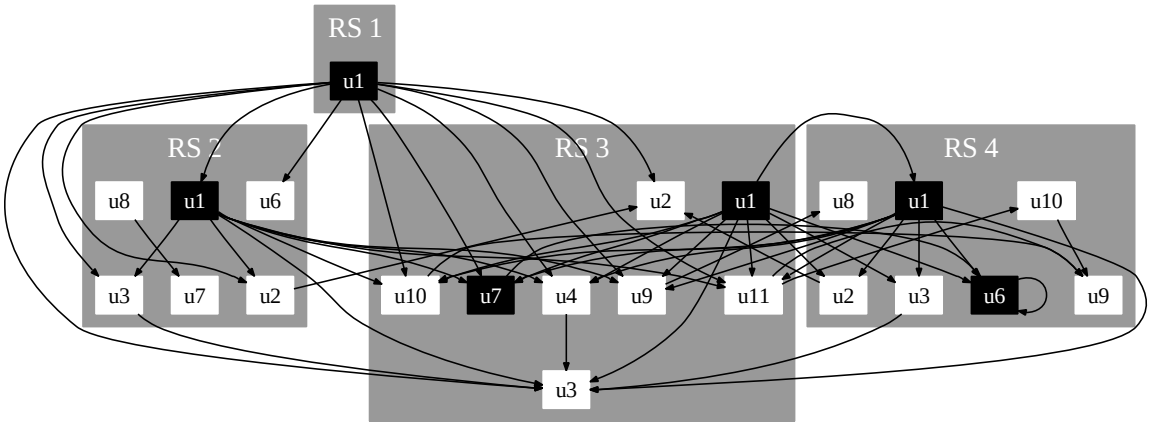
Initial state assignment. Initial state assignment is done according to the rules below:

1. Mark all freedom parameters f_i as "cold".
2. Mark all parameters that appear in the "applicable" and/or "freedoms" fields of the breakdown rules in the library plan as "cold". Examples of such parameters are u_1 in all recursion steps of the library plan in Table 5.1.
3. Mark all parameters that appear inside the pre(·) marker as "cold". Examples of such parameters are u_7 and u_1 in RS 3, and u_6 and u_1 in RS 4 in Table 5.1.
4. Put all other parameter nodes into "none" state.

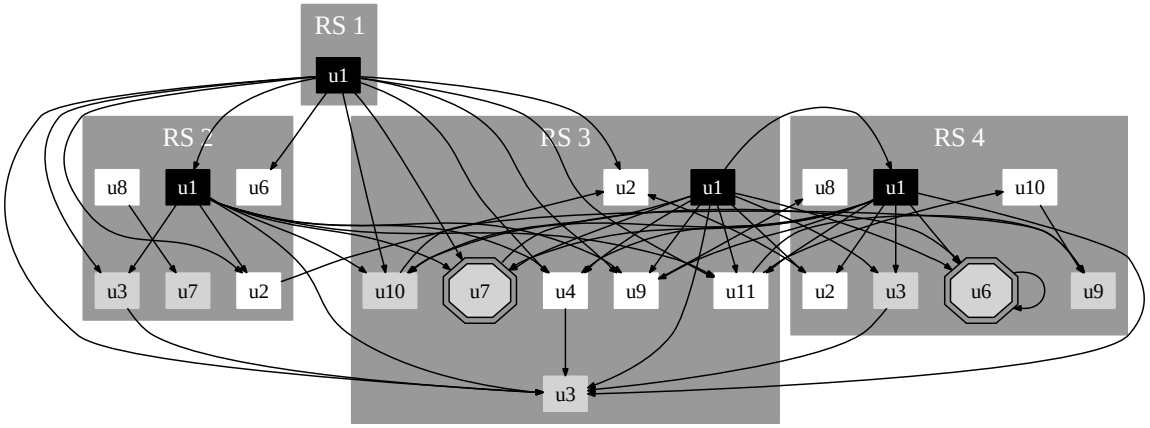
The initial assignment makes sure that the first two library requirements explained earlier are satisfied. Next, the phase 1 IDA will propagate the "coldness" backwards.



(a) Original unpartitioned graph



(b) After initial state assignment and phase 1 IDA (phase 1 does not change any states)



(c) After phase 2 IDA

Figure 5.2: Parameter flow graph obtained from the library plan from Table 5.1 and various stages of the automatic hot/cold partitioning algorithm. Node color and shape denotes state: white square = "none", black square ="cold", gray square = "hot", gray octagon = "reinit".

Phase 1: Backward IDA. The goal of this phase is to find all parameters that must be cold. A cold parameter is computed at initialization time. If its value depends on other parameters, then the other parameters must also be known at initialization time, i.e., they must be cold.

This is done by performing a backward IDA with one simple rule. For each dependence edge $p \rightarrow q$, if q is marked "cold", and p is marked "none", then mark p as "cold". The rule is applied until the fixed point is reached.

After the initial assignment in the parameter flow graph in Fig. 5.2 there are no such dependence edges. There are only edges with both nodes being cold already, for example u_1 of RS 1 to u_1 of RS 2.

Phase 2: Forward IDA. The goal of this phase is to fulfill the third library requirement, i.e., make the parameters that depend on loop indices hot. Similarly to phase 1, we need to propagate the "hotness" along the dependence edges, but now forward instead of backward. Clearly, a hot parameter can not be used to initialize the cold parameter, thus the "hotness" is propagated forward. Another caveat is that a parameter that depends on a loop index can be already marked "cold". In this case, we change the state to "reinit", which means that the enclosing environment is replicated to cover all possible values of such parameter. The set of possible values is bounded by the number of iterations of the loop. For all other purposes "reinit" parameters are treated as "cold", and thus no further propagation is necessary.

The phase 2 is defined by the following three IDA 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".

Note that dependence edges with p "hot" and q marked "cold" are not possible, because phase 1 propagates coldness backwards, and any edge $p \rightarrow q$ with q cold will result in p being marked "cold" as well. Also, if p is marked "cold" and depends on the loop index, it is marked "reinit", which is not propagated further.

ACAP vs AHAP policy. After the phase 2 IDA is complete some nodes may still be marked "none". Under the ACAP policy these nodes will be marked "cold", and under AHAP they will be marked "hot".

The ACAP policy exposes most parameters at the initialization time. The values of these parameters form important context information, which enables accurate search over the degrees of freedom. This search happens at the initialization time. With AHAP policy some parameters are not yet known at the initialization time, and thus the search must assume some default values for them. An example of parameters which can drastically change performance, are input and output strides. With the AHAP policy the search becomes context insensitive and thus less accurate.

The AHAP policy on the other hand, is useful when it is desired to change some parameters (for example, input and output data strides) without reinitializing. In addition, the AHAP policy also minimizes the size of the descriptor, since less parameters need to be stored after the initialization.

For libraries with small number of parameters, there may be no difference between ACAP and AHAP libraries' interfaces, but the choice of policy is still important. For example, in the library plan in Table 5.1 the user requested recursion step is RS 1, which has only one parameter (DFT size, u_1). Under both policies u_1 is necessarily cold, and thus the interface of RS 1 does not change

depending on the hot/cold policy chosen. However, the two libraries are still different, because the recursion steps used for implementing RS 1, will have different hot/cold partitions, based on the policy. In this case, ACAP policy enables more accurate runtime search over the degrees of freedom, and AHAP policy minimizes the descriptor size.

Other policies. It is possible to implement other policies for marking the remaining nodes. However, any node partition must satisfy the timimg requirement. Namely, a hot parameter can not be used to initialize a cold parameter. Formally, for each dependence each $p \rightarrow q$, if p is hot, than q has to be hot.

Suppose, that ACAP policy is desired, but one specific parameter p is desired to be hot. One way yo implement this is to perform the normal partitioning. If p is marked "cold" or "reinit", then signal an error. If p is marked "hot" then mark all other nodes cold (according to ACAP policy). Finally, if p is marked "none", mark it "hot", repeat phase 2 IDA, and mark remaining "none" nodes as "cold".

5.5 Code Generation

The final stage is to generate C++ classes for each recursion step in the library plan. Each C++ class will have a constructor and the `compute` method. The constructor performs the initialization tasks, and saves the values of the cold parameters. The `compute` method takes hot parameters as arguments, and uses the saved values of the cold parameters.

As we mentioned earlier, these classes are the equivalents of the "descriptors" in the regular transform libraries. Semantically, they serve the purpose of lexical closures, which in turn implement higher-order functions.

Non-adaptive library example. In Table 5.2 we show a code fragment of a generated non-adaptive DFT library based on the recursion step closure in (5.1) and library plan in Table 5.1. We show the C++ class generated for RS 1 (called `Env_1`, where "Env" stands for environment). RS 1 corresponds to the $\sum$-SPL formula $\mathbf{DFT}_{u_1}$, with only one parameter u_1, which is the size of the DFT. (We omit the `Env_1` destructor and auxiliary classes in the listing.)

In Table 5.2 the constructor `Env_1::Env_1` (line 9) and `Env_1::compute` (line 36) are formed by combining multiple alternative $\sum$-SPL implementations. The applicability conditions of alternative implementations are checked in the constructor and the first implementation with applicable condition being true is selected, saving the result in the class attribute `_rule`. In our case the constructor chooses out of three alternative implementations: the base case for DFT of size 2 (with condition $u_1 = 2$, line 11), base case for DFT of size 4 ($u_1 = 4$, line 15), and the recursive implementation for other sizes (line 19), which must be non-prime.

If the recursive implementation is selected, the child recursion steps must be created and initialized. As we can see from the library plan in Table 5.1 (and the call graph in Fig. 3.4(b)), RS 1 has two children – RS 2 and RS 3, which correspond to classes `Env_2` and `Env_3`. The initialization of RS 3 in lines 24–28 creates multiple copies due to the "reinit" parameter u_7 (diagonal generating function, which changes with the loop index). This generating function is also created at runtime, and thus is a higher-order function (just like DFT), and thus also requires a lexical closure, i.e., a class (here it is `Func_1`).

In this case the recursive implementation has a degree of freedom `f_1` and its value is determined in line 20, using a heuristic `chooseDivisor`, and without using any platform adaptation.

`Env_1::compute` dispatches to the right implementation based on the value of `_rule` which was

set during the initialization in the constructor. Lines 51–58 implement the recursion, by recursive calling the `compute` method of the child recursion steps.

This library is missing two important optimizations. First, the allocated buffer in the computation (in `Env_1::compute`) can be eliminated. Second, the recursion steps do not contain loops, and as a consequence there is no loop in the base case. Both of these optimization are implemented in our generator, as we explained in Chapter 3.

Using the generated library. The usage of the generated library is very simple, we show the usage example below:

```
Env_1 dft(1024);
dft.compute(Y, X);
```

The above fragment computes the DFT of X and stores the result in Y.

Adaptive libraries. In order for the library to be adaptive, it must automatically choose the value of `f_1`. It turns out that it is rather easy to incorporate into the generated libraries, by adding several extra methods to the classes, but preserving the overall code structure of Table 5.2. We will not show an example here.

```
class Env_1 : public Env {
  int u1, _rule, f1;
  char  *_dat;   // for precomputed data
  Env  *child1, *child2;
  Env_1(int u1);
  void compute(double  *Y, double  *X);
};

Env_1::Env_1(int u1) {
  this->u1 = u1;
  if (((u1) == (2))) {
    _dat = (char *) NULL;
    _rule = 1;
  }
  else if (((u1) == (4))) {
    _dat = (char *) NULL;
    _rule = 2;
  }
  else if (((isPrime(u1)) == (0))) {
    f1 = chooseDivisor(u1);

    // need multiple versions with different constants,
    // generated using different instances of Func_1
    child1 = new Env_3[(u1/f1)];
    for(int i40 = 0; i40 < (u1/f1); i40++) {
      *(cast<Env_3 *>(child1) + i40) = Env_3(f1,
         new Func_1(i40, f1, (u1/f1), u1, f1, (u1/f1)));
    }
    child2 = new Env_2((u1/f1));
    _dat = (char *) NULL;
    _rule = 3;
  }
  else error("no applicable rules");
}

void Env_1::compute(double  *Y, double  *X) {
  if (((_rule) == (1))) {  // Base case, DFT_2
    double a902, a903, a904, a905;
    a902 = *(X);
    a903 = *((2 + X));
    a904 = *((1 + X));
    a905 = *((3 + X));
    *(Y) = (a902 + a903);
    *((1 + Y)) = (a904 + a905);
    *((2 + Y)) = (a902 - a903);
    *((3 + Y)) = (a904 - a905);
  }
  else if (((_rule) == (2))) { // Base case, DFT_4
    .. skipped ..
  }
  else if (((_rule) == (3))) { // Recursion
    double * T35; // unnecessary buffer
    T35 = LIB_MALLOC(sizeof(double) * (2 * u1));
    for(int i41 = 0; i41 < f1; i41++) {
        cast<Env_2  *>(child2)->compute(T35, X, i41, u1, f1, ((u1*i41)/f1), u1);
    for(int i40 = 0; i40 < (u1/f1); i40++) {
        (cast<Env_3  *>(child1) + i40)->compute(Y, T35, i40, u1, (u1/f1), i40, u1, (u1/f1));
    LIB_FREE(T35, sizeof(double) * (2 * u1));
  }
  else error("no applicable rules");
}
```

Table 5.2: Fragment of the automatically generated code for $\mathbf{DFT}_{u_1}$ (with our comments added), based on the recursion step closure in Fig. 3.4 and library plan in Table 5.1. `Func_1(...)` creates a generating function for the diagonal elements, and is passed down to the child recursion step (RS 3), which will use it to generate the constants. Since each iteration of the loop uses a different portion of the constants, several copies of `Env_3` are created. Such copying is always required if a child recursion step has "reinit" parameters (u_7 in RS 3 in this case).

Chapter 6

Experimental Results

In this chapter we perform thorough performance evaluation of the library generator. We compare the generated libraries to the best hand-written implementations, including the appropriate vendor libraries. Next, we also discuss additional issues such as handling the higher-dimensional transforms, the efficiency of the vectorization and parallelization, and the flexibility of the generator.

6.1 Overview and Setup

Platforms. We ran our experiments on three different platforms, described in detail below:

- *3 GHz Intel Xeon 5160* (server version of Core 2 Duo) processor based computer. This machine had 2 dual-core processors (total of 4 processor cores) with 64 KB of private L1 cache per code and 4 MB of shared L2 cache per socket.
- *2.8 GHz AMD Opteron 2220* processor based computer. This machine also has 2 dual-core processors (total of 4 processor cores) with 256 KB of private L1 cache per core, 1 MB of private L2 cache per core (2 MB per socket), and a dedicated communication channel between cores within a pair.
- *3 GHz Intel Core 2 Extreme QX9650* (code name "Penryn") processor based computer. This is the latest Intel offering. The machine has 2 dual-core processors (total of 4 processor cores) with 64 KB of private L1 cache per core and 6 MB of shared L2 cache per socket. We have not exhaustively benchmarked on this machine, since it was released close to the completion of this thesis, and provide only a small number of benchmarks.

All three computers were running Linux in 64-bit mode. All of the generated libraries were in C++ compiled using the Intel C/C++ Compiler 10.1. Vectorized code was emitted using SSE and SSE2 intrinsics, and threading was implemented with OpenMP pragmas with explicit barriers.

Transforms. We generated general size libraries for the following transforms:

- Discrete Fourier transform (DFT);
- Walsh Hadamard transform (WHT) [17, 18];
- Real discrete Fourier transform (RDFT, also known as DFT of real data or real-valued DFT) [20, 98, 112, 125];

- Discrete Hartley transform (DHT) [31, 32, 98, 125, 126];
- DCT-2, DCT-3 and DCT-4 [33, 96, 97, 103];
- FIR filter [62, 93];
- Downsampled FIR filter (building block of the wavelet transform) [62, 113, 124];
- 2-dimensional DFT and DCT-2.

Hand-written libraries. We compared against FFTW 3.2 alpha [1], Intel IPP 5.2, and also AMD APL 1.1 on the AMD platform. We chose FFTW 3.2 alpha since it provided improved threading support and additional performance improvements. FFTW was compiled with pthreads, and not with OpenMP (as our generated libraries), since this resulted in some performance degradation.

We did not benchmark AMD APL on the Intel platforms, however, because AMD APL only implements the DFT and RDFT we did benchmark Intel IPP on the AMD platform.

Intel IPP 5.2 does not provide threading for DFT, RDFT, and DCTs. After we performed most of the benchmarks, we have learned that version 5.3 of IPP became available, which does include threading for these transforms, and in addition is slightly faster for single-threaded code. Unfortunately, we did not have enough time to benchmark the new IPP version in this thesis.

Generated libraries. We generated single and multi-threaded libraries with scalar, 2-way vectorized, and 4-way vectorized code. Table 6.1 shows a subset of the breakdown rules from which the trigonometric transform libraries were generated.

For the FIR filters we used only the time domain breakdown rules (not shown), that compute the filter by definition (i.e., by performing the direct matrix-vector product), and block the filter matrix in various ways. Spiral also contains frequency domain FIR filter rules, which use the DFT, but we did not generate the libraries from these rules due to the reason explained next.

One current limitation of the library generator is that it cannot generate code for the breakdown rules that must compute the DFT or some other transform as part of the initialization. Examples of such rules include frequency domain filtering rules, and Rader [102] and Bluestein [27] rules for the DFT. This limitation is only due to lack of time and can be easily fixed in the future. This limitation only affects the recursion, the rules can still be used inside the small fixed-size base cases, which are generated using standard fixed-size code Spiral capabilities.

To constrain the number of benchmarks, for trigonometric transforms we generated libraries which support 2-power sizes only. Libraries for larger classes of sizes can also be generated, as demonstrated in Fig. 6.5. Since Rader and Bluestein FFT algorithms currently can not be compiled as recursions, libraries which supported sizes with large prime factors cannot be yet generated.

Table 6.2 shows the number of recursion steps in our generated libraries. All libraries use recursion steps that include loops as explained in Section 3.6. This ensures better performance and enables parallelism optimizations. 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 leads to better performance.

In Table 6.2, the scalar **DFT** library uses the Cooley-Tukey FFT breakdown rule (2.1), to match our running example. Vectorized DFT libraries use the breakdown rule (6.1) from Table 6.1. (6.1) is different form (2.1), and requires less vector shuffles. It is based on the algorithm from [125].

[1] FFTW is partially generated. The library has hand-written recursion, and uses fully unrolled automatically generated code (codelets) for the base cases, i.e., small size transforms.

$$\mathbf{DFT}_n = P^\top_{k/2,2m} \left(\mathbf{DFT}_{2m} \oplus \left(I_{k/2-1} \otimes_i C_{2m}\, \mathbf{rDFT}_{2m}(i/k)\right)\right) \left(\mathbf{RDFT}'_k \otimes I_m\right), \quad k \text{ even}, \tag{6.1}$$

$$\begin{vmatrix} \mathbf{RDFT}_n \\ \mathbf{RDFT}'_n \\ \mathbf{DHT}_n \\ \mathbf{DHT}'_n \end{vmatrix} = (P^\top_{k/2,m} \otimes I_2) \left(\begin{vmatrix} \mathbf{RDFT}_{2m} \\ \mathbf{RDFT}'_{2m} \\ \mathbf{DHT}_{2m} \\ \mathbf{DHT}'_{2m} \end{vmatrix} \oplus \left(I_{k/2-1} \otimes_i D_{2m} \begin{vmatrix} \mathbf{rDFT}_{2m}(i/k) \\ \mathbf{rDFT}_{2m}(i/k) \\ \mathbf{rDHT}_{2m}(i/k) \\ \mathbf{rDHT}_{2m}(i/k) \end{vmatrix} \right) \right) \left(\begin{vmatrix} \mathbf{RDFT}'_k \\ \mathbf{RDFT}'^k \\ \mathbf{DHT}'_k \\ \mathbf{DHT}'_k \end{vmatrix} \otimes I_m \right), \quad k \text{ even}, \tag{6.2}$$

$$\begin{vmatrix} \mathbf{rDFT}_{2n}(u) \\ \mathbf{rDHT}_{2n}(u) \end{vmatrix} = L^{2n}_m \left(I_k \otimes_i \begin{vmatrix} \mathbf{rDFT}_{2m}((i+u)/k) \\ \mathbf{rDHT}_{2m}((i+u)/k) \end{vmatrix} \right) \left(\begin{vmatrix} \mathbf{rDFT}_{2k}(u) \\ \mathbf{rDHT}_{2k}(u) \end{vmatrix} \otimes I_m \right),$$

$$\mathbf{RDFT\text{-}3}_n = (Q^\top_{k/2,m} \otimes I_2)\, (I_k \otimes_i \mathbf{rDFT}_{2m})(i+1/2)/k))\, (\mathbf{RDFT\text{-}3}_k \otimes I_m), \quad k \text{ even},$$

$$\mathbf{DCT\text{-}2}_n = P^\top_{k/2,2m} \left(\mathbf{DCT\text{-}2}_{2m}\, K^{2m}_2 \oplus \left(I_{k/2-1} \otimes N_{2m}\, \mathbf{RDFT\text{-}3}^\top_{2m}\right)\right) B_n (L^{n/2}_{k/2} \otimes I_2)(I_m \otimes \mathbf{RDFT}'_k) Q_{m/2,k},$$

$$\mathbf{DCT\text{-}3}_n = \mathbf{DCT\text{-}2}^\top_n,$$

$$\mathbf{DCT\text{-}4}_n = Q^\top_{k/2,2m} \left(I_{k/2} \otimes N_{2m}\, \mathbf{RDFT\text{-}3}^\top_{2m}\right) B'_n (L^{n/2}_{k/2} \otimes I_2)(I_m \otimes \mathbf{RDFT\text{-}3}_k) Q_{m/2,k}.$$

Table 6.1: Breakdown rules for a variety of transforms: DFT for real input (RDFT), discrete Hartley transform (DHT), discrete cosine transforms (DCTs) of types 2–4. The other are auxiliary transforms needed to compute them. P, Q are permutation matrices, T are diagonal matrices, B, C, D, N are other sparse matrices. The first and second rule are respectively for four and two transforms simultaneously.

	Number of recursion steps		
Transform	scalar	vectorized	vectorized + parallelized
DFT	4 / 3	4 / 7	8 / 8
RDFT	4 / 6	10 / 10	12 / 10
DHT	4 / 6	10 / 10	12 / 10
DCT-2	5 / 9	11 / 13	13 / 13
DCT-3	5 / 9	12 / 16	14 / 16
DCT-4	4 / 4	6 / 4	8 / 4
WHT	4 / 3	6 / 4	7 / 4
2D DCT	10 / 14	12 / 13	14 / 13
2D DFT	7 / 9	11 / 13	13 / 13
FIR Filter	4 / 4	4 / 5	4 / 4
Downsampled FIR Filter	4 / 4	4 / 5	4 / 4

Table 6.2: 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.

	Code size	
Transform	non-parallelized	parallelized
no vectorization		
DFT	13.1 KLOC / 0.59 MB	10.3 KLOC / 0.45 MB
RDFT	8.5 KLOC / 0.36 MB	8.8 KLOC / 0.39 MB
DHT	9.1 KLOC / 0.40 MB	9.4 KLOC / 0.39 MB
DCT-2	12.0 KLOC / 0.55 MB	12.4 KLOC / 0.57 MB
DCT-3	12.0 KLOC / 0.56 MB	12.3 KLOC / 0.59 MB
DCT-4	6.8 KLOC / 0.33 MB	7.1 KLOC / 0.35 MB
WHT	5.6 KLOC / 0.21 MB	—
2-way vectorization		
DFT	14.8 KLOC / 0.73 MB	15.0 KLOC / 0.74 MB
RDFT	15.6 KLOC / 0.76 MB	16.0 KLOC / 0.81 MB
scaled RDFT	16.0 KLOC / 0.78 MB	—
DHT	16.9 KLOC / 0.83 MB	17.2 KLOC / 0.87 MB
DCT-2	20.7 KLOC / 1.10 MB	21.0 KLOC / 1.09 MB
DCT-3	27.9 KLOC / 1.56 MB	28.2 KLOC / 1.59 MB
DCT-4	7.8 KLOC / 0.47 MB	8.1 KLOC / 0.50 MB
WHT	6.9 KLOC / 0.32 MB	5.8 KLOC / 0.26 MB
FIR Filter	167 KLOC / 7.75 MB	120 KLOC / 5.12 MB
Downsampled FIR Filter	100 KLOC / 4.2 MB	68 KLOC / 2.76 MB
4-way vectorization		
DFT	17.9 KLOC / 1.09 MB	18.2 KLOC / 1.11 MB
RDFT	16.2 KLOC / 0.86 MB	16.5 KLOC / 0.91 MB
scaled RDFT	16.5 KLOC / 0.88 MB	—
DHT	17.9 KLOC / 1.02 MB	18.3 KLOC / 1.04 MB
DCT-2	23.3 KLOC / 1.50 MB	23.6 KLOC / 1.53 MB
DCT-3	32.0 KLOC / 2.17 MB	32.3 KLOC / 2.20 MB
DCT-4	8.3 KLOC / 0.63 MB	8.6 KLOC / 0.66 MB
WHT	8.5 KLOC / 0.53 MB	6.9 KLOC / 0.4 MB
2D DFT	20.6 KLOC / 1.32 MB	20.8 KLOC / 1.33 MB
2D DCT-2	27.0 KLOC / 2.1 MB	27.2 KLOC / 2.11 MB
FIR Filter	109 KLOC / 5.69 MB	74 KLOC / 3.44 MB
Downsampled FIR Filter	151 KLOC / 7.7 MB	92 KLOC / 4.61 MB

Table 6.3: Code size of our generated libraries. KLOC = kilo (thousands) lines of code. WHT, scaled and 2D transforms are not shown in all possible variants.

Table 6.3 shows the code size of different generated libraries.

Search. The generated libraries are partially adaptive, and provide a mechanism to search over degrees of freedom within each breakdown rule at library runtime (during the initialization). For example, the library can search for the best radix size in the Cooley-Tukey breakdown (2.1), and the best block size in the filter blocking rules. Currently, the library cannot automatically choose the best alternative out of several applicable breakdown rules.

The search mechanism keeps the value of the degree of freedom that yields the best runtime. The best choice is hashed against the recursion step number and its cold parameters. The search infrastructure was implemented by Frédéric de Mesmay, and thanks to him the effort to produce good runtime results was greatly reduced.

FFTW also uses runtime performance adaptation to search for the best recursion strategy. The FFTW search mechanism can find the best "solver" (equivalent of breakdown rule here) and search over the degrees of freedom within the solver (e.g. radix in Cooley-Tukey FFT). As far as we know, neither Intel IPP nor AMD APL use search or any other form of adaptation.

Sizes. For all transforms except FIR filters, we benchmarked only 2-power sizes (except in Fig. 6.5) smaller than 2^16. We benchmarked the 2-power sizes, as these are the most commonly used sizes, and provide a reasonable way to constrain the space of benchmarks.

We also limit the transform sizes so that the working set fits into L2 cache of the computer. On the Intel platforms the largest size which fits is about 2^{16}, and on the AMD platform it is smaller. Larger sizes, due to cache thrashing caused by large strides, require special optimizations, for example, copying the data into a buffer. These large optimizations are implemented in standard Spiral as breakdown rules, however, we did not yet port them to be compatible with the library generator.

Due to the lack of the proper algorithm, the performance of generated libraries for these larger size is very poor, and we don't think it makes sense to compare against properly optimized libraries. As soon as the large size breakdown rules are ported, we can generate the fast out-of-L2 cache library.

Performance metrics. To show performance of generated code, instead of runtime, we use pseudo Gflop/s (giga floating point operations per second), which allows us to use the same linear scale for different transform sizes. This is the standard practice in the field. "Pseudo" means that instead of the actual operations count we use a normalized value, so that pseudo Gflop/s are proportional to inverse runtime. Pseudo Gflop/s are computed as follows:

$$\text{pseudo Gflop/s} = \frac{\text{normalized arithmetic cost}}{\text{runtime [sec]}} \cdot 10^9.$$

The normalized arithmetic cost is computed as

- $5n \log_2 n$ for the 1-dimensional length n DFT;
- $5mn \log_2 mn$ for the 2-dimensional $m \times n$ DFT;
- $2.5n \log_2 n$ for the 1-dimensional length n RDFT, DHT and DCTs of all types;
- $2.5mn \log_2 mn$ for the 2-dimensional $m \times n$ RDFT, DHT and DCTs of all types;
- $n \log_2 n$ for the WHT;

- $2nk$ for the regular and downsampled FIR filter with n output samples and k taps. The number of input samples is hence $n + d(k-1)$, where d is the downsampling factor. For regular (not downsampled) filter $d = 1$.

6.2 Performance of Generated Libraries, Overview

In this section we provide a brief glimpse of the performance of generated libraries. We demonstrate a wide variety of transforms for which libraries can be generated, show generated libraries in different precisions, and compare scalar (non-vectorized) code performance to FFTW.

The complete set of benchmarks can be found in Appendix A.

6.2.1 Transform Variety and the Common Case Usage Scenario

In the first series of benchmarks we show a variety of useful functionality for which libraries can be generated, and focus on what we believe to be the most common case usage scenario. Namely, we assume that the typical user wants to compute a transform in double precision and as fast as possible.

For most users this means that the maximal level of optimizations should be used together with the most common platform configuration. This means 2-way vectorization (maximum for double precision), and up to 2 concurrent threads, since a single dual-core processor platform is the most common today.

In the experiments that follow, we generated 2-way vectorized and threaded libraries which support only 2-power size transforms. We always show one line per library, which shows the best performance between 1 and 2 threads. For all transforms the libraries switch to 2 threads at sizes 1024 or 2048. Intel IPP 5.2 does not support threading for the transforms we consider, and thus its performance is significantly worse for larger (>2048) sizes.

DFT and RDFT. The discrete Fourier transform is probably the most important linear transform used. Many hand-written optimized libraries exist for the complex DFT and the real-valued DFT (RDFT) and a lot of optimization effort went into maximizing their performance.

Today, fast DFT implementations are available from the hardware vendors, including Intel and AMD. These highly optimized implementations become the "baseline" of "good" performance.

We show the breakdown rules used for generation of libraries in Table 6.1. For the complex DFT, instead of the Cooley-Tukey FFT (2.1) we use the non-standard recursion based on the RDFT as shown in Table 6.1. It provides better performance for vector code, and provides a good motivation for the use of a library generator. For the real DFT, we use a similar breakdown rule, which is a general-radix RDFT algorithm from [125].

The respective performance plots are shown in Fig. 6.1. The generated DFT and RDFT libraries are slightly faster than the other libraries on the Intel Xeon and to a lesser extent on the AMD Opteron. The speedup comes from the more efficient DFT and RDFT algorithms (6.1) and (6.2), which minimize the amount of vector shuffles in the vectorized implementations. Both of these algorithms suffer from more expensive index computations at sizes beyond 4096 (for example, see Figure A.2 in Appendix A), however, in this case the index computation overhead is compensated by threading.

On the AMD platform APL has generally the best performance single thread DFT performance, except for smaller sizes. However, the generated library starts using 2 threads already at size 1024 wiping out the APL advantage completely.

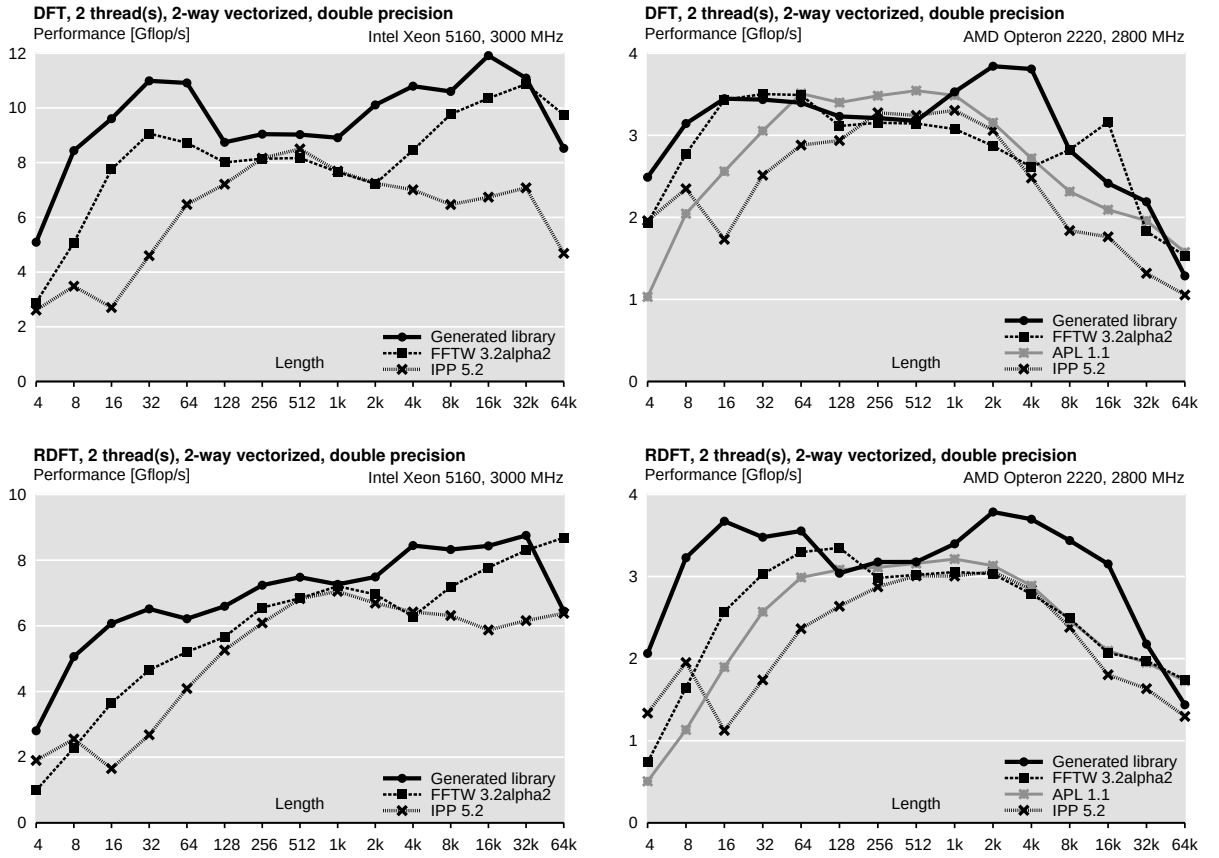


Figure 6.1: Generated libraries for complex and real Fourier transforms (DFT and RDFT). These are the most widely used transforms, and both vendor libraries and FFTW are highly optimized.

On both platforms, FFTW has suboptimal threading performance, perhaps due to using their own implementation of thread pooling using pthreads. The generated library benefits from the faster implementation in OpenMP. We did not enable OpenMP in FFTW, since it was slower than pthreads for some unknown reason, as we mentioned in Section 6.1.

DCT-2 and DCT-4. There is a total of 16 types of discrete cosine and sine transforms. Here we benchmark the two most commonly used, the type-2 and type-4 transforms. The type-2 DCT is sometimes called simply the forward DCT. We compare the performance in Fig. 6.2.

The results need a more detailed explanation. On the Intel Xeon, the generated library is on average about 2 times faster than FFTW, and up to 5 times faster than IPP. On the AMD platform the speedup is smaller but also sustained. The divergence between FFTW and IPP at larger sizes is due to the threaded DFT in FFTW. The reason for the big performance gap is twofold.

First, the DCTs are less commonly used than the DFT, and thus less resources are spent on their implementation and optimization. For example, not all DCT types are supported by IPP, and APL does not implement DCTs at all.

Second, our generated library uses a different algorithm, which is hard to implement without an full-program generator, as we explain next.

