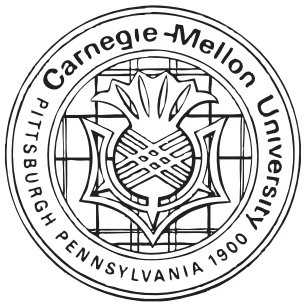
On the Computer Generation of Adaptive Numerical Libraries



A Dissertation Submitted in Partial Fulfillment of
the Requirements for the Degree of Doctor of Philosophy

FRÉDÉRIC DE MESMAY
Supervised by Markus Püschel
May 2010

Electrical and Computer Engineering,
Carnegie Mellon University,
Pittsburgh, Pennsylvania, USA

To Be Examined and Approved by the Doctoral Committee:

Professor ALBERT COHEN, INRIA Saclay, France
Professor FRANZ FRANCHETTI, Carnegie Mellon University
Professor JOSÉ MOURA, Carnegie Mellon University
Professor KESHAV PINGALI, University of Texas, Austin
Professor MARKUS PÜSCHEL (Advisor), Carnegie Mellon University

This work was supported by NSF through awards 0325687, 0702386 and by DARPA through DOI grant NBCH1050009.

In memory of my friends Félix and Rémi,
who both chose to leave the party a bit early

Abstract

Very fast runtime is crucial in many applications in scientific computing, multimedia processing, communication, and control. Most of these applications spend the bulk of the computation in well-known mathematical functions which are provided by highly optimized libraries. The development and maintenance of these libraries has become extraordinarily difficult. Optimal performance requires multiple-core balancing, careful use of vector instruction sets, and locality optimization. These optimizations require highly-skilled programmers and are often platform-specific, which means maintenance is a considerable effort given the short processor release cycles.

The Spiral system has successfully addressed these issues by automatically generating high performance libraries given only a high-level mathematical algorithm description in a language called SPL. Spiral produced high performance code using a number of techniques including SPL rewriting systems and a form iterative compilation. However, to date Spiral has been limited in two key aspects. First, Spiral could only generate libraries for the domain of linear transforms; second, all optimizations for a specific target platform are performed during the source code generation, that is, the produced libraries themselves had no dynamic platform-adaptation mechanism.

In this thesis we make progress on both fronts. We present a framework and its implementation for the computer generation of functionalities that are not transforms, specifically matrix multiplication and convolutional decoding. The framework builds on the operator language (OL) that we introduce and that extends SPL. Similar to prior work on transforms, we then develop OL rewriting system to explore algorithm choice, to vectorize and parallelize, and to derive the basic library structure called recursion step closure. The actual code is obtained through a backend that supports different target languages. The generated libraries exhibit a performance comparable to libraries that are hand-written for commodity workstations.

Further, we enable the generation of platform-adaptive libraries, through adaptation modules that can be inserted into our libraries, which are generated to support different ways to compute

the same function. We distinguish between online adaptation and offline adaptation and provide mechanism for both. Online adaptation happens during the actual user function call when the input size is provided. Given this size, the library searches for the best computation strategy inside the library, which can then be used for subsequent computations of this size. We provide the dynamic programming strategy used in prior work and introduce a novel kind of Monte-Carlo search on graphs. Finally, we present a machine learning approach that performs offline (during installation time) adaptation with an online adaptive library. First a search is run to produce solutions for a set of sizes. Based on the result, a learning algorithm derives solutions for all sizes in the form of a set of decision trees that are then inserted into the library to render it deterministic. Experiments show the viability of both approaches.

Table of Contents

List of Figures

List of Tables

> *Let the machine take care of the machines and I'll go spend more time with my family, or golf.*

– MARK GODDARD

Acknowledgments

I would like to express my deepest gratitude to all those that supported me during these past years. Faculty, staff, students, friends and family members helped me complete this dissertation and I would like to thank them all for their advice, their interest and their patience. Some of them have gone out of their ways for me and I mention them out here but, truly, lots contributed. Thank you all!

First things first, I'd like to thank both my parents Olivier and Nelly for making it all possible. During the long years of my education, they have provided me much of the moral support, encouragement, lots of freedom and trust that helped me making great strides in any direction I chose. Besides, they also gave me my younger brother Arnaud with whom they probably intended to correct most of my (secret?) design flaws. History being recursive, let me also have a special thought for their own parents, my Mom's who I never had the chance to meet and my Dad's who both passed away along the course of this PhD.

At the core of this dissertation lies the vision of my advisor Markus Püschel that I hope to push forward with this thesis. Throughout these years, he has been strong and understanding, always having faith in my work and being my best advocate. At times where my motivation was in the trough of the wave, his stubborn optimism always helped me return back to the crest. Thank you.

Franz Franchetti and Yevgen Voronenko have always been generous with their precious time and immense knowledge. They both played a key role in this thesis: Franz with the initial ideas for the operator language underlying part of this thesis work, and Yevgen for creating so many interesting challenges with his general input size library framework. Hours of discussions with them helped me draft the blueprint for this work and secure the foundations of this dissertation. They built so many of the tools I've used during these last years that I'm glad I payed them back by teaching them how to barbecue!

Committee member José Moura has been the wise person in the background, mostly calm but

quick and sharp when needed. Reviewing my work at periodical times, he helped me to set the bar high... and also paid for the beverages after technical demonstrations!

Albert Cohen and Keshav Pingali graciously agreed to be the external members of my thesis committee and I would like to thank them for taking the time. Their feedback on my thesis prospectus gave rise to many interesting ideas and helped structure this document. On a more personal note, I am very grateful to Albert for graciously offering me an office in France so I could be at home while writing this dissertation.

PhD candidates like to eat too! I would like to acknowledge here the sponsors who provided my funding and put the food on my table: the National Science Foundation (NSF) through awards 0325687 and 0702386 and the Defense Advanced Research Projects Agency (DARPA) through the DOI grant NBCH1050009.

Helping me organize my ideas, the Spiral group as a whole has been very helpful, suggesting many extensions, triggering interesting discussions and providing support in many ways. Among them, I would like to specifically mention Srinivas Chellappa, Marek Telgarsky, Peter Milder and Volodymyr Arbatov with whom I had the occasion to collaborate more closely. With lots of patience, they have reviewed and restructured my work, helping me transform raw products into finished goods. Of course, we sometimes ended up discussing less technical and more interesting subjects over less innocent and more spirited drinks... Oh, by the way, do you guys remember when we got escorted out of the hotel room? Those were great times indeed!

While in the US, I was offered the occasion to collaborate with a French team including my long time friend Arpad Rimmel and a colleague of his, Olivier Teytaud. It was a great deal of fun to work together and even more fun that it worked out in the end! Wanna know the the relationship between the board game of Go and adaptive libraries? Stay on board!

Some of my undergraduate teachers at the École Polytechnique have really been inspirational to me. Among them, Olivier Temam, Francois Pottier, Dale Miller, Behshad Behzadi, Jean-Jacques Levy and Albert (again!) have stirred and deepened my interest for computer science in general and compilers / computer architecture in particular. This triggered my decision to leave for the US where I also had the occasion to meet sharp professors like Babak Falsafi and James Hoe. In fact, Babak being the one that dug my CMU application out of the stack, he may have triggered the chain of events that is concluded with this dissertation.

I also would like to thank the anonymous users of the online interface to the software Viterbi decoder generator for their comments and continued interest in my work. They convinced me that some of my research is useful in the real world!

There were times where work didn't seem to go anywhere and, in these periods, it was always great to take my mind off things with all the friends I made in the 'Burgh. Joyful Italians, spotless housemates, sophisticated exiles, Dithridge girls, Firehouse rounders, Tarot players, ego volunteers, mighty climbers, screen-tanned office-mates, Harris grillers, rotten tomatoes enthusiasts, happy hour drinkers, Greek architects, and all the others I surely forgot, the cherished time I spent with

them gave me the strength to continue digging forward.

Despite my lengthy stay across the ocean, I have to mention these priceless friends in France who regularly welcomed me with open arms, as if I never left. Specialists of doing nothing around a sofa, amateurs of questionable movies and geeks from before the Internet (!), they know who they are and what they mean to me.

The acknowledgments wouldn't nearly be complete without mentioning my cousin Rémi Laure and my friend Félix Rubio-Nevado. I spent a big part of my childhood with each of them, and they taught me many of the tricks I still use every day. Even my very desire to do research originated in a conversation I had with Félix about dream jobs. It's hard to believe they left.

And finally, there's a very special thank you to my girlfriend Laure who weathered the storm with me for the past three years. She fought quasi-total absence, noisy communications, adverse time zones and hopeless flight attendant strikes with her pristine serenity, her good spirit and her crazy humor. One could even say she was kind of a lighthouse all this time, but, just in case you need directions too, let me warn you, she doesn't know her left from her right.

Paris, May 2010

FRÉDÉRIC DE MESMAY

CHAPTER 1

Introduction

1.1 Overview

Very high runtime performance is one of the key quality aspects for software in a wide range of domains, including scientific computing, image/video processing, communications, and control. In many of these applications, the bulk of the computation is spent inside well known mathematical functions such as matrix multiplications, discrete Fourier transforms, or others. For efficiency reasons, these functions are typically provided by external high-performance libraries. A good example is Intel's Integrated Performance Primitives (IPP), which implements around 10,000 functions from 16 different domains [Intel, 2009a].

Designing and implementing such libraries has become very difficult due to the complexity of current computing platforms. The optimal performance tolerates no bottleneck and every aspect of the code needs to be carefully designed. Memory hierarchy reuse, multiple core load-balancing, vector code scheduling, are but a few of the optimization techniques required to reach the best possible performance. To illustrate the impact of these optimizations, we show in Figure 1.1 the performance of four implementations of a square matrix-matrix multiplication on a commodity quad-core platform. Performance is measured in giga-floating point operations per second (GFlop/s) and all implementations are compiled with a state-of-the-art optimizing compiler from Intel. At the bottom is a naïve triple loop. Optimizing for the memory hierarchy yields about 20x improvement, explicit use of short vector instructions another 2x, and threading for 4 cores another 4x for a total speedup of 160x.

This large gap shows that the optimizing compiler does not perform the necessary optimizations automatically. There could be three fundamental reasons behind this failure: the lack of domain-specific algorithmic knowledge, the weaknesses of the loop optimizer framework, or the lack of a good performance model to decide which transformations are beneficial. The difficult optimization

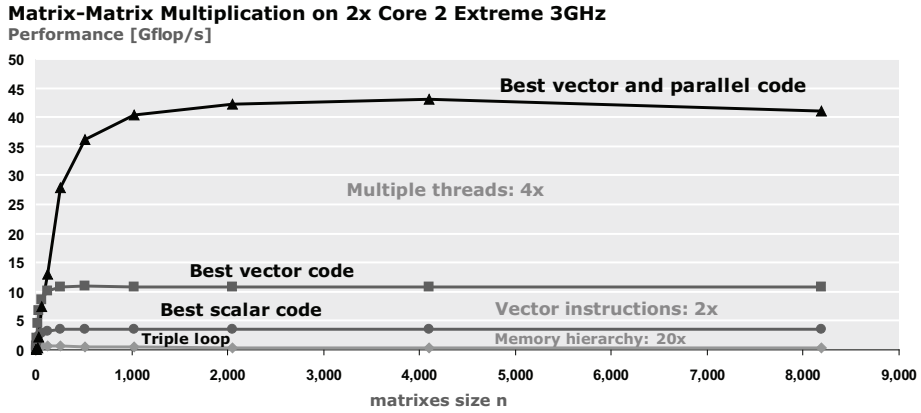


FIGURE 1.1: Underlying factors for performance. The plot shows four double precision implementations multiplying $n \times n$ matrices. The operations count is exactly the same in all four cases: $2n^3$.

task is therefore left with the library developer. Exacerbating the problem, optimal code is platform-dependent and thus, re-optimization and re-implementation is required for each new architecture or even microarchitecture.

Often, library performance objectives are achieved by having experts write different optimized routines for the same function, one for each of the supported architectures. Famous academic and commercial libraries (e.g., GotoBLAS [Goto, 2008], MKL [Intel, 2009b]) use this approach but it requires a continual development in order to support the constantly changing computing platforms. The resulting high development costs of this approach have spawned academic research efforts on mechanizing the optimization process.

One approach to tackle the problem has been to design *adaptive libraries*: In such libraries, the developers allows for a wide range of different implementations of the exact same function and the fastest one on the considered architecture is mechanically found by empirical search. This approach has been successfully demonstrated in various domains, including basic dense linear algebra (ATLAS [Whaley and Dongarra, 1998]), sparse linear algebra (OSKI [Vuduc et al., 2005]), sorting (Adaptive Sorting Library [Li et al., 2004]), and linear transforms (FFTW [Frigo and Johnson, 2005], UHFFT [Ali et al., 2007]).

Another approach to the problem is to raise the abstraction level and have machines, rather than experts, create optimized source code. Humans, of course, are still needed to formalize the algorithm space but the difficult and repetitive task of implementing them, optimizing them and combining them is then fully automatized. Such systems, that we call *library generators*, are domain-specific iterative compilers: their input is an algorithm specification given in a high-level mathematical form and their output is optimized source code implementing it. Along the way, multiple compilation phases are necessary to generate and evaluate a pool of variants. So far, progress has been reported in advanced dense linear algebra (FLAME [Bientinesi, 2006]), quantum chemistry (Tensor

Contraction Engine [Baumgartner et al., 2005]), and linear transforms (Spiral [Püschel et al., 2005], detailed later).

Library generators offer multiple advantages over standard performance libraries. First, they are easily maintainable, and thus extensible, since algorithms, optimizing transformations, and search mechanisms clearly reside in different layers. Second, they offer flexibility: it is for instance possible to target different languages, customize for different needs or even change the library interface. Third, they are rigorous in that the generated implementations and their proofs grow hand in hand and are therefore correct by construction [Dijkstra, 1972]. Of course, library generators also have drawbacks, mainly that the expressivity of their algorithm specification language inherently limits them to a specific domain and that their offline nature might forbid drop-in replacements in legacy user code.

1.2 Goal of the thesis

In this thesis, we push back both these limitations in the context of the library generator Spiral [Püschel et al., 2005]. As of 2010, Spiral has demonstrated that high-performance automatically generated code could make it into commercial performance libraries in the form of fixed input size functionalities [Intel, 2009a]. Recently, the capability of generating general input size libraries has been added to the system, but it comes at a price: automatic platform adaptation has been lost [Voronenko et al., 2008b]. Finally Spiral's domain has to date been restricted to linear transforms.

The goal of the thesis is to extend Spiral's high-performance library generation framework in two orthogonal ways:

1. We extend the formal framework and the generator to new functionalities *beyond linear transforms*, notably matrix-matrix multiplication and Viterbi decoding.

2. We enable the automatic generation of *adaptive general input size* libraries. We provide both novel *online* (at runtime) and *offline* (at installation time) adaptation mechanisms that can be inserted into Spiral-generated libraries.