The common methods for computing the DCTs focus on reusing optimized DFT implementations. They involve either a conversion to real DFT with pre- and post-processing passes [81], or a "radix-2" split into half-size DCTs, for example [37, 127]. The use of these methods is due to

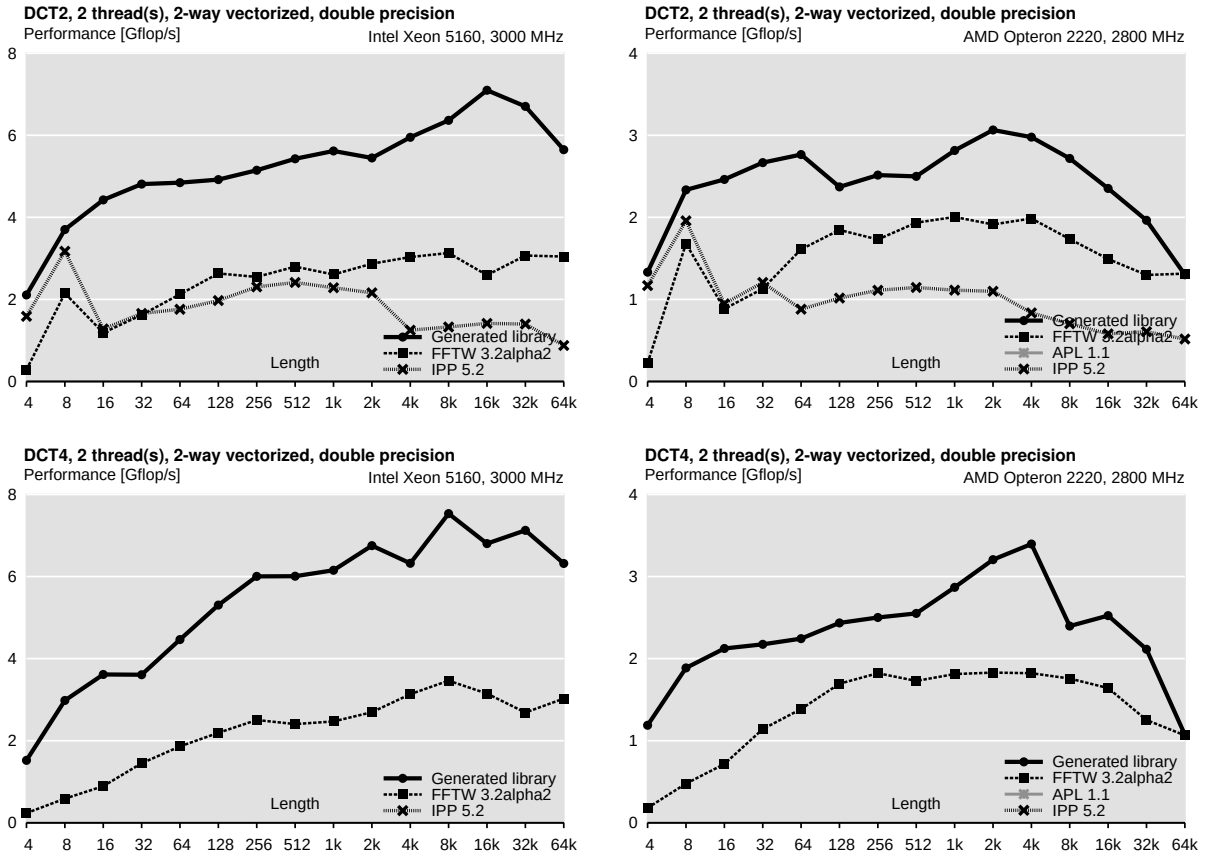


Figure 6.2: Generated libraries for discrete cosine and discrete Hartley transforms (DCT-2 and DHT). Less effort is spent on hand-optimizing these transforms, and thus hand-written libraries are either slower or lack the functionality. Neither IPP nor APL provide the DCT-4, and in addition APL does not implement DCT-2.

the common belief that the DFT (or RDFT) is the fundamental building block, on top of which the DCTs must be implemented. However, the explicit unmerged pre- and post-processing stages lead to suboptimal performance due to a number of factors. First, explicit passes over the array decrease memory reuse and result in memory hierarchy overhead. Second these extra passes need to be hand-written using SIMD vector instructions for best performance (which means that a different implementation for each vector length is necessary), and as far as we could tell, this was not done in FFTW. Finally, the extra passes limit the possible parallelization speedup since they are either implemented in serial code or require additional barriers.

Our generated library, on the other hand, employs native general-radix recursive Cooley-Tukey type algorithms for the DCTs (shown in Table 6.1), which are closely related to the algorithms in [97]. All of the permutations and other sparse matrices are appropriately merged with the adjacent loops resulting in the other recursion steps and thus other types of base cases (or "codelets" in FFTW terminology) than those occurring in the RDFT algorithm.

Manually implementing such DCT library based on this algorithm would have to be done from scratch, because very little code can be reused from any existing optimized DFT/RDFT implementation.

Discrete Hartley and Walsh Hadamard transforms. We show the results for the DHT and the WHT in Fig. 6.3. Neither of this transforms is provided by IPP and APL, so we only compare

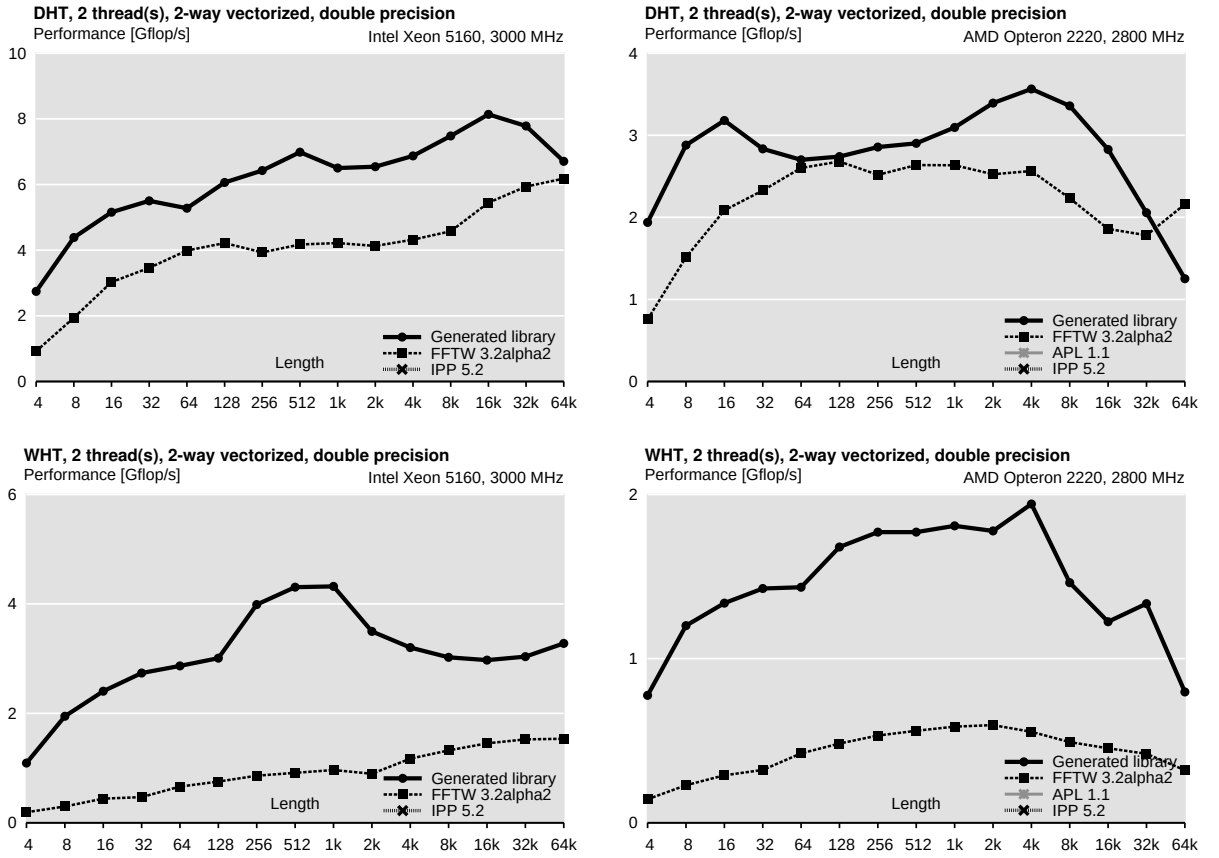


Figure 6.3: Generated libraries for discrete Hartley transform and Walsh Hadamard transform (DHT and WHT). Less effort is spent on optimizing these transforms, and thus hand-written libraries are much slower. IPP and APL do not provide DHT and WHT at all.

against FFTW. FFTW directly supports only DHT. However, since the 2^n WHT is equivalent to an n-dimensional $2 \times \cdots \times 2$ real DFT, we can still compute it using FFTW, because it supports DFTs of arbitrary dimensions.

Again, the generated library outperforms FFTW by a big margin. For example, the WHT on an Intel Xeon is up to 4 times faster in the generated library, and the DHT is up to 1.75 times faster. The reasons for the big difference are similar to those for the DCTs.

For the DHT, we employ the native general-radix algorithm from [125]. The generator performs loop merging, determines the right set of base cases (codelets) and generates the library. FFTW uses a conversion to the real DFT with a post-processing step, which results in performance degradation, and also limits the speedup achievable through parallelization.

For the WHT, we use a general-radix splitting algorithm which is equivalent to the FFTW's split along the dimensions of the $2 \times \cdots \times 2$ real DFT. However, FFTW performance is far suboptimal, because it does not provide codelets for the WHT (the only available codelet is $\mathbf{RDFT}_2 = \mathbf{WHT}_2$), and does not vectorize it.

FIR filters and wavelets. We compare the performance of FIR filters and downsampled filters in Fig. 6.4. A pair of downsampled filters constitutes a wavelet transform. On both platforms only IPP implements FIR filters. It supports both regular and downsampled (called "multirate") FIR filters. Our generated library computes all filters by definition, i.e., by directly evaluating the

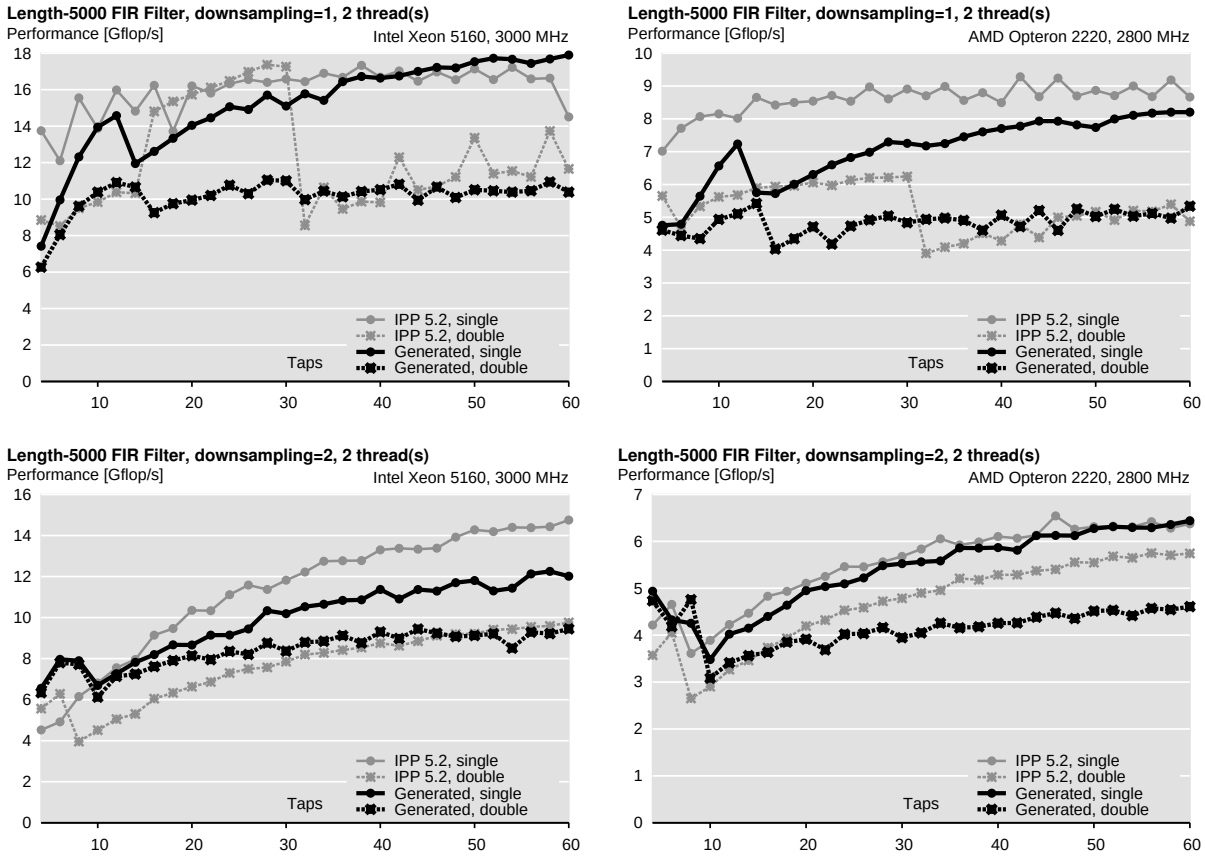


Figure 6.4: Generated libraries for FIR filters, first row is a plain filter, second row is a filter downsampled by 2, which is the building block for a 2-channel wavelet transform.

matrix-vector product and without using the DFT. The corresponding breakdown rules are called "time-domain" filtering rules, and they perform various forms of blocking of the FIR filter matrix. We also use some FIR filter specific vectorization breakdown rules.

Our generated libraries are only efficient for small number of filter taps (< 60), because they don't incorporate the "frequency-domain" breakdown rules due to a current limitation of the generator explained in Section 6.1. Frequency domain methods reduce the cost of the FIR filter by using a pair of DFTs, and are the best way to compute FIR filters with a larger number of taps.

Overall, the generated libraries are competitive with IPP on both platforms. On the Intel platform IPP is generally faster, and on the AMD platform the generated libraries are usually faster.

6.2.2 Non 2-power sizes

We made a decision to constrain the benchmarks for transforms that use algorithms relying on factoring the size to 2-power sizes only. This includes all transforms, except FIR filter. The limitation allows us to generate libraries with base cases for 2-power sizes only, and thus speeds up the generation.

The library generator, however, can produce libraries for any class of sizes for these transforms. To control the class of sizes supported by a library that uses a size factoring based algorithm, like

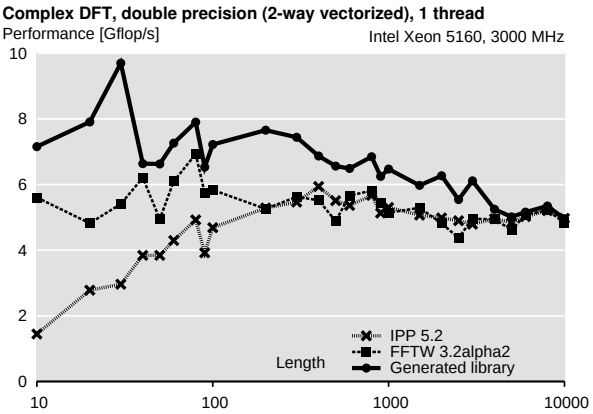


Figure 6.5: Generate DFT library for sizes $N = 2^i 3^j 5^k$. No threading is used.

Cooley-Tukey FFT, one merely needs to include additional base cases.

As an example, we generated a library for double precision DFT with base cases for $n \in \{2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 32, 64\}$ using the algorithm (6.1) from Table 6.1. This library supports DFT sizes of the form $N = 2^i 3^j 5^k$.

Fig. 6.5 shows the performance of this generated library. The library is faster due to the minimized shuffle overhead of (6.1), as is the case with the 2-power sizes in Fig. 6.1 in Section 6.2.1.

6.2.3 Different Precisions

Library generator can be used to produce the libraries for the same functionality using different precisions. For example, on the platforms we benchmarked, double precision and single precision floating point calculations are supported, as well as a variety of signed and unsigned integer number formats. In order to achieve the best performance, code cannot be reused between single and double precision libraries, because SSE/SSE2 instruction sets support double precision calculations with 2-way vectors, and single precision calculations with wider, 4-way vectors.

In Fig. 6.6 we show the performance of non-threaded single and double precision libraries generated for the DFT and DCT-2. Single precision libraries achieve better performance than the double precision counterparts due to wider 4-way vectorization, however, due to the two current limitations of the generator, they are not as fast as they could be in principle.

First, the library generator does not vectorize the base cases (with the exception of double precision DFT). Since, the base cases are fixed size transform, adding support for vectorization mostly requires some integration effort. As a consequence, due to the fact that non-vectorized single precision code is slower than the non-vectorized double precision code, single precision libraries are slower than the double precision ones for the transform sizes ≤ 64, which use non-vectorized base cases.

Second, some SPL vectorization rules reported in [54] relevant to the 4-way vectorization are not ported to $\sum$-SPL, and thus the 4-way vectorization is somewhat suboptimal.

As we previosly saw in Section 6.2.1 (with 2 threads and double precision), for both single and double precision the generated DFT libraries have performance close to FFTW and IPP, and the generated DCT-2 libraries are much faster, due to the native DCT algorithm.

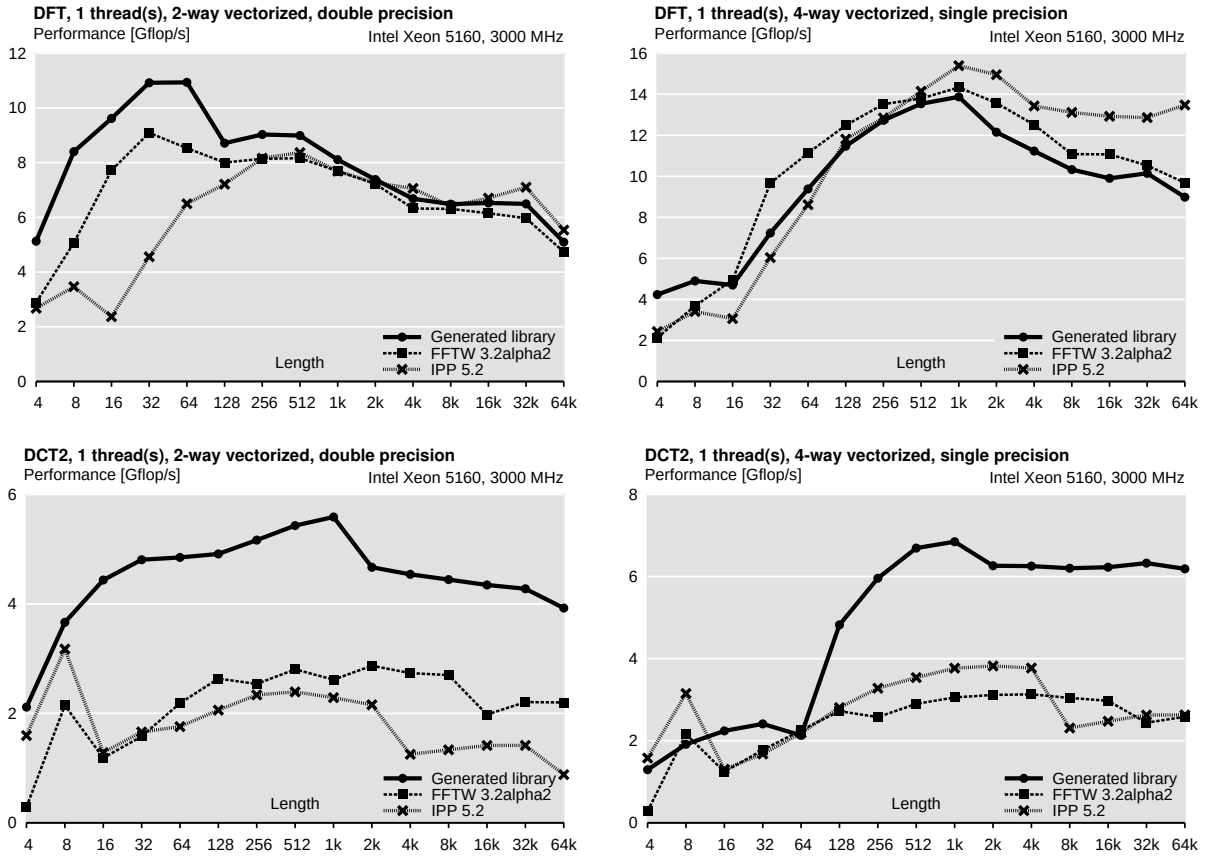


Figure 6.6: Comparison of generated libraries for DFT and DCT-2 in double (2-way vectorized) and single (4-way vectorized) precision. No threading.

6.2.4 Scalar Code

For best performance the library should be vectorized and multithreaded. However, it is also of interest to compare scalar (non-vectorized and non-parallelized) performance. Generally, there is more than one way to vectorize and to parallelize, which is further complicated by various threading overheads, which makes direct performance comparisons difficult to interpret. If, in contrast, both vectorization and parallelization are disabled, performance differences will be due only to different algorithm choices, the general code structure, and the overall level of library-specific and transform-specific optimizations. We compare to FFTW only, because a scalar version of IPP is not available.

Consider, the overall code structure. Both FFTW and the generated libraries decompose the original transform recursively and terminate the recursion with base cases ("codelets" in FFTW). For both libraries, the base cases are large blocks of unrolled code, with at most one outer loop.

Consider the algorithms. The recursive DFT algorithms used by both libraries for for the scalar code are the same (i.e. Cooley-Tukey), and the RDFT algorithms are very similar.

Now let's compare the performance. Both the DFT and the RDFT performance is shown in Fig. 6.7. The degree of similarity is striking. For sizes larger than 64, the performance of FFTW and of generated libraries for DFT and RDFT is practically identical. This led some people in our group to coin the term "to generate FFTW."

Both plots show a performance drop at transform size 128. The drop is due to the start of

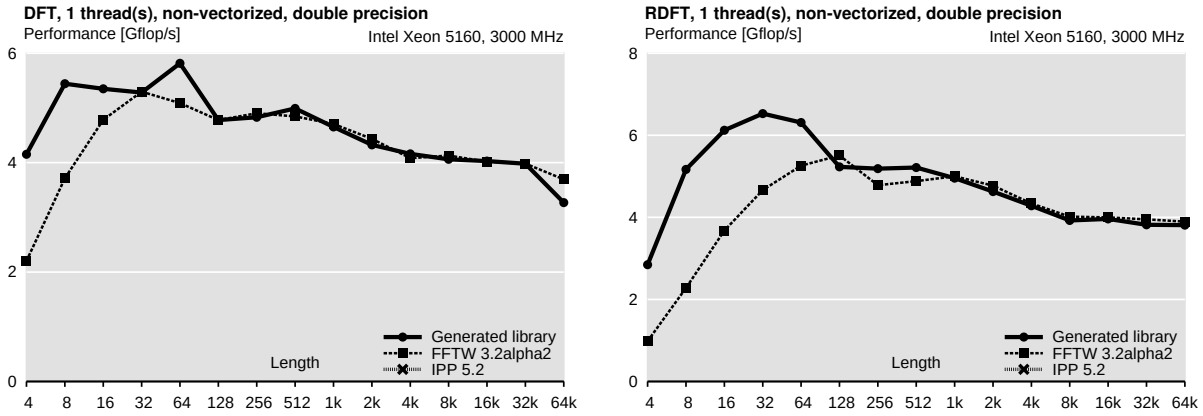


Figure 6.7: Generated scalar DFT and RDFT libraries versus scalar FFTW. No threading.

recursion, since both libraries provide base cases for sizes up to 64, and the size 128 has to go through at least one level of recursive split, introducing extra overhead.

For sizes 64 and smaller, however, the generated library is noticably faster, in particular in the case of the RDFT, even though both libraries use automatically generated fully unrolled base case code. We believe that this happens because the generated library uses base cases for a plain DFT/RDFT recursion step without any gather/scatter (recursion step 1 in Fig. 3.4), while FFTW reuses the base cases for strided gather/scatter DFTs (recursion step 2 in Fig. 3.4), which degrades performance due to unnecessary index computation. The index computation performance penalty is higher for smaller transforms, which have less floating point operations.

These performance results mean that not only the library generator does achieve the same overall optimization level as FFTW, it even can discover new kinds of optimizations. In this case, it even inadvertently discovered a new optimization, which increases performance of small transform sizes considerably.

From the practical standpoint this means that the library generator makes it possible to generate a library of the same quality as FFTW starting only from a high-level algorithm specification. It streamlines the generation process and makes it portable to other transforms and algorithms, without any human-effort. In particular, similar high quality libraries can be obtained for other transforms, as we already saw with DCTs.

6.3 Higher-Dimensional Transforms

All linear transforms that we consider extend to multiple dimensions. The multidimensional product is in most cases simply a separable product of 1-D transforms performed along each dimension. Exceptions to this rule are the RDFT and DHT, which require some additional post-processing.

The separable product along each dimension is simply a tensor product of matrices. For example, given a 1-dimensional transform T^1, its 2-dimensional and 3-dimensional variants are

$$T^2 = T^1 \otimes T^1, \quad \text{and} \quad T^3 = T^1 \otimes T^1 \otimes T^1.$$

Standard tensor product properties [21, 122] can now be used to manipulate multidimensional transforms and obtain breakdown rules. For example, the following decompositions are readily

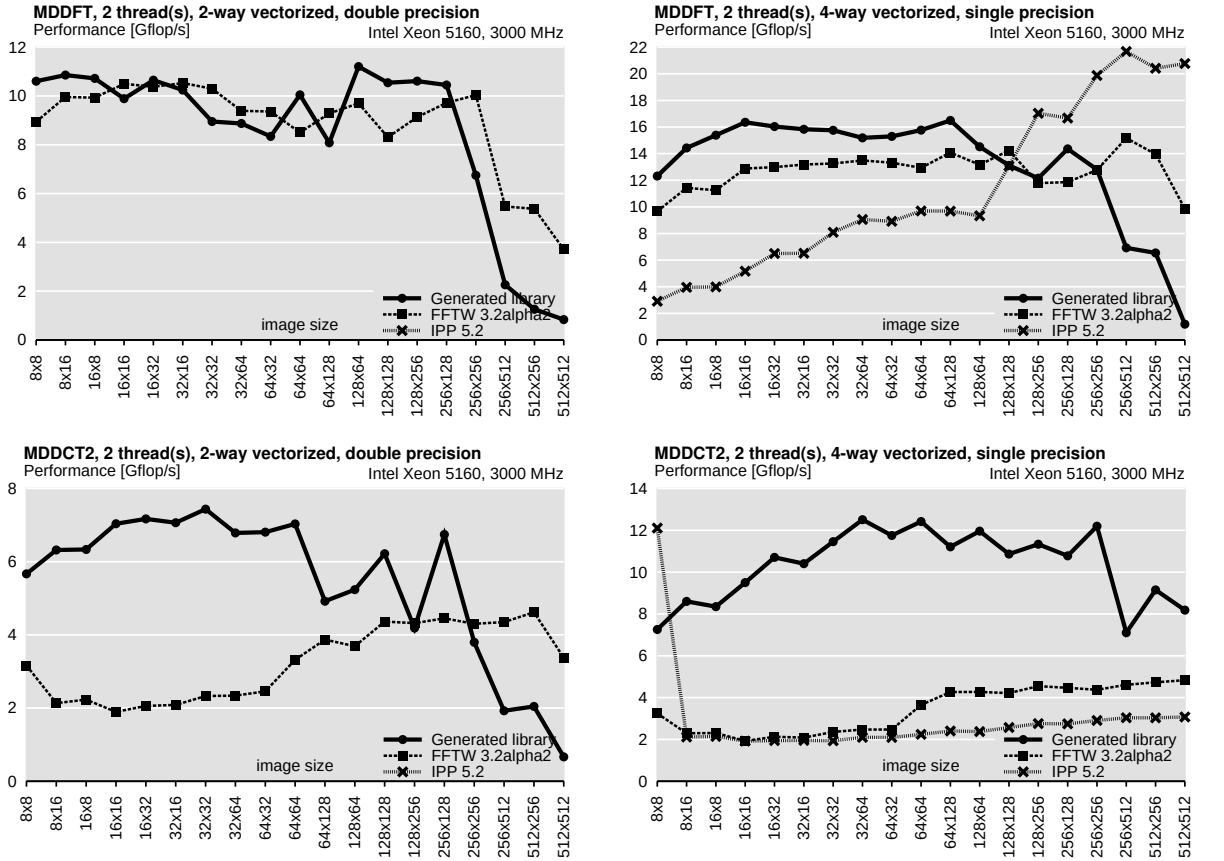


Figure 6.8: Generated 2-dimensional DFT and DCT-2 libraries, up to 2 threads. IPP does not provide double precision 2-dimensional transforms.

obtained:

$$T^2 = T^1 \otimes T^1 = (T^1 \otimes I)(I \otimes T^1), \tag{6.3}$$

$$T^3 = T^1 \otimes T^1 \otimes T^1 = (T^1 \otimes I \otimes I)(I \otimes T^1 \otimes I)(I \otimes I \otimes T^1). \tag{6.4}$$

The above rules are the standard way of implementing multidimensional transforms. However, it is just one possibility out of many. Tensor product identities can be used to obtain other algorithms. For example, if each T^1 is factored into a product of sparse matrices, $T^1 = A_1 \dots A_k$, we can obtain "vector-radix"-type multidimensional algorithms, by systematically interleaving the factors, as shown below for 2 dimensions:

$$(T^1 \otimes T^1) = (A_1 \dots A_k \otimes A_1 \dots A_k) = (A_1 \otimes A_1) \dots (A_k \otimes A_k) \tag{6.5}$$

The A_i are sparse, and the above split leads to a different algorithm. In some cases, it can reduce the arithmetic cost. For example, if some of A_i are diagonal matrices, say D, applying the identity $D \otimes D = D'$ (where D' is a new diagonal) saves operations compared to the standard algorithm (6.3).

We have generated 2-dimensional DFT and 2-dimensional DCT-2 libraries using (6.3). It is common practice to parallelize the outer loops over the 1-D transforms. However, an interesting

property of the library generator is that it can also vectorize these outer loops, instead of vectorizing the loops inside the transform. This leads to less overhead associated with vector shuffles.

Fig. 6.8 shows the performance of the generated 2-dimensional DFT and DCT-2 libraries on the Intel Xeon 5160. As a common case scenario the libraries use up to 2 threads and we show both double and single precision. FFTW supports both variants, and IPP has 2-dimensional transforms only as part of its image processing domain and does not support double precision.

For the DFT, the generated libraries are noticeable faster for smaller sizes, where the 1-dimensional building blocks are implemented as base cases, and the reduced shuffle overhead improves the performance. Once one of the dimensions is larger than 64 the 1-dimensional transforms need to further recurse to reach the base cases, and the increased overhead reduces the performance. At larger sizes the entire data set no longer fits into L2 cache, and at this point vectorizing the outer loop severely degrades the performance of the generated library. In addition, as we already discussed, some necessary optimizations to reduce cache thrashing are not implemented.

As the 1-dimensional DCT-2 described in the previous section, the generated libraries for 2-dimensional DCTs are much faster than the competition. The main reason is as before our use of the general-radix Cooley-Tukey type algorithm for the 1-dimensional DCT. In addition, the library automatically generates the appropriate DCT base cases (DCT codelets are missing in FFTW). Finally, the vectorization of the outer loop also improves the performance.

6.4 Efficiency of Vectorization and Parallelization

In this section, we briefly discuss the efficiency of our vectorization and parallelization.

Vectorization. We demonstrate the performance gain of vectorization in Fig. 6.9. The smaller transform sizes (≤ 64) are not vectorized in the generated libraries (except the double-precision DFT) due to a current limitation of the generator. As an artifact of this limitation, for small transform sizes it appears that 4-way vectorization leads to slowdown. This is not true, and the slowdown is due to the *lack* of vectorization, since scalar single precision code is slower than scalar double precision code, and 4-way vectorized libraries are single precision libraries.

Overall the vectorization payoff depends on the regularity of the algorithm. The best speedups are assocaiated with more regular algorithms, like the RDFT and DFT. The least regular algorithm is for the DCT-2, and the vectorization gain is lowest.

Parallelization. The parallelization speedup for the DFT and the real DFT (other trigonometric transforms have similar behavior) is shown in Figures 6.9–6.11. On our main benchmark platform (Intel Xeon 5160, Fig. 6.9) we obtain good performance scaling when going from 1 to 2 threads due to the shared L2 cache between the pair of cores, and no further speedup when going to 4 threads. The L2 cache is only shared between 2 cores, and the communication between the core pairs has to happen through the main memory bus. Thus, the CPU speed / memory bandwidth ratio appears to be the bottleneck.

To give some reference points we performed the same experiment on the new Intel platform, based on the Core 2 Extreme QX9650 processor, codenamed “Penryn”. The processor provides better memory bandwidth, and the plot in Fig. 6.11 now shows a speedup with 4 threads over 2 threads.

Our automatic parallelization was also evaluated on earlier platforms in [53] showing a linear speedup with 4 processors.

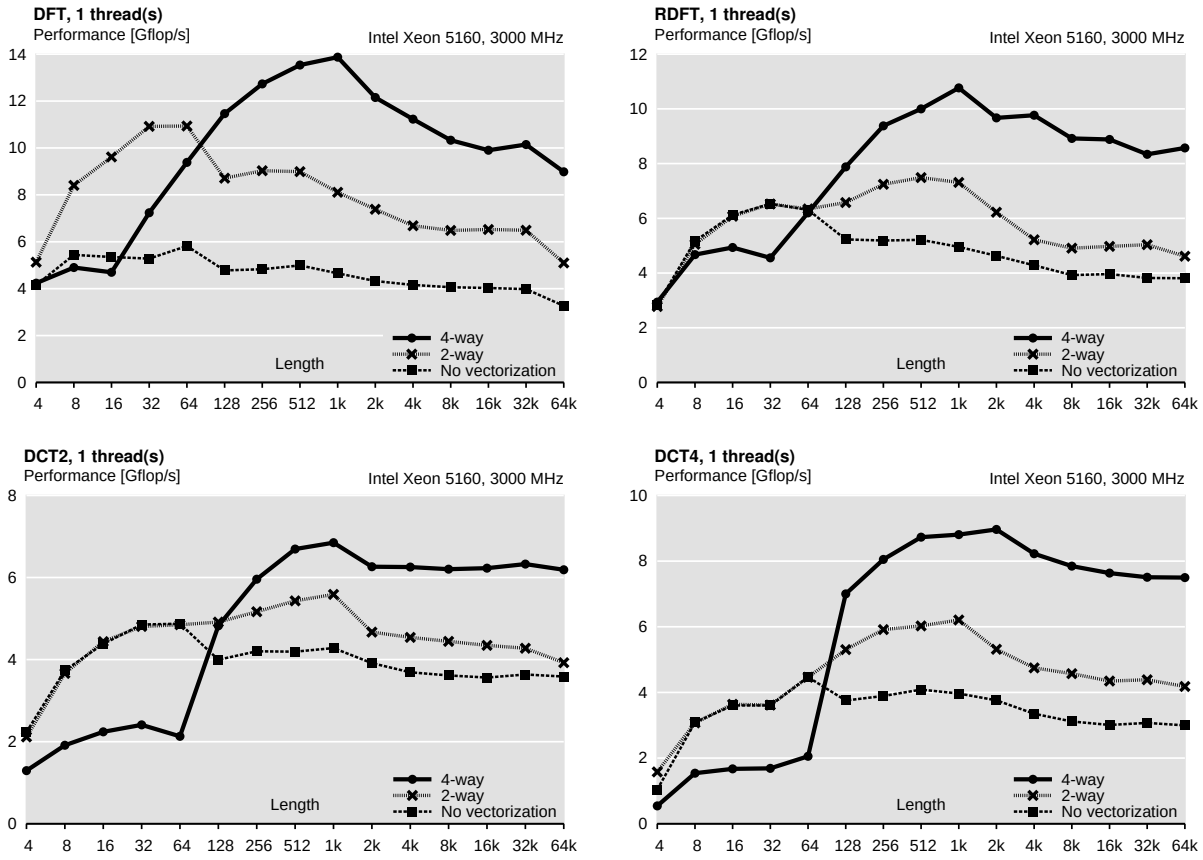


Figure 6.9: Vectorization efficiency. Platform: Intel Xeon.

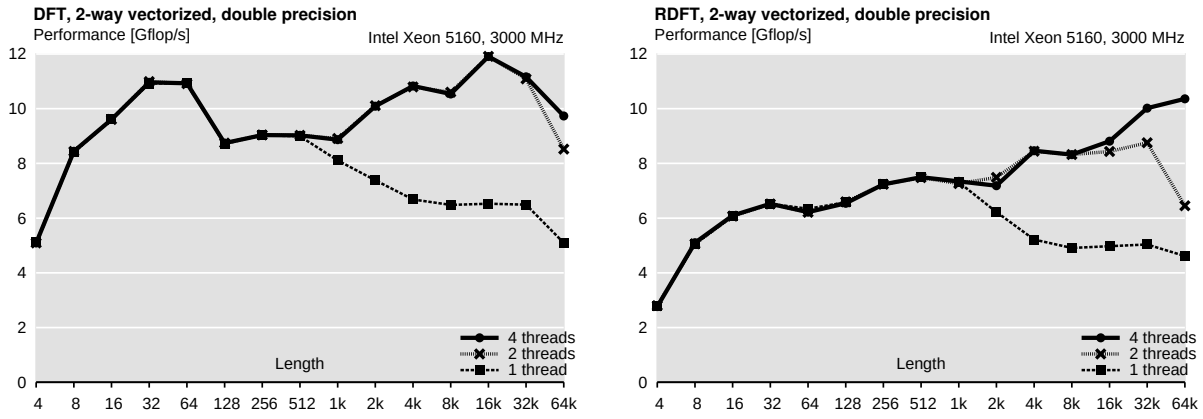


Figure 6.10: Parallelization efficiency. 4 threads give no speedup over 2 threads, due to low memory bandwidth. Platform: Intel Xeon 5160.

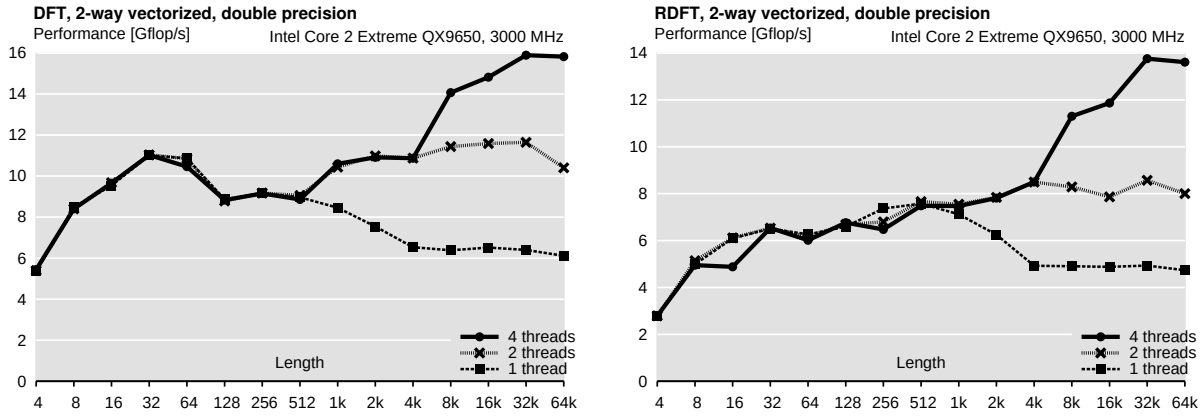


Figure 6.11: Better parallelization efficiency due to better memory bandwidth. Platform: Intel Core 2 Extreme QX9650.

6.5 Flexibility

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. We have already shown the versatility of the library generator with respect to the data type, novel transforms and novel algorithms, different types of vectorization and parallelization, and different transform size classes.

Here we overview the additional kinds of customization enabled by the library generator, discuss their benefits, and show several relevant examples.

We discuss three main types of customization: functional, qualitative and backend customization. *Functional customization* is concerned with extending or reducing the functionality of the libraru. *Qualitative customization* is used to modify some other non-functional properties of the library. When a library in different language is desired, *backend customization* is needed. In all three cases we give concrete examples of how our library generator can provide the given kind of customization.

6.5.1 Functional Library Customization

If some library functions do not exactly match the user requirements, there is often no other solution, but to either reimplement the desired functions or, if possible, preprocess the input or postprocess the output of the library function. Reimplementing is often not practical and might not result in good performance, and pre- or postprocessing also leads to a performance penalty.

In the case of open-source libraries, like FFTW, it is possible to modify the library itself, however, it is not practical for an average user, due to the high software complexity.

The most common examples of functionality customization include the choice of the data type (e.g. double or single precision, fixed or floating point), input or output data format (e.g. split or interleaved complex, different packing of symmetric complex sequences from the output of real DFT, out of order data), minor changes in the transform definition (e.g. scaling in DFTs and DCTs).

As an example of functional customization, we considered the well-known problem of scaling the outputs of the Fourier transforms (DFT and RDFT). The common definition of forward and

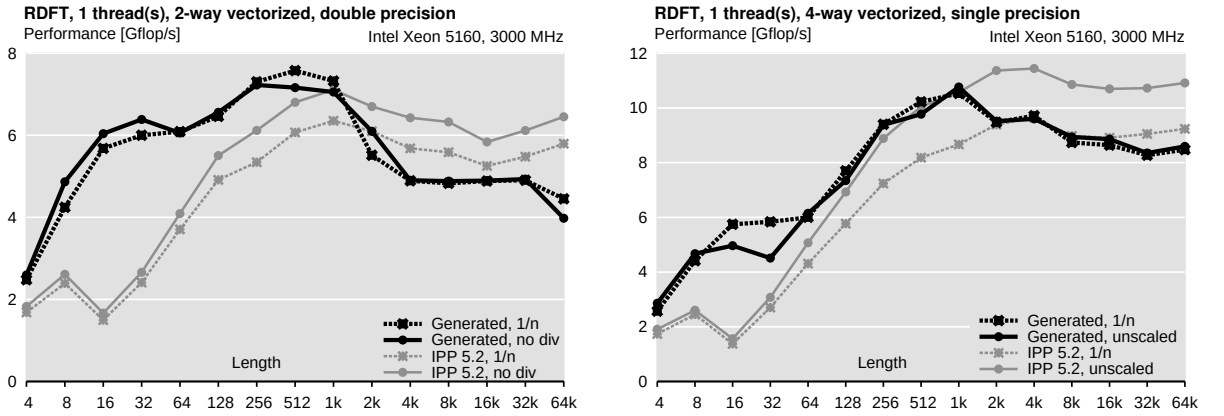


Figure 6.12: Generated RDFT libraries with built-in scaling. Generated libraries incur no performance penalty.

inverse DFT uses unscaled roots of unity, namely

$$\mathbf{DFT}_n = \left[\omega_n^{ij}\right], \quad \mathbf{DFT}_n^{-1} = \left[\omega_n^{-ij}\right], \quad \omega_n = e^{-2\sqrt{-1}\pi/n}.$$

Simple computation shows that applying forward and inverse DFT in succession result leads back to the original input scaled by n. For many applications this is acceptable, and these transforms are called unscaled . However, if the true inverse is desired the output of either the forward or inverse DFT must be scaled by $1/n$, or both must be scaled by $1/\sqrt{n}$.

Conider, for example, the real DFT in Intel IPP. The library provides different scaling modes. In Fig. 6.12 we show the performance of unscaled ("no div") and scaled ("1/n") IPP modes in both single and double precision. The scaling introduces an average performance penalty of 10% in the single precision case and 20% in the double precision case. Since the scalar multiplication can not possibly introduce a consistent 10% or 20% slow down to an $O(n \log n)$ algorithm, the scaling is probably implemented as a separate pass over the data set, which introduces unnecessary memory traffic.

The generated libraries in Fig. 6.12 incur no scaling overhead. The scaling is incorporated directly into the recursion steps (and thus codelets) using the rewrite rules which merge the diagonal matrices with adjacent loops.

6.5.2 Qualitative Library Customization

The common example of qualitative customization include trading off smaller code size for slower runtime, optimizing the average performance versus optimizing the best possible performance (e.g. at specific transform sizes), trading off accuracy for runtime, etc.

Most of the qualitative customization are not usually possible with fixed libraries, without reimplementation.

As an example of qualitative customization in Fig. 6.13 we show the code size versus performance tradeoff in generated libraries. The different lines in the plot are obtained by changing the number of generated base case sizes for each recursion step type.

The code sizes of generated libraries are rather suboptimal, so the absolute numbers from Fig. 6.13 are not directly comparable to optimized human-written code, and especially to the code optimized for size. The code size can be dramatically reduced by reducing the size of the recursion step closure. In this case and most other cases this reduction comes without any performance cost,

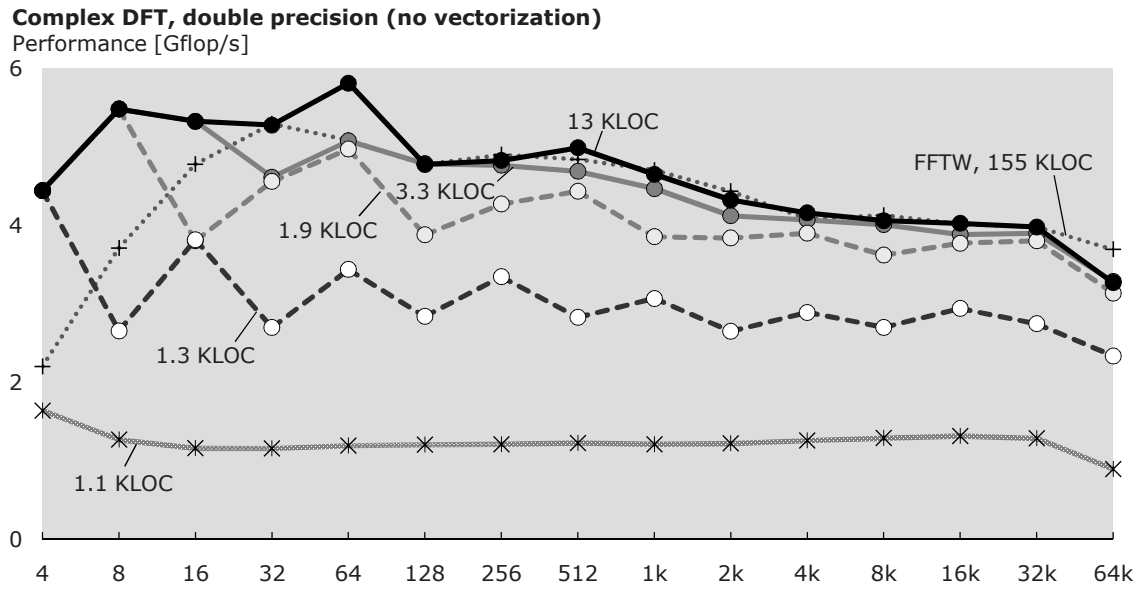


Figure 6.13: Performance vs. code size tradeoff in a generated DFT library. KLOC = kilo (thousands) lines of code. Generated libraries are for the DFT and 2-power sizes only. FFTW includes other transforms and supports all transforms sizes.

because many similar recursion steps can be unified into a single recursion step, similarly to how the scaling can be incorporated without any additional performance penalty, as shown in Fig. 6.12. Reducing the size of the closure reduces the code size due to the smaller number of recursions and base cases.

6.5.3 Backend Customization: Example, Java

Decoupling of transform specific and platform specific parts of the library generator allows us to easily retarget the generator to generate code in programming languages other than C++. In this section we describe our latest work, which provided the Java backend. The backend was implemented in a mere 2 days of work by Frédéric de Mesmay. However, the implications are tremendous. This allows, us to automatically generate the high-performance Java libraries for linear transforms.

In the rest of this section we describe the motivation for implementing numeric libraries in Java, alternative implementation approaches, and then benchmark our generated Java libraries.

Motivation. Recently, Java has emerged as one of the most popular programming languages for many applications. Java's slogan "Compile once — run everywhere" represents a paradigm shift from the traditional compiled programming languages. Java programs are compiled into portable byte-code, which can be executed by a special interpreter, called Java Virtual Machine (JVM). This allows Java byte-code files to be executed on a variety of platforms without recompilation.

Another important aspect of the Java is the extensive standardized APIs offering functionality ranging from standard data structures to graphical user interfaces (GUIs) and network access. Coupled with the portability provided by execution on the JVM, Java provides a true virtual platform.

In addition, JVM as the additional layer of indirection, offers additional important features. For example, JVM provides the controlled environment, where the running applications can be

restricted from full access to the host platform for security reasons; JVM performs consistency checks, which prevent common errors, such as out-of-bounds array accesses and buffer overflow attacks; security and robustness guarantees in addition to the true portability allows JVM to safely execute remote code. Thus, Java enables true heterogeneous distributed computing.

Thus, Java as a development platform became a popular choice in areas ranging from education to large-scale software development.

Interfaces to native libraries. Java programs can invoke native machine-specific libraries, such as FFTW and IPP, by using Java Native Interface (JNI). In this case, special wrappers must be provided for the native libraries. Even though this approach will most likely lead to the best performance, there are several disadvantages of this approach, stemming from the lack of virtualized JVM. Namely, the resulting Java application is no longer fully portable, it can not provide safety and robustness guarantees, and the resulting code is not "mobile", i.e. can not be executed by a remote machine, which does not have the machine-specific library.

Java numeric libraries. To fully take advantage of the Java and the JVM execution it makes sense to write numeric libraries in Java. In the context of numerical libraries and applications Java has been studied in [7, 26, 28, 29, 129].

Unfortunately, less effort is spent on Java numeric libraries and thus fewer optimized libraries exist, and a lot of important functionality is missing. In the domain of linear transforms, we found only two competing implementations: open-source JTransforms [128] and commercial JMSL [88]. JTransforms is a high performance library for linear transforms, which provides Java implementations of 1-, 2- and 3-dimensional real and complex DFT, DCT and DST. JTransforms is based on highly optimized C implementation from Takuya Ooura [92]. Wendykier in [129] describes the application of JTransforms to large scale image deblurring. JMSL is a commercial Java numeric library sold by Visual Numerics, JMSL has a wide range of functionality ranging from linear algebra to neural networks. JMSL has 1-dimensional real and complex DFT. We did not benchmark JMSL, due to lack of time.

We did not find any optimized libraries for FIR filters, wavelets, WHT and DCT-4.

Benchmarks. We benchmarked four generated Java libraries: DFT, RDFT, DCT-2 and an FIR filter. All libraries were non-threaded and non-vectorized. Vectorization is not possible in Java, because it is a cross-platform language, and thus does not provide hardware specific SIMD instructions. Threading, however, is supported, but our simple Java backend can't produce threaded Java code at the moment.

Tests were done on the Intel Xeon 5160 machine using Sun Java JDK 1.6 We compared DFT, RDFT and DFT-2 against JTransforms, which was restricted to 1 thread for an apples-to-apples comparison. FIR filter is compared to a naive double-loop implementation. The results are shown in Fig. 6.14.

The generated DFT, RDFT and DCT-2 libraries outperform JTansforms, except for the larger DFT sizes, where JTransforms is up to 10% faster. As before the generated libraries don't yet incorporate any optimization for out of L2 cache transforms, and thus we only show sizes that fit into the L2 cache. The slowdown at larger DFT sizes seems to come from the Java timer instability, which causes the runtime adaptation mechanism in the generated libraries to find suboptimal results (slower than the simplest model we have tried).

For FIR filters we did not find and competing high-performance Java library and thus we benchmark against a naïve double-loop Java implementation. The generated library is between 2 to 5.5 times faster than the naïve FIR code.

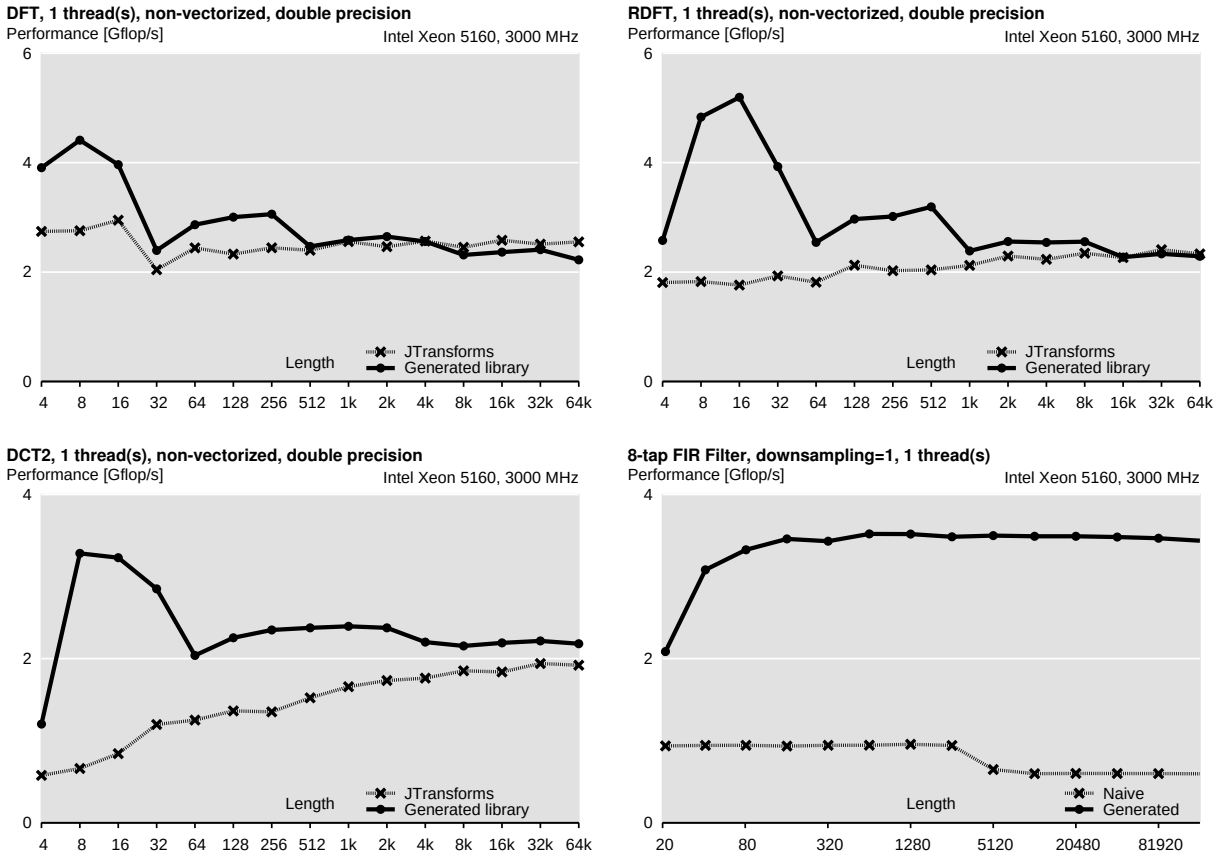


Figure 6.14: Generated Java libraries performance: trigonometric transforms.

While the overhead of the JVM interpreter is very high, today most JVMs employ JIT (just-in-time) compilers, which translate byte-code into native machine code at runtime. JIT compilers resulting in near native performance of Java applications, however some overhead is still unavoidable, because of additional runtime consistency and safety checks imposed by the JVM. In the case of numeric code, the most important check is the bounds check of array indices.

As a result of these checks, the generated DFT, RDFT and DCT-2 libraries perform about 1.5 – 2 times slower than the corresponding native (non-vectorized) C++ code. However, the FIR filter performs almost as well as the C++ implementation. We conjecture that most of the runtime array index bounds checking can be eliminated by the JIT compiler, since the FIR filter code has simple index expressions.

JTransforms complexity. JTransforms employs an iterative split-radix FFT algorithm with lots of specialized functions that implement different parts of the algorithm. Most of the functions represent a DFT transform possibly with a loop around it with some additional related computations, e.g., a permutation or a scaling by twiddle factors. These functions are equivalent to our recursion steps, and the full set of them to the recursion step closure. However, in this case the closure was derived manually. For comparison of the complexity of the implementation we show the recursion step count for JTransforms in Table 6.4. This table must be compared with Table 6.2.

The library has more recursion steps but overall less code than the generated libraries, because the base cases are provided for size-4 transforms only.

	number of recursion steps		total code size
Transform	scalar	parallelized	
DFT + RDFT	25	28	3.0 KLOC / 0.08 Mb
DCT-2 + DCT-3	25	28	2.8 KLOC / 0.08 Mb

Table 6.4: Number of recursion steps in JTransforms Java library.

6.5.4 Other Kinds of Customizaton

There are other numerous kinds of customization that are possible with the generator. In particular, it is possible to generate adaptive and non-adaptive libraries. More specifically, the standard libraries that we benchmarked all include three runtime phases, that the library goes through in order to compute the result:

1. *Planning phase.* In this phase the values for the degrees of freedom within the breakdown rules that lead to the best performance are chosen.

2. *Initialization phase.* Here, the necessary data (e.g. the twiddle factors for the Cooley-Tukey FFT) is precomputed, and if temporary memory buffers are needed, they are allocated.

3. *Compute phase.* The actual transform is computed.

It might be useful to generate, besides the regular three-phase library, also the two-phase and even the single-phase libraries. The two-phase library would exclude the planning phase, and thus it is a non-adaptive library. The single-phase library in addition to being non-adaptive, does not have the precomputation stage.

Another possible kind of customization is the generation of the specific library interface. This would be achieved by modifying the backend, and thus can also be called backend customization. For the DFT, for instance, there are three popular interfaces: DFTI [119], VSIPL [69, 78], and FFTW [61].

6.6 Detailed Performance Evaluation

In Appendix A we provide performance plots with exhaustive benchmarks for the following cross product:

$$\left\{\begin{array}{c}\text{no vectorization}\\ \text{2-way vectorization}\\ \text{4-way vectorization}\end{array}\right\} \times \left\{\begin{array}{c}\text{1 thread}\\ \text{2 threads}\\ \text{4 threads}\end{array}\right\} \times \left\{\begin{array}{c}\mathbf{DFT}\\ \mathbf{DHT}\\ \mathbf{RDFT}\\ \mathbf{DCT\text{-}2}\\ \mathbf{DCT\text{-}3}\\ \mathbf{DCT\text{-}4}\\ \text{FIR filter}\\ \text{Downsampled FIR filter}\end{array}\right\}$$

On three platforms: Intel Xeon 5160 (Section A.1), AMD Opteron 2220 (Section A.2), and Intel Core 2 Extreme QX9650 (Section A.3). We did not benchmark non-vectorized FIR filter libraries, and did not benchmark any FIR filter libraries on Core 2 Extreme QX9650.

No vectorization corresponds to double precision, *2-way vectorization* corresponds to double precision using 2-way SSE2 intrinsics, and *4-way vectorization* corresponds to single precision using 4-way SSE and SSE2 instrinsics.

Chapter 7

Future Work and Conclusions

We restate from the introduction

> **The goal of this thesis is the computer generation of high performance libraries for the entire domain of linear transforms. The input is a high level algorithm specification, request for multithreading, and vector length. The output is an optimized library with runtime platform adaptation mechanism.**

In this thesis we described a prototype that achieves the above goal. This goal and the automation of high-performance library development in general is a problem at the core of computer science. We have shown that for an entire domain of structurally complex algorithms, highest performance libraries can be generated automatically from the algorithm specification (e.g., breakdown rules in Table 2.3). The key is a properly designed domain-specific language ($\sum$-SPL in our case) that makes it possible to perform all difficult tasks (extraction of recursion steps, vectorization, etc.) at a high level of abstraction using rewriting and other techniques.

The generated libraries in all cases have performance comparable to hand-written code, and in many cases outperform the hardware vendor libraries, which are implemented by experts, who understand the platform very well.

We have based our work on Spiral, the code generator for linear transforms, and as part of this thesis extended Spiral capabilities in three key areas: 1) we enabled generating efficient loop code by introducing $\sum$-SPL and loop merging; 2) we enabled generating general-size transform code by automatically analyzing the breakdown rules and computing the recursion step closure; 3) we provided a general vectorization and parallelization framework to deal with increasing on-chip parallelism of modern computer platforms.

Clearly, a lot can be done to improve the existing functionality. Looking into the future, two main directions can be identified: extending the range of supported hardware platforms and computing paradigms and extending the supported functionality beyond transforms. In fact, first steps in both of these directions were already made by other members of the Spiral group.

Given the limitations of clock frequency scaling and of conventional vector and thread parallelism, it appears unavoidable that new computing paradigms will become adopted in the future. For example, today we see the emergence of hardware accelerator technologies, like on-chip graphics processor units (GPUs) being used for general numeric computing and on-board Field Programmable Gate Arrays (FPGAs). The high-level domain-specific and declarative nature of

SPL and $\sum$-SPL provide a convenient basis for capturing new paradigms. For example, [83, 101] uses Spiral, SPL breakdown rules and rewriting to generate efficient DFT implementations on FPGAs; [41] uses Spiral to perform automatic software-hardware partitioning; [30] describes the application of Spiral to generate code for distributed memory workstation clusters using MPI; [36] uses Spiral to generate code for the Cell processor; [58] shows some preliminary results on using Spiral to generate GPU-accelerated implementations.

To capture a larger domain of numeric functionality, [57] proposes the *operator language*, which extends SPL and $\sum$-SPL to non-transform non-linear functionality. If successful, this work may enable automation in other numerical domains, relevant for signal processing, communications, or scientific computing. linear transforms, such as, for instance, linear algebra.

Next we provide a more detailed overview of the contributions of this thesis, current limitations, and future work.

7.1 Major Contributions

We summarize the major contributions of this thesis below.

- **Looped code generation and $\sum$-SPL.** The loop merging framework described in Section 2.4 enables generating efficient looped code for a wide variety of transforms. The formalism enables elimination of redundant loops for well-known algorithms with well-understood data reordering like Cooley-Tukey FFT (strides), and lesser known algorithms, like the general-radix fast DCT, Rader FFT and prime-factor FFT. This part is joint work with Franz Franchetti, and is also reported in [52].

- **General size code generation via the recursion step closure.** Our approach to general size code generation is described in Chapter 3. The key to the ability to generate general size library is being able to statically (i.e., at code generation time) compute the finite recursion step closure. This enables the generation of full transform libraries similar to FFTW, rather than a collection of a few fixed transform sizes.

- **Parallelization and vectorization framework.** The parallelism framework is described in Chapter 4. It enables the generation of code that exploits two kinds of parallelism present in modern computers, namely SIMD vector parallelism and MIMD thread parallelism. Usually, high-performance numeric software relies on manual programming to exploit this available parallelism, instead of relying on a general purpose compiler. We automate this time consuming process, and have shown that the generated libraries have competitive performance with hand-written code. This part builds on work with Franz Franchetti, and is also reported in [53, 54].

- **Full automation.** We demonstrate a system prototype that achieves full automation in library implementation. This is facilitated by the library plan construction (described in Chapter 5), which combines multiple implementations of each recursion step and enables the generation of a single object (e.g., a C++ class) for each recursion step. Further, the library plan enables the "whole-library" analysis, which we use to automatically partition the parameters into hot (available at compute time) and cold (available at initialization time). Such analyses are not possible if the generator is not aware of the full set of recursion step implementations that form the library.

Finally, we want to note that generators by nature are much more flexible than hand-written code. The flexibility in terms of the choice of transform/algorithm is demonstrated through a number of different experiments in Chapter 6. The flexibility in customizing one specific type of functionality is discussed in Section 6.5.

7.2 Current Limitations

Our current prototype implementation has a number of limitations, all of which should be regarded as the short-term "future work".

- SPL vectorization rules are not completely ported to $\sum$-SPL, some breakdown rules are still missing. This results in suboptimal performance, in particular, with single precision (4-way vectorized) libraries.
- Base cases of the libraries are not vectorized (with the exception of double precision DFT). This results in suboptimal performance for small sizes in vectorized libraries, the problem is especially visible in single precision (4-way vectorized) libraries.
- The runtime search over alternative breakdown rules is not implemented, instead the library uses the first applicable breakdown rule. However, the library can search over the degrees of freedom within one breakdown rule (i.e., the radix in the Cooley-Tukey FFT (2.1)). Again, this may result in suboptimal performance in some cases. It also precludes using a different algorithm for transforms whose data set is not L2 cache resident, and results in severe performance degradation in this case. However, a working prototype of this kind of search was recently implemented by Frédéric de Mesmay, and promises to fix these problems.
- In large-size (non L2 cache resident) DFT related transforms standard algorithms are not always suitable due to large 2-power strides. We could not, however, provide alternative algorithms, since the libraries did not provide full adaptation as mentioned in the previous bullet, and thus could not choose the better algorithm.
- Transforms cannot be invoked from within the initialization of other transforms. This is needed to implement frequency domain FIR filtering algorithms, Rader and Bluestein FFT, and other algorithms. This limitation is due to the fact that the precomputations requiring transforms in Spiral are hard-coded to be performed at code generation time. To solve this problem the affected breakdown rules in Spiral must be updated.
- Constraint solving in parametrization (Section 3.3.3) requires at least a full linear system solver. We have not incorporated a full solver, since in most cases the systems are close to being trivial. In the most general case, the system of equations may not be linear. However, the non-linearity we have encountered was restricted to equations of the form $u_1 u_2 = u_3$, in which case we could simply treat the product $u_1 u_2$ as one variable, since in those cases there were no separate constraints on u_1 and u_2. Finally, we do not capture all of the constraints in the parametrization; for example, some $\sum$-SPL expressions that correspond to square matrices, become matrices of size $u_1 \times u_2$, with no known guarantee that $u_1 = u_2$. We have not encountered a problem related to this, and it does not happen if GTI (Section 3.8) is used.

7.3 Future Work

In the longer term, we see three main research directions. The first direction is implementing new optimizations, increasing the coverage of existing optimizations, and implementing other extensions within the current domain of linear transforms. The second direction is extending our results to new domains, for example, using the *operator language* [57]. The third direction is aligning the capabilities of the generator with the needs of the industry. Broadly speaking, this means increasing the robustness of the generator and the generated code.

New optimizations and other extensions. A number of important optimizations can be implemented within the generator to improve the generated code. We list some possibilities:

- **Integrate offline and online search.** By offline and online we mean the search that occurs at code generation time and library initialization time respectively. Currently, we have a very restricted integration between these two adaptation mechanisms. The transform base cases are platform-adapted using offline search, and the best recursion is determined using online search. However, much more is possible. For example, multiple alternative breakdown rules can be compiled into a single adaptive library. However, it might be the case that one breakdown rule consistently outperforms the others. If this can be determined offline, the library can be compiled with only one breakdown rule, reducing the overall code size.

- **Improve storage management.** The library generator needs more robust support of in-place transforms, and more efficient management of temporary storage.

- **Using automatically constructed performance models instead of runtime platform adaptation.** Originally we provided a hand-tuned model instead of search to select the best values for the degrees of freedom in the library. The hand-tuned model was always very successful for a given library (within 10% of search). However, changing the platform, data type, vectorization, number of threads, transform or breakdown rules always led to the need to adjust the model. Given the success of models, it would really be useful to automatically construct such models using machine learning techniques, in the spirit of work by Singer and others [109]. This could be considered a form of offline search.

- **Implement software pipelining.** All Intel and AMD processor based platforms that we benchmarked are out-of-order processors, with very small number of registers. On these platforms software pipelining is not beneficial. However, on other platforms with processors with in-order execution, it may make a noticeable performance difference.

- **Improve scheduling for straightline code.** Clearly, software pipelining is tied to code scheduling. However, scheduling is also useful without software pipelining. For example, even on out-of-order processors, scheduling can improve performance by grouping distant independent instructions in order to overcome the size limitation of the reorder buffer. Other factors may also contribute to performance. For example, Intel processors seem to favor blocks of loads followed by blocks of stores, while on other platforms code sequences with interleaved loads and stores may perform faster. Generally speaking, scheduling is the task of the C++ compiler, however, often it does not do it optimally, and the order of statements in the generated code can affect the final outcome. In a generator one could try different scheduling strategies on a given platform. One example of scheduling statements in straightline C code is described in [59].

Other domains. Some of the work we have done as part of this thesis is not specific to linear transforms. The essence of our approach are specially designed domain-specific languages (SPL, $\sum$-SPL, index-free $\sum$-SPL, intermediate code representation), multi-layered rewriting, and online and offline search mechanisms which use alternative breakdown rules and/or degrees of freedom within the breakdown rule. This approach can be applied to other domains, for which a suitable domain-specific language exists. One example is described in [57], which proposes an extension to SPL and $\sum$-SPL which can potentially capture several other numeric domains.

Needs of the industry. As we learned from [11] being able to generate fast numeric code is not yet sufficient to enable inclusion into a commercial numeric library, like Intel IPP or MKL.

We give just a small sample of issues that must be addressed:

- SIMD vectorized code must support both aligned and unaligned input and output vectors. (This is usually handled by specialization.)
- Code must adhere to preexisting library conventions, e.g., concerning memory allocation, in threading, etc.
- Besides supporting multitude data types, which Spiral does rather well, hybrid functions might be needed, where the input data type (say, 16-bit integer) is different from the output data type (say, 32-bit floating point).
- Due to the large amounts of generated code (potentially, thousands of functions), it must be directly pluggable into the existing library without any further modifications. In particular, function and file naming must conform to some standard naming scheme, processor dispatch must be handled, makefiles must be automatically generated, etc.
- The generated code must adhere to a certain predefined interface. For example, DFTI [119] in the case of DFTs, or the internal library interface, when a lower-level primitive is generated.

Any one of the individual issues above do not present a serious research challenge. However, in combination, they require a systematic approach for their efficient, extensible, and scalable solution. Our separation of the library generator into *Library Structure* and *Library Implementation* modules (Fig. 3.1 in Chapter 3) is the first step in this direction.

Appendix A

Detailed Performance Evaluation

A.1 Intel Xeon 5160

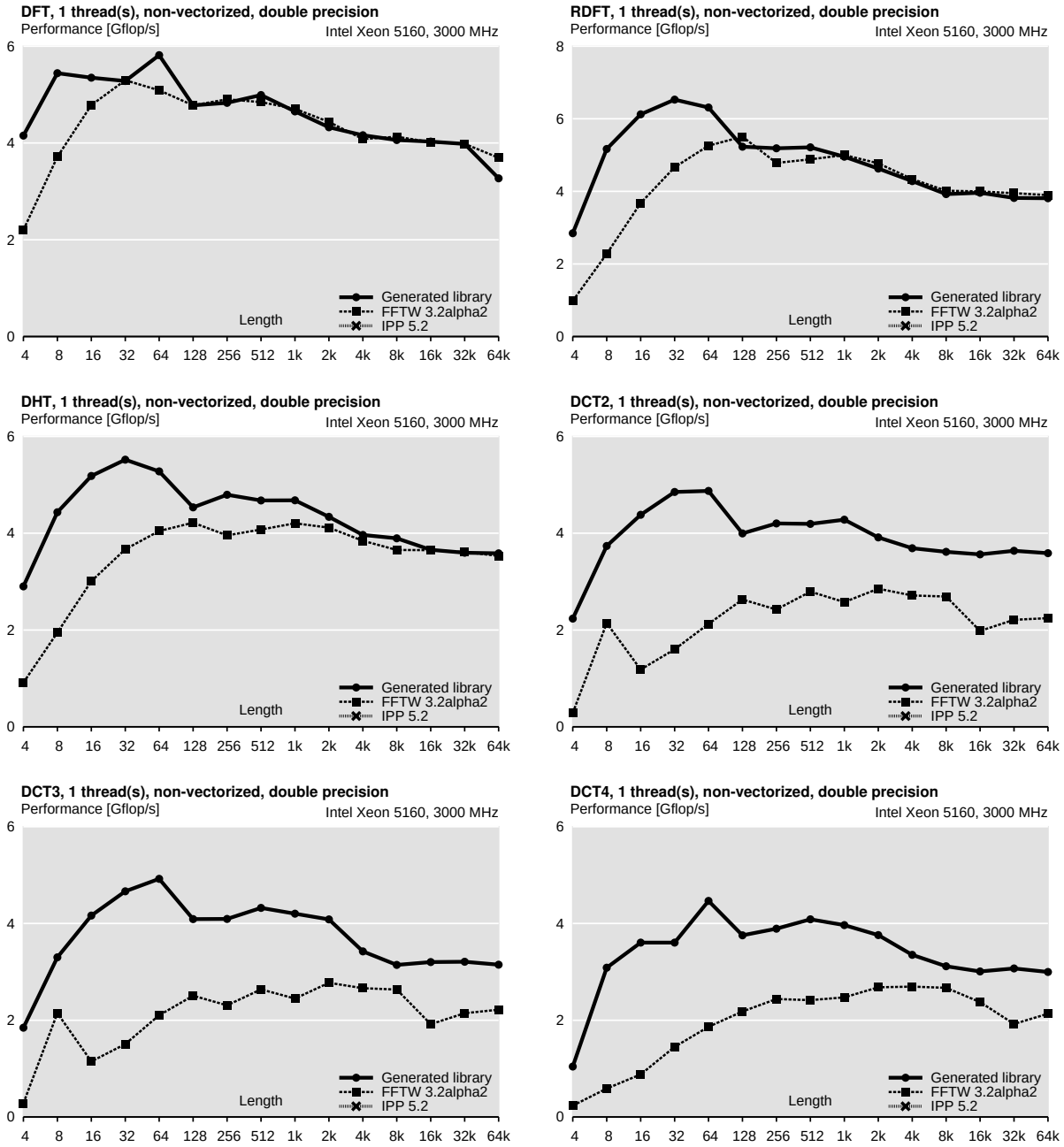


Figure A.1: Generated libraries performance: trigonometric transforms, no vectorization (double precision), no threading. Platform: Intel Xeon 5160.