The library generator we build is schematically displayed in Figure 1.2. It can first be seen as a "black box" with multiple inputs: the target library functionality, the target platform characteristics and the pool of algorithms.

- The *target library functionality* describes the precise operations that the generated library will provide to its users (e.g., row-major or column-major matrix multiplication, forward or

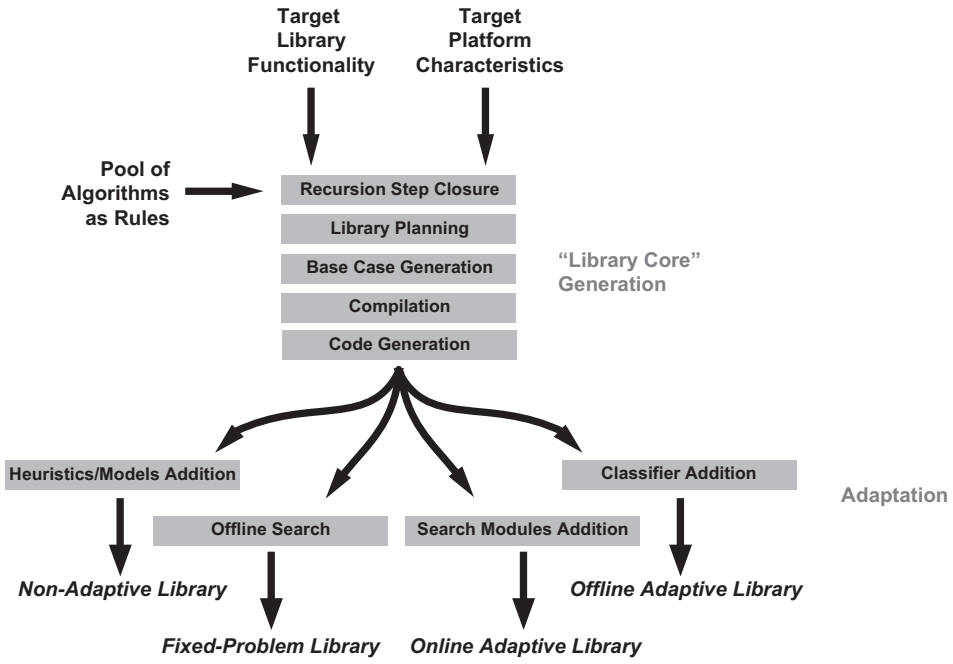


FIGURE 1.2: Architecture of the library generator.

inverse discrete Fourier transform). It is represented as an expression in a domain-specific language, called OL, that we introduce.

- The *target platform characteristics* captures specifics about the architecture that we are generating the library for. It consists of tags that we attach to expressions in our domain-specific language. These specify if the code should be vectorized or threaded.

- The *pool of algorithms* is a set of widely known algorithms such as blocked matrix multiplication, fast Fourier transforms or Viterbi decoding that we represent as rules in OL, extending prior Spiral work.

The system is able to produce libraries of four different types: heuristic-based general input size libraries, fixed input size libraries, online adaptive general input size libraries, and offline adaptive general input size libraries. Before we explain these types of libraries, we first use Figure 1.3 to give possible examples of generated libraries.

In Figure 1.3a, the system is configured to generate a matrix-multiplication library conforming with the general matrix multiplication (GEMM) specification which is a popular interface in the linear algebra community [Dongarra et al., 1990]. The algorithm space consists of recursive block matrix multiplications and the library should be optimized for commodity processors using important features such as multicores and short vector extensions (here, 2-way SSE). The generator outputs a *library core* which consists of a dozen of mutually recursive functions and corresponding

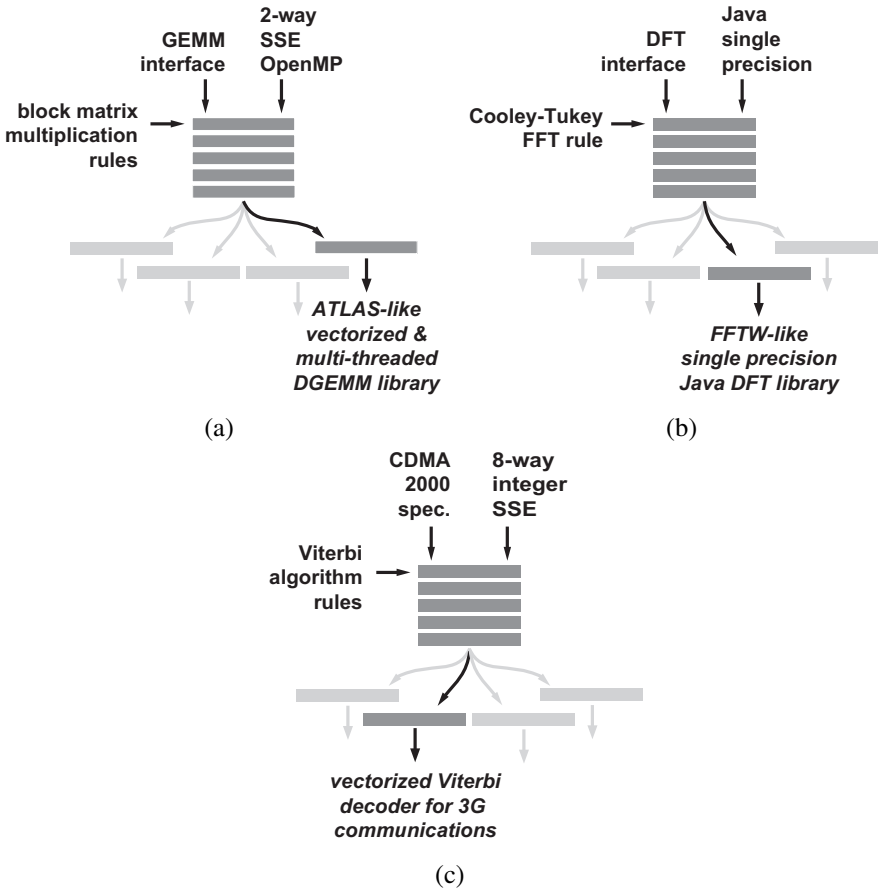


FIGURE 1.3: Some possible inputs to the library generator and the corresponding generated libraries.

base cases, totalizing more than 10,000 lines of C++ code, filled with SSE vector intrinsics and OpenMP threading pragmas. It is not yet usable though: since it can capture a vast domain of algorithmic variants, it has many degrees of freedom that require decisions before any computation can occur. In the presented example, the system has been configured to generate an *offline adaptive* library, which means that these degrees of freedom are to be determined on the target platform during installation time (similarly to ATLAS [Whaley and Dongarra, 1998]). We achieve this by deploying the library core to the user with a statistical classifier and the necessary installation scripts. During installation on a user platform, the classifier will generate multiple decision trees that will be plugged back inside the library core to take the required decisions and hence render the computation deterministic.

In Figure 1.3b, the system is configured to generate a discrete Fourier transform (DFT) library using the Cooley-Tukey algorithm, which is the most famous fast Fourier transform (FFT) [Cooley and Tukey, 1965]. In this case, the target platform is a Java virtual machine and compu-

tations should use single-precision floating-point numbers. In this case, the generator produces a library core that consists of multiple Java classes. Again, it is not directly usable since any single function inside still has multiple open degrees of freedom (e.g, the choice of radix). In this example, the system has been configured to generate an *online adaptive* library, which means that the degrees of freedom are fixed on the target platform at runtime (similarly to FFTW [Frigo and Johnson, 2005]). The core has therefore to be bundled with one of the available search modules that we provide before being released.

In Figure 1.3c, the system is configured to generate a Viterbi decoder [Viterbi, 1995] for the CDMA 2000 standard that is used in 3G cellphones communications. The selected target platform supports 8-way SSE integer operations. In this embedded systems setup, we assume that the computation and platforms are fully specified before generation. In this case, we search offline, during the generation process, and produce a deterministic library specifically tailored for this problem and platform.

1.3 Compared benefits of different library types

As shown in Figure 1.2, one of the goals of the thesis is to enable the computer generation of libraries of four different types within the same framework: fixed input size (prior work, [Püschel et al., 2005]), heuristics based general input size (prior work, [Voronenko, 2008]), online adaptive general input size (*this thesis*) and offline adaptive general input size (*this thesis*). Each of these library flavors inherently possesses different properties that favor or prevent a given usage scenario. We have collected the advantages and limitations of each type of library in Table 1.1.

The first column represents *fixed input size* libraries such as the ones provided by the Spiral-generated IPPgen [Intel, 2009a]. Such libraries are inherently designed to perform a specific computation (or, alternatively, a collection of specific computations) on a specific platform. Without a generator, they are expensive to develop and of limited use which restricts their scope to high-performance kernels on high-profile applications such as the ones that are used inside the GotoBLAS [Goto, 2008]. These restrictions are considerably relaxed when a generator is made directly available to the user which is for instance what we propose, as part of this work, for Viterbi decoders [de Mesmay et al., 2010a]. However, library users might still not be able to accommodate the change in the interface that requires the problem to be of fixed input size. In comparison, all other columns capture a different kind of libraries that we denote as *general input size*. Such libraries are more flexible since the users can dynamically choose the parameters of the computation that has to be performed.

The second column captures general input size non-adaptive performance libraries in which the developers have fixed the inherent degrees of freedom of the implementation space through *heuristics*. This category, which contains most commercial and scientific libraries, is, by nature, extremely sensible to changes in hardware. The whole library may have to be rewritten if the

		General input size		
	Fixed input size	Heuristics based	Online adaptive	Offline adaptive
Example library	IPPgen-like [Intel, 2009a]	MKL-like [Intel, 2009b]	FFTW-like [Frigo, 1999]	ATLAS-like [Whaley et al., 2001]
Example interface	`f_8(X,Y)`	`f(8,X,Y)`	`d = f(8)` `d(X,Y)`	`f(8,X,Y)`
Supports legacy interface	N	Y	N	Y
Adaptation	none	none	online (planner)	offline (setup)
User view				
When size changes	discard	-	replan	-
When platform changes	pray	pray	replan	reinstall
When paradigm changes	discard	discard	discard	discard
Developer view (without generator)				
When size changes	rewrite the library	-	-	-
When platform changes	rewrite the library	retune heuristics	-	-
When paradigm changes	rewrite the library	rewrite the library	rewrite the library	rewrite the library
Developer view (with generator)				
When size changes	regenerate	-	-	-
When platform changes	regenerate	retune heuristics	-	-
When paradigm changes	retune the generator	retune the generator	retune the generator	retune the generator
Generatable by Spiral	[Püschel et al., 2005]	[Voronenko, 2008]	*this dissertation*	*this dissertation*

TABLE 1.1: Different libraries for a hypothetical functionality `f` and their properties. The second row shows the corresponding user interface for a call of `f` with input `X` of size 8 and output `Y`. "Pray" upon a platform change means nothing in the library is changed; hence, the resulting performance may or may not be good. Observe that wrappers can fake the legacy interface for both the fixed input size and the online adaptive library if the sizes of interest are known but they cannot handle arbitrary sizes.

platform substantially changes and even minor architectural modifications may require a new tuning of the heuristics. The recent doctoral thesis of Voronenko developed an extension to Spiral that has enabled the computer generation of such libraries for linear transforms [Voronenko, 2008]. This way, the library developers can update the generator and regenerate the library instead of rewriting the library if major changes are needed. However, this generative approach still has one limitation which is that the automation is complete: the heuristics still have to be designed by hand.

The third column captures *online adaptive libraries* such as FFTW [Frigo and Johnson, 2005] that split the computation into two phases: the creation of a problem descriptor or *plan* (called `d` in the table) and the use of that descriptor to actually perform the computation. The idea there is that a space of different algorithms can be explored during plan creation and that overhead can be compensated over time since users usually perform the desired computations more than once for the same problem size (here: 8). The thesis intends to enable the generation of such libraries which would free the library developers of the heuristics design. To achieve this, we design different search strategies for the planner, from the "traditional" dynamic programming to advanced decision making strategies based on reinforcement learning. These are then implemented in a way that they can be inserted into the libraries generated by us or the transform libraries from [Voronenko, 2008].

The fourth column captures *offline adaptive libraries* such as ATLAS [Whaley and Dongarra, 1998] that adapt to the platform during installation. The difference to online adaptive libraries is that, after setup, the library is readily available to the user who can now use it through the legacy interface. This dissertation presents a mechanism for the computer generation of offline adaptive libraries from online adaptive libraries by creating deterministic decision-trees during the setup of the library.

1.4 Contributions of the Thesis

The main contributions of this thesis include the following:

- A domain-specific mathematical language, called the Operator Language (OL), to symbolically represent numerical algorithms that have data-independent control flows. OL is a superset of SPL [Xiong et al., 2001], the signal processing language at the core of Spiral. We show that OL can describe linear transforms (the domain of SPL), matrix-multiplications and Viterbi decoding [de Mesmay et al., 2010a; Franchetti et al., 2009].

- Compiler transformations that enable the generation of high-performance code directly from OL specifications. This includes optimization passes on intermediate languages and rewriting systems to formally vectorize and parallelize at a high level of abstraction.

- OL extensions to Voronenko's *recursion step closure* which is the key phase in deriving the set of mutually recursive functions necessary to provide efficient general input size libraries for

linear transforms [Voronenko, 2008; Voronenko et al., 2009]. In particular, data reorderings across recursion steps for OL are supported.

- Modular platform-adaptation mechanisms that search the space of alternative algorithms. Notably, we generate offline adaptive (at installation time) and online adaptive (at runtime) libraries. Offline adaptation is achieved using a statistical classifier that generates decision trees at installation time [de Mesmay et al., 2010b]. Online adaptation is performed by runtime planning using several search modules that include dynamic programming and a novel algorithm based on Monte-Carlo tree search [de Mesmay et al., 2009].

- A generator prototype that extends previous work and implements the above contributions to allow "push-button" adaptive library generation for different domains.

Evaluation. We evaluate the results of this thesis on two key aspects, the performance of the generated libraries and the versatility of the code generator. Performance is assessed with comparisons against relevant academic and industrial competitors on commodity platforms. Versatility is shown by the range of different libraries that can be generated. In particular, we demonstrate the following:

- A library generator for different domains: linear transforms, basic linear algebra, and convolutional decoding. For linear transforms, the general input size library generation problem was solved by [Voronenko, 2008]. In this case, our work makes it possible to render these libraries adaptive (see below).

- Experimental results that show that the generated libraries have a performance that is competitive with existing academic and industrial competitors.

- A system that can generate different types of libraries: collections of fixed input size problem (prior work), heuristic based libraries (prior work), planner-based adaptive libraries or adaptive libraries that infer a decision tree at installation time.

- A back-end that can target different languages: Besides C++, we will produce C code and Java code.

- A customization mechanism that allows interesting tradeoffs, notably trading library code size for performance or for problem coverage.