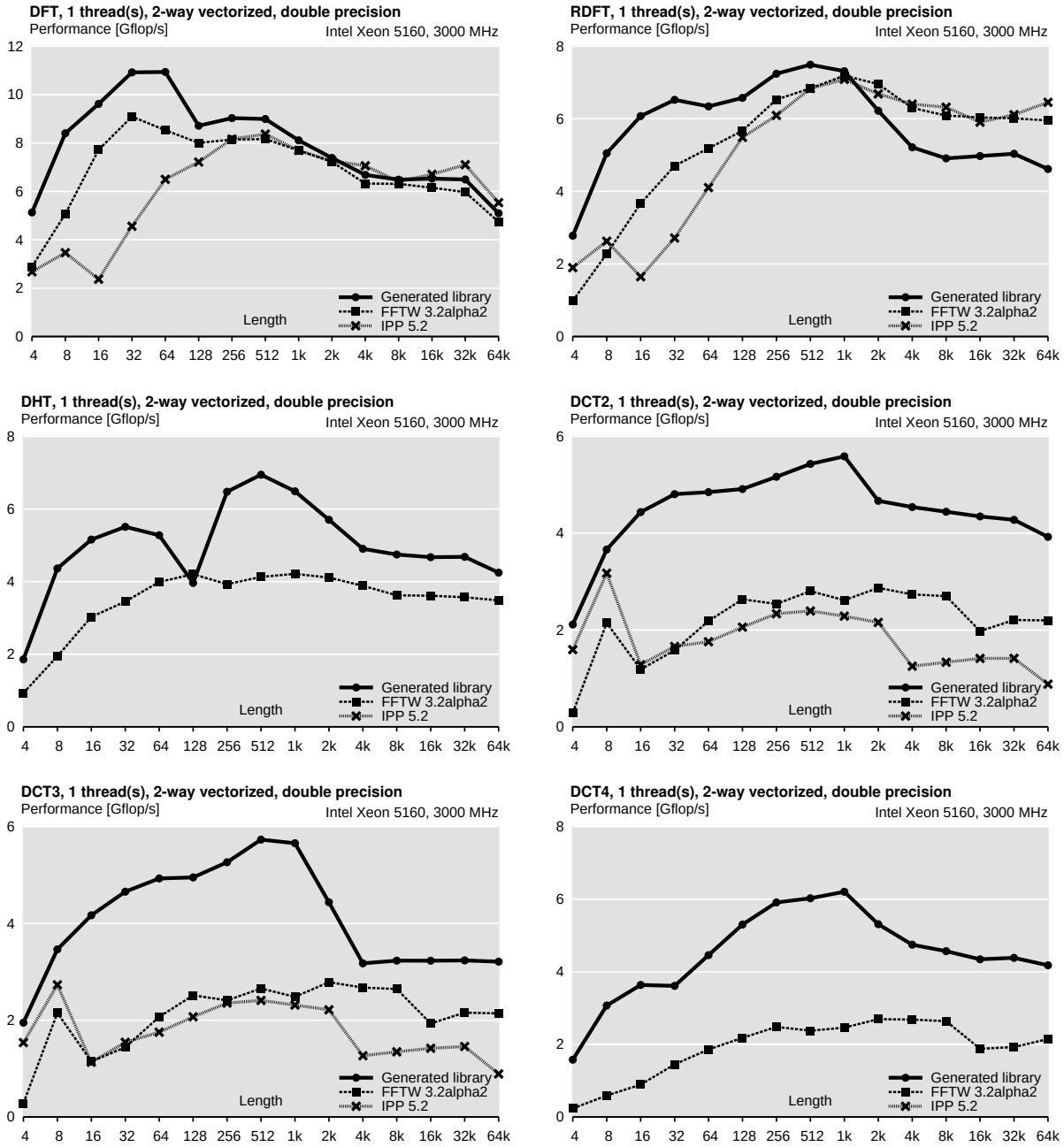


Figure A.2: Generated libraries performance: trigonometric transforms, 2-way vectorization (double precision), no threading. Platform: Intel Xeon 5160.

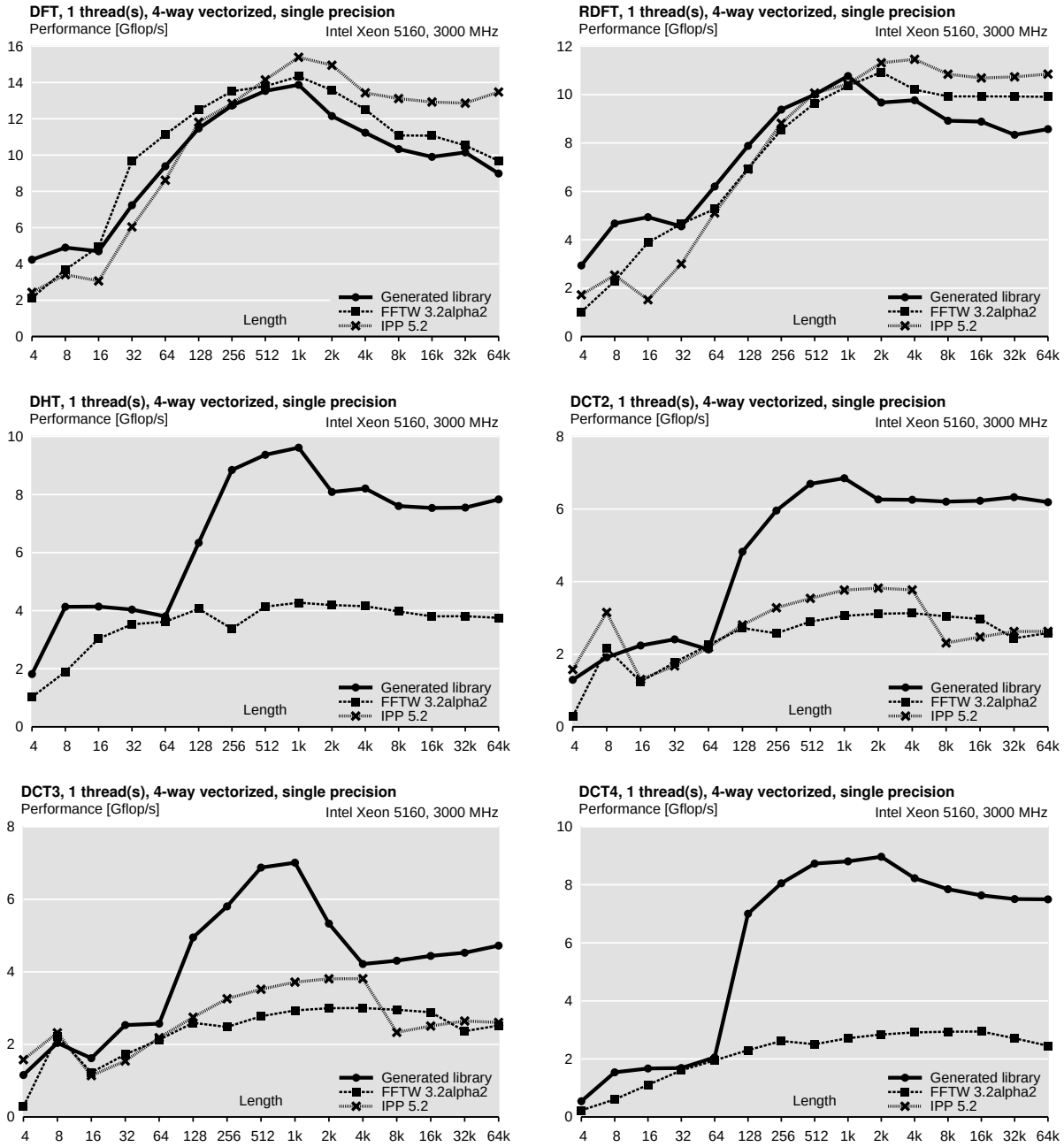


Figure A.3: Generated libraries performance: trigonometric transforms, 4-way vectorization (single precision), no threading. Platform: Intel Xeon 5160.

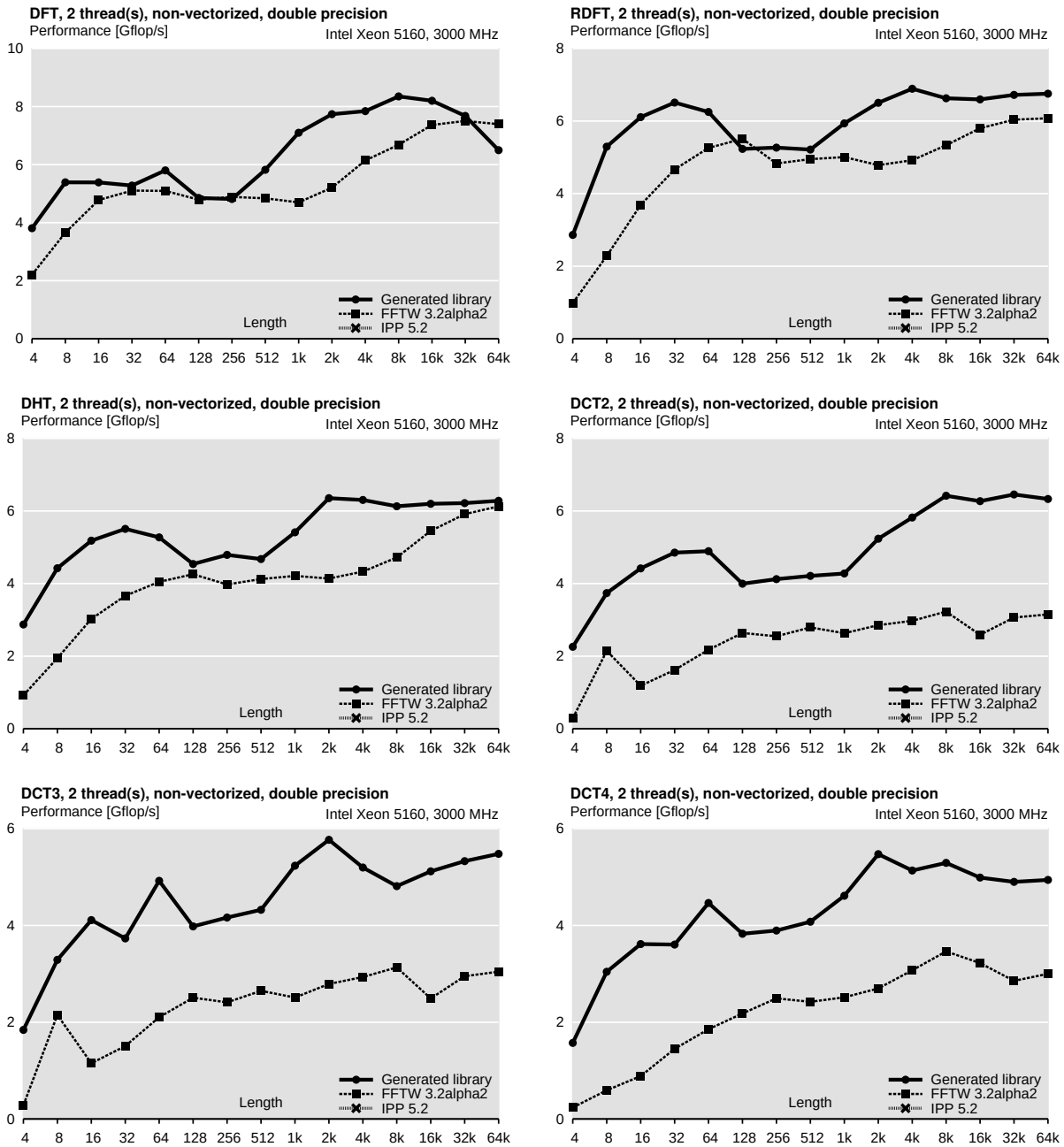


Figure A.4: Generated libraries performance: trigonometric transforms, no vectorization (double precision), up to 2 threads. Platform: Intel Xeon 5160.

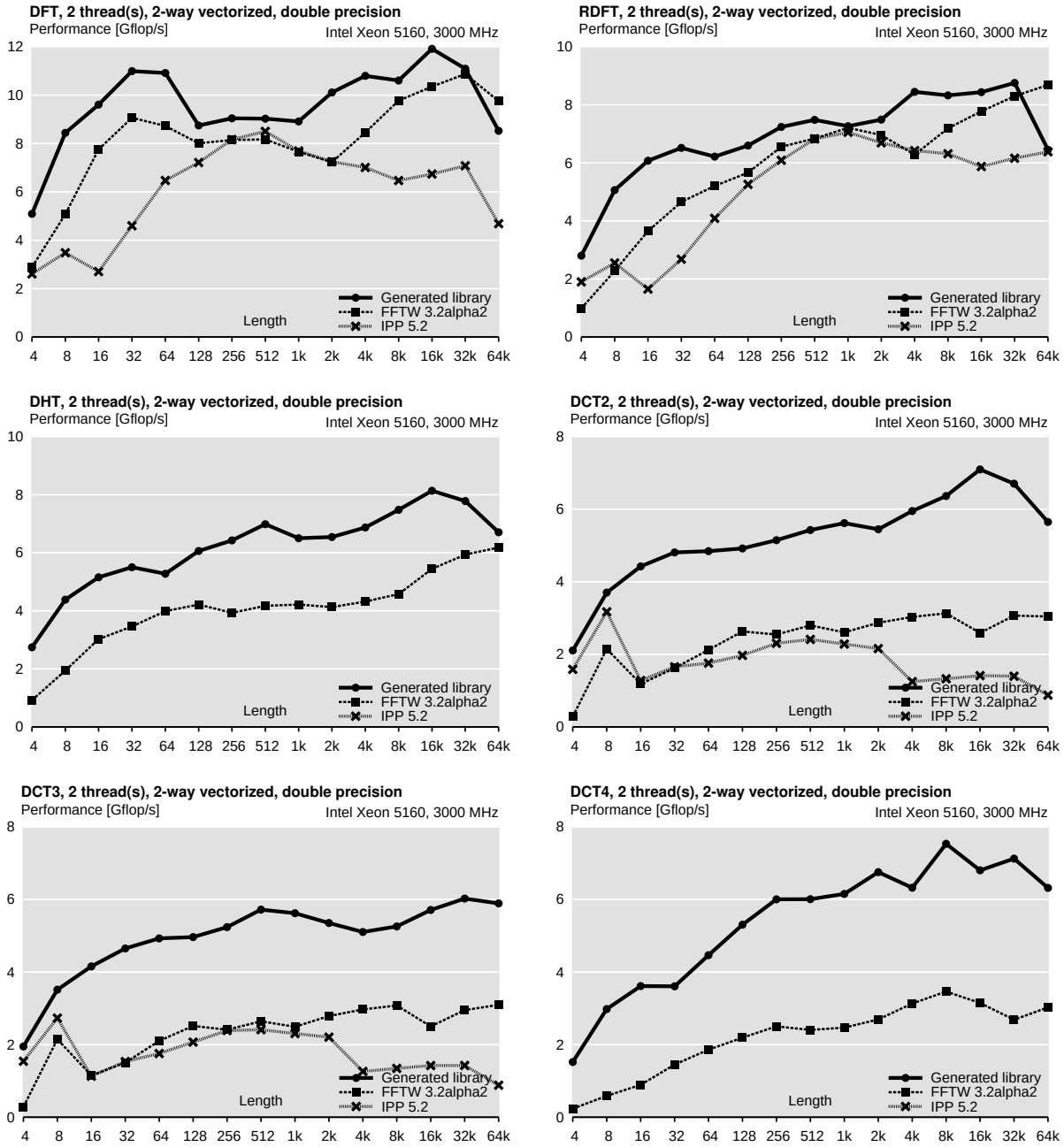


Figure A.5: Generated libraries performance: trigonometric transforms, 2-way vectorization (double precision), up to 2 threads. Platform: Intel Xeon 5160.

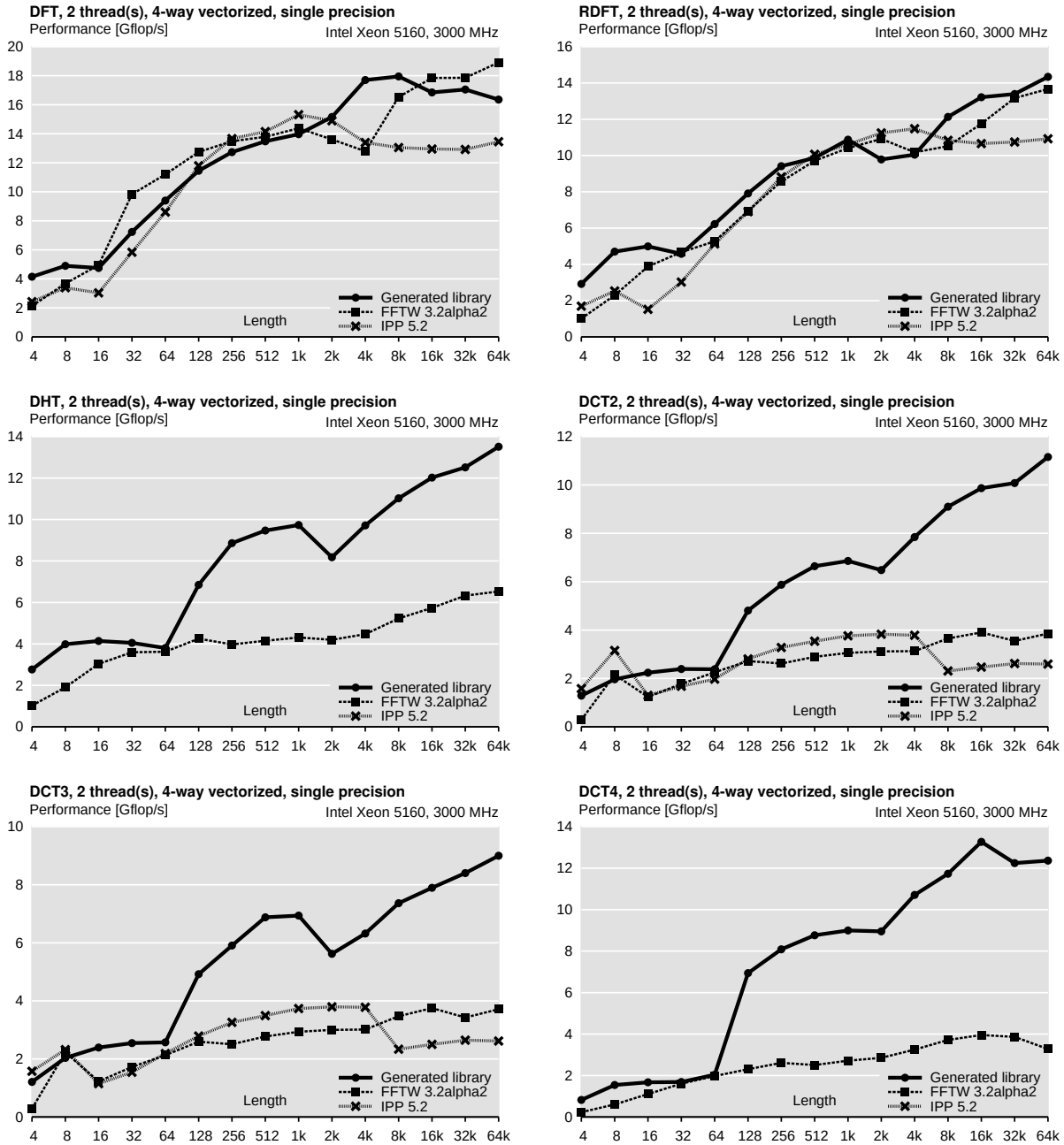


Figure A.6: Generated libraries performance: trigonometric transforms, 4-way vectorization (single precision), up to 2 threads. Platform: Intel Xeon 5160.

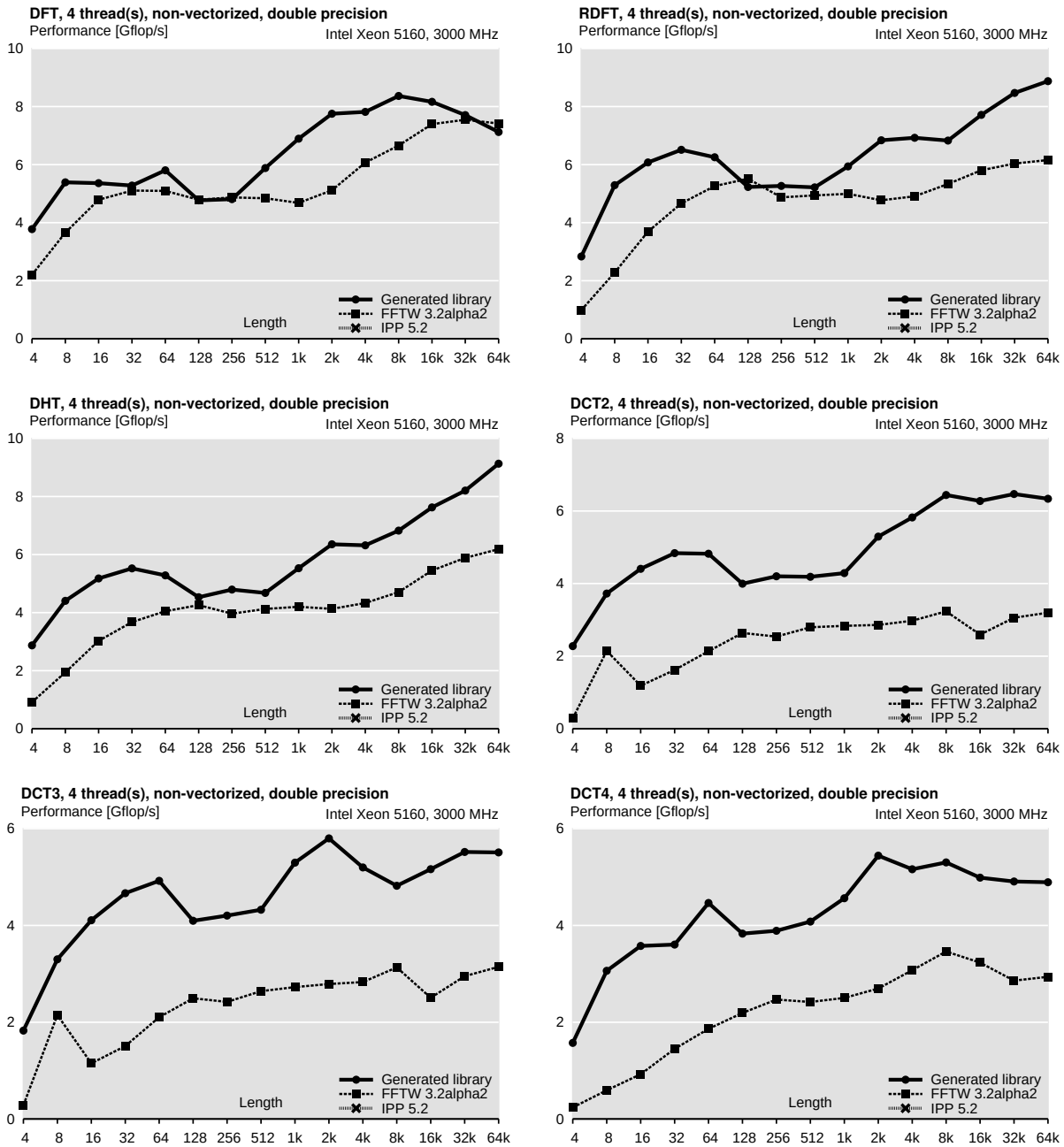


Figure A.7: Generated libraries performance: trigonometric transforms, no vectorization (double precision), up to 4 threads. Platform: Intel Xeon 5160.

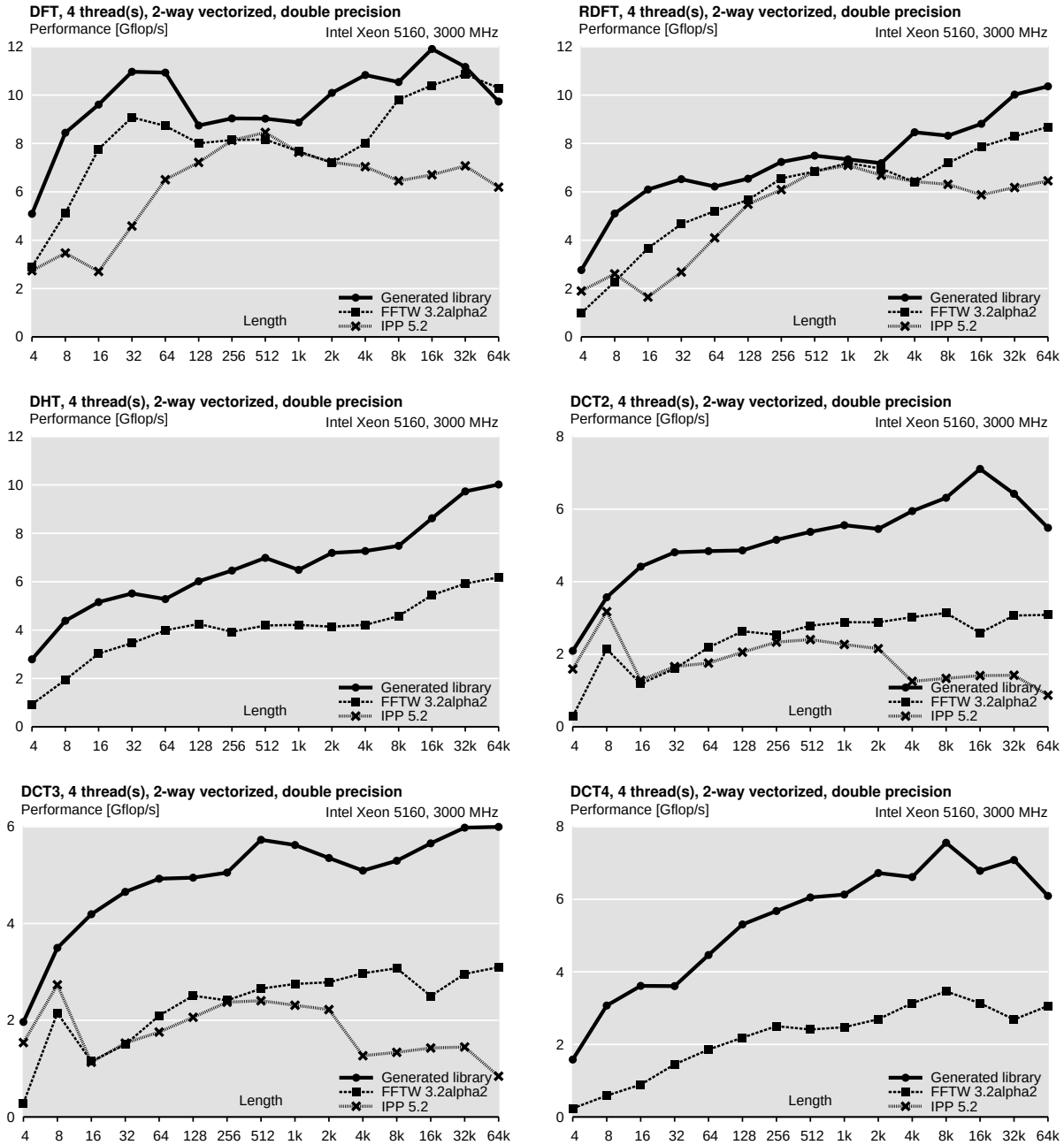


Figure A.8: Generated libraries performance: trigonometric transforms, 2-way vectorization (double precision), up to 4 threads. Platform: Intel Xeon 5160.

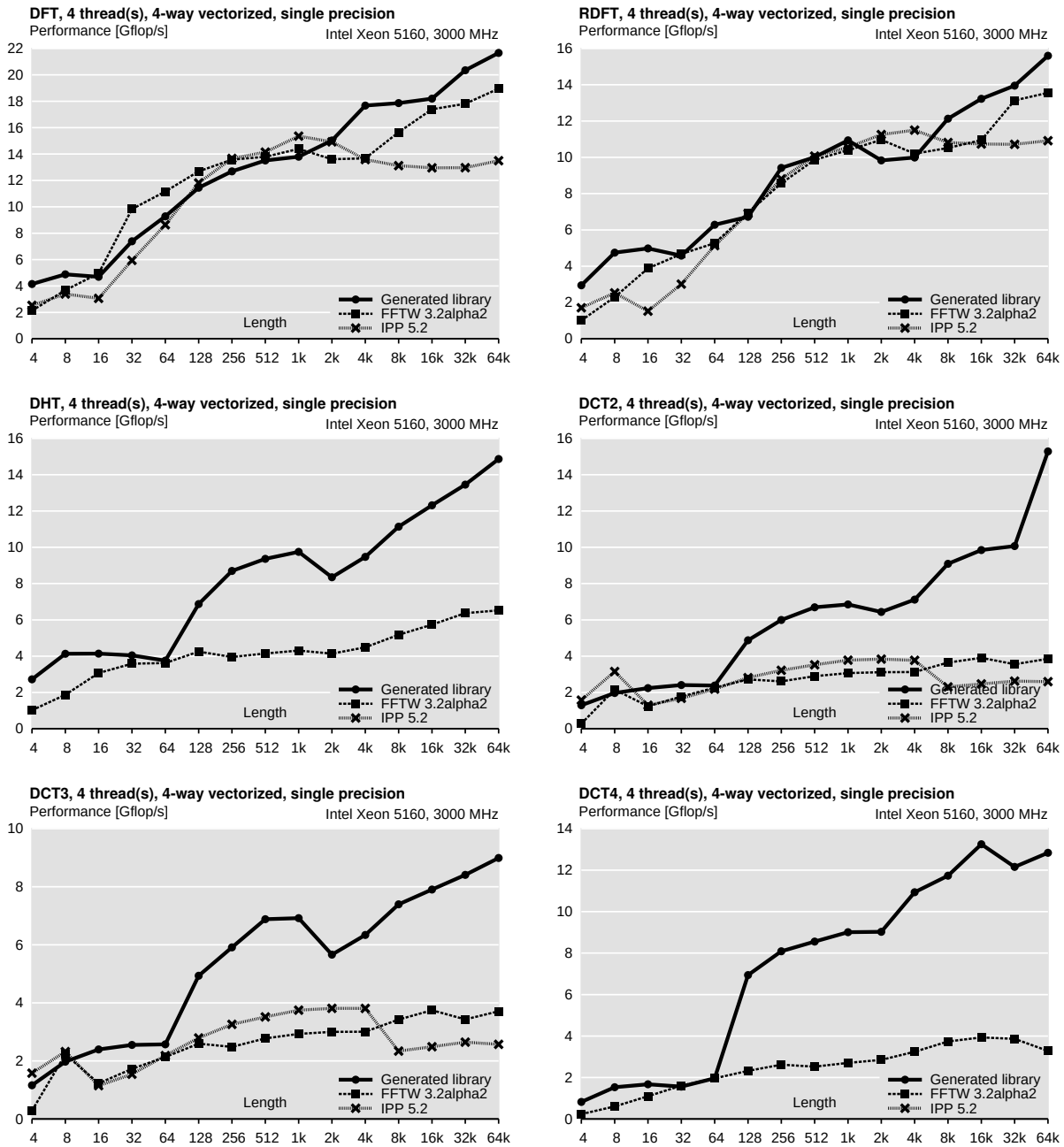


Figure A.9: Generated libraries performance: trigonometric transforms, 4-way vectorization (single precision), up to 4 threads. Platform: Intel Xeon 5160.

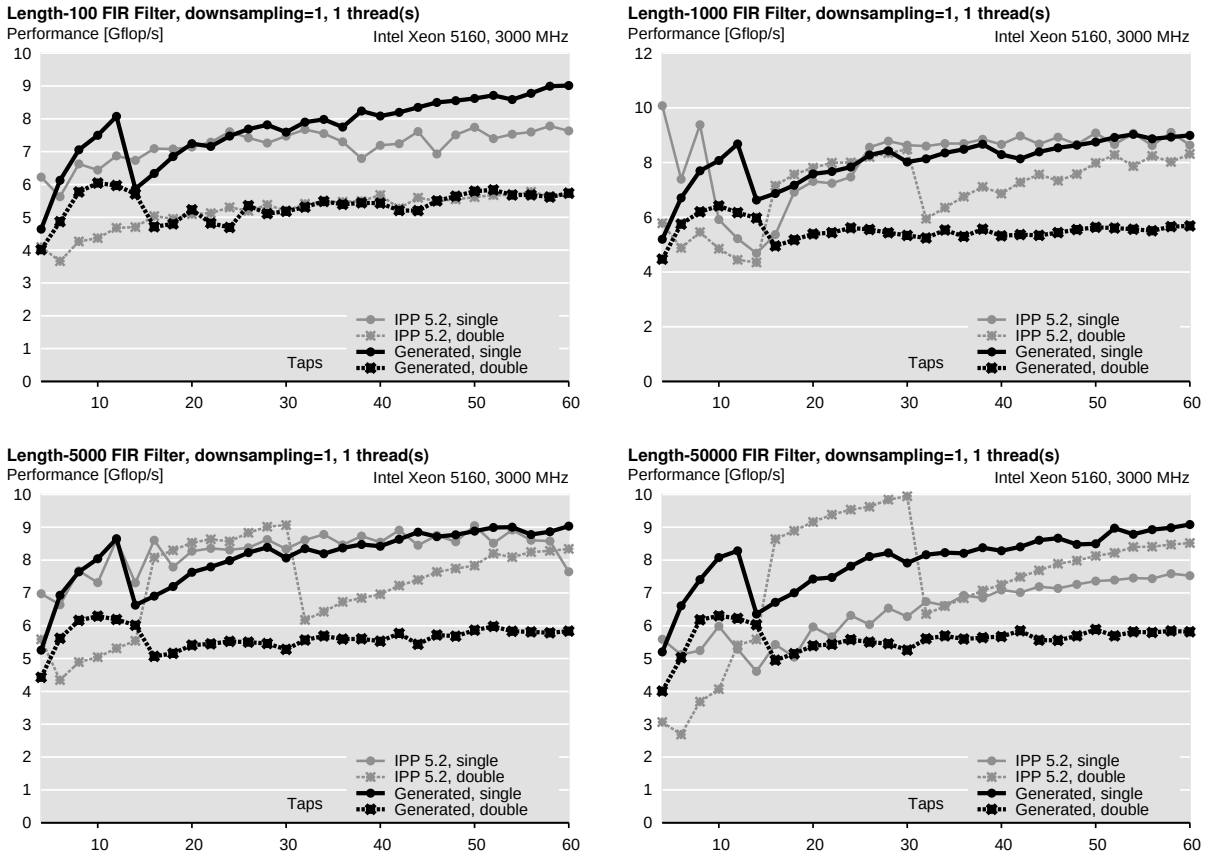


Figure A.10: Generated libraries performance: FIR filter, downsampling = 1, varying number of taps, up to 1 thread(s). Platform: Intel Xeon 5160.

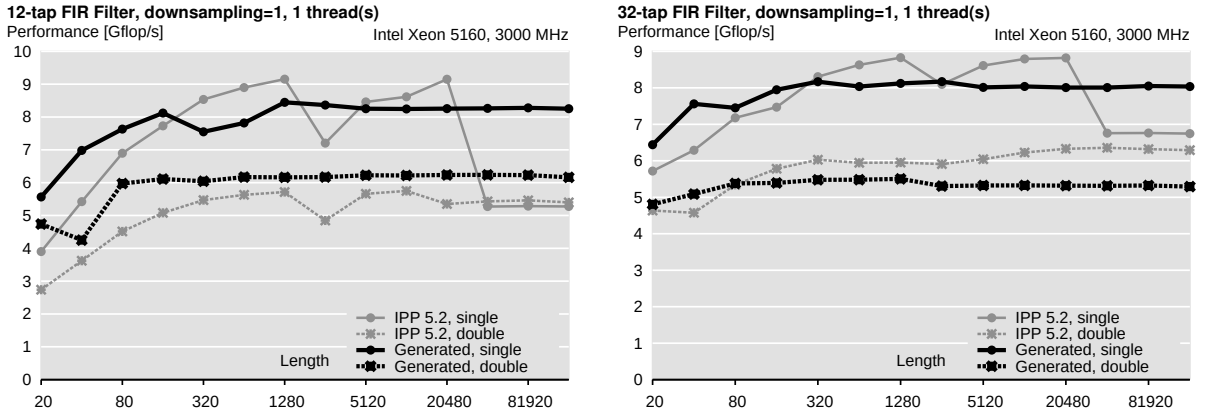


Figure A.11: Generated libraries performance: FIR filter, varying length, up to 1 thread(s). Platform: Intel Xeon 5160.

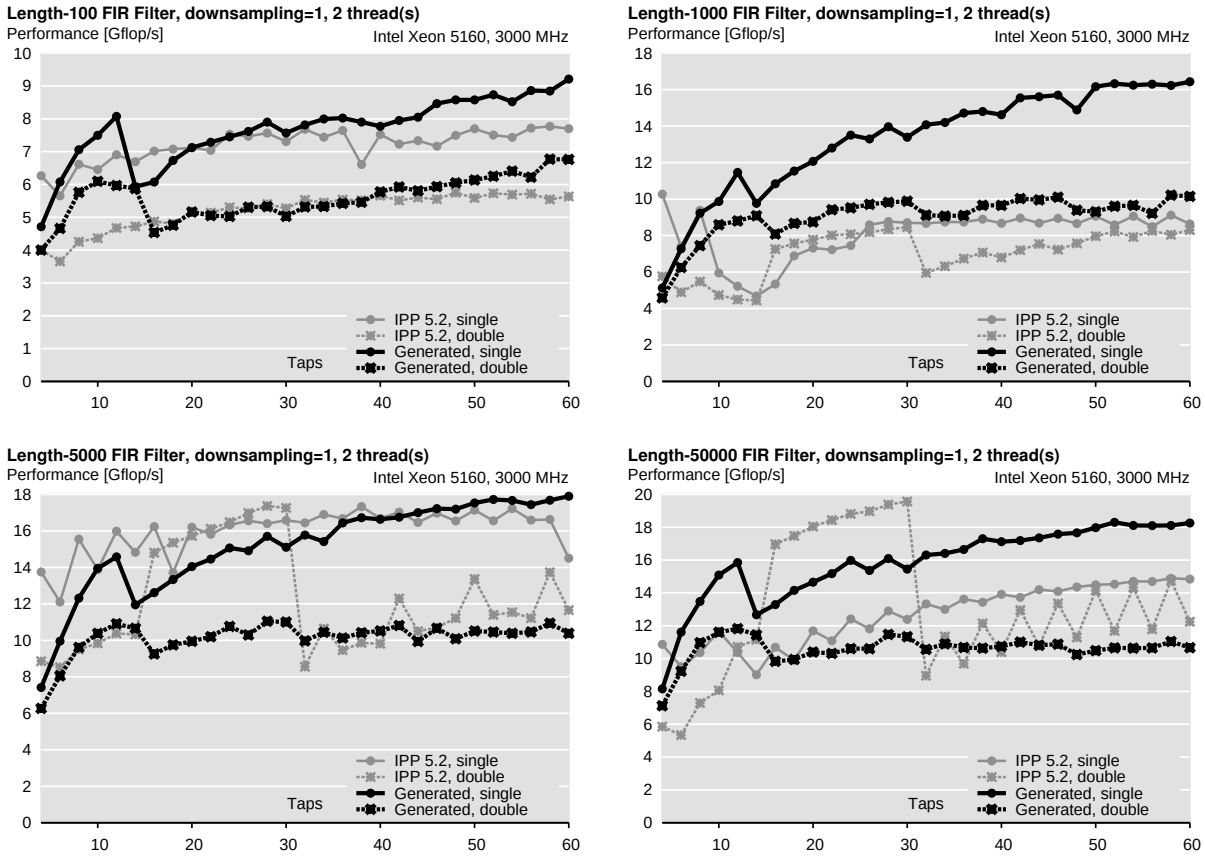


Figure A.12: Generated libraries performance: FIR filter, downsampling = 1, varying number of taps, up to 2 thread(s). Platform: Intel Xeon 5160.

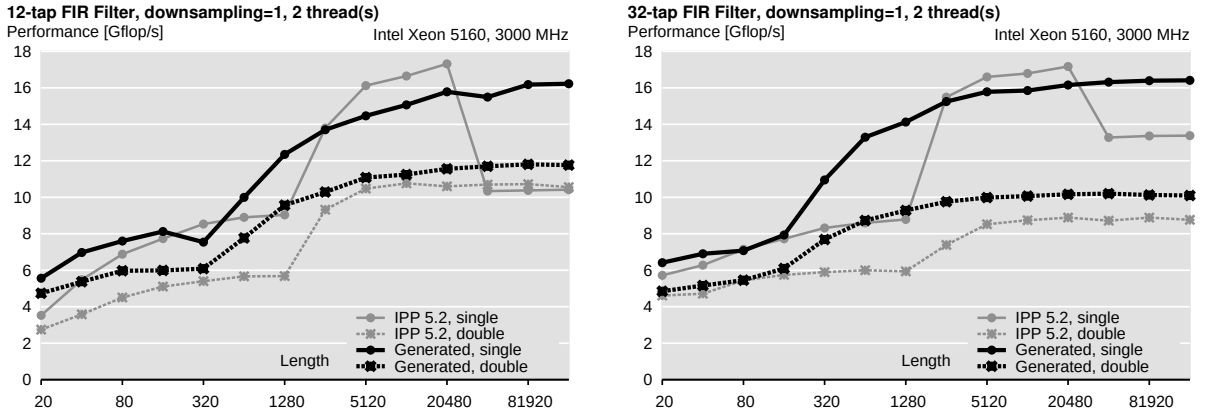


Figure A.13: Generated libraries performance: FIR filter, varying length, up to 2 thread(s). Platform: Intel Xeon 5160.

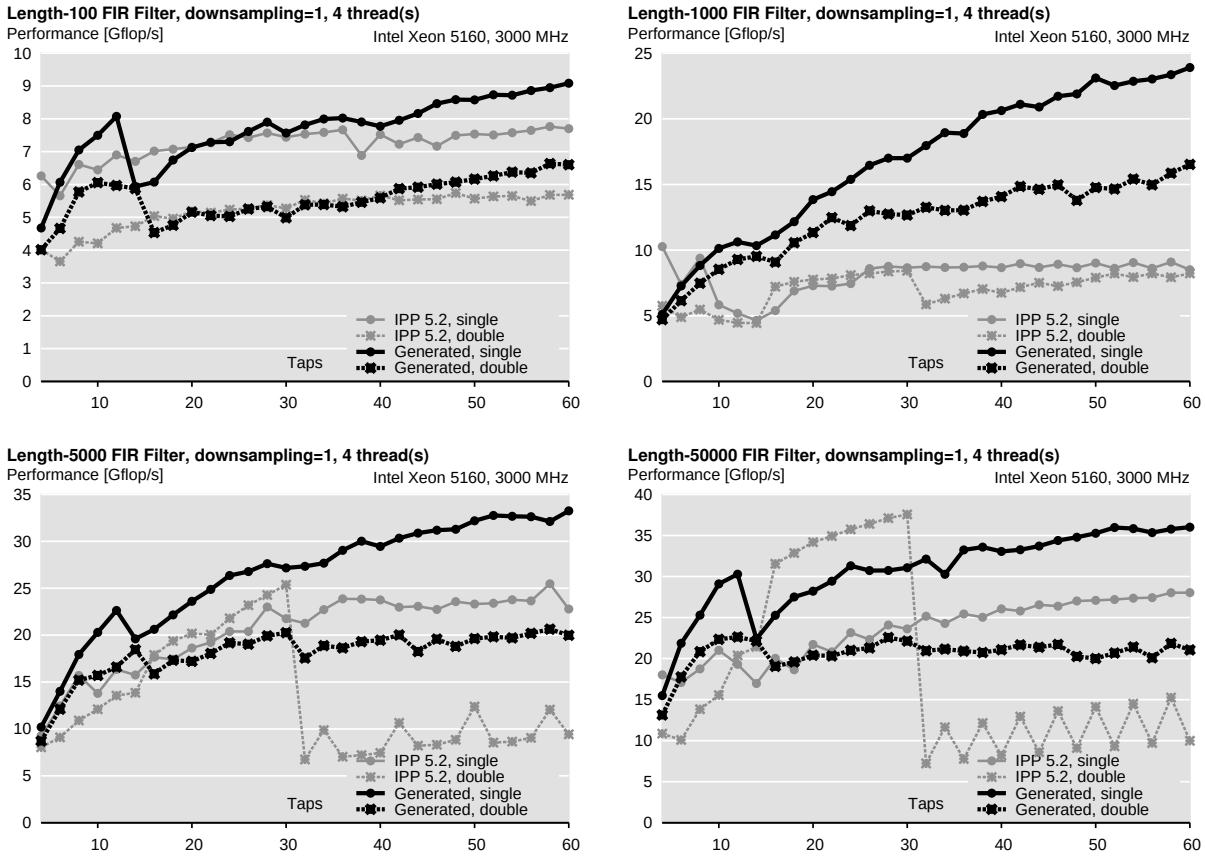


Figure A.14: Generated libraries performance: FIR filter, downsampling = 1, varying number of taps, up to 4 thread(s). Platform: Intel Xeon 5160.

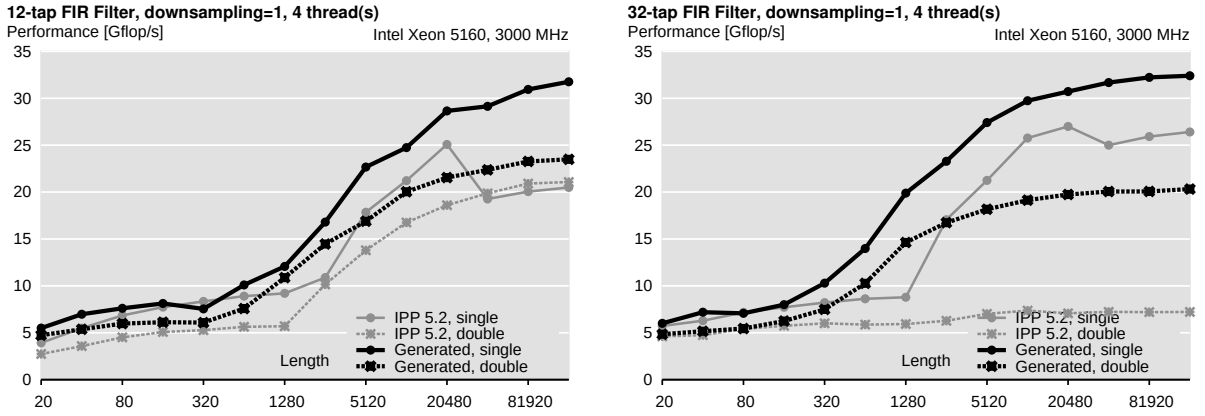


Figure A.15: Generated libraries performance: FIR filter, varying length, up to 4 thread(s). Platform: Intel Xeon 5160.

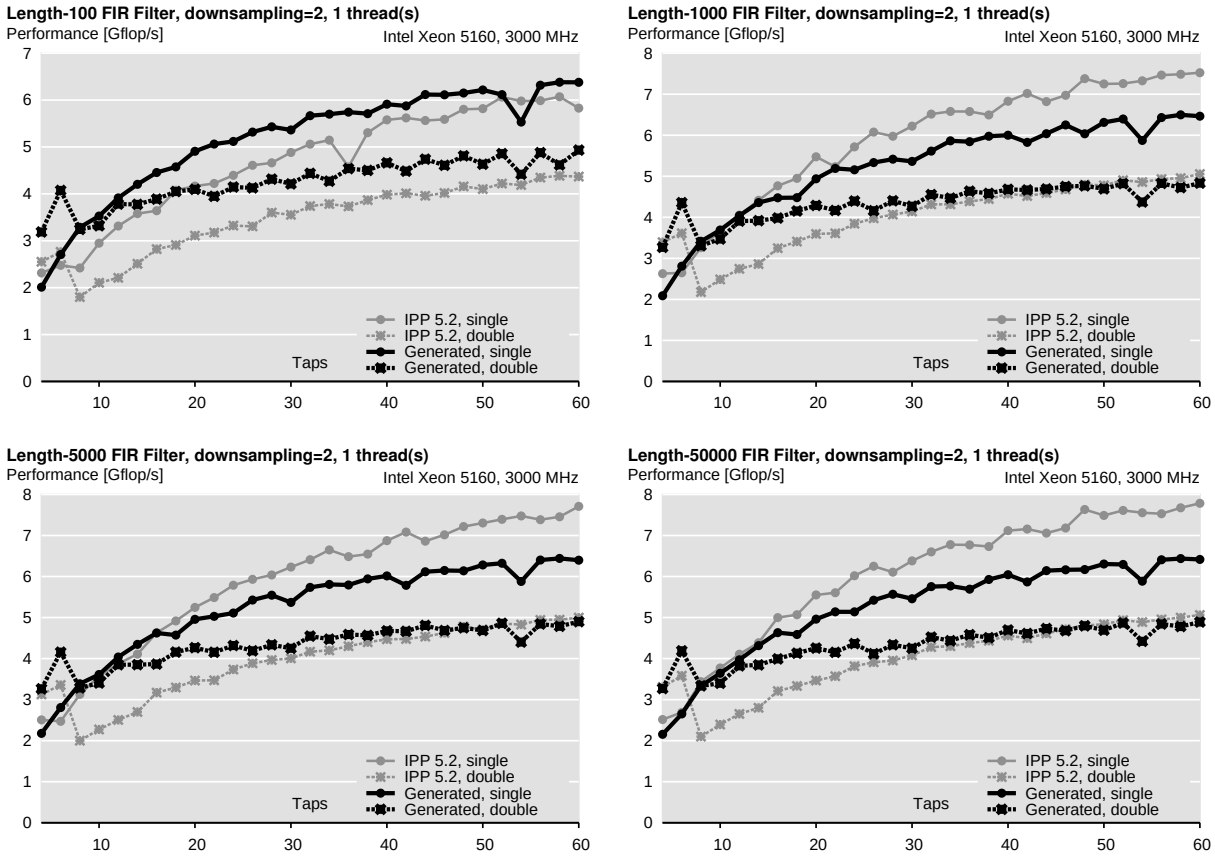


Figure A.16: Generated libraries performance: FIR filter, downsampling = 2, varying number of taps, up to 1 thread(s). Platform: Intel Xeon 5160.

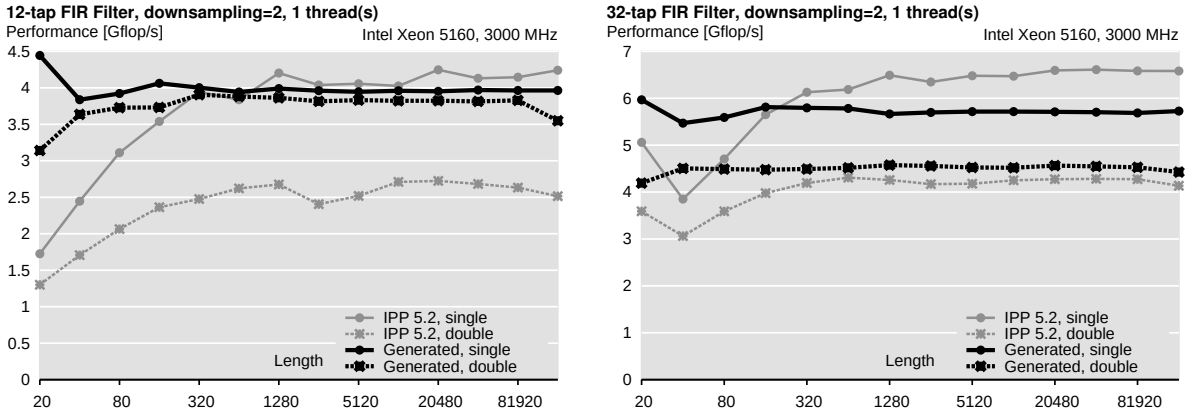


Figure A.17: Generated libraries performance: FIR filter, varying length, up to 1 thread(s). Platform: Intel Xeon 5160.

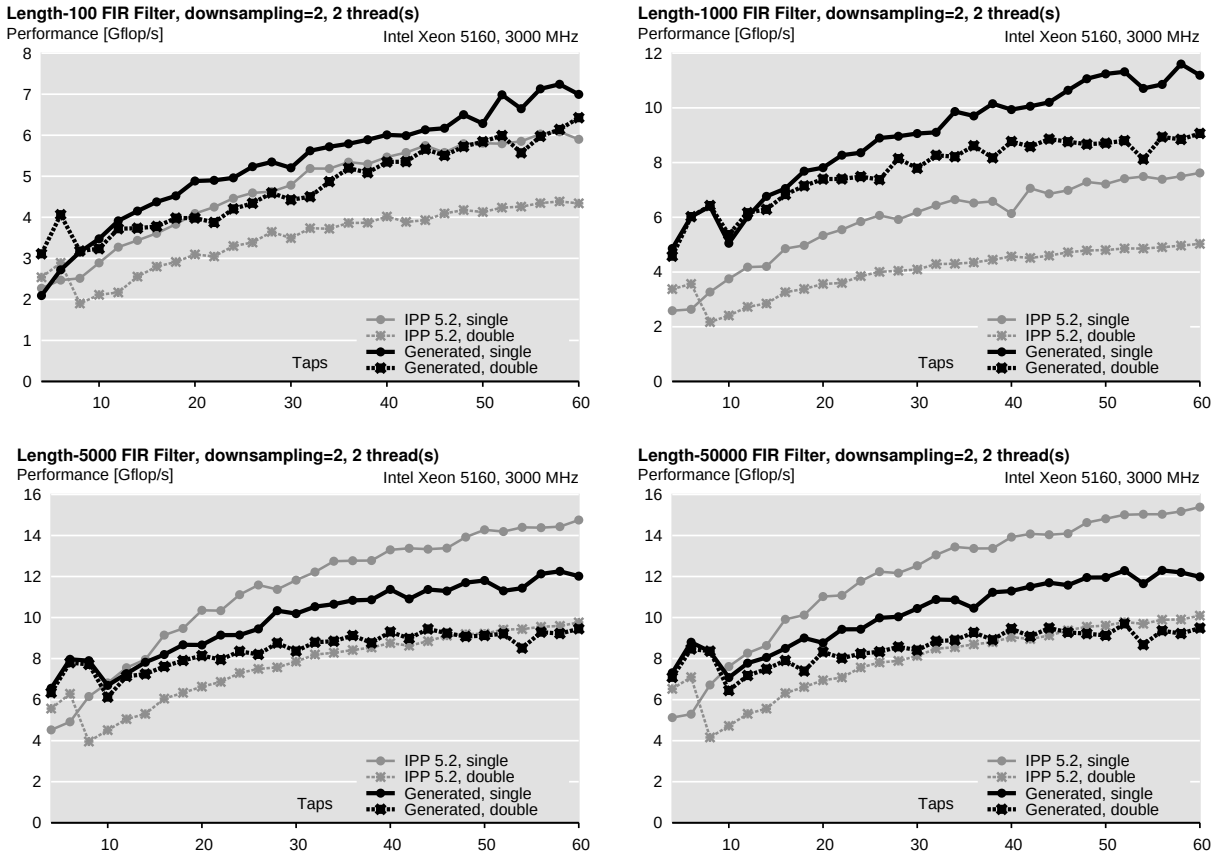


Figure A.18: Generated libraries performance: FIR filter, downsampling = 2, varying number of taps, up to 2 thread(s). Platform: Intel Xeon 5160.

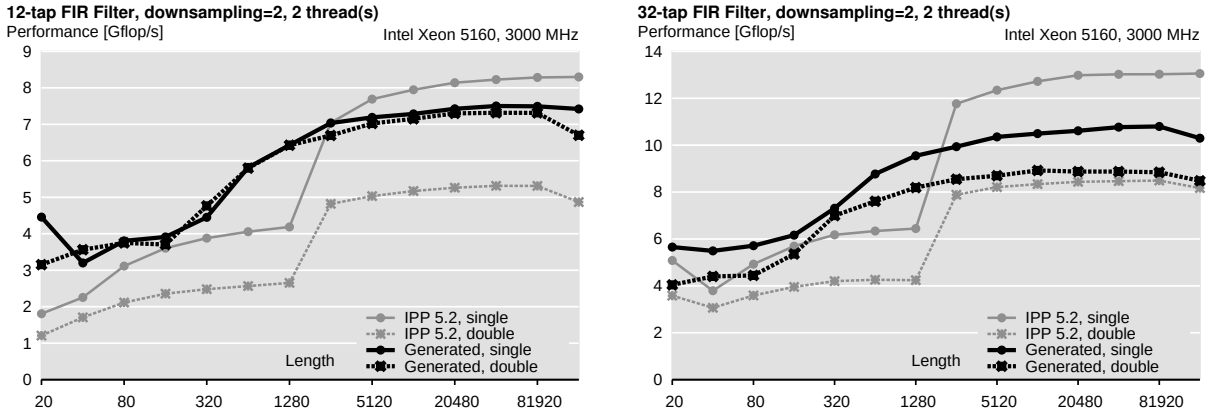


Figure A.19: Generated libraries performance: FIR filter, varying length, up to 2 thread(s). Platform: Intel Xeon 5160.

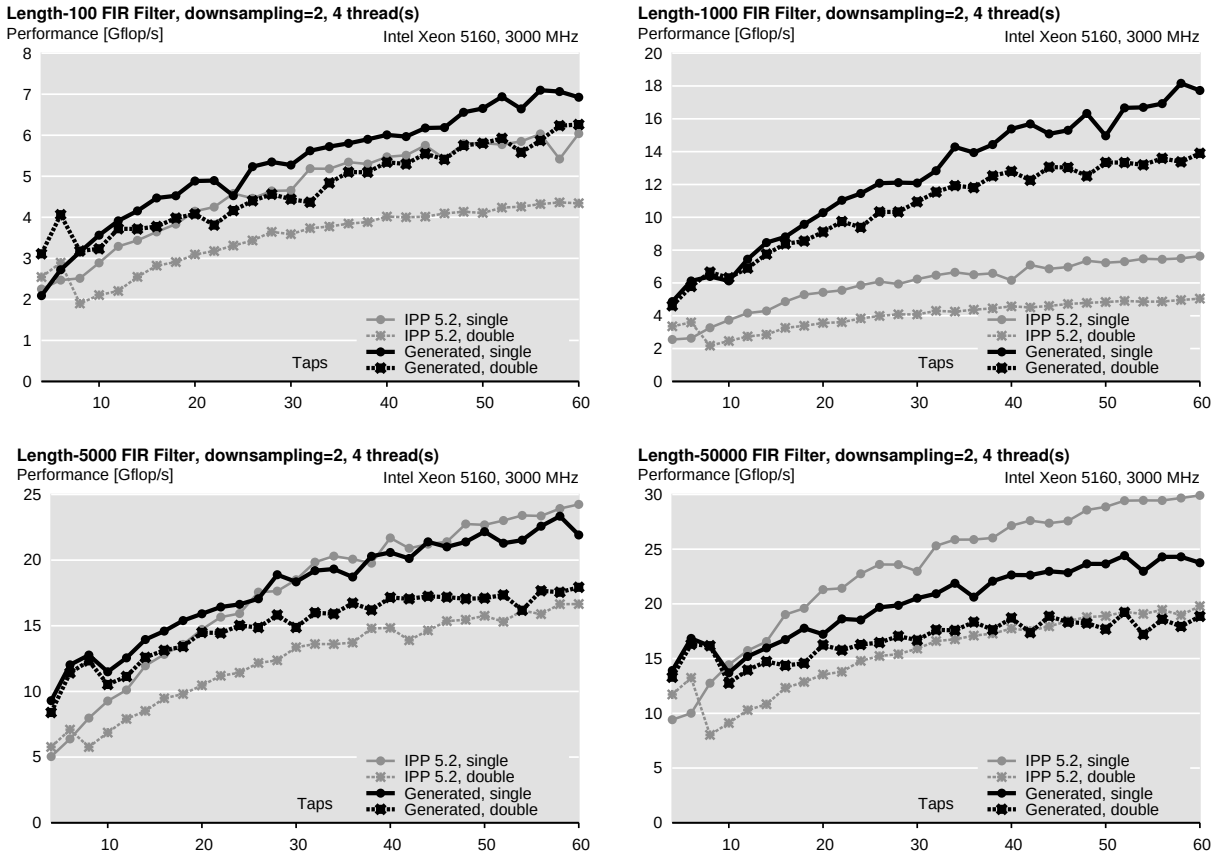


Figure A.20: Generated libraries performance: FIR filter, downsampling = 2, varying number of taps, up to 4 thread(s). Platform: Intel Xeon 5160.

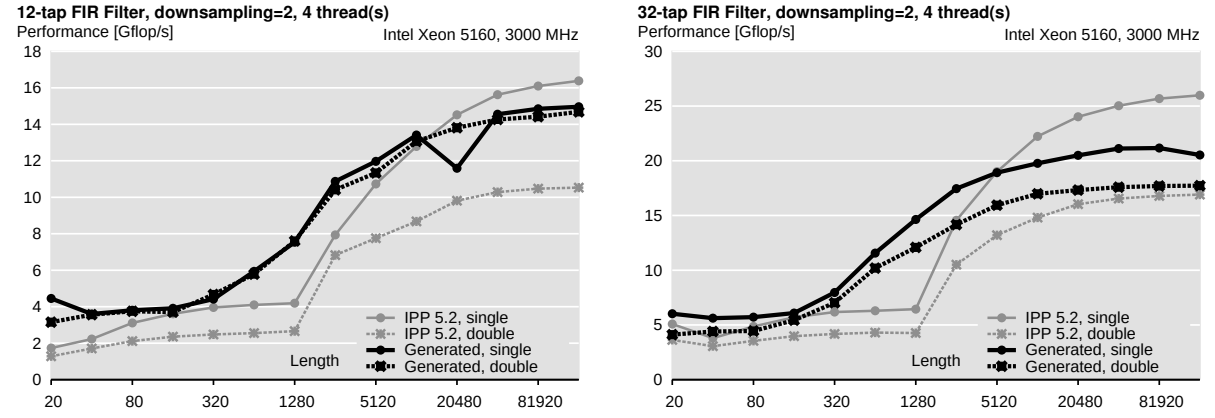


Figure A.21: Generated libraries performance: FIR filter, varying length, up to 4 thread(s). Platform: Intel Xeon 5160.

A.2 AMD Opteron 2220

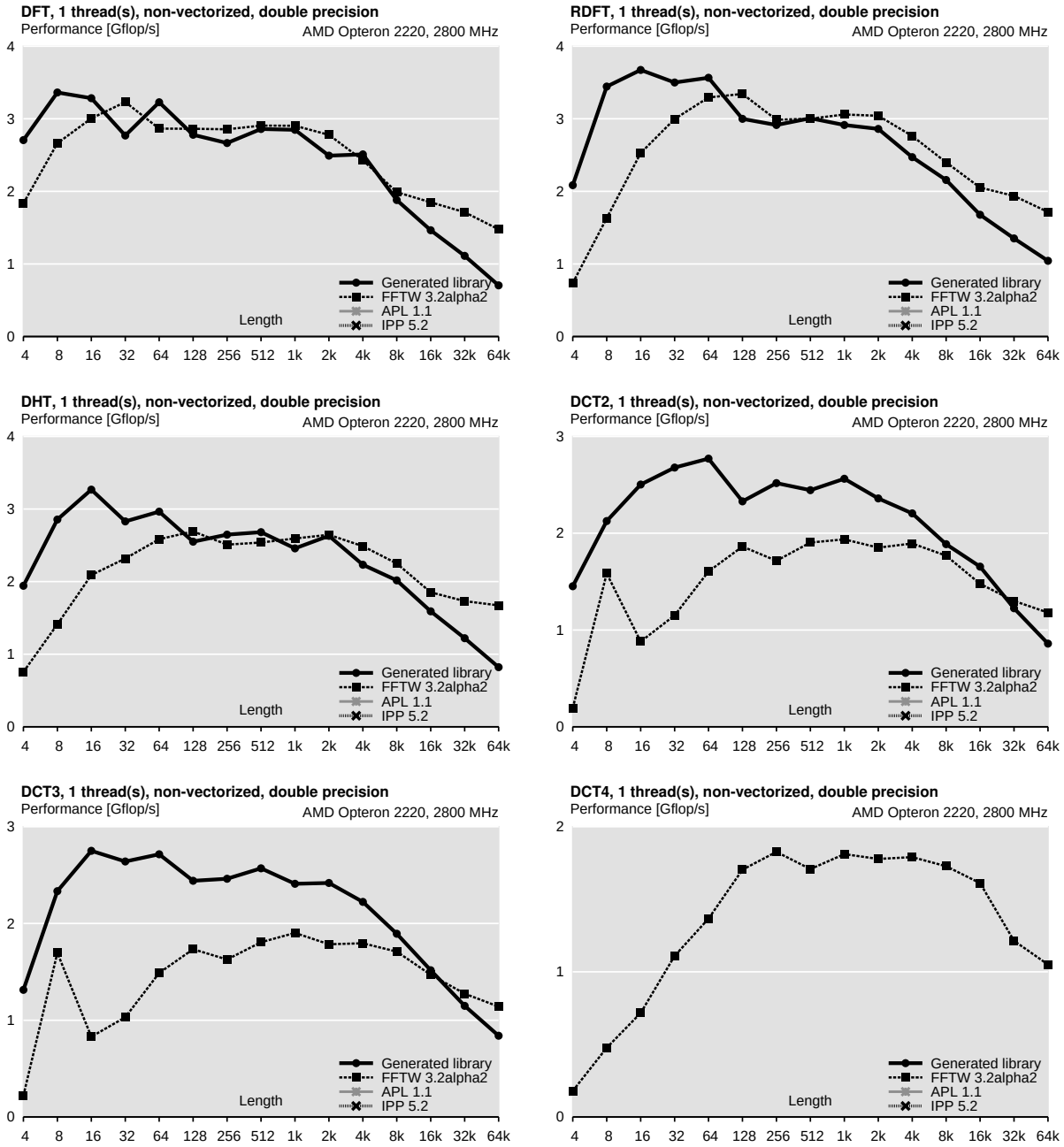


Figure A.22: Generated libraries performance: trigonometric transforms, no vectorization (double precision), no threading. Platform: AMD Opteron 2220.

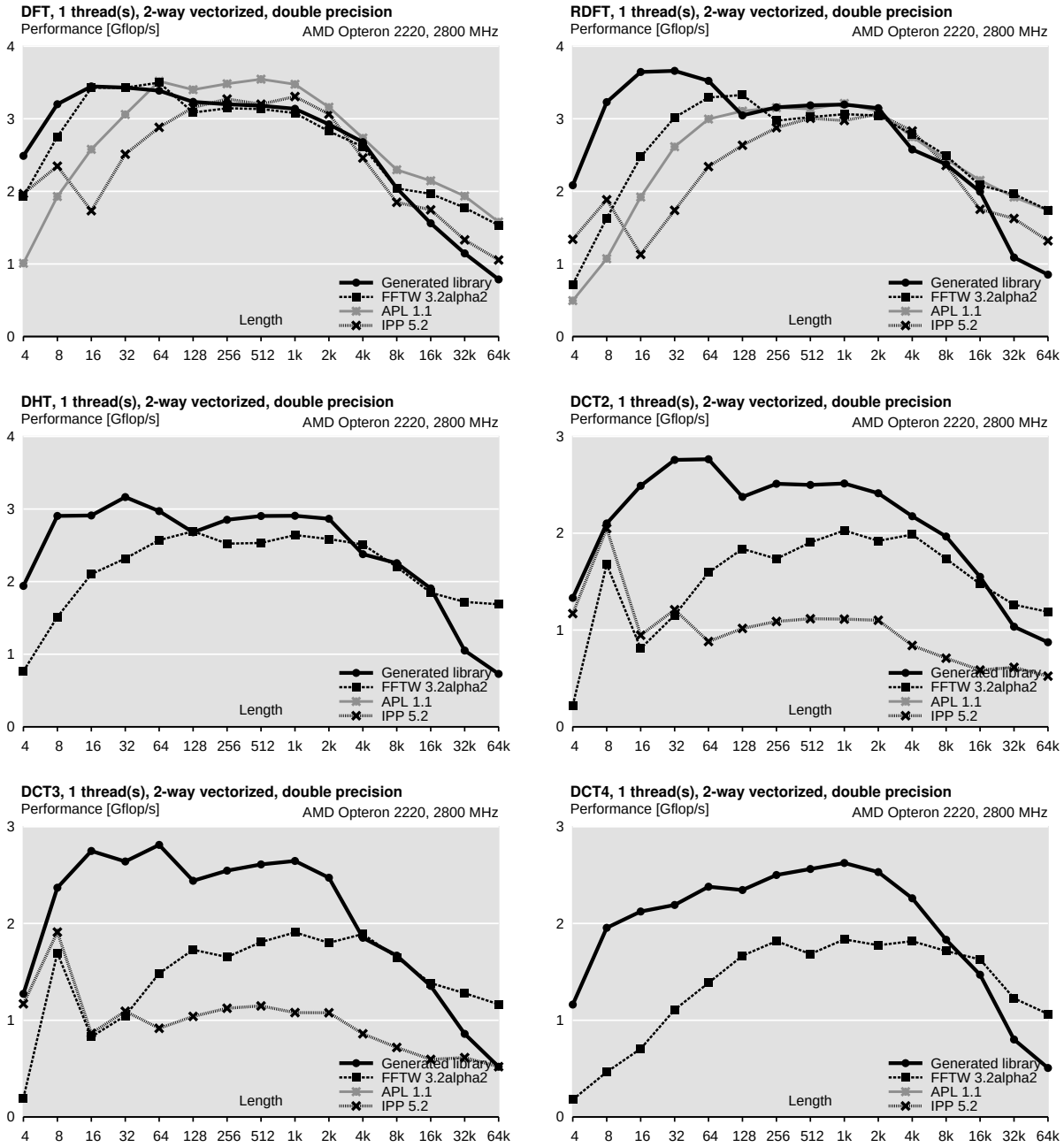


Figure A.23: Generated libraries performance: trigonometric transforms, 2-way vectorization (double precision), no threading. Platform: AMD Opteron 2220.

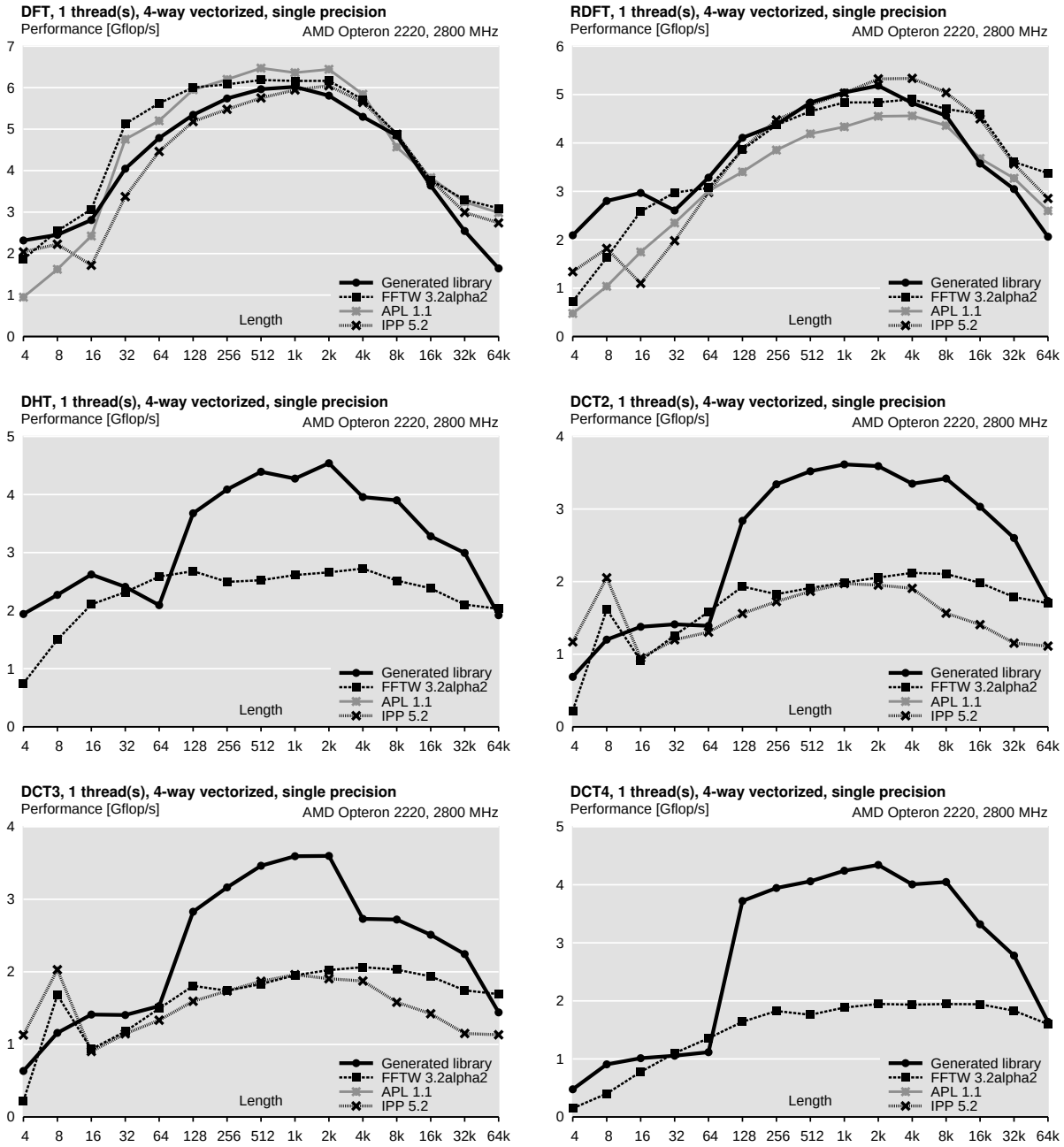


Figure A.24: Generated libraries performance: trigonometric transforms, 4-way vectorization (single precision), no threading. Platform: AMD Opteron 2220.