- A compiler that can optimize for different targets most notably scalar, vectorized, and parallel code.

- Different planners that provide different online search strategies, including dynamic programming, Monte-Carlo, and a novel technique based on reinforcement learning.

- A mechanism that automatically generates customized heuristics in order to generate offline adaptive libraries.

1.5 Related work

The motivation for program generation and automatic platform adaptation arises from the inability of optimizing compilers to achieve the performance levels of hand-optimization (see, for example, Figure 1.1). This is evidenced by the fact that most platform vendors offer specific *performance* libraries: Intel (Math Kernel Library & Integrated Performance Primitives), AMD (Core Math Library & Framewave), IBM (Engineering and Scientific Subroutine Library), and others.

The core reason for this shortcoming of compilers is that the increasing complexity of computing platforms prevents the development of accurate platform models that are needed to assess the profitability of compiler transformations and hence choose among the large set of otherwise legal optimizations. Two different avenues of research related to our work have started to address these problems: iterative compilation and automatic performance tuning / program generation.

1.5.1 Iterative Compilation

Several researchers have proposed *iterative compilation* which places the compiler at the center of a feedback-driven optimization loop [Bodin et al., 1998; Fursin et al., 2005b; Kisuki et al., 2000a]. The idea is attractive but, in practice, going beyond the research demonstrator is difficult because there are actually many compiler parameters to evaluate and even more if we start looking at sequences of transformations. As of 2010, several propositions have been made to render iterative compilation practical and we distinguish four major research directions:

- **Machine learning** techniques can be used to model the search space and therefore reduce the number of actual evaluations needed [Cavazos and O'Boyle, 2006; Cooper et al., 2002; Leather et al., 2009; Monsifrot et al., 2002; Stephenson and Amarasinghe, 2005].
- **Sampling** methodologies propose to reduce the duration of each run by evaluating only a carefully chosen subset of the computation phases of the benchmark program, therefore scanning more points within the same amount of time [Fursin et al., 2005a; Lau et al., 2005; Sherwood et al., 2002].
- **Using developer hints** can direct the compiler towards the right transformations and therefore reduce the number of points to evaluate. This is done by providing a framework to help developers express their own transformations [Barthou et al., 2007; Beckmann et al., 2004; Cohen et al., 2006; Donadio et al., 2006].
- **Structuring** the transformation space can directly reduce the number of points to evaluated. Typically, structuration is a "grassroot" approach where one focuses on a specific domain and

gradually grows it towards more general cases. The polyhedral model framework is possibly the most general approach in that it can deal with a certain type of loops and can describe entire compositions of optimizing transformations for them [Ahmed et al., 2000; Bastoul, 2004; Cohen et al., 2005].

1.5.2 Automatic Performance Tuning and Program Generation

A different approach (even though in some aspects related to iterative compilation) to overcoming compiler shortcomings has been the development of automatic performance tuning and program generation techniques. We overview some prominent examples starting with Spiral which underlies this thesis.

Spiral. Our work builds on Spiral, a library generator for linear transforms [Püschel et al., 2005]. Representing linear transforms using a declarative domain-specific language [Xiong et al., 2001] based on Van Loan's formalism [Van Loan, 1992], Spiral formally manipulates the structure of the algorithms to map them to various targets: parallel shared memory code [Franchetti et al., 2006b], distributed message passing code [Bonelli et al., 2006], vectorized code [Franchetti et al., 2006c], and hardware descriptions [Milder et al., 2008]. The associated search space has been modeled using machine learning techniques [Singer and Veloso, 2001, 2002]. Spiral has long been limited to generating collections of fixed input size functionalities but the recent doctoral thesis of Voronenko [Voronenko, 2008] enables the generation of general input size libraries. Such libraries achieve very high performance [Voronenko et al., 2009] but, since they are generated *without feedback*, the degrees of freedom have to be determined with properly chosen heuristics. In this thesis, we extend Spiral in two orthogonal directions: 1) the computer generation of non-transform functionalities, notably matrix-multiplication libraries and Viterbi decoders, and 2) the computer generation of online and offline adaptive general input size libraries.

ATLAS. ATLAS is an offline adaptive library providing automatically tuned basic linear algebra operations (BLAS) [Whaley and Dongarra, 1998; Whaley et al., 2001]. The different implementations of the matrix multiplication operation arise from different combinations of important parameters such as the block sizes or the unrolling factor. Interesting parameter combinations are tested on the target platform at installation time and the best ones are selected for inclusion in the final library. This approach has been very successful for years but lacks expressivity to represent more advanced transformations such as vectorization or packing that are required on new commodity architectures. As a result, ATLAS is now mainly used as an infrastructure around user-contributed kernels [Whaley, 2001]. In comparison, this thesis targets the generation of vectorized GEMM libraries. Considering that a high-performance BLAS 3 can be built from a high-performance GEMM [Kågström et al., 1998], this thesis could serve as a basis for an offline adaptive BLAS library.

FFTW and UHFFT. FFTW and UHFFT are online adaptive libraries for linear transforms [Frigo and Johnson, 2005; Mirković, 2001]. The core idea of both libraries is to provide a set

of different recursions for the computation of the DFT. At runtime, the libraries use a so-called *planner* to select the best recursions with dynamic programming. Using the work from [Voronenko, 2008] as a basis to generate linear transform recursive libraries, this thesis builds the infrastructure to add a runtime planner to our generated libraries, making them generated online adaptive libraries. Multiple search mechanisms are proposed to drive the planner.

FLAME. FLAME is a library generator for advanced dense linear algebra algorithms (e.g., LU Factorization) [Gunnels et al., 2001; van de Geijn and Quintana-Ortí, 2008]. Algorithms are described using blocked matrix equations and transformed all the way to source code implementations with key steps being the discovery of possible loop invariants and the parallelization using task queues [Bientinesi, 2006]. Note that FLAME relies on an existing high-performance BLAS library.

TCE. TCE is a library generator for tensor contractions that are needed in quantum chemistry [Baumgartner et al., 2005]. The input to the system is a tensor expression and main optimization steps include operation count minimization, locality optimization, and problem partition among different processors [Gao et al., 2005]. It also relies on an external performance library to supply it with BLAS functionality.

1.6 Organization of the Thesis

In the remainder of this document, we will explain the mechanisms at work inside the library generator. For the explanation, we will mostly focus on a single functionality, the matrix-matrix multiplication but other functionalities are in principle possible.

In Chapter 2, we present the domain-specific language OL that we use to describe algorithms and we show how to automatically generate imperative code from OL. In Chapter 3, we derive *library cores* from our algorithms: undetermined optimized recursive libraries that naturally capture a spectrum of different implementations. We then study how to determine the degrees of freedom inside these library cores using online and offline exploration mechanisms in Chapter 4. Finally, we present results in Chapter 5 before concluding.

CHAPTER 2

Representing Algorithms

This chapter introduces the Operator Language (OL), a mathematical domain-specific language designed to describe structured algorithms for data-independent functions. It is used as the input language of our library generator, which generates an optimized library through a sequence of transformations.

We start by providing background on the Signal Processing Language (SPL, [Xiong et al., 2001]), which is a subset of OL, and describes algorithms for linear transforms and underlies Spiral [Püschel et al., 2005]. We then extend SPL beyond the transform domain by introducing OL primitives needed for matrix-matrix multiplication. Finally, we show how OL can be used to model subsets of more involved functions using a Viterbi decoder as example.

2.1 Signal Processing Language (SPL)

2.1.1 Linear Transforms

Although often presented in the literature as summations, linear transforms can be equivalently defined as a matrix-vector products

$$y = Mx,$$

where x and y are, respectively, the input and the output vectors, and M is a *fixed* matrix that, by slight abuse of notation, we also call the transform.

Many transforms are important for signal processing and data compression, notably:

- the Discrete Fourier transform (DFT), widely used for spectral analysis [Tolimieri et al., 1997; Van Loan, 1992] and its real-valued counterpart, the real DFT (RDFT) [Bergland, 1968; Voronenko and Püschel, 2009];

$$\mathbf{DFT}_n = \left[e^{-2\pi jk\ell/n}\right]_{0\le k,\ell<n}, \quad j^2=-1 \qquad \mathbf{RDFT}_n = \left[\begin{cases} \cos\frac{2\pi k\ell}{n}, & \text{if } k\le\lfloor\frac{n}{2}\rfloor \\ -\sin\frac{2\pi k\ell}{n}, & \text{else}\end{cases}\right]_{0\le k,\ell<n}$$

$$\mathbf{WHT}_n = \begin{bmatrix}\mathbf{WHT}_{n/2} & \mathbf{WHT}_{n/2}\\ \mathbf{WHT}_{n/2} & -\mathbf{WHT}_{n/2}\end{bmatrix}, \text{ with} \qquad \mathbf{WHT}_2 = \begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}$$

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

$$\mathbf{DCT\text{-}3}_n = \mathbf{DCT\text{-}2}_n^{\mathrm{T}} = \left[\cos\frac{(2k+1)\ell\pi}{2n}\right]_{0\le k,\ell<n} \qquad \mathbf{DCT\text{-}4}_n = \left[\cos\frac{(2k+1)(2\ell+1)\pi}{4n}\right]_{0\le k,\ell<n}$$

$$\mathbf{MDCT}_n = \left[\cos\frac{(2k+1)(2\ell+1+n)\pi}{4n}\right]_{\substack{0\le k<n\\ 0\le\ell<2n}}$$

TABLE 2.1: Definition of important linear transforms.

- the Walsh-Hadamard transform (WHT), used for instance inside the high-definition video standards Blu-Ray and HD-DVD [Beauchamp, 1984];
- the four main types of discrete cosine transforms (DCTs), used for instance in the JPEG image format and MPEG video format [Rao and Yip, 1990] and [Püschel and Moura, 2008];
- the modified discrete cosine transform (MDCT), used in numerous modern lossy audio formats such as MP3, AC-3 or WMA [Malvar, 1992].

These transforms are all defined for every input size n, except the WHT which only exists for two-powers $n = 2^k$. Further, all transforms are $n \times n$ square matrices with the exception of the MDCT. Their general form is presented in Table 2.1. Note that many more transforms exist, but are possibly less relevant.

2.1.2 Fast Transform Algorithms and SPL

$\Theta(n^2)$ arithmetic operations are required for a generic dense matrix-vector product [Burgisser et al., 1997] but the particular structure of many transforms, including those in Table 2.1, allows their computation with $O(n \log n)$ operations.

Fast transform algorithms can be represented as factorizations of the typically dense transform into a product of sparse structured matrices. We provide a simple example with the DCT-2 of size 4, defined by:

$$\mathbf{DCT\text{-}2}_4 = \begin{bmatrix} 1 & 1 & 1 & 1\\ \cos\frac{1\pi}{8} & \cos\frac{3\pi}{8} & \cos\frac{5\pi}{8} & \cos\frac{7\pi}{8}\\ \cos\frac{2\pi}{8} & \cos\frac{6\pi}{8} & \cos\frac{6\pi}{8} & \cos\frac{2\pi}{8}\\ \cos\frac{3\pi}{8} & \cos\frac{7\pi}{8} & \cos\frac{1\pi}{8} & \cos\frac{5\pi}{8}\end{bmatrix}. \tag{2.1}$$

Computing $y = \mathbf{DCT\text{-}2}_4\, x$ naïvely requires 12 additions and 12 multiplications[1]. Yet, it is easy to verify (but not easy to derive) that the following factorization holds:

$$\mathbf{DCT\text{-}2}_4 = \begin{bmatrix} 1 & . & . & . \\ . & . & 1 & . \\ . & 1 & . & . \\ . & . & . & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & . & . \\ \cos\frac{2\pi}{8} & \cos\frac{6\pi}{8} & . & . \\ . & . & \cos\frac{1\pi}{8} & \cos\frac{3\pi}{8} \\ . & . & \cos\frac{3\pi}{8} & \cos\frac{7\pi}{8} \end{bmatrix} \begin{bmatrix} 1 & . & . & 1 \\ . & 1 & 1 & . \\ 1 & . & . & -1 \\ . & 1 & -1 & . \end{bmatrix} \tag{2.2}$$

Using Equation 2.2, $y = \mathbf{DCT\text{-}2}_4\, x$ can be computed in 3 successive matrix-vector products and reduces the number of operations to 8 additions and 6 multiplications.

The Signal Processing Language (SPL) extends the Kronecker product formalism introduced by [Johnson et al., 1990a; Van Loan, 1992]. It was first presented in [Xiong et al., 2001] and further extended in [Püschel et al., 2005]. It is a domain-specific language that captures transform algorithms such as Equation 2.2 using *formulas* that are constructed from *transforms*, *matrices*, and *matrix constructs*.

Matrices. Basic matrices are the building blocks of SPL. They are described using standard mathematical notations. Examples include the $n \times n$ identity I_n and the $n \times n$ "flip" matrix J_n:

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

Important permutations are also given symbols such as the *stride permutation* L^n_m, defined by its underlying permutation

$$L^n_m : jk + i \mapsto im + j \text{ for } 0 \le i < k, 0 \le j < m \text{ with } n = mk. \tag{2.3}$$

Matrix constructs. The strength of SPL lies in its ability to capture the structure of the transform algorithms. Matrix constructs are used for this purpose and the most important ones are the matrix product, the direct sum, and the tensor product.

We have already used the matrix product $\mathrm{A} \cdot \mathrm{B} = \mathrm{A}\,\mathrm{B}$ in the $\mathbf{DCT\text{-}2}_4$ example (Equation 2.2). We note that, in SPL, a matrix product is not supposed to be computed using a matrix-matrix computation but is viewed as a two stage matrix-vector multiplication algorithm: $t = \mathrm{B}\, x, y = \mathrm{A}\, t$. The iterative product $\Pi_{i=0}^{n-1} \mathrm{A}_i$ extends the matrix product and expresses that the computation be done in n successive steps.

[1] We assume that multiplications by 1 and -1 do not count.

The direct sum $\oplus$ composes two matrices into a block diagonal matrix:

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

The most important matrix construct in SPL is the tensor (or Kronecker) product $\otimes$, defined as:

$$\mathrm{A} \otimes \mathrm{B} = [a_{k,l}\,\mathrm{B}], \quad \text{where } \mathrm{A} = [a_{k,l}]. \tag{2.4}$$

Two important special cases arise when A or B is the identity matrix. If the identity appears on the left side, then the tensor product degenerates into a direct sum that is a simple block diagonal matrix:

$$I_n \otimes \mathrm{B} = \underbrace{\mathrm{B} \oplus \cdots \oplus \mathrm{B}}_{n \text{ times}} = \begin{bmatrix} \mathrm{B} & & \\ & \ddots & \\ & & \mathrm{B} \end{bmatrix}.$$

As we will see later $I_n \otimes \mathrm{B}$ can be interpreted as a computation that is perfectly parallelizable on a machine using n processors. If the identity appears on the right side, as in $\mathrm{A} \otimes I_n$, the tensor product also presents a very particular structure where each entry is replicated n times, for instance,

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \otimes I_2 = \begin{bmatrix} a & & b & \\ & a & & b \\ c & & d & \\ & c & & d \end{bmatrix}. \tag{2.5}$$

We will see later that computations of this type can be perfectly mapped to machine supporting SIMD vector extensions.