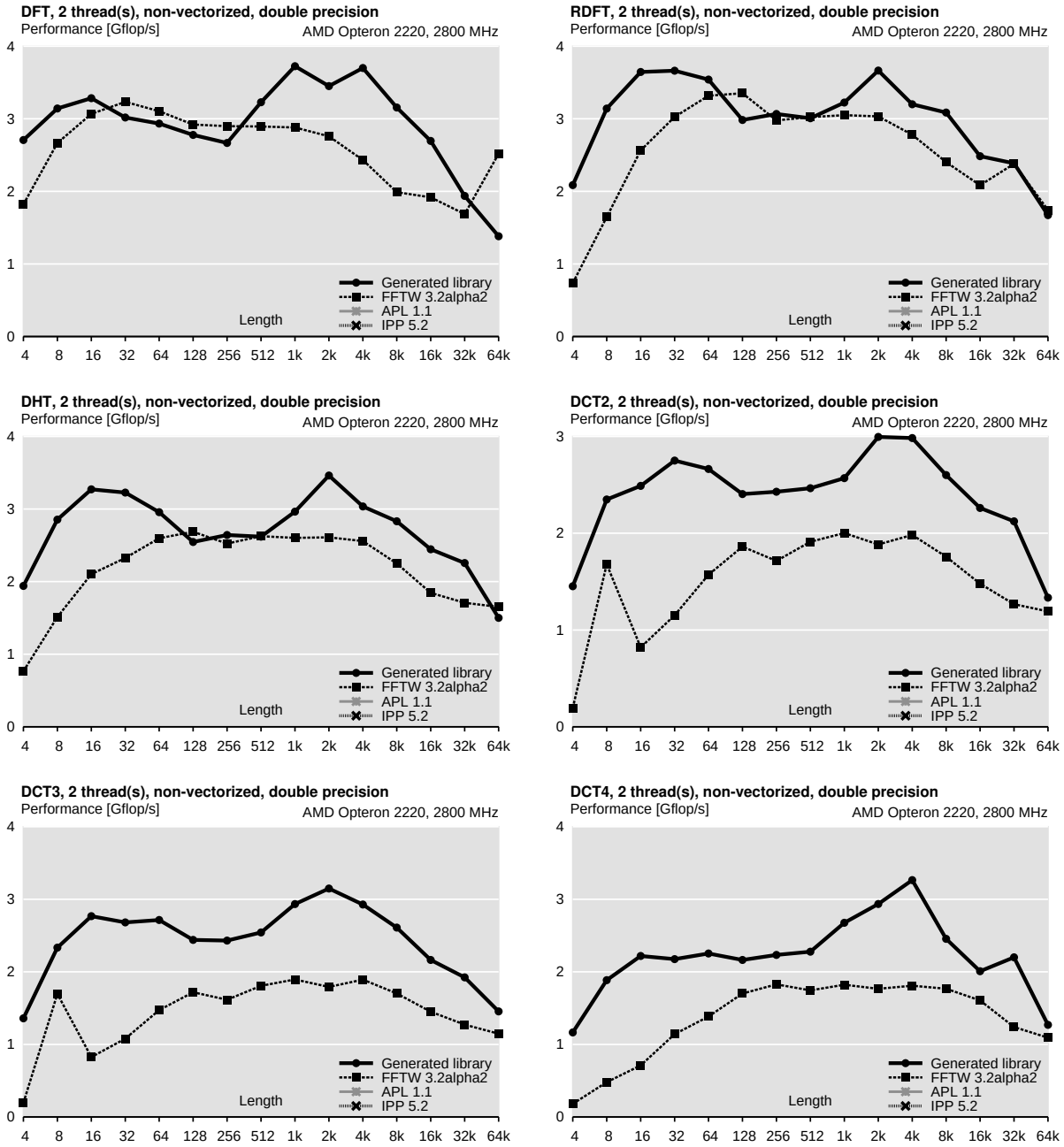


Figure A.25: Generated libraries performance: trigonometric transforms, no vectorization (double precision), up to 2 threads. Platform: AMD Opteron 2220.

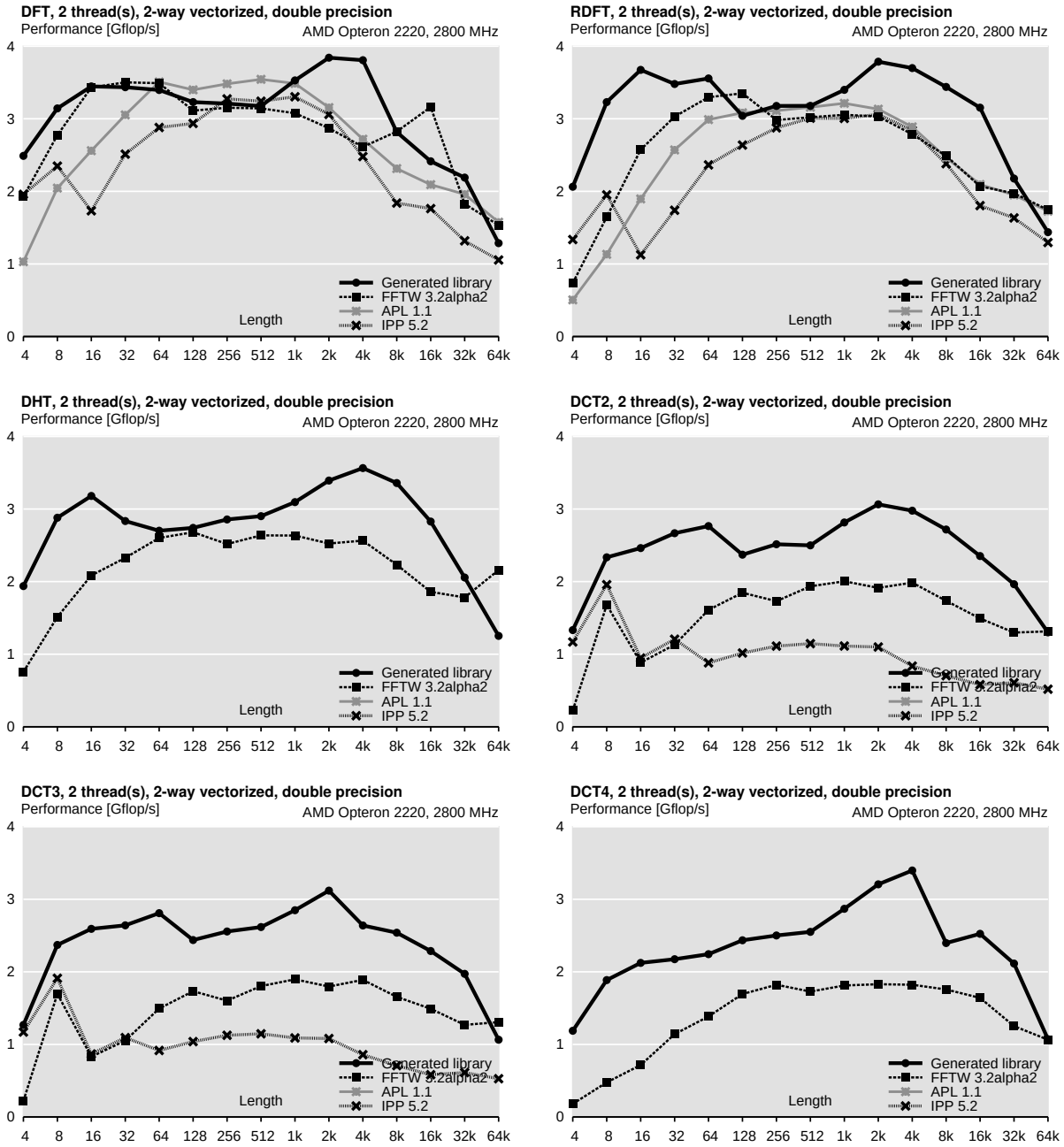


Figure A.26: Generated libraries performance: trigonometric transforms, 2-way vectorization (double precision), up to 2 threads. Platform: AMD Opteron 2220.

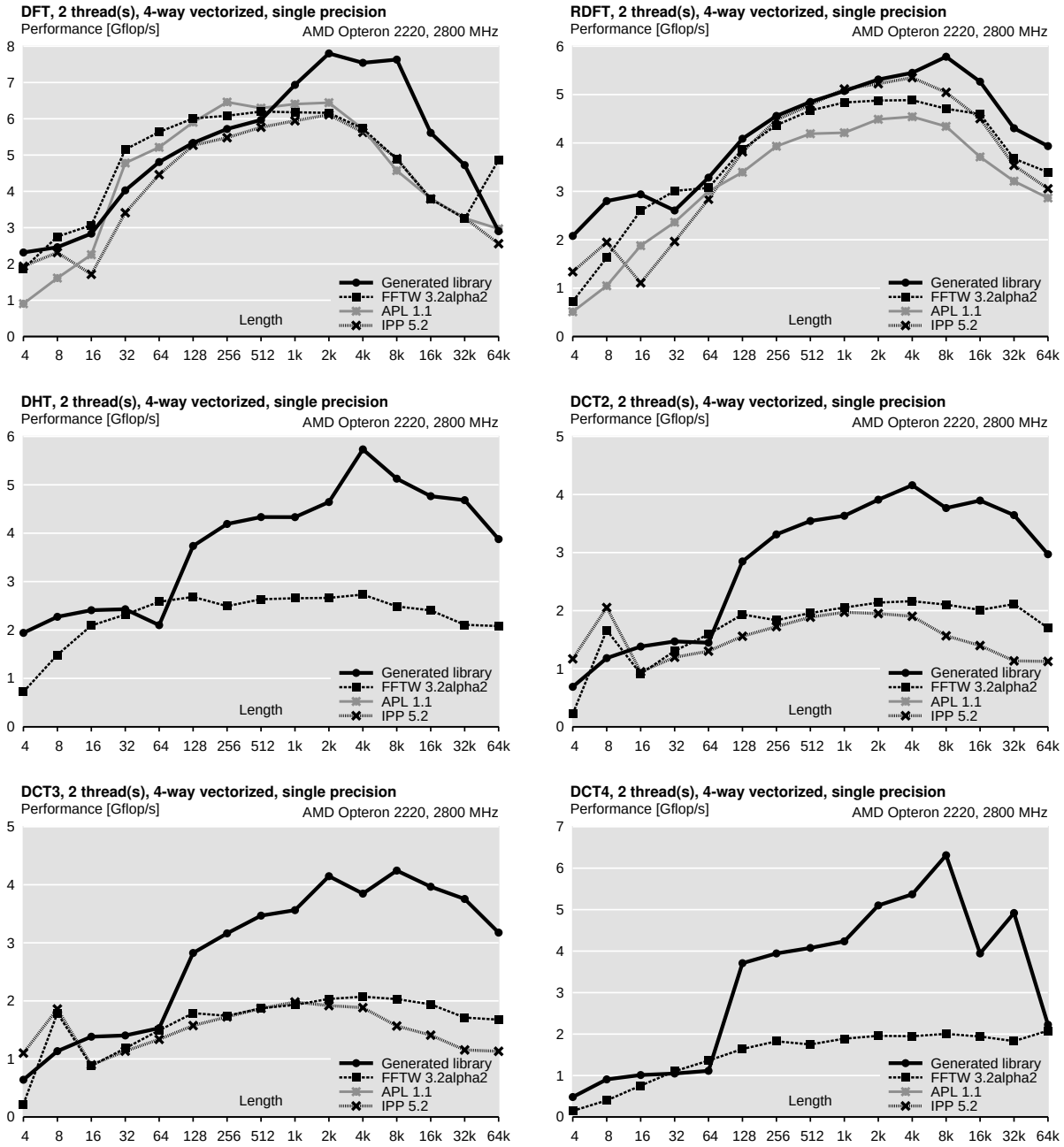


Figure A.27: Generated libraries performance: trigonometric transforms, 4-way vectorization (single precision), up to 2 threads. Platform: AMD Opteron 2220.

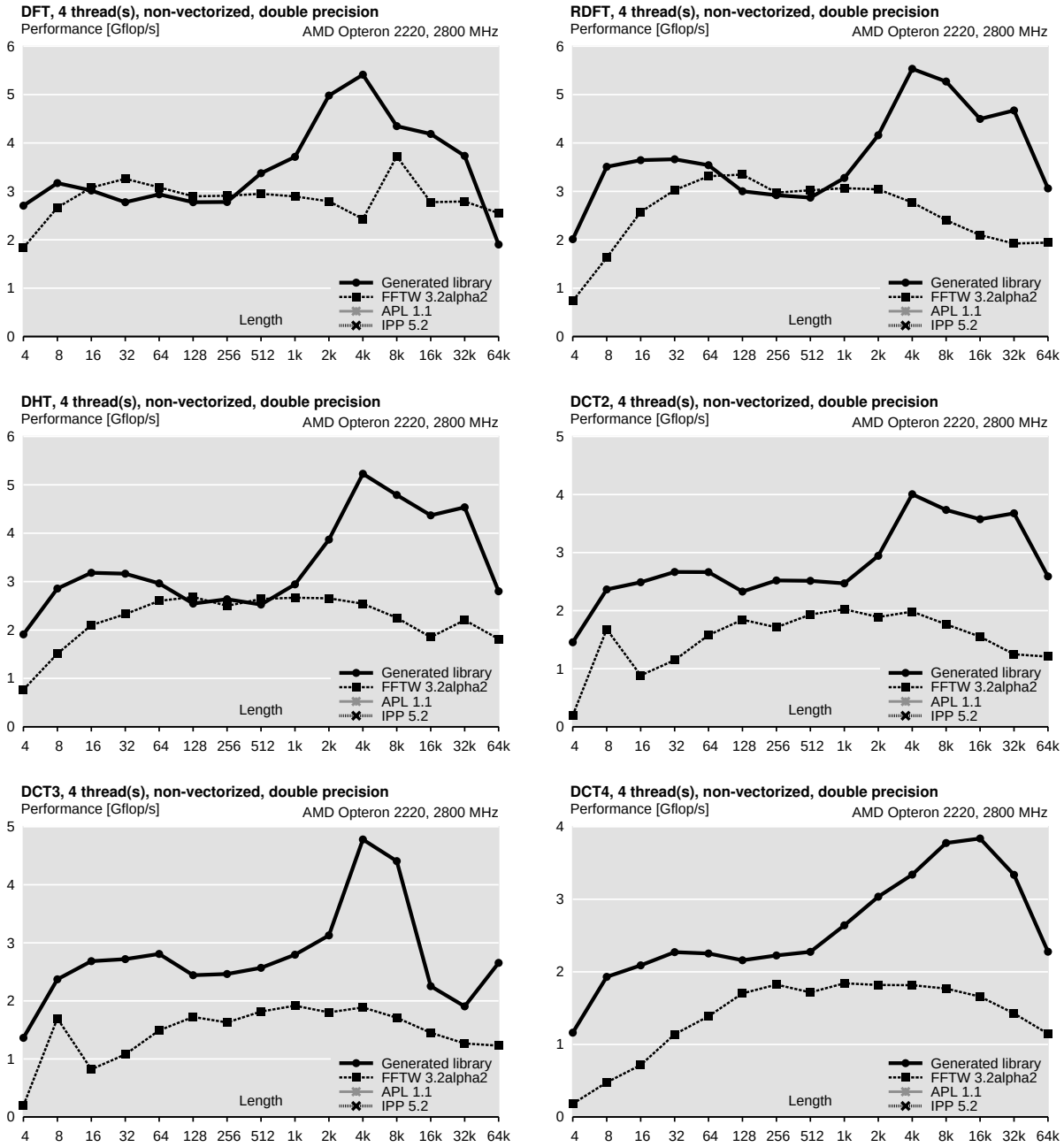


Figure A.28: Generated libraries performance: trigonometric transforms, no vectorization (double precision), up to 4 threads. Platform: AMD Opteron 2220.

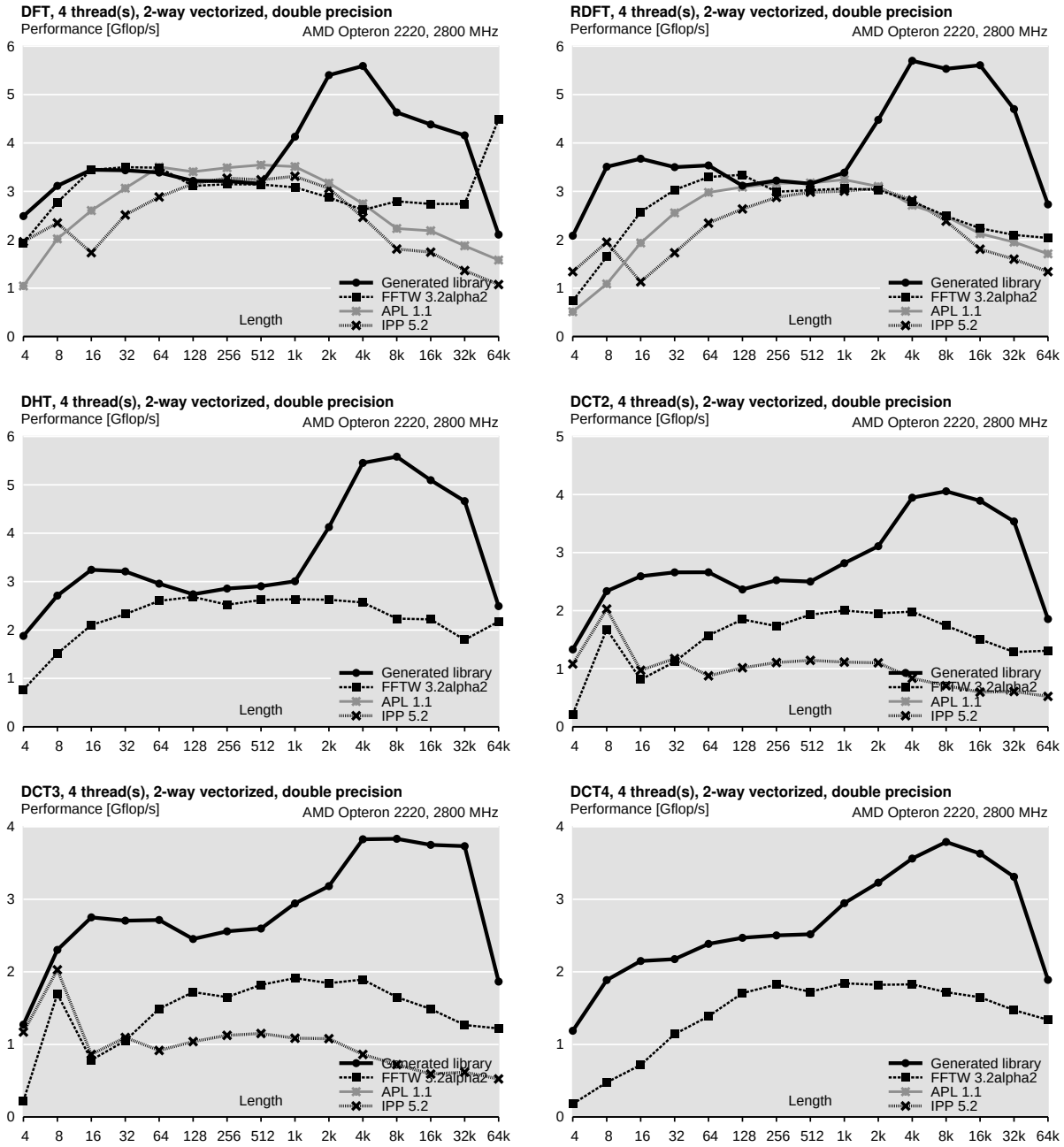


Figure A.29: Generated libraries performance: trigonometric transforms, 2-way vectorization (double precision), up to 4 threads. Platform: AMD Opteron 2220.

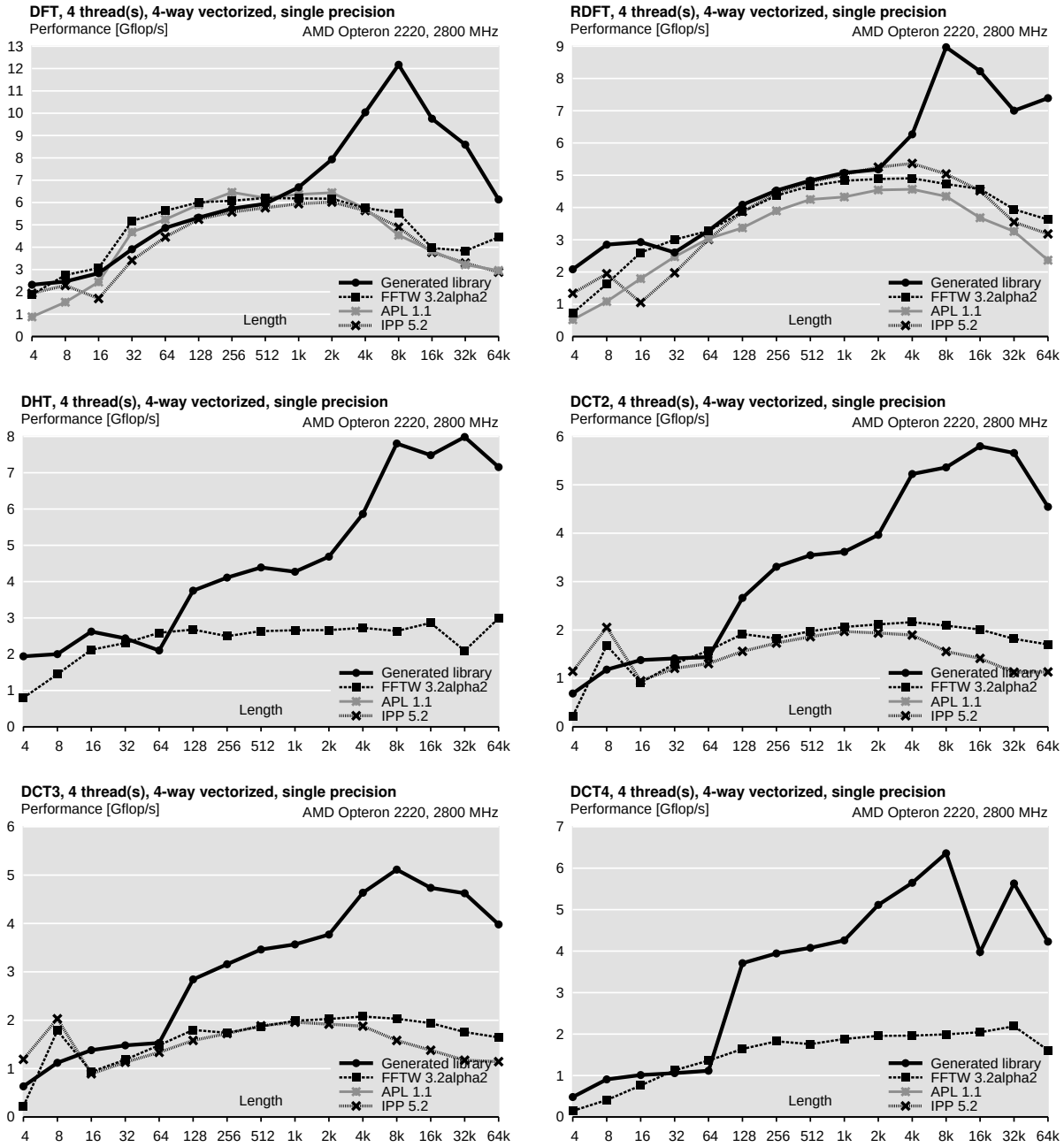


Figure A.30: Generated libraries performance: trigonometric transforms, 4-way vectorization (single precision), up to 4 threads. Platform: AMD Opteron 2220.

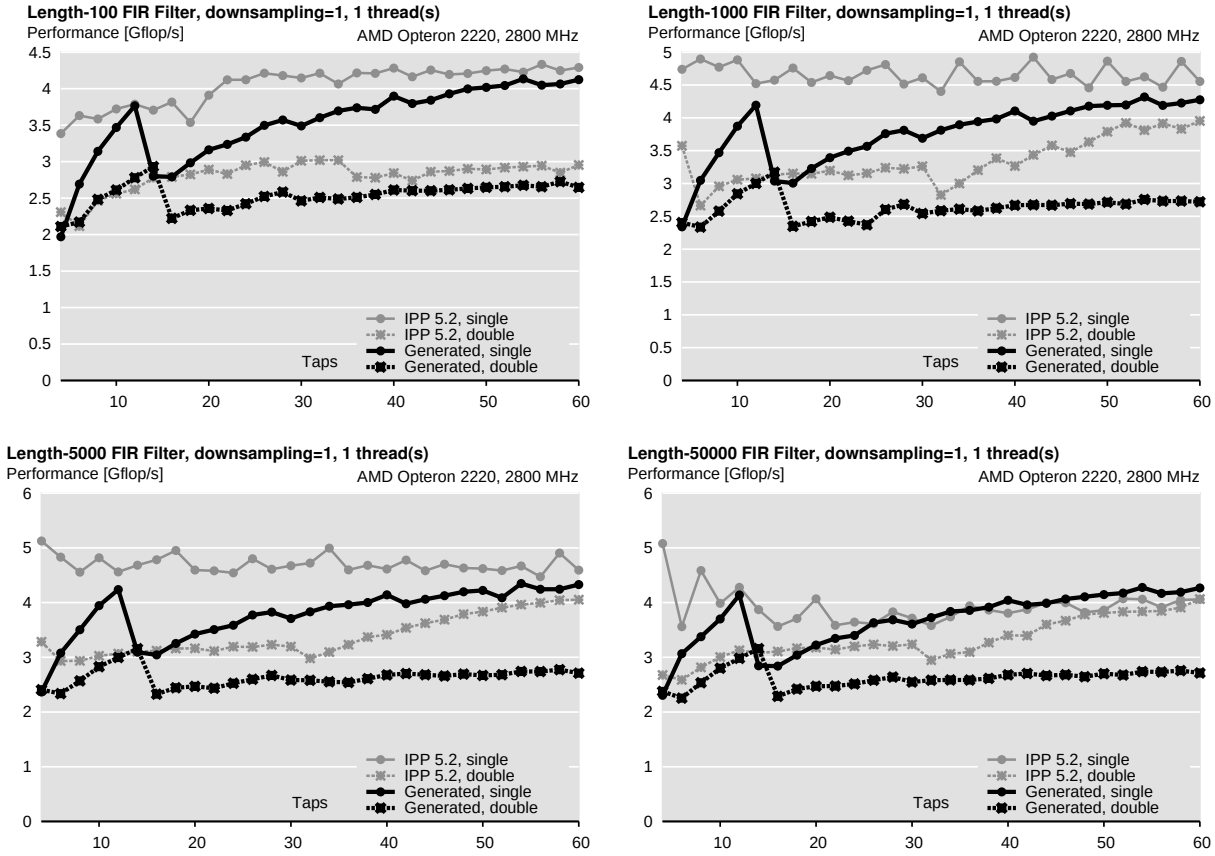


Figure A.31: Generated libraries performance: FIR filter, downsampling = 1, varying number of taps, up to 1 thread(s). Platform: AMD Opteron 2220.

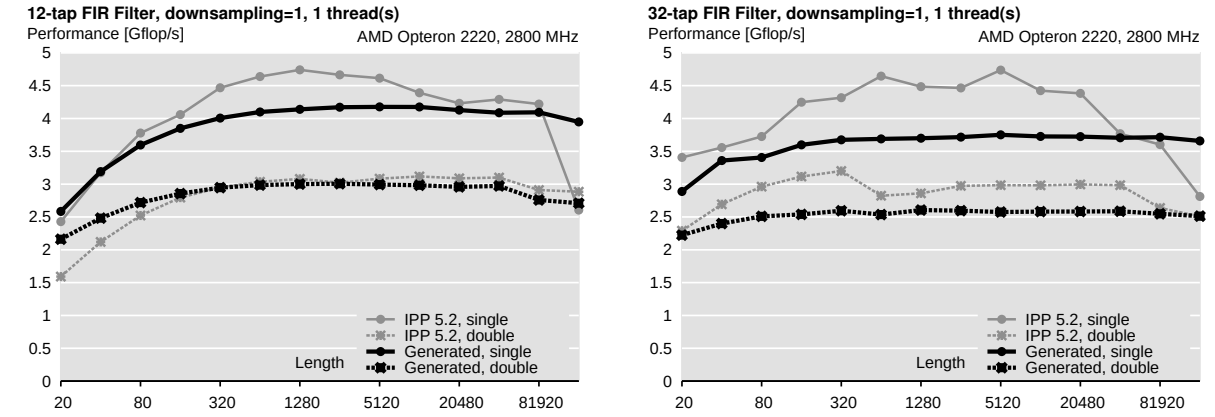


Figure A.32: Generated libraries performance: FIR filter, varying length, up to 1 thread(s). Platform: AMD Opteron 2220.

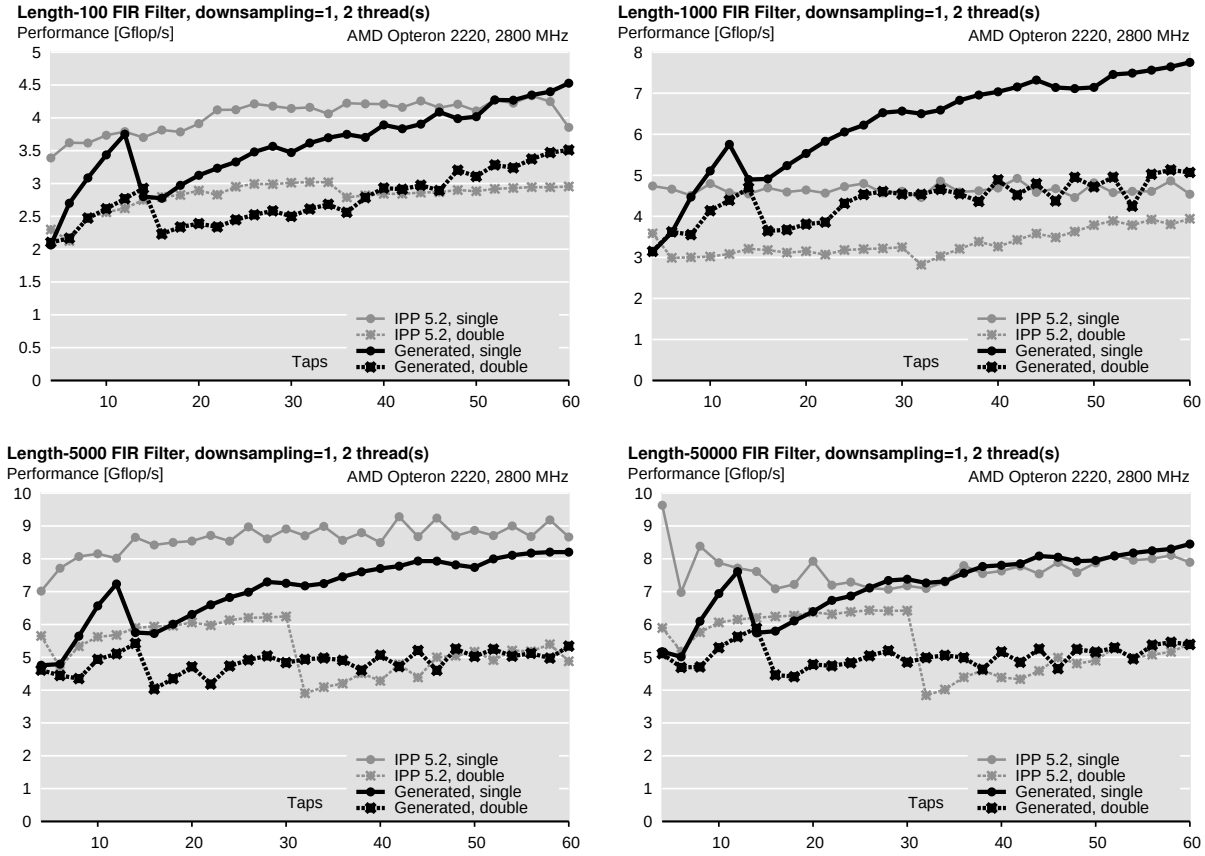


Figure A.33: Generated libraries performance: FIR filter, downsampling = 1, varying number of taps, up to 2 thread(s). Platform: AMD Opteron 2220.

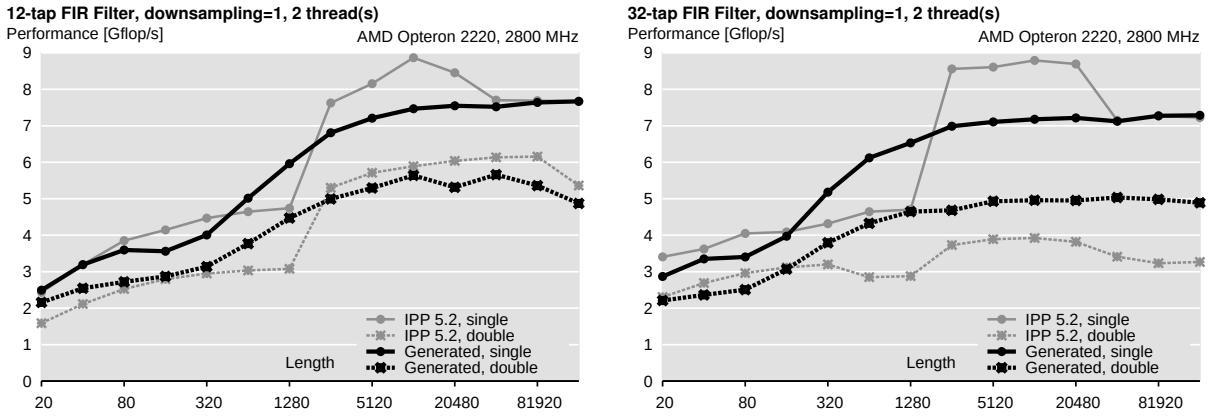


Figure A.34: Generated libraries performance: FIR filter, varying length, up to 2 thread(s). Platform: AMD Opteron 2220.

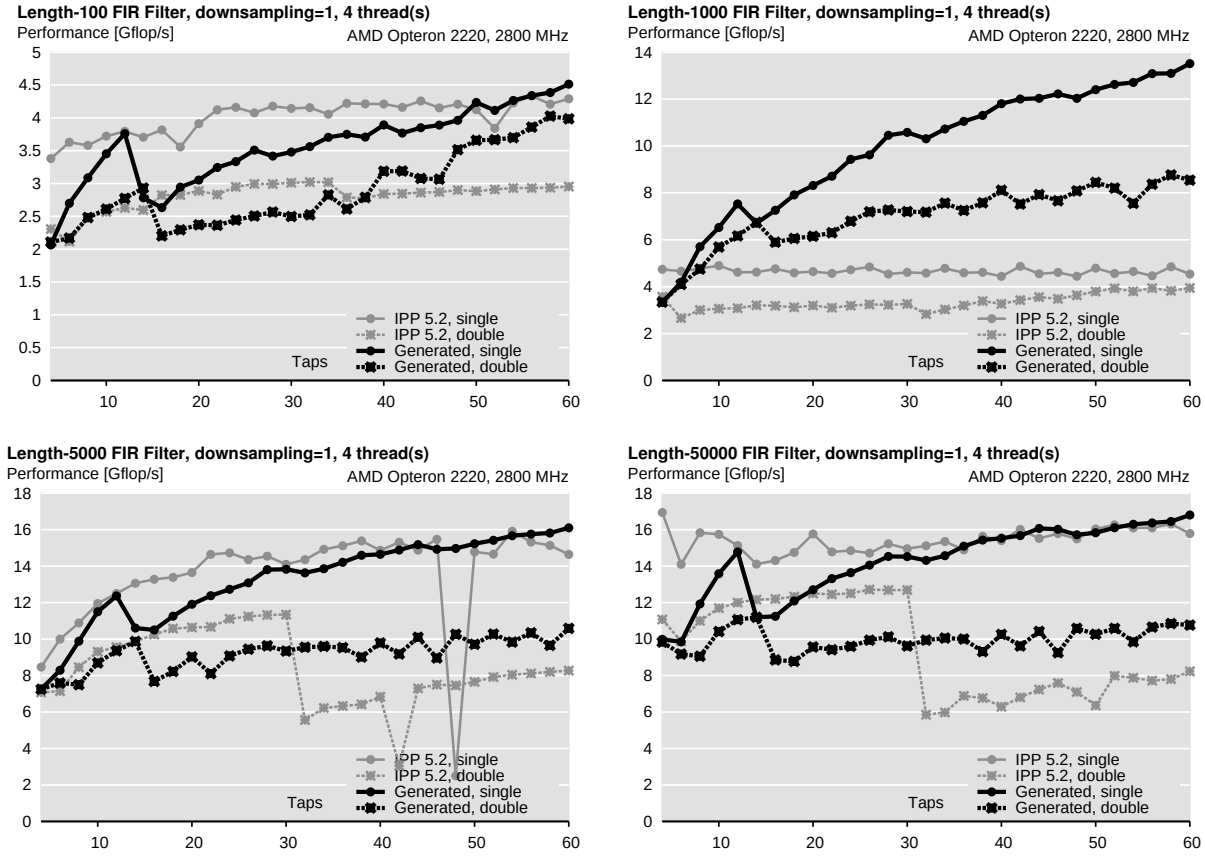


Figure A.35: Generated libraries performance: FIR filter, downsampling = 1, varying number of taps, up to 4 thread(s). Platform: AMD Opteron 2220.

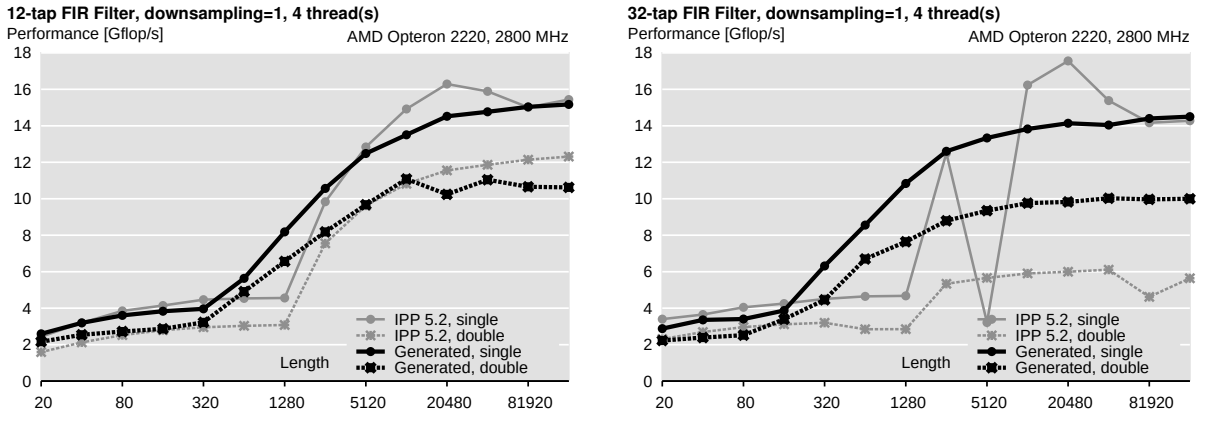


Figure A.36: Generated libraries performance: FIR filter, varying length, up to 4 thread(s). Platform: AMD Opteron 2220.

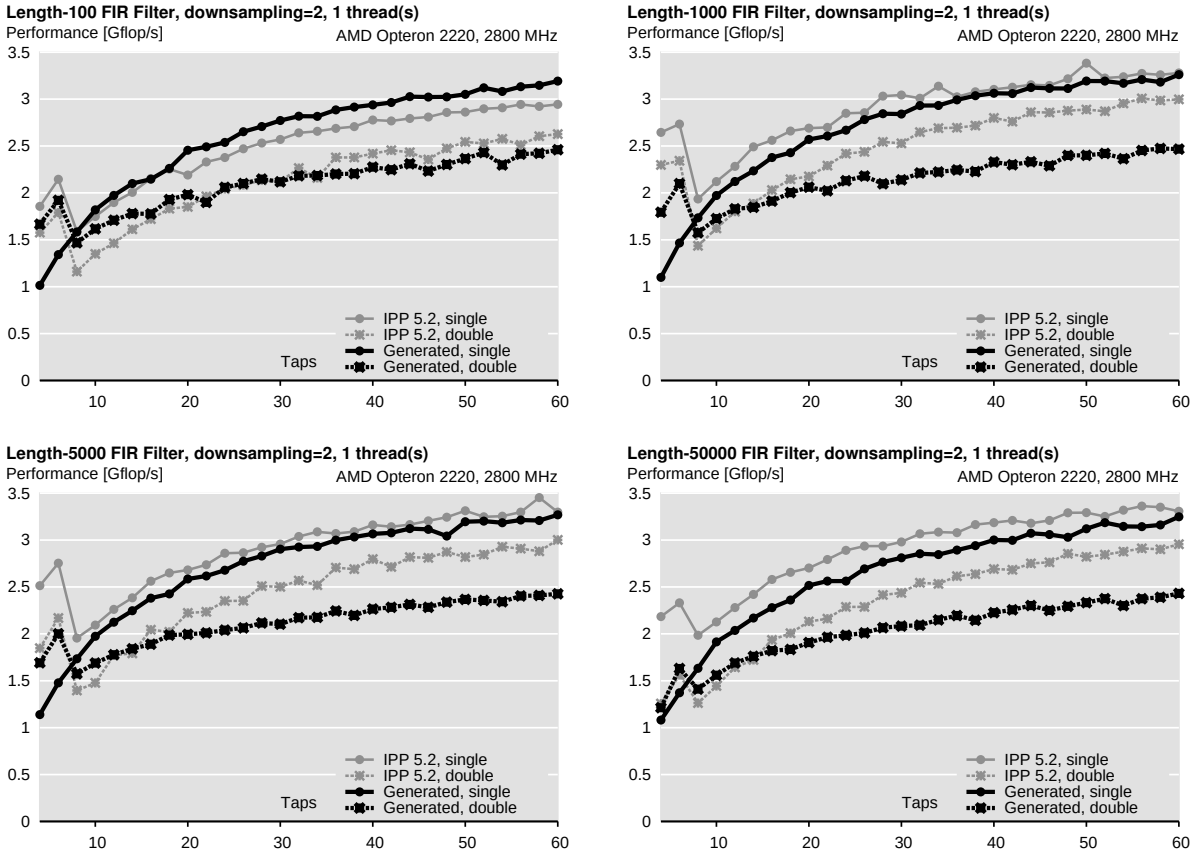


Figure A.37: Generated libraries performance: FIR filter, downsampling = 2, varying number of taps, up to 1 thread(s). Platform: AMD Opteron 2220.

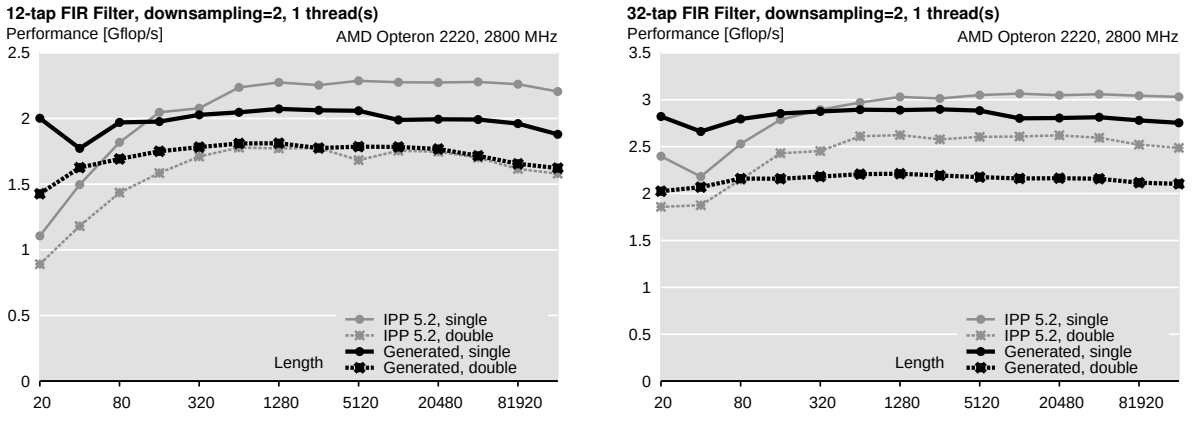


Figure A.38: Generated libraries performance: FIR filter, varying length, up to 1 thread(s). Platform: AMD Opteron 2220.

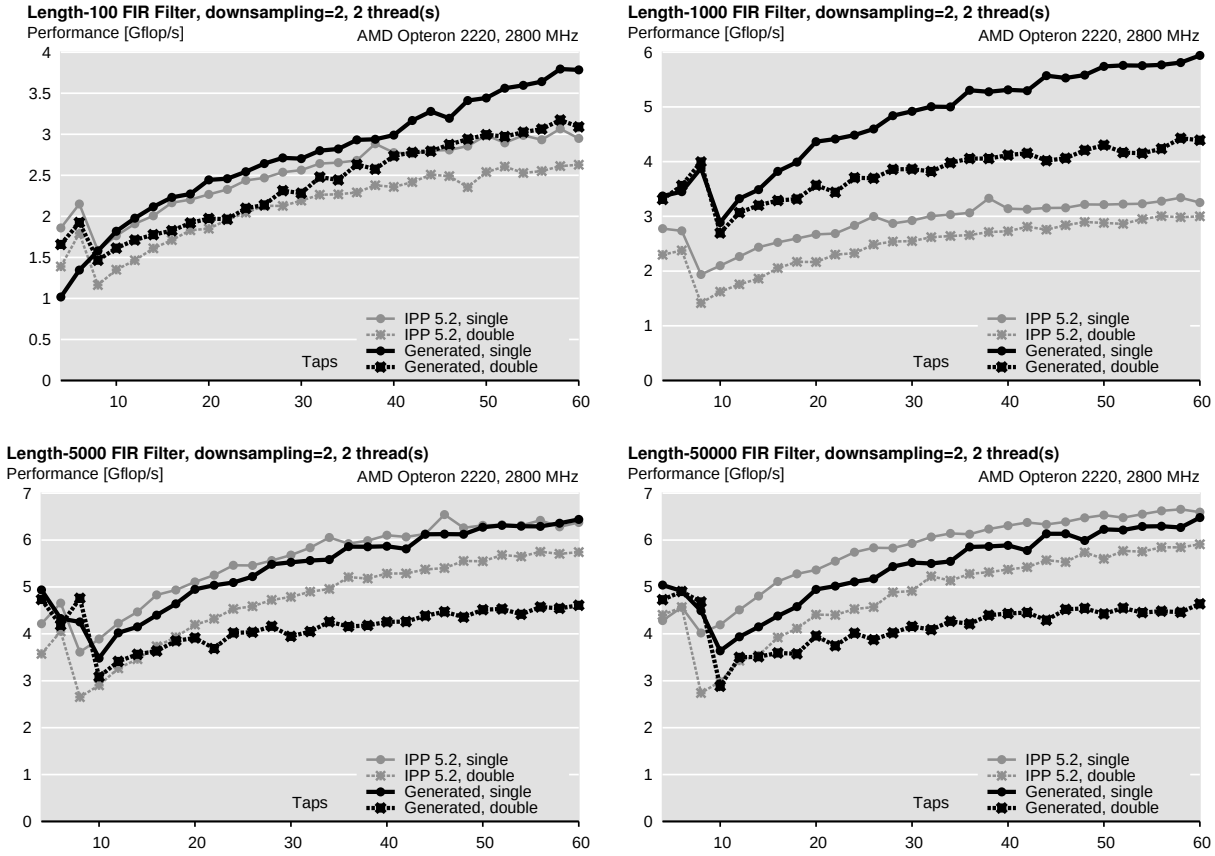


Figure A.39: Generated libraries performance: FIR filter, downsampling = 2, varying number of taps, up to 2 thread(s). Platform: AMD Opteron 2220.

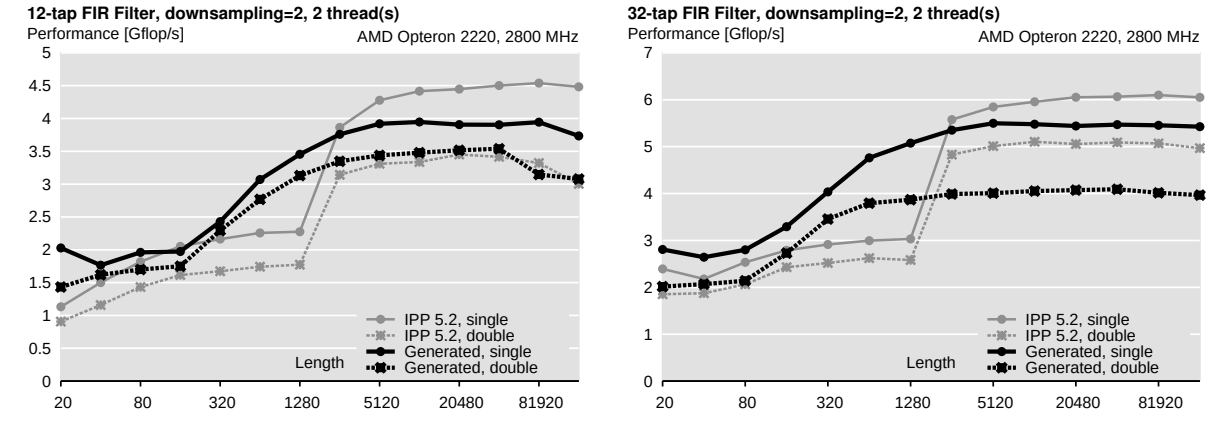


Figure A.40: Generated libraries performance: FIR filter, varying length, up to 2 thread(s). Platform: AMD Opteron 2220.

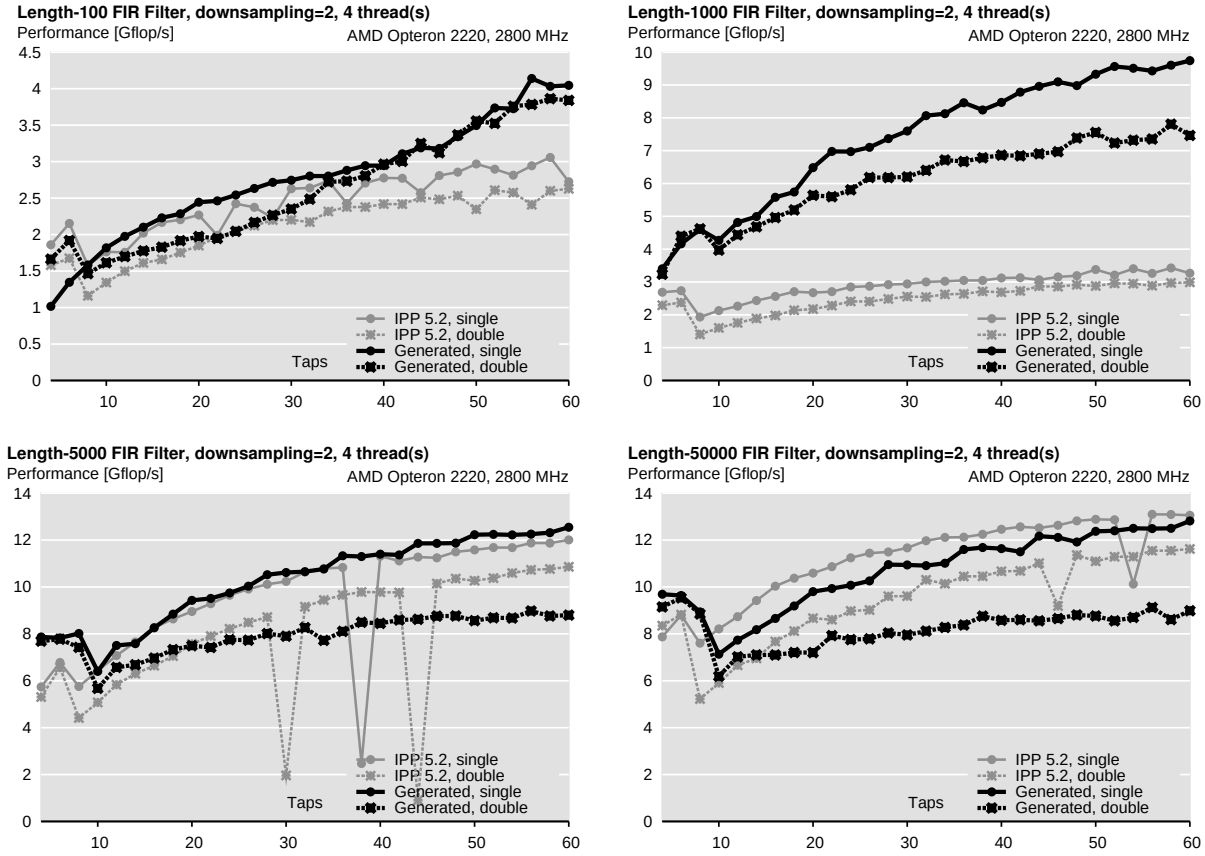


Figure A.41: Generated libraries performance: FIR filter, downsampling = 2, varying number of taps, up to 4 thread(s). Platform: AMD Opteron 2220.

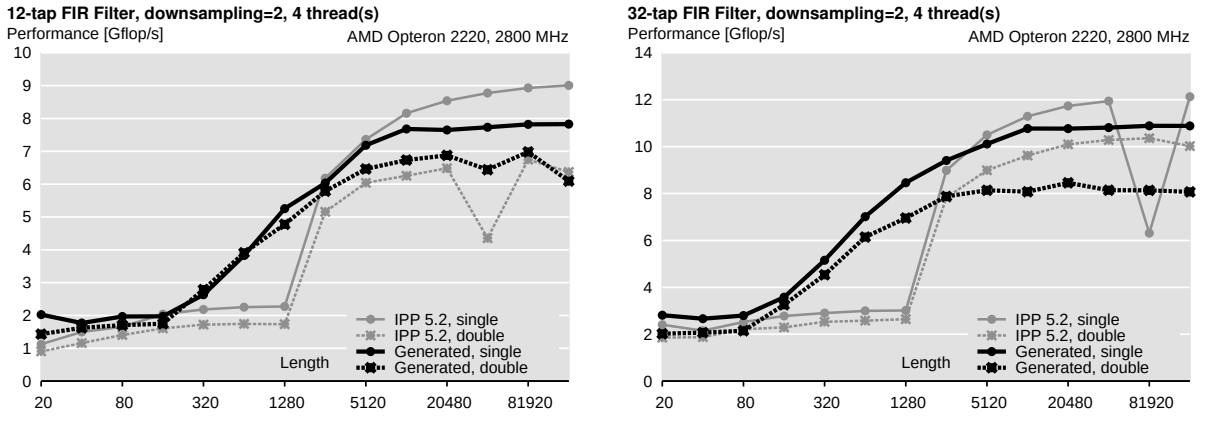


Figure A.42: Generated libraries performance: FIR filter, varying length, up to 4 thread(s). Platform: AMD Opteron 2220.

A.3 Intel Core 2 Extreme QX9650

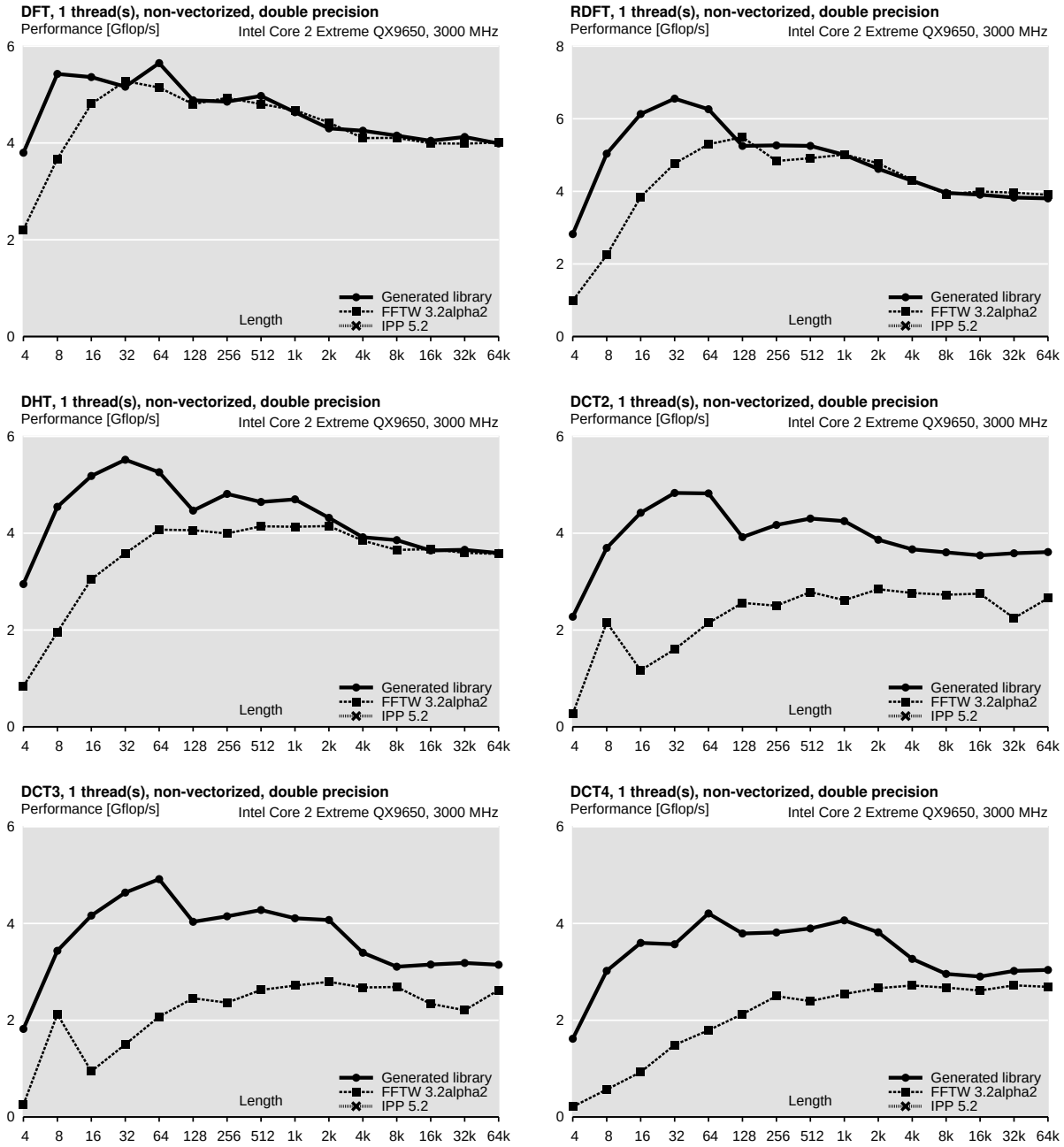


Figure A.43: Generated libraries performance: trigonometric transforms, no vectorization (double precision), no threading. Platform: Intel Core 2 Extreme QX9650.

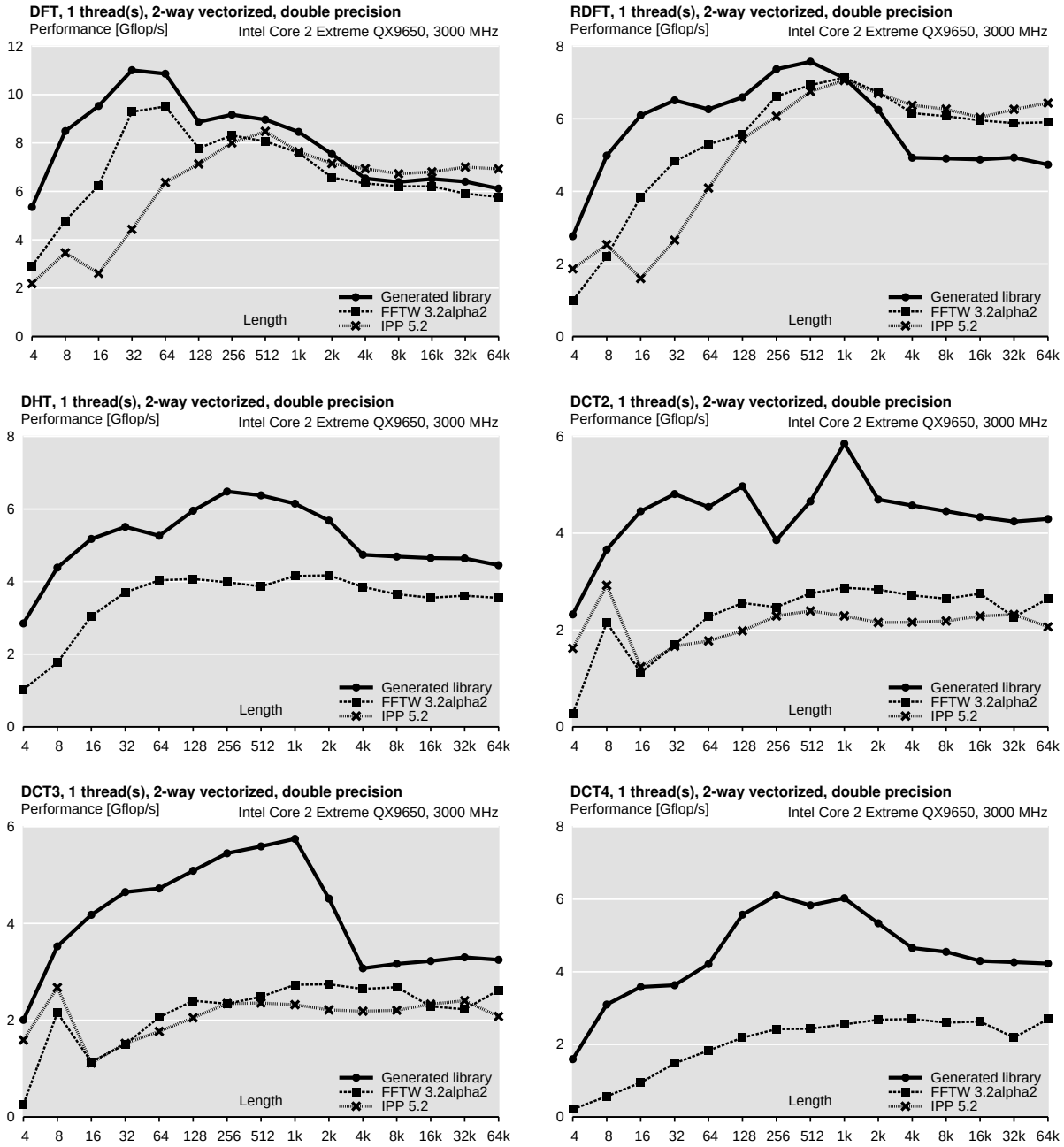


Figure A.44: Generated libraries performance: trigonometric transforms, 2-way vectorization (double precision), no threading. Platform: Intel Core 2 Extreme QX9650.

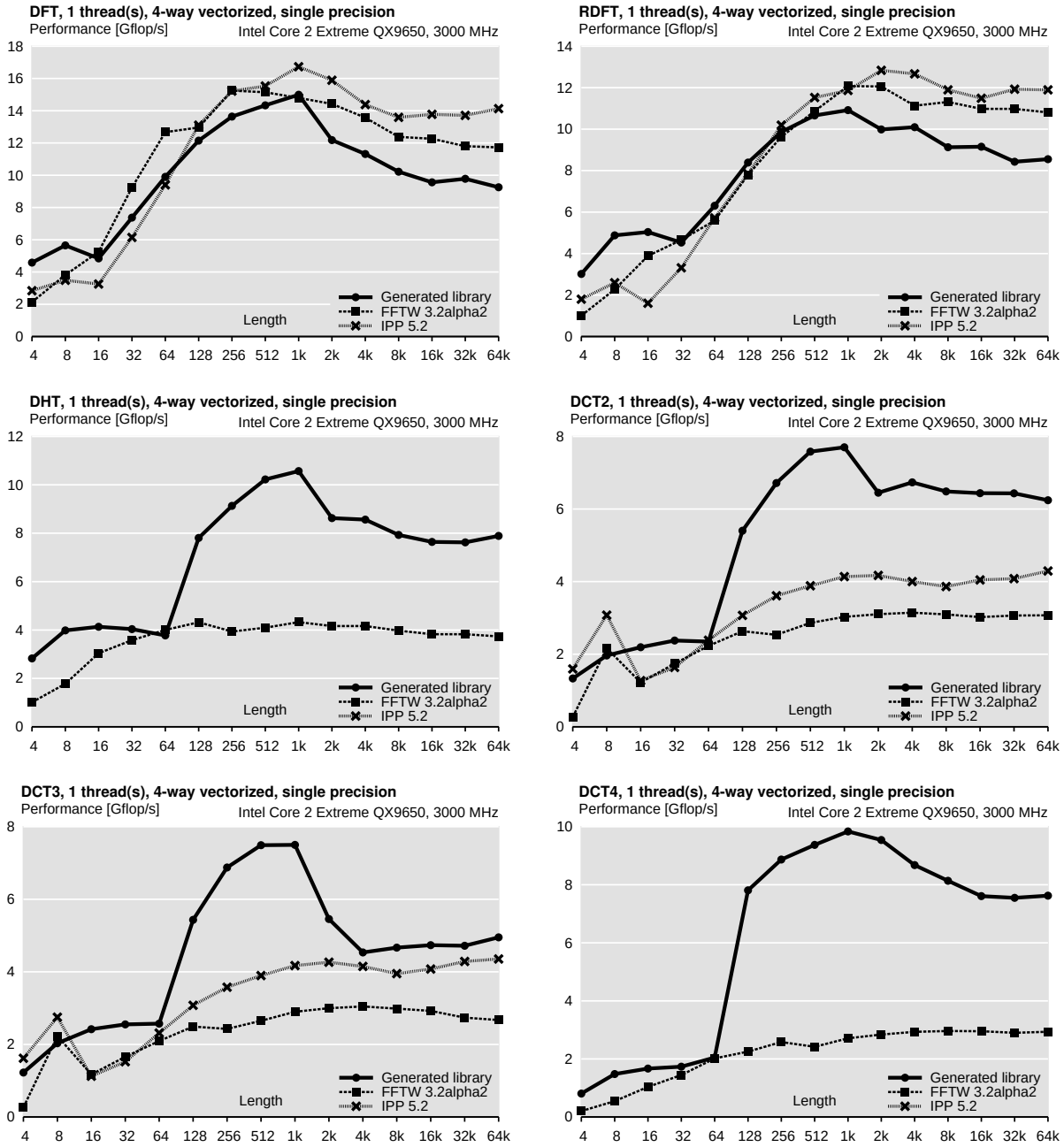


Figure A.45: Generated libraries performance: trigonometric transforms, 4-way vectorization (single precision), no threading. Platform: Intel Core 2 Extreme QX9650.

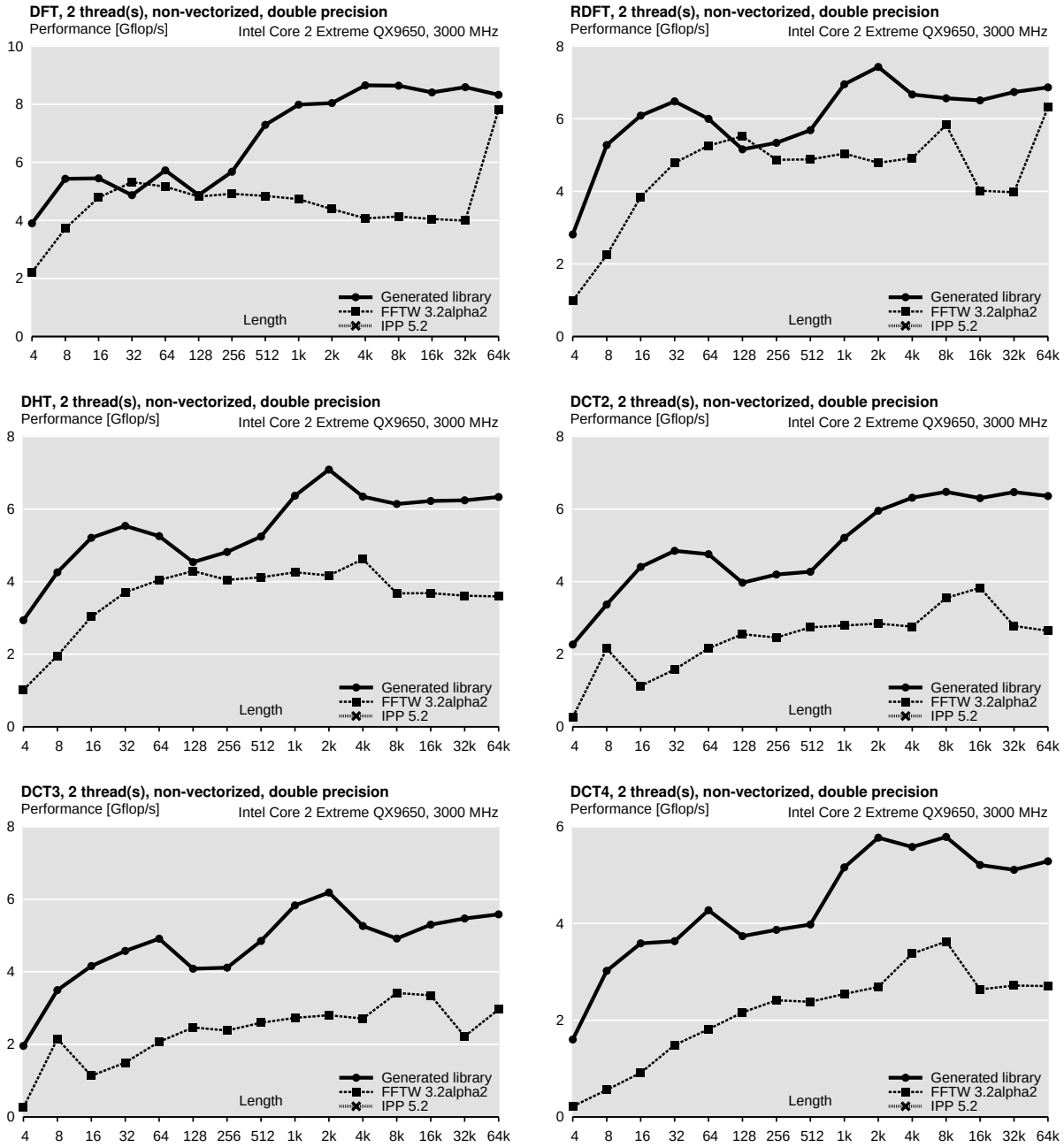


Figure A.46: Generated libraries performance: trigonometric transforms, no vectorization (double precision), up to 2 threads. Platform: Intel Core 2 Extreme QX9650.

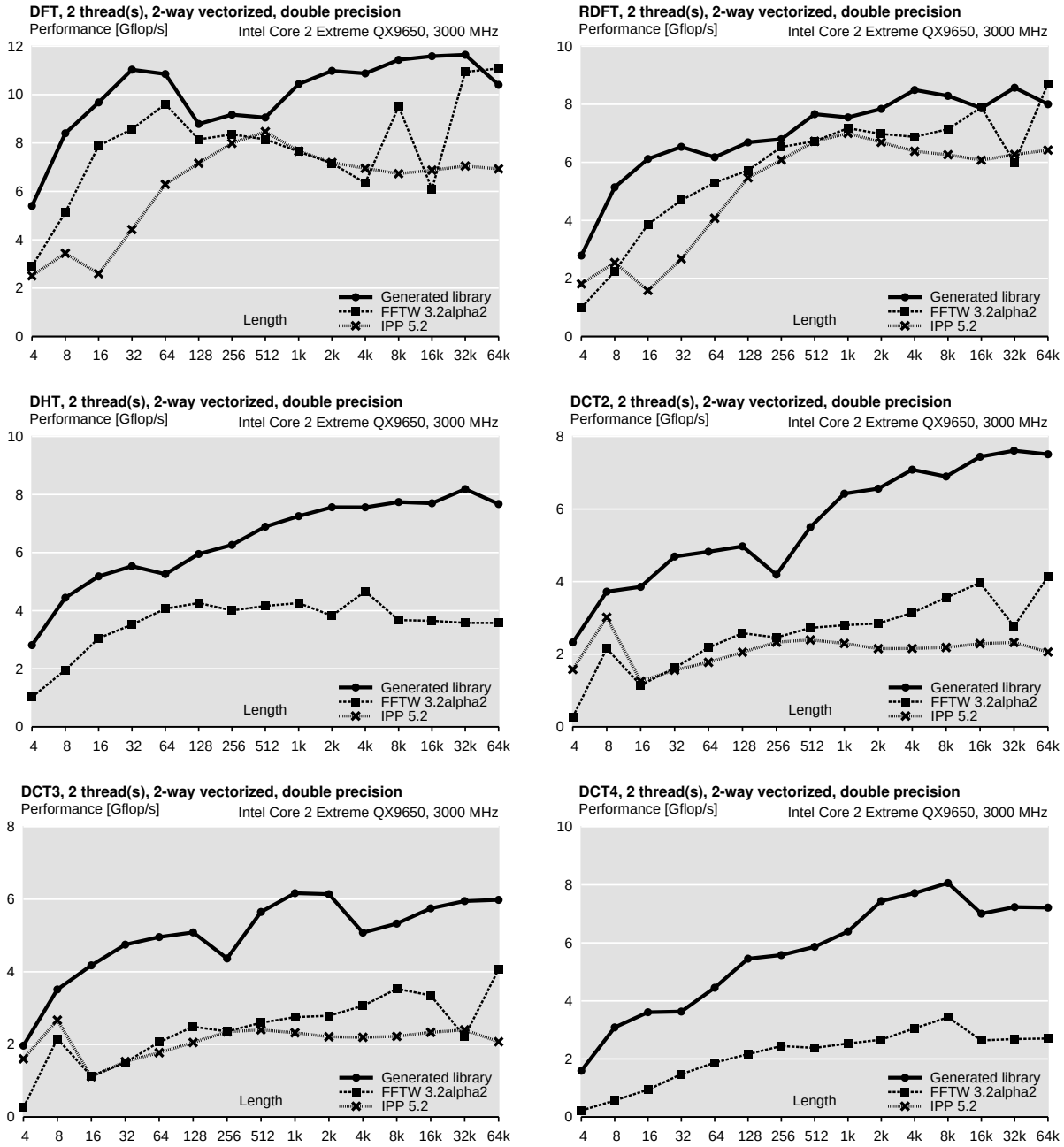


Figure A.47: Generated libraries performance: trigonometric transforms, 2-way vectorization (double precision), up to 2 threads. Platform: Intel Core 2 Extreme QX9650.