Finally, we introduce the indexed tensor product for n matrices B_j, $j = 0 \ldots n-1$, of the same size:

$$I_n \otimes_j \mathrm{B}_j = \mathrm{B}_0 \oplus \cdots \oplus \mathrm{B}_{n-1} = \begin{bmatrix} \mathrm{B}_0 & & \\ & \ddots & \\ & & \mathrm{B}_{n-1} \end{bmatrix}.$$

$\mathrm{B}_j \otimes_j I_n$ is defined analogously.

Formulas and breakdown rules. SPL expressions are called *formulas* and their grammar is sketched in Table 2.2. It is possible for two different formulas to be mathematically equal in which case each formula can be read as an algorithm and both algorithms ultimately perform the same computation. For instance, the definition for $\mathbf{DCT\text{-}2}_4$ (Equation 2.1) presents a trivial strategy to

$$
\begin{array}{rcll}
\langle \text{formula} \rangle & ::= & \langle \text{matrix} \rangle \mid \langle \text{transform} \rangle \mid & \\
 & & \langle \text{formula} \rangle \langle \text{formula} \rangle \mid & \textit{product} \\
 & & \langle \text{formula} \rangle \oplus \langle \text{formula} \rangle \mid & \textit{direct sum} \\
 & & \langle \text{formula} \rangle \otimes \langle \text{formula} \rangle & \textit{tensor product} \\
\langle \text{matrix} \rangle & ::= & \left[\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right] \mid I_n \mid J_n \mid L^n_k \mid \ldots & \\
\langle \text{transform} \rangle & ::= & \mathbf{DFT}_n \mid \mathbf{RDFT}_n \mid \mathbf{DCT\text{-}2}_n \mid \ldots &
\end{array}
$$

TABLE 2.2: SPL grammar [Püschel et al., 2005] in Backus-Naur form [Harrison, 1978] showing that an SPL formula is either a matrix, a transform, or is constructed using matrix constructs. a, b, c, d, n and k are integers.

compute it, whereas the factorization Equation 2.2 describes a fast algorithm for it.

A *breakdown rule* is a formula equality involving a transform of arbitrary size such as $\mathbf{DCT\text{-}2}_n$ (as opposed to fixed size such as $\mathbf{DCT\text{-}2}_4$). It captures a divide-and-conquer algorithm for the transform by breaking it down into several other transforms.

For example, the breakdown rule that generalizes Equation 2.2 is

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

Note that we write "$\rightarrow$" instead of "$=$" to denote that it is a rule. The most iconic example, and also a recurring character of this dissertation, is the rule corresponding to the well known general-radix Cooley-Tukey Fast Fourier transform (FFT), the first special case of which was introduced in [Cooley and Tukey, 1965]:

$$\mathbf{DFT}_n \rightarrow (\mathbf{DFT}_k \otimes I_m) T^n_m (I_k \otimes \mathbf{DFT}_m) L^n_k, \quad n = km. \tag{2.6}$$

The above equality shows that $\mathbf{DFT}_n$ can be computed in four successive steps, namely, a permutation, k $\mathbf{DFT}_m$s, a scaling[2] and m $\mathbf{DFT}_k$s.

The literature contains many dozens of such fast algorithms, for instance [Elliott and Rao, 1983; Nussbaumer, 1982; Püschel and Moura, 2008; Tolimieri et al., 1997; Voronenko and Püschel, 2009]. Table 2.3 presents a few of these breakdown rules for $\mathbf{DFT}_n$ and $\mathbf{DCT\text{-}2}_n$, including the famous prime-factor [Good, 1958] (Equation 2.8), Rader [Rader, 1968] (Equation 2.9), and Bluestein [Bluestein, 1970] FFTs.

2.1.3 Insights from Spiral

Spiral [Püschel et al., 2005; Voronenko et al., 2009] automatically generates high-performance libraries for linear transforms. The generated programs are automatically tuned to a given target

[2] T^n_m is a diagonal matrix whose precise entries are irrelevant here.

$$\mathbf{DFT}_n \longrightarrow (\mathbf{DFT}_k \otimes I_m) T^n_m (I_k \otimes \mathbf{DFT}_m) L^n_k, \quad n{=}km \tag{2.7}$$

$$\mathbf{DFT}_n \longrightarrow V^{-1}_{m,k} (\mathbf{DFT}_k \otimes I_m)(I_k \otimes \mathbf{DFT}_m) V_{m,k}, \quad n{=}km,\ \gcd(k,m){=}1. \tag{2.8}$$

$$\mathbf{DFT}_n \longrightarrow W^{-1}_n (I_1 \oplus \mathbf{DFT}_{n-1}) E_n (I_1 \oplus \mathbf{DFT}_{n-1}) W_n, \quad n \text{ prime} \tag{2.9}$$

$$\mathbf{DFT}_n \longrightarrow B^{\mathrm{T}}_{n,m} D_m \, \mathbf{DFT}_m \, D'_m \, \mathbf{DFT}_m \, D''_m B_{n,m}, \quad m{\geq}2n{-}1 \tag{2.10}$$

$$\mathbf{DFT}_n \longrightarrow P^{\mathrm{T}}_{k/2,2m} \left(\mathbf{DFT}_{2m} \oplus \left(I_{k/2-1} \otimes_i C_{2m} \, \mathbf{rDFT}_{2m}((i+1)/k) \right) \right) \left(\mathbf{RDFT}'_k \otimes I_m \right) \tag{2.11}$$

$$\mathbf{DCT\text{-}2}_n \longrightarrow L^n_{n/2} (\mathbf{DCT\text{-}2}_{n/2} \oplus \mathbf{DCT\text{-}4}_{n/2}) \begin{bmatrix} I_{n/2} & J_{n/2} \\ I_{n/2} & -J_{n/2} \end{bmatrix} \tag{2.12}$$

$$\mathbf{DCT\text{-}2}_n \longrightarrow S_n \, \mathbf{RDFT}_n \, K^n_2 \tag{2.13}$$

$$\mathbf{DCT\text{-}2}_n \longrightarrow P^{\mathrm{T}}_{k/2,2m} \left(\mathbf{DCT\text{-}2}_{2m} \, K^{2m}_2 \oplus \left(I_{k/2-1} \otimes N_{2m} \, \mathbf{RDFT\text{-}3}^{\mathrm{T}}_{2m} \right) \right) G_n (L^{n/2}_{k/2} \otimes I_2)(I_m \otimes \mathbf{RDFT}'_k) Q_{m/2,k} \tag{2.14}$$

TABLE 2.3: Examples of SPL breakdown rules for $\mathbf{DFT}_n$ and $\mathbf{DCT\text{-}2}_n$. Above, Q, P, K, V, W are various permutation matrices, D are diagonal matrices, and B, C, E, G, N, S are other sparse matrices, whose precise form is irrelevant. Arrows are used in place of equalities to emphasize that the left-hand side should be replaced by the right-hand side.

platform using both a rewriting system that structurally optimizes algorithms and a feedback directed search in the space of alternative algorithms.

More precisely, Spiral can be viewed as an iterative compiler for SPL[3] as shown in Figure 2.1. The input to Spiral is a transform of fixed size such as $\mathbf{DCT\text{-}2}_4$ (that we used earlier) or $\mathbf{DFT}_{1024}$; the output is an optimized C program that computes the transform. Like other iterative compilation frameworks [Fursin et al., 2005b; Kisuki et al., 2000b], Spiral replaces traditional static cost models for optimization choices by providing an *evaluation* and a *search* module that work in unison: the evaluation feeds back timing information that are exploited by the search in order to drive the current optimization strategy towards the best implementations.

One of the key characteristics of the project is that it is limited to a specific domain, linear signal transforms. Because generality has been forfeited, the level of abstraction of the source language of the compiler, SPL, is much higher than the one of traditional languages like C. This has two interesting consequences: first, entire *algorithms*, not just loop transformations, can be captured and searched upon and second, optimizations, during the *implementation* stage, can be made more general and more powerful due to the limited variety of inputs coming into the compiler.

In this dissertation, our first goal is to enlarge the source language of Spiral in a way that different primitives, non-transforms, can be captured. The major difficulty in doing this is that each layer of the system is tightly integrated: adding a primitive to the description language requires to propagate

[3] As we will see later, this affirmation is only partially true for the later versions of Spiral

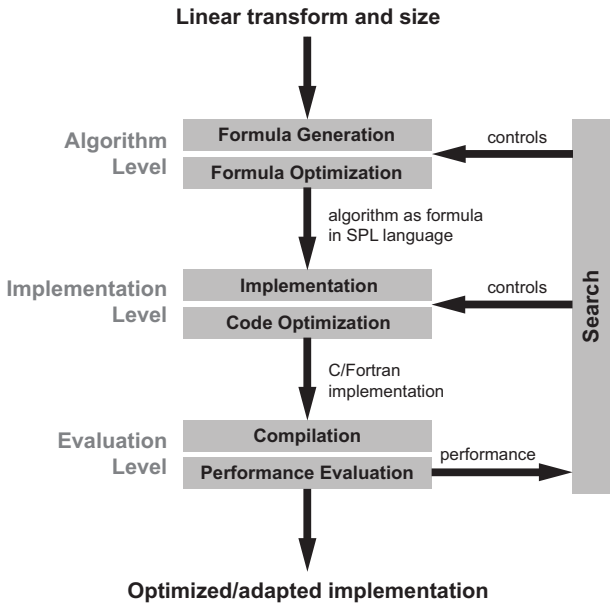


FIGURE 2.1: The architecture of the early versions of Spiral.

changes throughout the compiler. We present our candidate language, the Operator Language, in the next section.

2.2 Operator Language (OL)

The Operator Language (OL) is a domain-specific language that extends SPL. Just as SPL, it aims to express algorithms at a high abstraction level and to be mathematical and declarative in that it describes the structure and the index space of the data layout of computations without specifying how to perform them. The language to date is restricted to express computations that are data independent and that possess a suitable structure to be handled efficiently.

This section starts by presenting the building blocks of the language followed by an overview of how to capture a simple dense matrix-matrix multiplication using the language.

2.2.1 Elements of the Language

OL is a language of mathematical nature that is a superset of SPL (Subsection 2.1.2). Its main building blocks are *operators*, combined into *formulas* by *higher-order operators*.

Operators. An operator is a function that consumes and produces a set of vectors. An operator of arity (r, s) consumes r vectors and produces s vectors. An operator can be (multi)linear or not. Linear operators of arity $(1, 1)$ are precisely linear transforms, i.e., mappings $x \mapsto Mx$, where M is a fixed matrix.

Name	Definition
Linear, arity (1,1)	
Identity	$I_n : \mathbb{C}^n \to \mathbb{C}^n;\ x \mapsto x$
Transposition of an $m \times n$ matrix	$L^{mn}_m : \mathbb{C}^{mn} \to \mathbb{C}^{mn};\ A \mapsto A^T$
Matrix $M \in \mathbb{C}^{m\times n}$	$M : \mathbb{C}^n \to \mathbb{C}^m; x \mapsto Mx$
Multilinear, arity (2,1)	
Point-wise product	$P_n : \mathbb{C}^n \times \mathbb{C}^n \to \mathbb{C}^n;\ ((x_i),(y_i)) \mapsto (x_i y_i)$
Scalar product	$R_n : \mathbb{C}^n \times \mathbb{C}^n \to \mathbb{C};\ ((x_i),(y_i)) \mapsto \sum(x_i y_i)$
Kronecker product	$K_{m\times n} : \mathbb{C}^m \times \mathbb{C}^n \to \mathbb{C}^{mn};\ ((x_i), y)) \mapsto (x_i y)$

TABLE 2.4: Definition of basic OL operators. The operators are here assumed to operate on complex numbers but extension to other base sets is straightforward. Boldface fonts represent vectors or matrices linearized in memory. A vector is sometimes written as $x = (x_i)$ to identify the components.

Matrices are viewed as vectors stored linearized in memory in row major order. For example, the operator that transposes an $m \times n$ matrix[4], denoted with L^{mn}_n, is of arity $(1, 1)$. Table 2.4 defines a set of basic operators that we use.

Functionalities. A functionality is an operator for which we want to generate fast code. For instance, the *matrix-matrix multiplication* $\mathbf{MMM}_{m,k,n}$ is an operator that consumes two matrices and produces one[5]:

$$\mathbf{MMM}_{m,k,n} : \mathbb{R}^{mk} \times \mathbb{R}^{kn} \to \mathbb{R}^{mn};\ (A, B) \mapsto AB.$$

The *discrete Fourier transform* $\mathbf{DFT}_n$ is another functionality example. It is a linear operator of arity $(1, 1)$ that performs the following matrix-vector product:

$$\mathbf{DFT}_n : \mathbb{C}^n \to \mathbb{C}^n;\ x \mapsto [e^{-2\pi ikl/n}]_{0\le k,l<n} x.$$

In this document, functionalities are bold-faced.

Higher-order operators. Higher-order operators are functions on operators. A simple example is the *composition*, denoted in standard infix notation by $\circ$. For instance, using the P_{mn} operator defined in Table 2.4,

$$L^{mn}_n \circ P_{mn}$$

is the arity $(2, 1)$ operator that first multiplies point-wise two matrices of size $m \times n$, and then transposes the result.

[4]The transposition operator is functionality equivalent to the *stride* permutation introduced in Equation 2.3 and is sometimes referred to as a *corner turn* in the literature.

[5]Note that this definition is purely mathematical and thus differs from commonly used interfaces such as the DGEMM [Dongarra et al., 1990]. This will be taken into account later in this thesis.

The *cross product* of two operators applies the first operator to the first input set and the second operator to the second input set, and then combines the outputs. For example,

$$L_n^{mn} \times P_{mn}$$

is the arity $(3, 2)$ operator that transposes its first argument and multiplies the second and third argument pointwise, producing two output vectors.

The most important higher order operator in this language is the *tensor product*. For linear operators A, B of arity (1,1) (i.e., matrices), the new tensor product corresponds to the tensor or Kronecker product of matrices defined in Equation 2.4.

As we have seen in Table 2.3, the Kronecker product is very useful for concisely describing transform algorithms. Its usefulness resides in its ability to captures loop structures, data independence, and parallelism. Therefore, it really is a key construct in SPL and we now formally extend its definition to more general operators, focusing on the case of two operators with arity (2,1); the generalization is straightforward.

Let $\mathcal{A} : \mathbb{C}^p \times \mathbb{C}^q \to \mathbb{C}^r$ be a *multi-linear* operator and let $\mathrm{B} : \mathbb{C}^m \times \mathbb{C}^n \to \mathbb{C}^k$ be *any* operator. We stress that $\mathcal{A}$, being multi-linear, is not interchangeable with B by using a calligraphic font for it. We denote the ith canonical basis vector of $\mathbb{C}^n$ with $\mathbf{e}_i^n$. Then

$$(\mathcal{A} \otimes \mathrm{B})(x, y) = \sum_{i=0}^{p-1} \sum_{j=0}^{q-1} \mathcal{A}(\mathbf{e}_i^p, \mathbf{e}_j^q) \otimes \mathrm{B}((\mathbf{e}_i^{p^T} \otimes I_m)x, (\mathbf{e}_j^{q^T} \otimes I_n)y),$$

$$(\mathrm{B} \otimes \mathcal{A})(x, y) = \sum_{i=0}^{p-1} \sum_{j=0}^{q-1} \mathrm{B}((I_m \otimes \mathbf{e}_i^{p^T})x, (I_n \otimes \mathbf{e}_j^{q^T})y) \otimes \mathcal{A}(\mathbf{e}_i^p, \mathbf{e}_j^q).$$

Intuitively, $\mathcal{A}$, whether on the left or on the right of the tensor product, describes how to operate on the chunks of data produced by B. If it is on the left, $\mathcal{A}$ describes the coarse structure of the computation. If it is on the right, $\mathcal{A}$ describes the internal structure (as in Equation 2.5). Obviously, if B is also multilinear, then both properties hold simultaneously.

We give a few examples to better illustrate the tensor product. First, let $\mathcal{A} = P_2$ be the pointwise product of two vectors of length 2. Then, the relationship between P_2 and $P_2 \otimes \mathrm{B}$ (arbitrary B) is as follows (the superscripts U and L denote the upper and lower halves of a vector):

$$\begin{pmatrix} x_0 \\ x_1 \end{pmatrix} \times \begin{pmatrix} y_0 \\ y_1 \end{pmatrix} \xrightarrow{P_2} \begin{pmatrix} x_0 \cdot y_0 \\ x_1 \cdot y_1 \end{pmatrix}, \qquad \begin{pmatrix} x^U \\ x^L \end{pmatrix} \times \begin{pmatrix} y^U \\ y^L \end{pmatrix} \xrightarrow{P_2 \otimes \mathrm{B}} \begin{pmatrix} \mathrm{B}(x^U, y^U) \\ \mathrm{B}(x^L, y^L) \end{pmatrix}. \quad (2.15)$$

Next, we choose $\mathcal{A} = R_2$, the scalar product of vector of length 2:

$$\begin{pmatrix} x_0 \\ \hline x_1 \end{pmatrix} \times \begin{pmatrix} y_0 \\ \hline y_1 \end{pmatrix} \xrightarrow{R_2} \sum_{i \in \{0,1\}} x_i \cdot y_i, \qquad \begin{pmatrix} x^U \\ \hline x^L \end{pmatrix} \times \begin{pmatrix} y^U \\ \hline y^L \end{pmatrix} \xrightarrow{R_2 \otimes \mathrm{B}} \sum_{i \in \{U,L\}} \mathrm{B}(x^i, y^i).$$

Finally, we choose $\mathcal{A} = K_{2\times 2}$, the Kronecker product $K_{2\times 2}$ (now viewed as operator of arity $(2, 1)$ on vectors, see Table 2.4, not viewed as higher order operator):

$$\begin{pmatrix} x_0 \\ \hline x^1 \end{pmatrix} \times \begin{pmatrix} y_0 \\ \hline y_1 \end{pmatrix} \xrightarrow{K_{2\times 2}} \begin{pmatrix} x_0 \cdot y_0 \\ \hline x_0 \cdot y_1 \\ \hline x^1 \cdot y_0 \\ \hline x^1 \cdot y_1 \end{pmatrix}, \qquad \begin{pmatrix} x^U \\ \hline x^L \end{pmatrix} \times \begin{pmatrix} y^U \\ \hline y^L \end{pmatrix} \xrightarrow{K_{2\times 2} \otimes \mathrm{B}} \begin{pmatrix} \mathrm{B}(x^U, y^U) \\ \hline \mathrm{B}(x^U, y^L) \\ \hline \mathrm{B}(x^L, y^U) \\ \hline \mathrm{B}(x^L, y^L) \end{pmatrix}.$$

In all three cases, the multilinear part $\mathcal{A}$ of the tensor product describes how blocks are arranged and B prescribes the local operations to perform on the blocks. Comparing these three examples, $\mathcal{A} = P$ yields a tensor product in which only corresponding parts of the input vectors are computed on and results are juxtaposed, $\mathcal{A} = R$ does essentially the same computation but results are accumulated and $\mathcal{A} = K$ yields a tensor product in which *all* combinations are computed on and stored.

2.2.2 Matrix-Matrix Multiplication

This subsection outlines how OL can be used by developing a whole *space* of alternative implementations for a simple example functionality: the dense matrix-matrix multiplication. This is achieved by plugging divide-and-conquer *algorithms* into each other and terminating the recursion using *base cases*.

Breakdown rules. We express recursive algorithms for functionalities as OL equations and call them *breakdown rules*. As example, we consider a blocked matrix multiplication. While it does not improve the arithmetic cost over a naïve implementation, blocking increases reuse and therefore can improve performance [Whaley et al., 2001; Yotov et al., 2005]. Note that we do not draw a firm line between cache and register blocking since this difference is related to unrolling which is only performed later in our framework. We start with blocking along one dimension.

Figure 2.2a shows a picture of a horizontally blocked matrix. Each part of the result C is produced by multiplying the corresponding part of A by the whole matrix B. In OL, this computation is expressed by a tensor product with a Kronecker product:

$$\mathbf{MMM}_{m,k,n} \to K_{m/m_b \times 1} \otimes \mathbf{MMM}_{m_b,k,n}\,. \tag{2.16}$$

Note that the number of blocks m/m_b is a degree of freedom under the constraint that m is divisible by m_b[6]; in the picture, m/m_b is equal to 2 (white block and black block).

[6] In general, blocks may be of different sizes but this is not easily expressible with the tensor product.

(a) Horizontal blocking (b) Vertical blocking (c) Depth blocking

FIGURE 2.2: Blocking matrix multiplication along each one of the three dimensions. For the horizontal and vertical blocking, the white (black) part of the result is computed by multiplying the white (black) part of the blocked input with the other, gray, input. For the depth blocking, the result is computed by multiplying both white parts and both black parts and adding the results.

Figure 2.2b shows a picture of a vertically tiled matrix. The result is computed by multiplying parts of the matrix B with A so the underlying tensor product again uses a Kronecker product. However, since matrices are linearized in row-major order, we now need two additional stages: a pre-processing stage where the parts of B are de-interleaved and a post-processing stage where the parts of C are re-interleaved[7]:

$$\mathbf{MMM}_{m,k,n} \rightarrow \big(I_m \otimes L^n_{n/n_b}\big) \circ \big(\mathbf{MMM}_{m,k,n_b} \otimes K_{1\times n/n_b}\big) \circ \big(I_{km} \times (I_k \otimes L^n_{n_b})\big). \tag{2.17}$$

Finally, Figure 2.2c shows a picture of a matrix tiled in the "depth." This time, parts of one input corresponds to parts of the other input but all results are added together. Therefore, the corresponding tensor product is not done with a Kronecker product but with a scalar product:

$$\mathbf{MMM}_{m,k,n} \rightarrow \big(R_{k/k_b} \otimes \mathbf{MMM}_{m,k_b,n}\big) \circ \big((L^{mk/k_b}_{k/k_b} \otimes I_{k_b}) \times I_{kn}\big). \tag{2.18}$$

The three blocking rules we just described can actually be combined into a single rule with three degrees of freedom:

$$\begin{aligned}\mathbf{MMM}_{m,k,n} \rightarrow (I_{m/m_b} \otimes L^{m_b n/n_b}_{m_b} \otimes I_{n_b}) \circ (\mathbf{MMM}_{m/m_b,k/k_b,n/n_b} \otimes \mathbf{MMM}_{m_b,k_b,n_b}) \\ \circ((I_{m/m_b} \otimes L^{m_b k/k_b}_{k/k_b} \otimes I_{k_b}) \times (I_{k/k_b} \otimes L^{k_b n/n_b}_{n/n_b} \otimes I_{n_b})).\end{aligned} \tag{2.19}$$

The above rule captures the well-known mathematical fact that a matrix multiplication of size (m, k, n) can be done by repeatedly using block multiplications of size (m_b, k_b, n_b). Interestingly, a blocked matrix multiplication can be described as a tensor product in which both the coarse and fine structures are themselves matrix multiplications[8]. This fact was already observed by [Johnson et al., 1990b], albeit expressed somewhat differently.

Base cases. Recursive algorithms need to be terminated by base cases that correspond to sizes

[7] As we will explain later, the three stages here are fused during the loop merging optimization, so three OL stages do not necessary imply three different passes through the data.

[8] Observe that any one of the matrix multiplications can actually be chosen as the coarse structure since both operators are multilinear.

```
for (p=0; p<k; p++)
   for (i=0; i<m; i++)
      for (j=0; j<n; j++)
         C[i*n+j] += A[i*k+p] * B[p*n+j] ;
```

FIGURE 2.3: Naïve triple loop implementation of a matrix-matrix multiplication in C. We assume the output matrix to be initialized with zeros.

for which the computation of the functionality is done straightforward.

In the blocked multiplication case, the three dimensions can be reduced independently. Therefore, it is sufficient to know how to handle each one to be able to tackle any size. In the first two cases, the matrix multiplication degenerates into Kronecker products; in the last case, it simplifies into a scalar product:

$$\mathbf{MMM}_{m,1,1} \to K_{m\times 1}, \tag{2.20}$$

$$\mathbf{MMM}_{1,1,n} \to K_{1\times n}, \tag{2.21}$$

$$\mathbf{MMM}_{1,k,1} \to R_k. \tag{2.22}$$

Note that these three rules are degenerate special cases of the blocking rules in Equation 2.16, Equation 2.17 and Equation 2.18.

Algorithm space. A complete algorithm to compute a functionality is obtained by inserting the breakdown rules into each other and varying the degrees of freedom. For instance, Figure 2.3 presents the naïve algorithm that accumulates outer products in order to compute a matrix multiplication.

In OL, the equivalent algorithm can be obtained by successively applying to $\mathbf{MMM}_{m,k,n}$ the rule in Equation 2.18 with $k_b = 1$, the rule in Equation 2.16 with $m_b = 1$ and finally the rule in Equation 2.21. The derivation is shown below (obvious simplifications are performed for improved readability):

$$\begin{aligned}\mathbf{MMM}_{m,k,n} &\to \left(R_k \otimes \mathbf{MMM}_{m,1,n}\right) \circ \left(L_k^{mk} \times I_{kn}\right)\\ &\to \left(R_k \otimes \left(K_{m\times 1} \otimes \mathbf{MMM}_{1,1,n}\right)\right) \circ \left(L_k^{mk} \times I_{kn}\right)\\ &\to \left(R_k \otimes \left(K_{m\times 1} \otimes K_{1\times n}\right)\right) \circ \left(L_k^{mk} \times I_{kn}\right).\end{aligned} \tag{2.23}$$

The last line exactly corresponds to the four lines of pseudocode above. This exhibits the three strengths of OL:

1. it concisely captures the structure of the computation,

2. it is index-free (no dummy indices i, j and p),

3. it is point-free[9] (no explicit input-output arrays A, B, C).

Also note that we actually *derived* the final algorithm, thereby giving a proof of correctness, assuming Equation 2.16, Equation 2.17 and Equation 2.18 are correct.

And beyond. More complex algorithms can be generated by following the same principle. For instance, the above rules capture both the standard "recursive" and "iterative" class of algorithms [Yotov et al., 2007]. Furthermore, algorithms with sub-cubic cost can also be described in OL. For instance, Strassen's method to multiply 2×2 matrices using only 7 multiplications (instead of 8) is captured as a base case for $\mathbf{MMM}_{2,2,2}$ [Strassen, 1969]:

$$\mathbf{MMM}_{2,2,2} \to \begin{bmatrix} 1 & 0 & 0 & 1 & -1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & -1 & 1 & 0 & 0 & 1 & 0 \end{bmatrix} \circ P_7 \circ \left(\begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ -1 & 1 & 0 & 0 \\ 0 & 0 & 1 & -1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 \\ -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix} \right). \quad (2.24)$$

Since blocking is a tensor product of two $\mathbf{MMM}$s (Equation 2.19), the Strassen base case can then be used in the coarse part of the tensor while standard matrix blocking rules are used in the fine structure. Such algorithms are called *hybrid Strassen* and they are among the fastest (albeit numerically unstable) methods to compute the matrix multiplication [D'Alberto and Nicolau, 2007]. However, this will unfortunately remain purely speculative for this dissertation as current limitations in the compiler prevent us from generating code for non canonical tensor products such as this one.

2.3 OL for Applications Example: Viterbi Decoding

Many real world applications have little global structure and therefore OL cannot reasonably be extended to cover them. However, for many compute intensive applications, most of the time is spent within a single functionality which is the performance bottleneck [Intel, 2009d; Jain, 1991]. Sometimes OL can be enlarged to capture these kernels. Spiral can then be used to deliver highly optimized performance critical parts of full applications. There are two main reasons for using a domain-specific language. First, it structures and simplifies the implementation of our software generator. Second, it can enable the automatic SIMD vectorization of the bottleneck. This section presents one such application that was considerably sped up with OL, namely *Viterbi decoding*.

Note that OL has been also extended to support other applications, notably synthetic aperture radar [McFarlin et al., 2009] and sorting [Franchetti et al., 2009]. The most demanding part of the encoding algorithm for the image compression standard JPEG 2000 [ISO, 2004] has also been captured [Shen, 2008].

Viterbi decoding. Viterbi decoding is a maximum likelihood sequence decoder method introduced in [Viterbi, 1967], and finds wide usage in communications, speech recognition, and sta-

[9] For a good example of point-free notation, see [Gibbons, 1999].

tistical parsing. As a decoder for convolutional codes, it is used in a broad range of everyday applications and telecommunication standards including wireless communication (e.g., cell phones and satellites) and high-definition television [Viterbi, 2006]. In the past, the high throughput requirements for decoding demanded dedicated hardware implementations [Black and Meng, 1992, 1997; Kang and Willson, 1998; Lin et al., 2005]. However, the dramatically growing processor performance has started to change this situation: expensive processing is now often done in software for reasons of cost and flexibility. A prominent example is software defined radio [Mitola, 2002].

In this section, we start by introducing convolutional codes and the Viterbi algorithm. We then explain how the Viterbi algorithm is usually split it into two parts, the forward pass and the traceback. We finally rewrite the forward pass, which is the computation bottleneck, and capture it in OL.