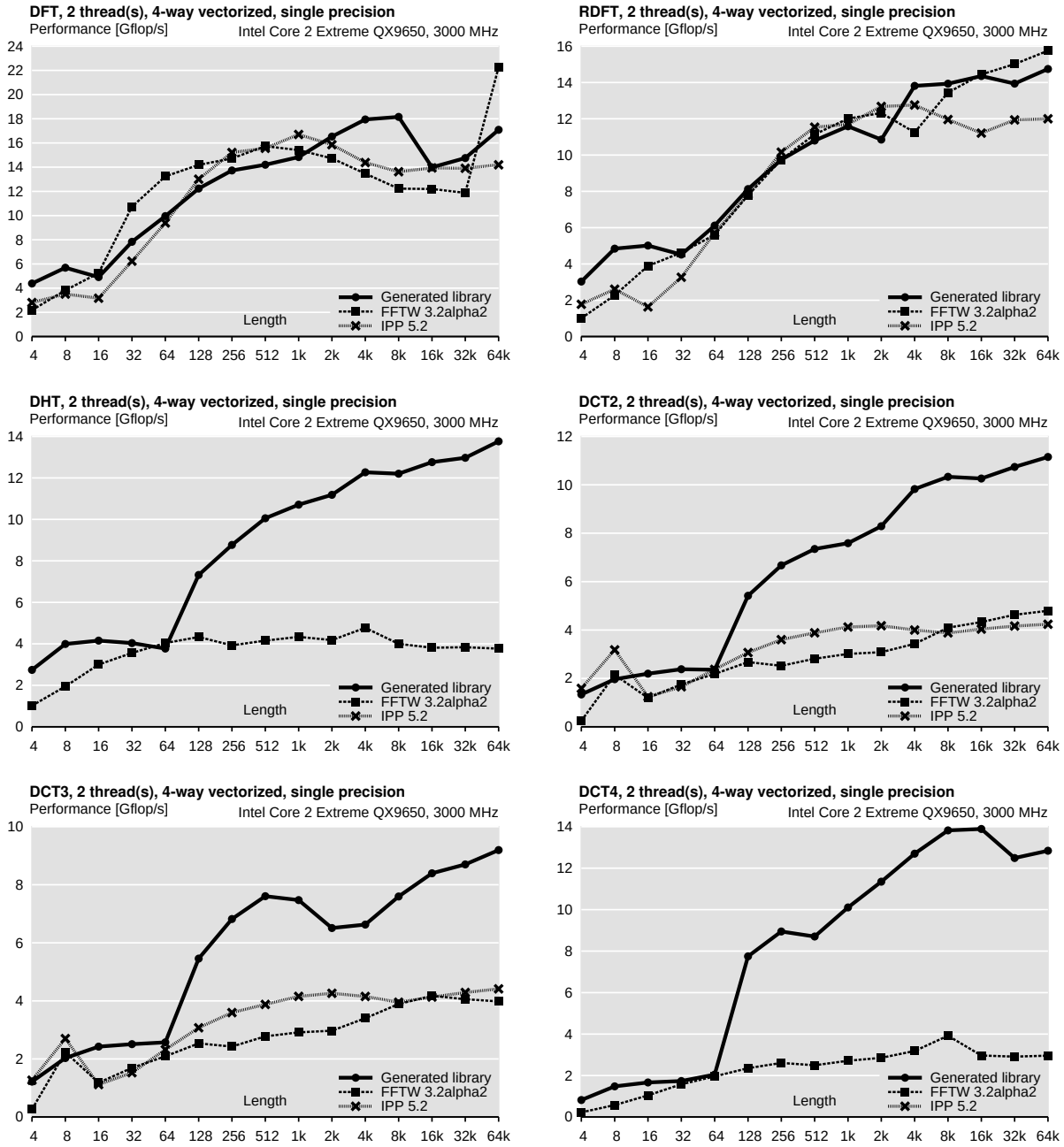


Figure A.48: Generated libraries performance: trigonometric transforms, 4-way vectorization (single precision), up to 2 threads. Platform: Intel Core 2 Extreme QX9650.

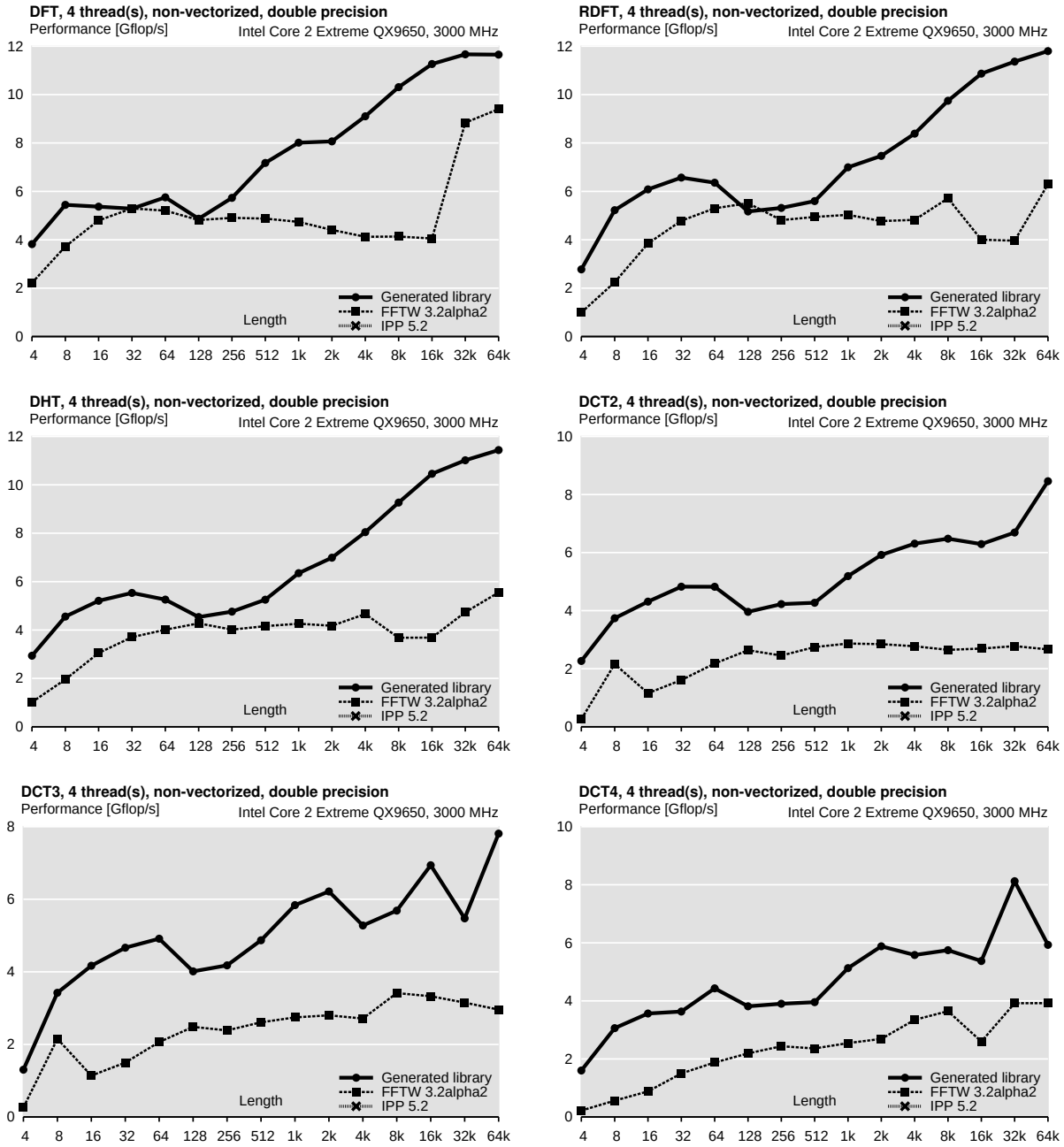


Figure A.49: Generated libraries performance: trigonometric transforms, no vectorization (double precision), up to 4 threads. Platform: Intel Core 2 Extreme QX9650.

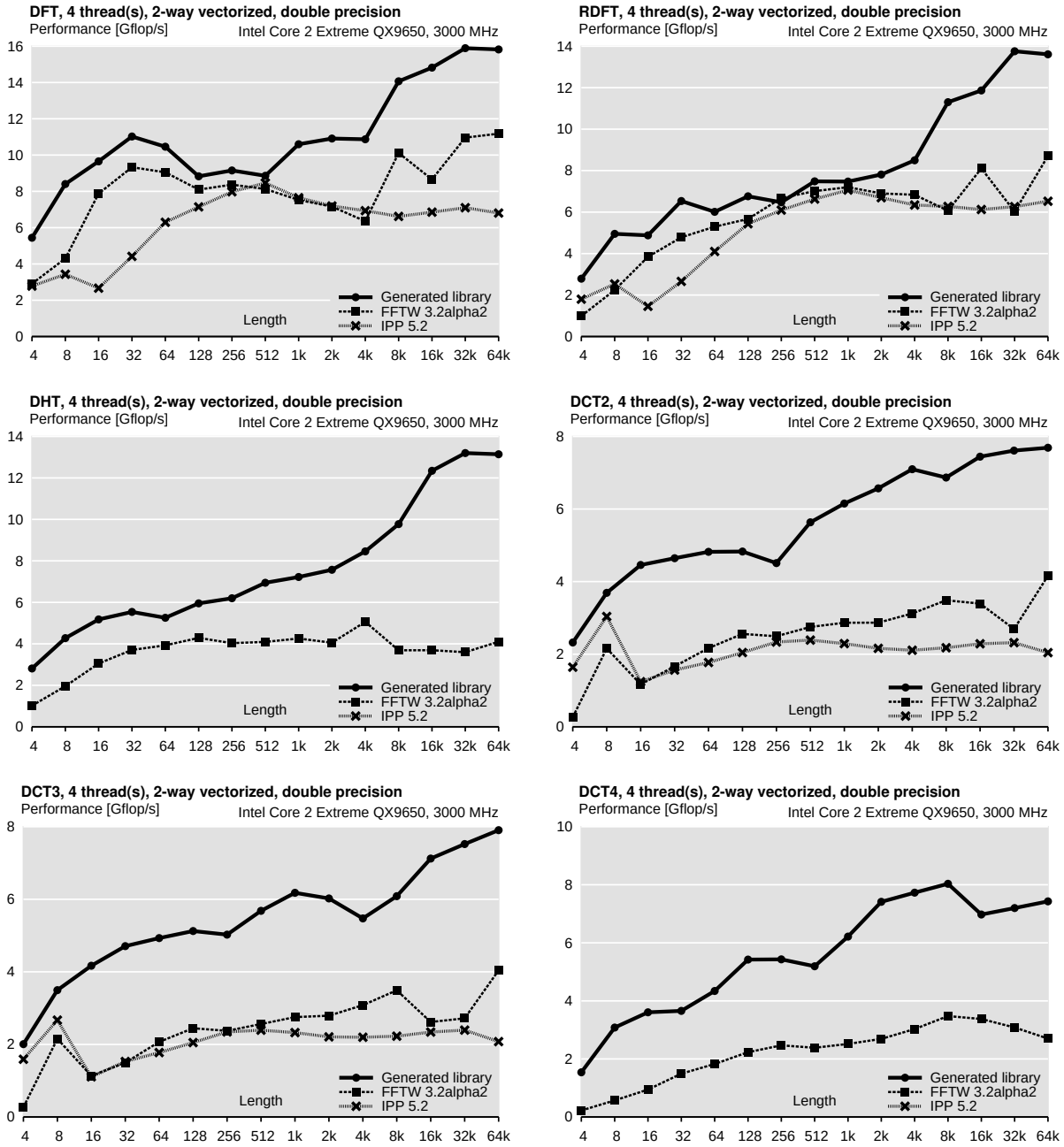


Figure A.50: Generated libraries performance: trigonometric transforms, 2-way vectorization (double precision), up to 4 threads. Platform: Intel Core 2 Extreme QX9650.

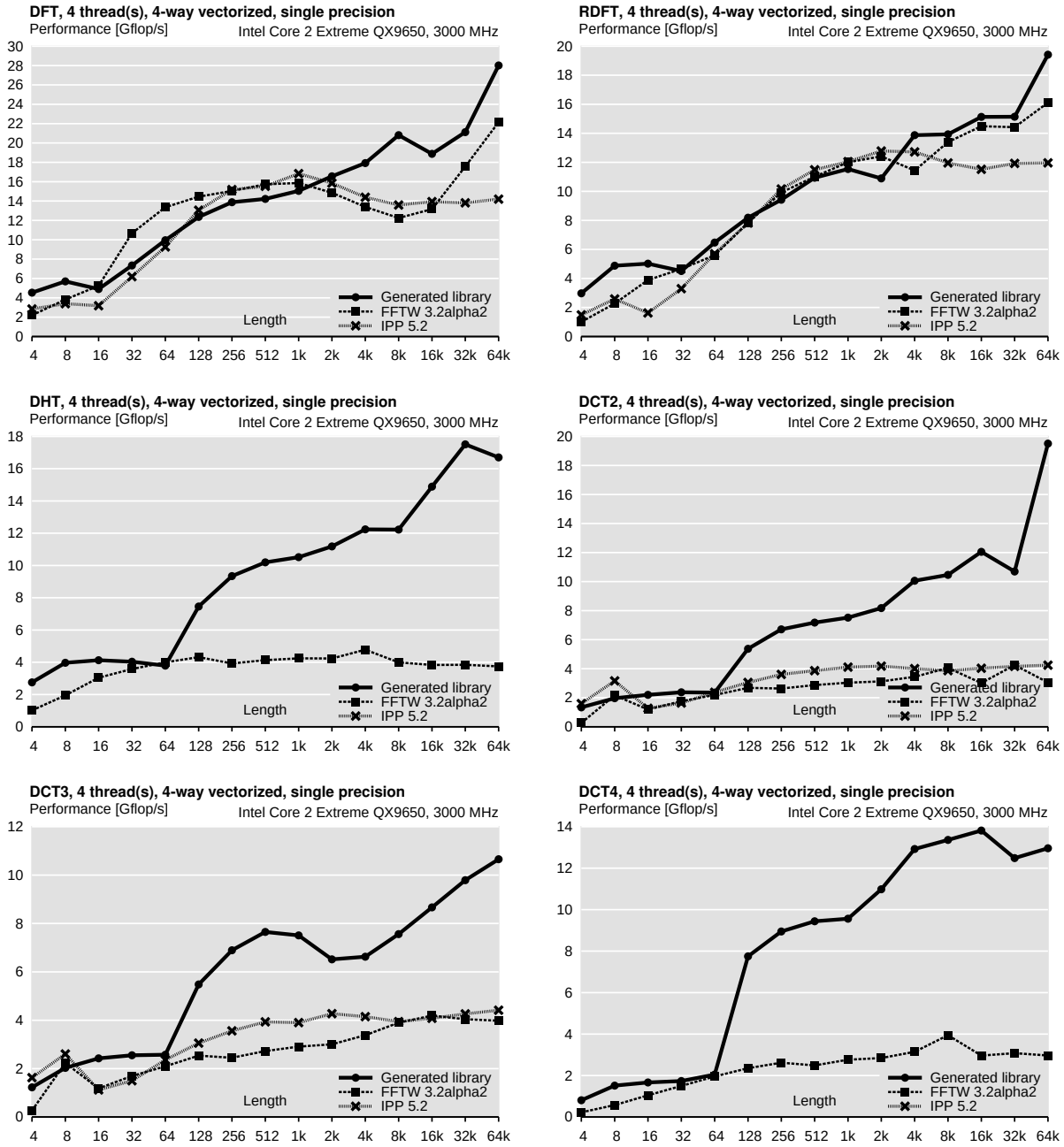


Figure A.51: Generated libraries performance: trigonometric transforms, 4-way vectorization (single precision), up to 4 threads. Platform: Intel Core 2 Extreme QX9650.

Bibliography

[1] *ACM Conference Generative Programming and Component Engineering (GPCE)*, since 2002.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., 1986.

[3] Rami A. Al Na'mneh, W. D. Pan, and R. Adhami. Communication efficient adaptive matrix transpose algorithm for FFT on symmetric multiprocessors. In *Proc. Southeastern Symposium on System Theory (SSST)*, pages 312–315, 2005.

[4] Rami A. Al Na'mneh, W. D. Pan, and R. Adhami. Parallel implementation of 1-D fast Fourier transform without inter-processor communications. In *Proc. Southeastern Symposium on System Theory (SSST)*, pages 307–311, 2005.

[5] A. Ali, L. Johnsson, and D. Mirković. Empirical auto-tuning code generator for FFT and trigonometric transforms. In *Proc. ODES: 5th Workshop on Optimizations for DSP and Embedded Systems*, March 2007. in conjunction with International Symposium on Code Generation and Optimization (CGO).

[6] Randy Allen and Ken Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, 1987.

[7] G. Almasi, F. G. Gustavson, and J. E. Moreira. Design and evaluation of a linear algebra package for java. In *Proc. ACM conference on Java Grande*, pages 150–159, 2000.

[8] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 3rd edition, 1999.

[9] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[10] D. H. Bailey. FFTs in external or hierarchical memory. *J. Supercomputing*, 4:23–35, 1990.

[11] Dmitry Baksheev, Victor Pasko, Andrey Bakshaev, Peter Tang, Boris Sabanin, and David Kuck. Integration of Spiral-generated kernels into Intel MKL and IPP libraries, 2005–2008. collaborative effort with the Spiral group.

[12] U. Banerjee, R. Eigenmann, A. Nicolau, and D. A. Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243, 1993.

[13] D. Batory, C. Johnson, B. MacDonald, and D. von Heeder. Achieving extensibility through product-lines and domain-specific languages: A case study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):191–214, 2002.

[14] D. Batory, R. Lopez-Herrejon, and J-P. Martin. Generating product-lines of product-families. In *Proc. Automated Software Engineering Conference (ASE)*, 2002.

[15] G. Baumgartner, A. Auer, D. E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. J. Harrison, S. Hirata, S. Krishanmoorthy, S. Krishnan, C.-C. Lam, Q. Lu, M. Nooijen, R. M. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov. Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. *Proceedings of the IEEE*, 93(2), 2005. special issue on "Program Generation, Optimization, and Adaptation".

[16] G. Baumgartner, D. Bernholdt, D. Cociovora, R. Harrison, M. Nooijen, J. Ramanujan, and P. Sadayappan. A performance optimization framework for compilation of tensor contraction expressions into parallel programs. In *Proc. Int'l Workshop on High-Level Parallel Programming Models and Supportive Environments (held in conjunction with IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS))*, 2002.

[17] K. Beauchamp. *Applications of Walsh and Related Functions.* Academic Press, 1984.

[18] T. Beer. Walsh transforms. *American Journal of Physics*, 49(5):466–472, 1981.

[19] Jon Bentley. Programming pearls: little languages. *Communications of the ACM*, 29(8):711–721, 1986.

[20] Glenn D. Bergland. Numerical analysis: A fast Fourier transform algorithm for real-valued series. *Commun. ACM*, 11(10):703–710, 1968.

[21] Dennis S. Bernstein. *Matrix Mathematics.* Princeton University Press, 2005.

[22] Eran Bida and Sivan Toledo. An automatically-tuned sorting library. *Software — Practice & Experience*, 37(11):1161–1192, 2007.

[23] Paolo Bientinesi, John A. Gunnels, Margaret E. Myers, Enrique Quintana-Orti, and Robert van de Geijn. The science of deriving dense linear algebra algorithms. *ACM Transactions on Mathematical Software*, 31(1):1–26, March 2005.

[24] Paolo Bientinesi, Enrique Quintana-Orti, and Robert van de Geijn. Representing linear algebra algorithms in code: The FLAME APIs. *ACM Transactions on Mathematical Software*, 31(1), March 2005.

[25] Aart J. C. Bik, Milind Girkar, Paul M. Grey, and Xinmin Tian. Automatic intra-register vectorization for the intel architecture. *International Journal of Parallel Programming*, 30(2):65–98, 2002.

[26] Brian Blount and Siddhartha Chatterjee. An evaluation of Java for numerical computing. *Journal of Scientific Programming*, 7(2):97–110, 1999.

[27] L. I. Bluestein. A linear filtering approach to the computation of the discrete Fourier transform. *IEEE Trans. on Audio and Electroacoustics*, AU-18(4):451–455, 1970.

[28] R. F. Boisvert, J. Moreira, M. Philippsen, and R. Pozo. Java and numerical computing. *Computing in Science & Engineering*, 3(2):18–24, Mar/Apr 2001.

[29] Ronald F. Boisvert, Jack J. Dongarra, Roldan Pozo, Karin A. Remington, and G. W. Stewart. Developing numerical libraries in Java. *Concurrency — Practice and Experience*, 10(11–13):1117–1129, 1998.

[30] A. Bonelli, F. Franchetti, J. Lorenz, M. Püschel, and C. W. Ueberhuber. Automatic performance optimization of the discrete Fourier transform on distributed memory computers. In *Proc. International Symposium on Parellel and Distributed Processing and Applications (ISPA)*, 2006. to appear.

[31] R. N. Bracewell. Discrete Hartley transform. *J. Optical Society America*, 73(12):1832–1835, 1983.

[32] R. N. Bracewell. The fast Hartley transform. *Proceedings of the IEEE*, 72, 1984.

[33] V. Britanak and K. R. Rao. The fast generalized discrete Fourier transforms: A unified approach to the discrete sinusoidal transforms computation. *Signal Processing*, 79(2):135–150, 1999.

[34] P. Burgisser, M. Clausen, and M. A. Shokrollahi. *Algebraic Complexity Theory*. Springer, 1997.

[35] Rohit Chandra, Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan, and Jeff McDonald. *Parallel Programming in OpenMP*. Morgan Kaufman, 2000.

[36] Srinivas Chellappa, Franz Franchetti, and Markus Püschel. FFT program generation for the Cell BE. In *International Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA)*, 2008.

[37] W. H. Chen, C. H. Smith, and S. C. Fralick. A fast computational algorithm for the discrete cosine transform. *IEEE Trans. on Communications*, COM-25(9):1004–1009, 1977.

[38] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. of Computation*, 19:297–301, 1965.

[39] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[40] Krzysztof Czarnecki, John O'Donnell, Jörg Striegnitz, and Walid Taha. DSL implementation in metaocaml, template haskell, and C++. In Batory, Consel, Lengauer, and Odersky, editors, *Dagstuhl Workshop on Domain-specific Program Generation*, LNCS. 2004.

[41] Paolo D'Alberto, Peter A. Milder, Aliaksei Sandryhaila, Franz Franchetti, Ja mes C. Hoe, José M. F. Moura, Markus Püschel, and Jeremy Johnson. Generating fpga accelerated dft libraries. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2007.

[42] Alain Darte. On the complexity of loop fusion. *Parallel Computing*, 26(9):1175–1193, 2000.

[43] Ingrid Daubechies and Wim Sweldens. Factoring wavelet transforms into lifting steps. *Journal of Fourier Analysis and Applications*, 4(3):247–269, 1998.

[44] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, C. Whaley, and K. Yelick. Self adapting linear algebra algorithms and software. *Proceedings of the IEEE*, 93(2), 2005. special issue on "Program Generation, Optimization, and Adaptation".

[45] Nachum Dershowitz and David A. Plaisted. Rewriting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume 1, chapter 9, pages 535–610. Elsevier, 2001.

[46] Alexandre E. Eichenberger, Peng Wu, and Kevin O'brien. Vectorization for SIMD architectures with alignment constraints. In *Proc. ACM PLDI*, June 2004.

[47] D. F. Elliott and K. R. Rao. *Fast Transforms: Algorithms, Analyses, Applications*. Academic Press, 1982.

[48] FFTW web site. `www.fftw.org`.

[49] F. Franchetti, S. Kral, J. Lorenz, and C. W. Ueberhuber. AURORA report series on simd vectorization techniques for straight line code. AURORA Technical Report TR2003-01, TR2003-10, TR2003-11, TR2003-12, Institute for Applied Mathematics and Numerical Analysis, Vienna University of Technology, 2003.

[50] F. Franchetti and M Püschel. A SIMD vectorizing compiler for digital signal processing algorithms. In *Proc. IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS)*, pages 20–26, 2002.

[51] F. Franchetti and M Püschel. Short vector code generation for the discrete Fourier transform. In *Proc. IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS)*, pages 58–67, 2003.

[52] F. Franchetti, Y. Voronenko, and M. Püschel. Loop merging for signal transforms. In *Proc. Programming Language Design and Implementation (PLDI)*, pages 315–326, 2005.

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

[54] F. Franchetti, Y. Voronenko, and M. Püschel. A rewriting system for the vectorization of signal transforms. In *Proc. High Performance Computing for Computational Science (VECPAR)*, 2006.

[55] Franz Franchetti, 2005. Personal communication.

[56] Franz Franchetti, Stefan Kral, Juergen Lorenz, and Christoph Ueberhuber. Efficient utilization of SIMD extensions. *Proceedings of the IEEE*, 93(2), 2005. special issue on "Program Generation, Optimization, and Adaptation".

[57] Franz Franchetti and Markus Püschel. Program generation for parallel platforms, 2007. NSF proposal and grant 0702386.

[58] Franz Franchetti, Yevgen Voronenko, Peter A. Milder, Srinivas Chellappa, Marek Telgarsky, Hao Shen, Paolo D'Alberto, Frédéric de Mesmay, James C. Hoe, José M. F. Moura, and Markus Püschel. Domain-specific library generation for parallel software and hardware platforms. In *NSF Next Generation Software Program Workshop (NSFNGS) colocated with IPDPS*, 2008.

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

[60] M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proc. IEEE Int'l Conf. Acoustics, Speech, and Signal Processing (ICASSP)*, volume 3, pages 1381–1384, 1998.

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

[62] Aca Gačić. *Automatic Implementation and Platform Adaptation of Discrete Filtering and Wavelet Algorithms*. PhD thesis, Carnegie Mellon University, 2004.

[63] Neal Glew. Object closure conversion. *Electronic Notes in Theoretical Computer Science*, 26, 1999.

[64] I. J. Good. The interaction algorithm and practical Fourier analysis. *Journal Royal Statist. Soc.*, B20:361–375, 1958.

[65] K. J. Gough. Little language processing, an alternative to courses on compiler construction. *SIGCSE Bulletin*, 13(3):31–34, 1981.

[66] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Commun. ACM*, 35(8):66–80, 1992.

[67] Paul Hudak. Domain specific languages. Available from author on request, 1997.

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

[69] R. Janka, R. Judd, J. Lebak, M. Richards, and D. Campbell. VSIPL: an object-based open standard API for vector, signal, and image processing. In *Proc. ICASSP*, volume 2, pages 949–952, 2001.

[70] 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.

[71] Thomas Johnsson. Lambda lifting: transforming programs to recursive equations. In *Proc. Conference on Functional programming languages and computer architecture*, pages 190–203, New York, NY, USA, 1985. Springer-Verlag.

[72] Ken Kennedy and Kathryn S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *1993 Workshop on Languages and Compilers for Parallel Computing*, number 768, pages 301–320, Portland, Ore., 1993. Berlin: Springer Verlag.

[73] Oleg 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`.

[74] S. Kral, F. Franchetti, J. Lorenz, and C. W. Ueberhuber. SIMD vectorization of straight line FFT code. In *Proceedings of the EuroPar '03 Conference on Parallel and Distributed Computing LNCS 2790*, pages 251–260, 2003.

[75] Andreas Krall and Sylvain Lelait. Compilation techniques for multimedia processors. *International Journal of Parallel Programming*, 28(4):347–361, 2000.

[76] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proc. ACM PLDI*, 2000.

[77] Samuel Larsen, Emmett Witchel, and Saman P. Amarasinghe. Increasing and detecting memory address congruence. In *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 18–29, 2002.

[78] James Lebak, Jeremy Kepner, Henry Hoffmann, and Edward Rutledge. Parallel VSIPL++: An open standard software library for high-performance parallel signal processing. *Proceedings of the IEEE*, 93(2), 2005. special issue on "Program Generation, Optimization, and Adaptation".

[79] Xiaoming Li, María J. Garzarán, and David Padua. A dynamically tuned sorting library. In *Proc. International Symposium on Code Generation and Optimization (CGO)*, pages 111–124, 2004.

[80] Xiaoming Li, María J. Garzarán, and David Padua. Optimizing sorting with genetic algorithm. In *International Symposium on Code Generation and Optimization (CGO)*, pages 99–110, 2005.

[81] J. Makhoul. A fast cosine transform in one and two dimensions. *IEEE Trans. on Acoustics, Speech, and Signal Processing*, ASSP-28(1):27–34, 1980.

[82] Henrique S. Malvar. *Signal Processing with Lapped Transforms.* Artech House Publishers, 1992.

[83] 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)*, 2008.

[84] D. Mirković and S. L. Johnsson. Automatic performance tuning in the UHFFT library. In *Proc. Int'l Conf. Computational Science (ICCS)*, volume 2073 of *LNCS*, pages 71–80. Springer, 2001.

[85] Dorit Naishlos. Autovectorization in GCC. In *Proc. GCC Developers Summit*, June 2004. Online: `ftp://gcc.gnu.org/pub/gcc/summit/2004/Autovectorization.pdf`.

[86] Dorit Naishlos, Marina Biberstein, Shay Ben-David, and Ayal Zaks. Vectorizing for a SIMdD DSP architecture. In *Proc. International Conference on Compilers, architecture and synthesis for embedded systems (CASES)*, pages 2–11, 2003.

[87] A. Norton and A. J. Silberger. Parallelization and performance analysis of the Cooley-Tukey FFT algorithm for shared-memory architectures. *IEEE Trans. Comput.*, 36(5):581–591, 1987.

[88] Visual Numerics. JMSL numerical library for Java applications, April 2008. `http://www.vni.com/`.

[89] H. J. Nussbaumer. *Fast Fourier Transformation and Convolution Algorithms*. Springer, 2nd edition, 1982.

[90] Dorit Nuzman and Richard Henderson. Multi-platform auto-vectorization. In *Proc. International Symposium on Code Generation and Optimization (CGO)*, pages 281–294, 2006.

[91] Dorit Nuzman, Ira Rosen, and Ayal Zaks. Auto-vectorization of interleaved data for SIMD. In *Proc. Programming Language Design and Implementation (PLDI)*, pages 132–143, 2006.

[92] Takuya Ooura. General purpose FFT (fast Fourier/cosine/sine transform) package, April 2008. `http://www.kurims.kyoto-u.ac.jp/~ooura/`.

[93] Alan V. Oppenheim and Ronald W. Schafer. *Discrete-Time Signal Processing*. Prentice-Hall, 2nd edition, 1999.

[94] 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.

[95] William Pugh and David Wonnacott. Constraint-based array dependence analysis. *ACM Trans. Progr. Lang. Syst.*, 20(3):635–678, 1998.

[96] M. Püschel and J. M. F. Moura. Algebraic signal processing theory: 1-D space. *IEEE Transactions on Signal Processing*, 2008. to appear.

[97] M. Püschel and J. 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. a longer version is available at `http://arxiv.org/abs/cs.IT/0702025`.

[98] M. Püschel and J. M. F. Moura. Algebraic signal processing theory: Foundation and 1-D time. *IEEE Transactions on Signal Processing*, 2008. to appear.

[99] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005. special issue on "Program Generation, Optimization, and Adaptation".

[100] M. Püschel, B. Singer, M. Veloso, and J. M. F. Moura. Fast automatic generation of DSP algorithms. In *Proc. Int'l Conf. Computational Science (ICCS)*, volume 2073 of *LNCS*, pages 97–106. Springer, 2001.

[101] Markus Püschel, Peter A. Milder, and James C. Hoe. Permuting streaming data using rams. submitted for publication.

[102] C. M. Rader. Discrete Fourier transforms when the number of data samples is prime. *Proceedings of the IEEE*, 56:1107–1108, 1968.

[103] K. R. Rao and P. Yip. *Discrete Cosine Transform: Algorithms, Advantages, Applications*. Academic Press, 1990.

[104] Gang Ren, Peng Wu, and David Padua. An empirical study on the vectorization of multimedia applications for multimedia extensions. In *Proc. IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS)*, 2005.

[105] G. E. Révész. *Introduction to Formal Languages*. McGraw-Hill, 1983.

[106] Paul N. Schwarztrauber. Multiprocessor FFTs. *Parallel Computing*, 5:197–210, 1987.

[107] Jaewook Shin, Mary Hall, and Jacqueline Chame. Superword-level parallelism in the presence of control flow. In *Proc. International Symposium on Code Generation and Optimization (CGO)*, pages 165–175, 2005.

[108] B. Singer and M. Veloso. Stochastic search for signal processing algorithm optimization. In *Proc. Supercomputing*, 2001.

[109] B. Singer and M. M. Veloso. Automating the modeling and optimization of the performance of signal transforms. *IEEE Trans. Signal Processing*, 50(8):2003–2014, 2002.

[110] B. Singer and M. M. Veloso. Learning to Construct Fast Signal Processing Implementations. *Journal of Machine Learning Research*, 3:887–919, 2002.

[111] D. R. Smith. Mechanizing the development of software. In M. Broy, editor, *Calculational System Design, Proc. of the International Summer School Marktoberdorf*. NATO ASI Series, IOS Press, 1999. Kestrel Institute Technical Report KES.U.99.1.

[112] H. V. Sorensen, D. L. Jones, M. T. Heideman, and C. S. Burrus. Real-valued fast Fourier transform algorithms. *IEEE Trans. on Acoustics, Speech, and Signal Processing*, ASSP-35(6):849–863, 1987.

[113] Gilbert Strang and Trong Nguyen. *Wavelets and Filter Banks*. Wellesley-Cambridge Press, 1998.

[114] P. N. Swarztrauber. Vectorizing the fast Fourier transforms. *Parallel Computations*, pages 51–83, 1982.

[115] P. N. Swarztrauber. Fast Fourier transform algorithms for vector computers. *Parallel Computing*, pages 45–63, 1984.

[116] P. N. Swarztrauber. FFTPACK, 2006. `www.netlib.org/fftpack/`.

[117] Daisuke Takahashi. A blocking algorithm for parallel 1-D FFT on shared-memory parallel computers. *Lecture Notes in Computer Science*, 2367:380–389, 2002.

[118] Daisuke Takahashi, Mitsuhisa Sato, and Taisuke Boku. An OpenMP implementation of parallel FFT and its performance on IA-64 processors. *Lecture Notes in Computer Science*, 2716:99–108, 2003.

[119] Ping Tak Peter Tang. DFTI — a new interface for fast Fourier transform libraries. *ACM Trans. Math. Softw.*, 31(4):475–507, 2005.

[120] Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, 2nd edition, 1999.

[121] R. Tolimieri, M. An, and C. Lu. *Algorithms for Discrete Fourier Transforms and Convolution*. Springer, 2nd edition, 1997.

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

[123] M. Vetterli and H.J. Nussbaumer. Simple FFT and DCT Algorithms with reduced Number of Operations. *Signal Processing*, 6:267–278, 1984.

[124] Martin Vetterli and Jelena Kovačević. *Wavelets and Subband Coding*. Prentice-Hall, 1995.

[125] Yevgen Voronenko and Markus Püschel. Algebraic signal processing theory: Cooley-Tukey type algorithms for real DFTs. *IEEE Transactions on Signal Processing*, 2008. accepted for publication.

[126] Z. Wang. Fast algorithms for the discrete W transform and for the discrete Fourier transform. *IEEE Trans. Acoustics, Speech, and Signal Processing*, ASSP-32(4):803–816, 1984.

[127] Z. Wang. On computing the discrete Fourier and cosine transforms. *IEEE Trans. on Acoustics, Speech, and Signal Processing*, ASSP-33(4):1341–1344, 1985.

[128] Piotr Wendykier. JTransforms 1.2, April 2008. `http://piotr.wendykier.googlepages.com/jtransforms`.

[129] Piotr Wendykier and James G. Nagy. Large-scale image deblurring in Java. In *Proc. Int. Conf. on Computational Science*, June 2008.

[130] R. C. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software (ATLAS). In *Proc. Supercomputing*, 1998. `math-atlas.sourceforge.net`.

[131] R. C. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.

[132] Wikipedia. Closure (computer science), 2008. [Online; accessed 16-April-2008].

[133] Wikipedia. Closure (mathematics), 2008. [Online; accessed 16-April-2008].

[134] Michael Wolfe. *High performance compilers for parallel computing*. Addison-Wesley, Redwood City, CA, 1996.

[135] Peng Wu, Alexandre E. Eichenberger, and Amy Wang. Efficient SIMD code generation for runtime alignment and length conversion. In *Proc. International Symposium on Code Generation and Optimization (CGO)*, pages 153–164, 2005.

[136] Peng Wu, Alexandre E. Eichenberger, Amy Wang, and Peng Zhao. An integrated simdization framework using virtual vectors. In *Proc. 19th Annual International Conference on Supercomputing (ICS)*, pages 169–178, 2005.

[137] J. Xiong, J. Johnson, R. Johnson, and D. Padua. SPL: A language and compiler for DSP algorithms. In *Proc. PLDI*, pages 298–308, 2001.

[138] Field Van Zee, Paolo Bientinesi, Tze Meng Low, and Robert van de Geijn. Scalable parallelization of FLAME code via the workqueuing model. *ACM Transactions on Mathematical Software*, 34(2), June 2008.

[139] Hans Zima and Barbara Chapman. *Supercompilers for parallel and vector computers.* ACM Press, New York, 1990.