2.3.1 Convolutional Codes

The purpose of forward error-correcting codes (FEC) is to prevent the corruption of a message by adding redundant information before the message is sent over a channel. At the receiver side the redundant data is used to reconstruct the original message. In this dissertation, we focus on a single type of FEC, namely convolutional codes. These codes are heavily used in telecommunications including Global System for Mobile communications (GSM) and Code Division Multiple Access (CDMA).

A convolutional encoder takes as input a bit stream and convolves it with a number of fixed bit sequences to obtain the output bit stream. Since convolution is equivalent to polynomial multiplication, the fixed bit sequences are often called polynomials.

Formal specification. Formally, a convolutional code is specified by N polynomials of degree $K-1$, denoted with $p_1, \ldots, p_N$. Such a code is said to have a *constraint length* K and a *rate* $1/N$, i.e., for each input bit, the encoder produces N output bits.

We view each polynomial p_ℓ alternatively as bit sequence[10], integer[11], or actual polynomial $p_\ell(x)$ with binary coefficients. An example polynomial for $K = 3$ is

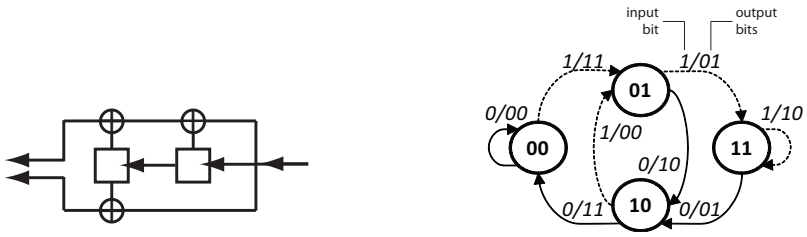
$$p_\ell = 101\,_2 \quad \Leftrightarrow \quad p_\ell = 5 \quad \Leftrightarrow \quad p_\ell(x) = x^2 + 1\,.$$

More generally, if $a = a_0, \ldots, a_{S-1}$ is a sequence of bits, then $a(x) = \sum_{i=0}^{S-1} a_i x^i$.

Encoding now works as follows. The bit stream to be encoded is divided into blocks of length S. Such a block of bits $a = a_0, \ldots, a_{S-1}$ is convolved (denoted with $\star$) with each p_ℓ, $\ell = 1, \ldots, N$,

[10] We denote bit sequences like this: $101\,_2$.

[11] In the literature, integers for polynomials are often expressed in octal or hexadecimal format.



(a) (b)

FIGURE 2.4: (a) Hardware implementation with a shift register and (b) Finite state machine representations of the encoder $r = 1/2$, $K = 3$ with polynomials $7 \Leftrightarrow x^2 + x + 1$ and $5 \Leftrightarrow x^2 + 1$.

which is equivalent to polynomial multiplication:

$$\begin{aligned} b^1 = p_1 \star a \quad &\Leftrightarrow \quad b^1(x) = p_1(x)a(x) \\ &\dots \\ b^N = p_N \star a \quad &\Leftrightarrow \quad b^N(x) = p_N(x)a(x)\,. \end{aligned}$$

The bit streams $b^1, \dots, b^N$ are interleaved to yield the output bit stream b. Each b^ℓ has length $F = S + K - 1$, called the *frame length*. It is convenient to view the output stream as a sequence of F many words of N bits each.

Example. Assume we want to encode the bit-stream $a = 1010_2$ with polynomials $p_1 = 7$ and $p_2 = 5$, i.e., $N = 2$ and $K = 3$:

$$\begin{aligned} b^1(x) &= (x^2 + x + 1)(x^3 + x) \\ &= x^5 + x^4 + x^2 + x \\ \Leftrightarrow b^1 &= 110110_2\,. \end{aligned}$$

Similarly, $b^2(x) = (x^2 + 1)(x^3 + x) \Leftrightarrow b^2 = 100010_2$. Therefore the final output stream is

$$b = 11\,10\,00\,10\,11\,00_2\,. \tag{2.25}$$

Hardware implementation. A different way to look at a convolutional code is to consider its actual hardware implementation, represented for our example in Figure 2.4a.

A shift-register with $K-1$ flip-flops is used to delay the input stream so that modulo-2 additions can be performed between the K newest bits. The actual wiring of the adders is determined by the bit representation of the polynomials p_ℓ. The initial content of all registers is 0 and that the input stream is padded with $K - 1$ trailing zeros to flush the message through the channel.

For instance, assume the two registers in Figure 2.4a contain 0 and 1 and a 1 enters the encoder. In this case, the top output bit is a $0\,(\equiv 0 + 1 + 1)$ and the bottom output bit is a $1\,(\equiv 0 + \quad + 1)$ so

the aggregated output is 01_2. Shifting the values to the left, the registers now contains 1 and 1.

Finite state machine (FSM) representation. Equivalently, the encoding process can be represented by a finite state machine with 2^{K-1} states (the possible states of the shift register) that outputs N bits on each transition. The FSM equivalent to Figure 2.4a is shown in Figure 2.4b.

Each state in the finite state machine has a 0-transition (input bit is 0, solid arrow) and a 1-transition (input bit is 1, dashed arrow) to other states. More precisely, there exists a 0-transition between states n and m if $m \equiv 2n \mod 2^{K-1}$. Similarly, there exists a 1-transition between states n and m if $m \equiv (2n+1) \mod 2^{K-1}$. The output bit in b^ℓ (corresponding to the polynomial p_ℓ) when transitioning from state n to state m is computed as

$$b^\ell_{n\to m} = \bigoplus \Big(p_\ell \& \big(2n \oplus (m \& 1)\big) \Big) .$$

Here, $\&$ is the bit-wise AND, $\oplus$ is the bit-wise XOR, and the initial $\bigoplus$ performs an XOR on all bits. As said before, the initial state is assumed to be 0 and the input stream is padded with $K-1$ trailing zeros.

In our running example (Figure 2.4b), assume the current state is 01_2. If the input bit is 1 then the FSM outputs 01_2 and transitions to state 11_2.

Viterbi trellis. A third representation of the encoding process "unrolls" the finite state machine in time to yield the Viterbi *trellis*, shown in Figure 2.5 for our running example. Each path from the initial state to the final state represents a possible message that can be sent in a single frame.

The different states of the encoder are placed vertically, the different time steps, or *stages* are placed horizontally. For example the first line contains state 00_2 at different stages, the second line contains state 01_2 and so on. The initial state (first stage) when starting a frame is 0. The zero padding explained previously implies that the last $K-1$ transitions are 0-transitions. Therefore, there is a unique final state and it is zero.

In our example, the message 1010_2 is first padded to 101000_2 then encoded in the finite state machine. The path that results is highlighted in Figure 2.5. Collecting the corresponding output bits yields Equation 2.25 again.

2.3.2 Viterbi Decoding

The Viterbi algorithm is a dynamic programming method that performs maximum likelihood sequence decoding on a convolutionally encoded stream. Intuitively, the decoder receives a bit stream from the receiver and has to find the path in the Viterbi trellis that best corresponds to it, which, ideally, would be the same path the encoder took. It is composed of three phases, the *branch metric* computation, the *path metric* computation, and the *traceback*. The best visualization of the Viterbi algorithm again uses the Viterbi trellis but its purpose is now reversed: the incoming message is fixed and the path is to be found.

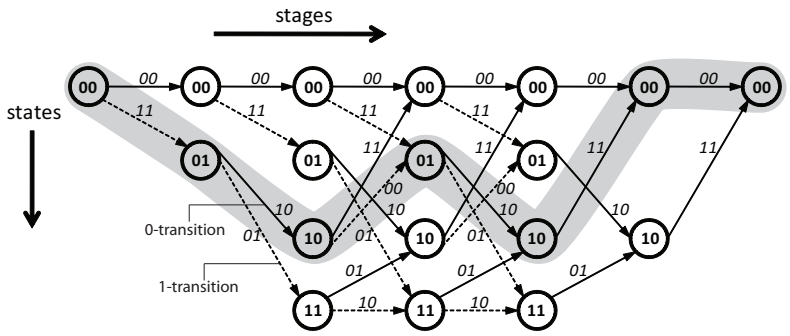


FIGURE 2.5: Viterbi trellis representation of the encoder $r = 1/2$, $K = 3$ with polynomials $7 \Leftrightarrow x^2 + x + 1$ and $5 \Leftrightarrow x^2 + 1$ encoding 1010_2 (padded to 101000_2).

Branch metrics computation. In the first phase, the Viterbi algorithm assigns a cost, called the *branch metric*, to each edge in the trellis. This value represents how well the received bits would match if we knew the encoder took the transition corresponding to a given edge. It is computed by taking the Hamming distances between the bits the transition should output and the actually received ones.

Building on our example, we assume the decoder just received the message $11\,10\,10\,00\,11\,00_2$ (which is the message in Equation 2.25 with two bit flips corresponding to injected errors) and we aligned it vertically with the corresponding stages (top row in Figure 2.6a). The branch metrics that have been placed on each arrow are the Hamming distance between the actually received bits and the output bits of the transition (shown in Figure 2.5)

Path metrics computation. After the previous phase, the problem is equivalent to finding the shortest path between the entry and the exit vertices on a directed acyclic graph with weighted edges. Therefore, the second phase is a breadth-first forward traversal of the graph. It progressively computes the *path metric*, which is the shortest path to get from the root to each vertex. If a state has the path metric π, there exists one message that ends in the state with π corrupted bits and this message is less or equally corrupted than all other messages. While computing this, the predecessor of each node is remembered as a *decision bit*[12].

In our example (Figure 2.6b), path metrics have been written inside each state and decision bits are represented by removing the discarded incoming edge.

Traceback. The decision bits describe the ancestor of each vertex. Given this information and the final state, one can reconstruct the shortest path, called the *survivor path* by reading off predecessors.

In our example, remembering that solid lines correspond to the zero input bit and dashed lines to the one input bit, one can simply read off the shortest path on the Figure 2.6b by starting in the

[12]The structure of the FSM guarantees that there are exactly two incoming edges into each vertex, except for the leftmost nodes in the trellis where there is only one.

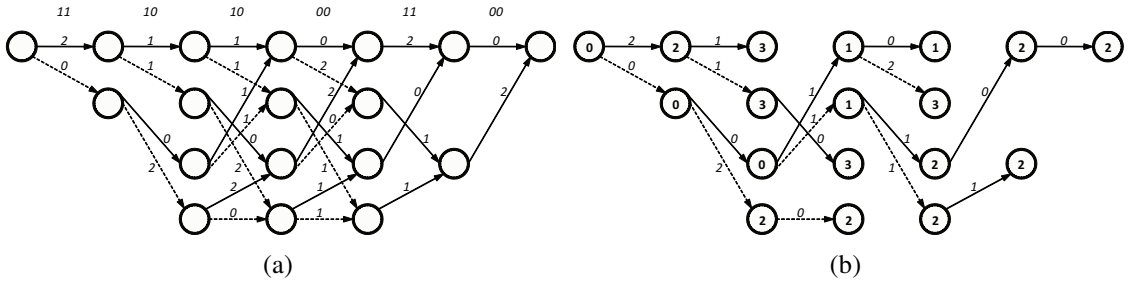


FIGURE 2.6: The same Viterbi trellis at different phases in the algorithm: (a) After branch metrics have been computed and (b) After path metrics have been propagated and predecessors have been decided.

final state and reading backwards to get 101000_2.

In a software Viterbi decoder, it is important to perform branch and path metrics computations simultaneously to improve the ratio of operations over memory accesses. The fusion of these two phases is called the *forward pass*. For more information about the Viterbi algorithm, we refer the reader to [Fleming, 2006].

Soft decisions. In actual applications, Viterbi decoders are usually placed downstream of a receiver that converts analog channel symbols to a digital format. If the receiver classifies incoming symbols as either zero or one, the system is said to be using *hard decisions*. However, doing so hides uncertainty which can be useful for decoding. For example, assuming symbols on the channel are internally mapped by the receiver to reals between zero and one, 0.9 makes a more convincing 1 than 0.6.

This observation underlies a *soft-decision* receiver, which directly encodes its confidence level and passes this information to the decoder. Let Q be the number of levels in which the information is encoded: for instance, if $Q = 8$, the receiver classifies in 8 different levels and if $Q = 2$ the system is actually making hard decisions. The higher Q, the less emphasis is put on ambiguous symbols, resulting in an overall better decoding.

In a hard decision decoder as shown in Figure 2.6, the decoder receives 1 bit per polynomial per stage and branch metrics are therefore comprised between 0 (perfect match) and N (bits differ on all polynomials). Formally, let $\beta^i_{n\to m}$ be the branch metric for the transition from state n to m at stage i and let s^i_ℓ denote the bit for the ℓ-th polynomial at stage i. Then, the Hamming distance expression is

$$\beta^i_{n\to m} = \sum_{\ell=1}^{N} \left| s^i_\ell - b^\ell_{n\to m} \right| .$$

In a soft-decision system with Q levels of quantization, the branch metrics range between 0 and $(Q-1)N$ since the maximal mistake that can be done on each polynomial is $Q-1$. Redefining s^i_ℓ to be the *symbol* (i.e., a value between 0 and $Q-1$) for the ℓ-th polynomial at stage i, the Hamming

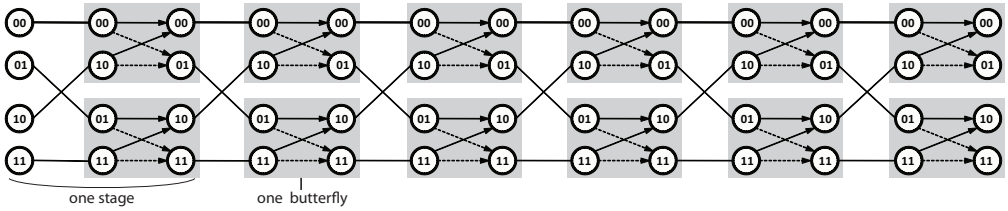


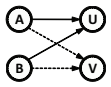
FIGURE 2.7: Each stage in the Viterbi trellis consists of a perfect shuffle and 2^{K-2} parallel butterflies (here $K = 3$ and $F = 6$).

distance generalizes to

$$\beta^i_{n\to m} = \sum_{\ell=1}^{N} |s^i_\ell - (Q-1)b^\ell_{n\to m}| \,.$$

Viterbi butterflies. The trellis shown in Figure 2.5 has a regular structure except for the initial and final stages. The initial stage can be handled like all other stages by inserting prohibitively high path metrics as appropriate. Handling the final stage like all other stages simply involves computing all path metrics—the useless ones are automatically discarded.

Closer inspection of the trellis structure now shows that each stage of the forward pass can be decomposed in two phases: a fixed permutation called a *perfect shuffle* and a parallel operation on 2^{K-2} 2-by-2 substructures called *butterflies* (see Figure 2.7). In the following, we denote the states of a butterfly as shown below:



During the path metric computation, each butterfly does two *Add-Compare-Select* operations to compute the path metrics π_U and π_V from the path metrics π_A and π_B and the branch metrics $\beta_{A\to U}$, $\beta_{A\to V}$, $\beta_{B\to U}$ and $\beta_{B\to V}$:

$$\begin{cases} \pi_U = \min_{d_U}(\pi_A + \beta_{A\to U}, \pi_B + \beta_{B\to U}) \\ \pi_V = \min_{d_V}(\pi_A + \beta_{A\to V}, \pi_B + \beta_{B\to V}) \,. \end{cases}$$

Note that the minimum operator $\min_d(a, b)$ actually performs both the compare and select operations simultaneously. It returns the actual minimum of a and b and stores the binary decision in the decision bit d.

Symmetries. Code designers usually impose additional properties on the polynomials that trigger symmetries in the trellis and simplify the computation [Taipale, 2004].

In particular, if states A and B can both transition into the same state U, then all lower bits of A and B must be the same. In fact, the two states differ precisely in their most significant bit, the one that is shifted out (Figure 2.4a). Now, if a polynomial has degree $K - 1$, the outgoing bit

is guaranteed to be taken into account and thus, the corresponding output bits for A and B have to be complements of each other. Therefore, if all polynomials have this property[13], then the two incoming transitions into the same state always have symbols that complement each other (it can be observed in Figure 2.5). Therefore, one branch metric computation can be deduced from the other one: $\beta_{B\to U} = (Q-1)N - \beta_{A\to U}$.

In our running example, since $Q = 2$ and $N = 2$, we can verify in Figure 2.6a that all pairs of arrows entering the same state either have branch metrics (0,2) or (1,1).

Similarly, if state A can reach both state U and state V, then U and V only differ by the lowest significant bit. Therefore, if all polynomials have their constant coefficient equal to one[14], the output bits for all pairs of arrow that stem from the same state are complement of each other. Therefore, the branch computation can also be reduced: $\beta_{A\to V} = (Q-1)N - \beta_{A\to U}$.

In our running example, since $Q = 2$ and $N = 2$, we can verify in Figure 2.6a that all pairs of arrows outgoing from the same state either have branch metrics (0,2) or (1,1).

By combining both symmetries, the computation carried out by the butterflies during the pass metric propagation can be simplified to use only one branch metric instead of four:

$$\begin{cases} \pi_U = \min_{d_U}(\pi_A + \beta_{A\to U}, \pi_B + (Q-1)N - \beta_{A\to U}) \\ \pi_V = \min_{d_V}(\pi_A + (Q-1)N - \beta_{A\to U}, \pi_B + \beta_{A\to U}) \,. \end{cases}$$

Forward pass with both symmetries. Since the transition $A \to U$ is always a 0-transition, the computation carried out by the j-th butterfly at stage i during the forward pass (branch and path metrics computation), assuming both symmetries explained above hold, can therefore be simplified to

$$\begin{cases} \beta_+ = \sum_{\ell=1}^{N} s_\ell^i \oplus \big[(Q-1)\big(\oplus (p_\ell \& 2j)\big)\big] \\ \beta_- = (Q-1)N - \beta_+ \\ \pi_{2j}^i = \min_{d_{2j}^i}(\pi_{2j}^{i-1} + \beta_+, \pi_{2j+1}^{i-1} + \beta_-) \\ \pi_{2j+1}^i = \min_{d_{2j+1}^i}(\pi_{2j}^{i-1} + \beta_-, \pi_{2j+1}^{i-1} + \beta_+) \,. \end{cases}$$

Note that the computation can be reduced further by introducing the *tabulated branch* t, a pre-computed two-dimensional array containing numbers resulting from the polynomials in the following way:

$$t_{2j}^{\ell} = (Q-1)\big(\oplus (p_\ell \& 2j)\big) \,. \tag{2.26}$$

[13]This condition is equivalent to saying that the integers representing the polynomials are all bigger than 2^{K-1} and also equivalent to saying that the final register is always connected to one of the adders.

[14]This condition is equivalent to saying that the integers representing the polynomials are all odd and also equivalent to saying that the input is always connected to one of the adders.

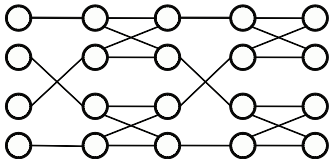


FIGURE 2.8: Dataflow of the Pease algorithm for computing the $\mathbf{WHT}_4$.

2.3.3 Forward Pass Formulation in OL

From now on, we only consider the forward pass, excluding the traceback. The reason is that the traceback is computationally much cheaper than the forward pass, requiring $O(F)$ operations versus $O(2^K F)$ for the forward pass. Hence, in practice, except for very short constraint lengths, a generic traceback is not the performance bottleneck.

Intuition: Pease FFT algorithms. Among the many fast (i.e., $O(n \log n)$) algorithms to compute the DFT and WHT (see Table 2.1), the so-called Pease algorithms [Pease, 1968] stand out for having maximal regularity, which makes them good candidates for hardware implementation [Milder et al., 2008].

We denote the $n \times n$ *bit-reversal* permutation with R_n and the *twiddle* diagonal matrix with T_i^n. Their exact definition is not important here. The DFT *butterfly*[15] matrix F_2 corresponds to a DFT on two points: $F_2 = \left[\begin{smallmatrix} 1 & 1 \\ 1 & -1 \end{smallmatrix}\right]$. In SPL, the Pease algorithms for WHT and DFT are written as:

$$\mathbf{WHT}_{2^n} \rightarrow \prod_{i=0}^{n-1} \left((I_{2^{n-1}} \otimes F_2) L^{2^n}_{2^{n-1}} \right) \tag{2.27}$$

$$\mathbf{DFT}_{2^n} \rightarrow R_{2^n} \prod_{i=0}^{n-1} \left(T_i^n (I_{2^{n-1}} \otimes F_2) L^{2^n}_{2^{n-1}} \right). \tag{2.28}$$

The dataflow for the Pease WHT of size 4 is shown in Figure 2.8 (the Pease DFT dataflow is analogous, except for the final bit-reversal). Note the similarity to the Viterbi trellis shown in Figure 2.7, but remember that the butterflies operate differently. The resemblance between the DFT, the WHT and the Viterbi forward pass was already noticed in [Forney, 1973], [Rader, 1981] and [Gilhousen et al., 1971]. Omega networks also share this dataflow [Lawrie, 1975].

Forward pass. The forward pass of the Viterbi algorithm is not linear and therefore cannot be expressed in the SPL framework. It is however captured by the OL tensor product which can handle non-linear operators.

Observe that the DFT butterfly F_2 is a two-by-two matrix but can equivalently be seen as an

[15] The Viterbi butterfly is not to be confused with the DFT butterfly.

operator that takes two inputs x_0 and x_1 and produces two outputs y_0 and y_1:

$$\begin{cases} y_0 = x_0 + x_1 \\ y_1 = x_0 - x_1 \,. \end{cases}$$

Similarly, we view the j-th Viterbi butterfly decoding the i-th codeword as an operator $B_{i,j}$ that takes two inputs x_0 and x_1 and produces two outputs y_0 and y_1. The difference between F_2 and $B_{i,j}$ is that, depending on its position, the Viterbi butterfly uses values from two global arrays, namely the received symbols s and the tabulated branch t, and it also writes values to the decision bit array d (through the "select" part of the minimum operator). Formally, it computes

$$\begin{cases} \beta_+ = \sum_{\ell=1}^{N} s_\ell^i \oplus t_{2j}^\ell \\ \beta_- = (Q-1)N - \beta_+ \\ y_0 = \min_{d_{2j}^i}(x_0 + \beta_+,\, x_1 + \beta_-) \\ y_1 = \min_{d_{2j+1}^i}(x_0 + \beta_-,\, x_1 + \beta_+) \,. \end{cases}$$

Using OL, the forward pass of a Viterbi decoder with constraint length K, frame length F, denoted $\mathbf{Vit}_{K,F}$ can therefore be expressed as

$$\mathbf{Vit}_{K,F} \to \prod_{i=1}^{F} \left((I_{2^{K-2}} \otimes_j B_{F-i,j}) L_{2^{K-2}}^{2^{K-1}} \right) . \tag{2.29}$$

Not surprisingly, the VL formulation of the forward pass looks very similar to the Pease algorithms (Equation 2.27 and Equation 2.28).

CHAPTER 3

Library Core Generation

In the library generation framework that we present (Figure 1.2), the first block (Library Core Generation) takes as input functionality, algorithms, and platform characteristics and derives the implementation of the *library core* for this functionality. The library core can be viewed as an adaptive library that has been stripped of its search logic. It is therefore capable of computing a given functionality using different methods that typically all yield a different performance. Various adaptation mechanisms that can be added to the library core will be presented in the next chapter.

Overview of the problem. High-performance library users want the best code possible for their machine and their problem set. The popularity of libraries such as FFTW ([Frigo and Johnson, 2005]) and ATLAS ([Whaley and Dongarra, 1998]) shows that most of them are willing to pay a fixed cost for this specialization, either at runtime (online adaptation) or at installation time (offline adaptation). We want the library generation framework that we develop to support both cases. It therefore needs to support the most constraining case which is to specialize based on parameters specified by the user at runtime.

The runtime requirement imposes the adaptation to be robust, portable and fast, ruling out the possibility of invoking any compiler. Hence, this adaptation problem creates the paradoxical situation where the code has to be generated first, but the algorithmic transformation space is determined only later. In this thesis, we solve this apparent contradiction by operating at a *meta*-level[1]: what we really compute during the library core generation is the transformation space for *all* possible parameters. This generic space is then instantiated at runtime with the user-provided parameters using the search mechanisms.

In practice, it is not possible to generate every variant of a code, so the generic transformation space has to be somehow closed during the library generation. This closure problem is difficult in

[1] See *meta*-programming [Sheard and Jones, 2002]

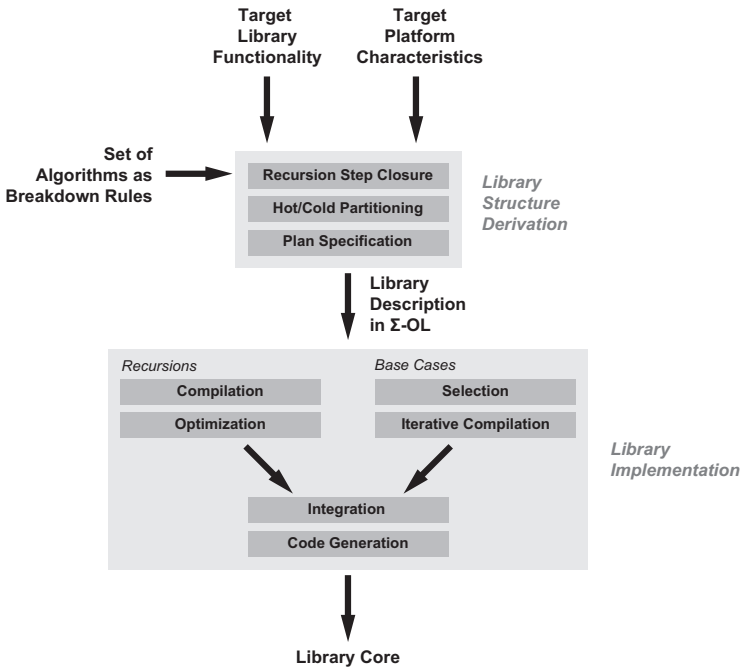


FIGURE 3.1: Library Core Generation Overview.

the general case and might not even have solutions. However, by restricting to SPL and the specific domain of linear transforms, [Voronenko, 2008] solved this problem by deriving the *recursion step closure*, which is the set of mutually recursive functions that are needed to capture the interactions of multiple transform algorithms after locality optimizations.

Our approach. In this thesis, we recognize the recursion step closure derivation as a critical step in the automatic generation of libraries, even beyond linear transforms. The existing SPL framework was therefore enlarged to support OL which can capture additional functionalities such as the matrix-matrix multiplication. This extension leveraged some of the previous work but also requires an expansion of its foundations.

The library core generation from Figure 1.2 is depicted in larger detail in Figure 3.1 and can be decomposed in two components: the *library structure derivation* and the *library implementation*. The first section of this chapter covers the library structure derivation and focuses on the generation of the global structure of the library using the aforementioned recursion step closure. It produces a precise description of each recursion step, or function, in the library. This description uses the intermediary language $\sum$-OL that we introduce and which is derived from $\sum$-SPL [Franchetti et al., 2005].

The second section covers the library implementation, which translates the $\sum$-OL description to the target language C, C++ or Java. Specifically, it produces the recursive functions in the closure and combines them with properly selected base cases to form the library core. These base

cases are generated directly from $\sum$-OL, similar to the generation of fixed input size transforms [Püschel et al., 2005]. To achieve these goals, we expanded the capabilities of the original iterative compiler to support OL and included a few new optimization passes.

Finally, the last section explains how parallelization and vectorization are performed. In practice, these are captured as rules that directly manipulate loop nests. New operators are therefore introduced that extend the GT index-free notation [Voronenko, 2008] to $\sum$-OL.

3.1 Library Structure Derivation

The goal of this section is to automatically derive the structure of the library that is being generated (first block in Figure 3.1). More precisely, the mechanisms presented here are responsible for determining the specifications of all the functions that compose the library based on the target functionality and the breakdown rules used. In practice, the target platform characteristics (e.g., vectorization, parallelization) also play a role in the specification but this part will be specifically covered in Section 3.3.

We start with a brief example. Assume that:

- the target functionality is a dense row major matrix multiplication $\mathrm{C} = \mathrm{A}\,\mathrm{B}$,
- the target platform does not have SIMD or multiprocessing capabilities,
- and the chosen breakdown rules are the three standard matrix blocking rules: Equation 2.16, Equation 2.17, and Equation 2.18.

The target functionality that interests the user constitutes what we call a *recursion step*: a class of problems that only differ by some parameters, here, the dimensions of the matrices and the actual pointers to the data. Typically, a recursion step can be thought of as being an equivalent to the specification of a function in a traditional imperative language. For example, in this case, the target functionality is the first recursion step (`RS1`) and would have the following signature in C:

```
void RS1(int m, int k, int n, double* A, double* B, double* C);
```

After blocking, the smaller matrix multiplications are performed on matrices that are not contiguous in memory. Hence, this function alone is not suitable for a recursive implementation without generating a prohibitive number of explicit copies. A helper recursion step is therefore needed and it takes the following signature, assuming `lda`, `ldb`, and `ldc` are the leading dimensions of the matrices A, B, and C[2]:

```
void RS2(int m, int k, int n, int lda,
                int ldb, int ldc, double* A, double* B, double* C);
```

[2]Our denominations for leading dimensions are intentionally compatible with the ones from the commonly used DGEMM interface [Dongarra et al., 1990].

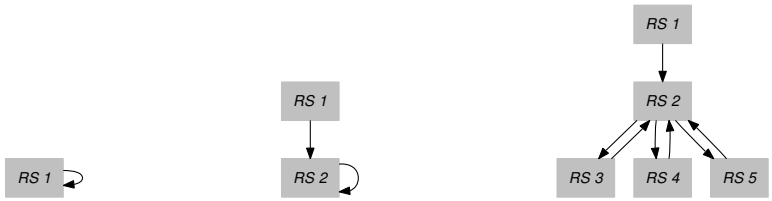


FIGURE 3.2: Static call graphs of three matrix-matrix multiplication libraries. We call the set that comprises all recursion steps of a given library the *recursion step closure* of the library. The first recursion step (RS1) captures the functionality that the user is interested in and is therefore part of the API (Application Program Interface): it is common to all libraries. However, the strategy to compute it is not fixed and, in particular, it can be decomposed into other recursion steps which are internal to the library. (a) is the naïve implementation and (b) is obtained after removing unnecessary explicit copies from it. Note that the closure can become more complicated, for instance (c) is obtained after an optimization to provide looped base cases.

The differences in the structure of the two possible libraries are best understood with the static call graphs of the libraries, presented in Figure 3.2. In these graphs, the nodes are the recursion steps and the arrows show which steps call which other steps. The set of all recursion steps for a given library is called the *recursion step closure* of the library. The first library closure is trivial: it is uniquely composed of `RS1` and is displayed in Figure 3.2a. The second library closure was obtained by removing explicit copies, it is composed of `RS1` and `RS2` and is displayed in Figure 3.2b. Note that other optimizations can further complicate the closure: for instance, using the generalized tensor operator to capture loops (Subsection 3.3.1) enables the generation of looped base cases and leads to the closure in Figure 3.2c.

Also note the following:

- Ultimately, only the original recursion step (`RS1` in Figure 3.2) ever needs to be exposed to the final user through the library interface since it is the only one that actually captures the problems he or she is interested in.

- There might be different rules to break down a given recursion step. Choosing which rule to apply is considered a degree of freedom. Chapter 4 will explain mechanisms that make good decisions.

- In the present case (Figure 3.2c), the existence of the five different recursion steps allows the library to use every possible regular[3] blocking strategy, for any given input.

In the first part of this section, we introduce $\sum$-OL, a lower-level extension of OL that makes loops and index mappings explicit. Using rewriting systems, $\sum$-OL solves the problem of loop

[3]By regular, we mean that no blocking step produces leftover rows or columns.

merging whose derivation is a key issue in recursive library generation as we will see later. $\sum$-OL is a generalization of $\sum$-SPL, which was designed for linear transforms and introduced in [Franchetti et al., 2005].

The second part of this section presents the *recursion step closure* which is the key step of the library generation since it automatically provides the specification of all the recursion steps needed in the process. It was first introduced by [Voronenko, 2008] for linear transforms.

Lastly, we explain how to partition function parameters into two classes, the *cold parameters* which provide degrees of freedom and the *hot parameters* which don't. This partition impacts the function signatures in general and the user interface in particular.

3.1.1 Loop Merging with Sigma-OL

OL describes the data-flow of algorithms. In particular, data accesses are represented as explicit permutations as, for example, can be seen in the right-hand sides of the Cooley-Tukey FFT rule (Equation 2.6) and the MMM vertical blocking rule (Equation 2.17). A straightforward mapping to code would explicitly perform these permutations which would be detrimental to the performance. The goal of loop merging is to fuse these permutations with adjacent computations to increase locality. We perform this loop merging is performed by rewriting expressions in $\sum$-OL, which is an extension of $\sum$-SPL [Franchetti et al., 2005] that supports operators of higher arities. Note that, while maximizing reuse through loop merging has been proved NP-hard in its most general form [Kennedy and McKinley, 1994], it becomes tractable in our domain-specific framework because the structure of the computation is explicit.

Motivation. In the previous chapter, we derived in Equation 2.23 a naïve OL algorithm to compute the matrix-multiplication. We restate it here:

$$\mathbf{MMM}_{m,k,n} \rightarrow \left(R_k \otimes (K_{m\times 1} \otimes K_{1\times n})\right) \circ \left(L^{mk}_k \times I_{kn}\right). \tag{3.1}$$

Parsing this expression, we observe the two following distinct stages: *first*, the first input matrix is transposed and *second*, an accumulation of Kronecker products is performed. A direct mapping of the algorithm to code would therefore yield the code in Figure 3.3 which is different and worse than the expected code for this naïve algorithm (presented earlier in Figure 2.3). This extra stage is, in a sense, the price paid by the expressiveness of OL. In this subsection, we describe a rewriting system that fuses data reorderings into the subsequent computational loops.

The loop merging process is a rewriting system that can conceptually decomposed into two steps. First, the OL expressions are converted into $\sum$-OL, which introduces explicit loops. For example, after the conversion into $\sum$-OL, the right-hand side of Equation 3.1 becomes:

$$\left(\sum_{0\le p<k}\ \sum_{0\le i<m}\ \sum_{0\le j<n} \mathrm{S}(h^{1\rightarrow mn}_{in+j,1}) \circ P_1 \circ (\mathrm{G}(h^{1\rightarrow mk}_{pm+i,1}) \times \mathrm{G}(h^{1\rightarrow kn}_{pn+j,1}))\right) \circ \left(L^{mk}_k \times I_{kn}\right).$$

```
for (t=0; t<m*k; t++)
   T[(t%k)*m + t/k] = A[t] ;
for (p=0; p<k; p++)
   for (i=0; i<m; i++)
      for (j=0; j<n; j++)
         C[i*n+j] += T[p*m+i] * B[p*n+j] ;
```

FIGURE 3.3: Direct mapping of Equation 3.1 to code. The matrix `A` is explicitly transposed to a temporary matrix `T`, a step that could be merged with the subsequent computation and yield more efficient code shown in Figure 2.3. The purpose of the $\sum$-OL rewriting system is to perform this kind of loop merging automatically.

Then, a set of rewriting rules propagates the data shuffles (here: $\left(L^{mk}_k \times I_{kn}\right)$) into the sums which represent loops. In our example, we get:

$$\sum_{0\le p<k} \sum_{0\le i<m} \sum_{0\le j<n} \mathrm{S}(h^{1\to mn}_{in+j,1}) \circ P_1 \circ (\mathrm{G}(h^{1\to mk}_{ik+p,k}) \times \mathrm{G}(h^{1\to kn}_{pn+j,1})). \tag{3.2}$$

In summary, the OL tensors, which expressed the structure of the algorithm, are converted into $\sum$-OL *iterative sums* and indexing. Within these sums, the data points are first *gathered* using index mapping functions, multiplied together and *scattered* into the result array. Note that, in Equation 3.2, the triple loop nature of the algorithm becomes apparent.

Next, we provide a more formal description of $\sum$-OL and loop-merging.

Definition. $\sum$-OL is composed of three components: index mapping functions, parametrized operators, and iterative sums.

Index mapping functions are functions mapping interval into intervals. In this thesis, we will only use three of them, the *identity* $\imath$[4], the *stride* h, and the *transposition* ℓ. Let $\mathbb{I}_n$ denote the integer interval $\{0, ..., n-1\}$, then

$$\begin{aligned} \imath^{n\to n} &: \mathbb{I}_n \to \mathbb{I}_n;\ i \mapsto i, \\ h^{d\to r}_{b,s} &: \mathbb{I}_d \to \mathbb{I}_r;\ i \mapsto b + is, \\ \ell^{mk\to mk}_k &: \mathbb{I}_{mk} \to \mathbb{I}_{mk};\ i \mapsto \lfloor \tfrac{i}{m} \rfloor + k(i \bmod m). \end{aligned}$$

The stride index mapping $h_{b,s}$ *strides* the input by a factor s starting at b. The transposition is the permutation underlying the transposition operator L introduced in Equation 2.3.

Parametrized operators are operators whose definition depends on an index mapping function. We will define three of them: the *gather* $\mathrm{G}(f)$, the *scatter* $\mathrm{S}(f)$, and the *permutation* $\mathrm{perm}(f)$. We define here arity-(1,1) parametrized operators (i.e., parametrized matrices); higher-arities parametrized operators follow. Let $f^{d\to r}$ be an index mapping function from $\mathbb{I}_d$ into $\mathbb{I}_r$ and let $\mathbf{e}^n_i$ be the ith

[4]For backwards compatibility with legacy notations, the symbol of the identity index mapping is an i without dot [Franchetti et al., 2005].

canonical basis vector of $\mathbb{C}^n$, then

$$\begin{aligned} \mathrm{S}(f^{d\to r}) &= [\mathbf{e}^r_{f(0)} \mid ... \mid \mathbf{e}^r_{f(d-1)}], \\ \mathrm{G}(f^{d\to r}) &= \mathrm{S}(f)^T, \\ \mathrm{perm}(f^{d\to d}) &= \mathrm{G}(f) = \mathrm{S}(f^{-1}). \end{aligned}$$

To give the reader some intuition, consider a simple example where one wants to extract $d = 2$ data points at stride $s = 4$ among a set of $r = 8$ points. In $\sum$-OL, this translates to a gather operator which corresponds to a simple rectangular matrix:

$$\mathrm{G}(h^{2\to 8}_{0,4}) = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix},$$

If the data points are gathered at the same stride but are also offset by $b = 1$, we get:

$$\mathrm{G}(h^{2\to 8}_{1,4}) = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}.$$

The last component of $\sum$-OL is the *iterative sum*, denoted with the usual $\sum$ symbol. Sums are indexed over an interval and express the concepts of loops and iterations. Formally, we define for a series of arity-(1,1) operators A_i and a series of arity-(2,1) operators B_i:

$$\left(\sum_i A_i\right) x = \sum_i (A_i x) \qquad\qquad \left(\sum_i B_i\right)(x, y) = \sum_i (B_i(x, y))$$

Observe that the original definition of iterative sums [Franchetti et al., 2005] guarantees that the different $A_i x$ are non-overlapping and that therefore, the resulting iterative sum is a convenient mathematical representation that does not incur any additional computational cost. We do not take such a view in our definition and stick with the mathematical definition. We therefore default the optimization of useless sums onto the compiler.

Rewriting OL into $\sum$-OL. The purpose of $\sum$-OL is to be an intermediate language between OL and imperative code. In this paragraph we explain how to transform OL expressions into $\sum$-OL expressions.

As we have seen earlier, the main difference between OL and $\sum$-OL is that tensors are converted into iterative sums which represent loops. The transformation from one into the other is done by parsing the OL expression tree top down, recursively matching and replacing subtrees with the parametrized $\sum$-OL templates listed in Table 3.1.

Simultaneously with the introduction of loops that tend to complicate the expression, other sets

$$I_m \otimes \mathrm{A}^{d\to r} \to \sum_{0\le i<m} \mathrm{S}(h^{r\to mr}_{ir,1}) \circ \mathrm{A} \circ \mathrm{G}(h^{d\to md}_{id,1}) \tag{3.3}$$

$$\mathrm{A}^{d\to r} \otimes I_m \to \sum_{0\le i<m} \mathrm{S}(h^{r\to mr}_{i,m}) \circ \mathrm{A} \circ \mathrm{G}(h^{d\to md}_{i,m}) \tag{3.4}$$

$$P_p \otimes \mathrm{A}^{m\times n\to k} \to \sum_{0\le i<p} \mathrm{S}(h^{k\to pk}_{ik,1}) \circ \mathrm{A} \circ (\mathrm{G}(h^{m\to pm}_{im,1}) \times \mathrm{G}(h^{n\to pn}_{in,1})) \tag{3.5}$$

$$\mathrm{A}^{m\times n\to k} \otimes P_p \to \sum_{0\le i<p} \mathrm{S}(h^{k\to pk}_{i,p}) \circ \mathrm{A} \circ (\mathrm{G}(h^{m\to pm}_{i,p}) \times \mathrm{G}(h^{n\to pn}_{i,p})) \tag{3.6}$$

$$R_p \otimes \mathrm{A}^{m\times n\to k} \to \sum_{0\le i<p} \mathrm{A} \circ (\mathrm{G}(h^{m\to pm}_{im,1}) \times \mathrm{G}(h^{n\to pn}_{in,1})) \tag{3.7}$$

$$\mathrm{A}^{m\times n\to k} \otimes R_p \to \sum_{0\le i<p} \mathrm{A} \circ (\mathrm{G}(h^{m\to pm}_{i,p}) \times \mathrm{G}(h^{n\to pn}_{i,p})) \tag{3.8}$$

$$K_{p\times q} \otimes \mathrm{A}^{m\times n\to k} \to \sum_{0\le i<p} \sum_{0\le j<q} \mathrm{S}(h^{k\to pqk}_{(iq+j)k,1}) \circ \mathrm{A} \circ (\mathrm{G}(h^{m\to pm}_{im,1}) \times \mathrm{G}(h^{n\to qn}_{jn,1})) \tag{3.9}$$

$$\mathrm{A}^{m\times n\to k} \otimes K_{p\times q} \to \sum_{0\le i<p} \sum_{0\le j<q} \mathrm{S}(h^{k\to pqk}_{iq+j,pq}) \circ \mathrm{A} \circ (\mathrm{G}(h^{m\to pm}_{i,p}) \times \mathrm{G}(h^{n\to qn}_{j,q})) \tag{3.10}$$

$$K_{p\times q} \to K_{p\times q} \otimes P_1 \tag{3.11}$$

$$R_p \to R_p \otimes P_1 \tag{3.12}$$

TABLE 3.1: Rules to convert OL to $\sum$-OL. Depending on the superscripts, A is assumed to be either a (1,1)-operator from $\mathbb{C}^d$ into $\mathbb{C}^r$ or a (2,1)-operator from $\mathbb{C}^m \times \mathbb{C}^n$ into $\mathbb{C}^k$

$$\Big(\sum\nolimits_j \mathrm{A}_j\Big) \circ \mathrm{B} \to \Big(\sum\nolimits_j \mathrm{A}_j \circ \mathrm{B}\Big) \tag{3.13}$$

$$\mathrm{B} \circ \Big(\sum\nolimits_j \mathrm{A}_j\Big) \to \Big(\sum\nolimits_j \mathrm{B} \circ \mathrm{A}_j\Big) \tag{3.14}$$

$$(\mathrm{A} \times \mathrm{B}) \circ (\mathrm{C} \times \mathrm{D}) \to (\mathrm{A} \circ \mathrm{C}) \times (\mathrm{B} \circ \mathrm{D}) \tag{3.15}$$

$$\mathrm{S}(f_1) \circ \mathrm{S}(f_2) \to \mathrm{S}(f_1 \circ f_2) \tag{3.16}$$

$$\mathrm{G}(f_1) \circ \mathrm{G}(f_2) \to \mathrm{G}(f_2 \circ f_1) \tag{3.17}$$

$$\mathrm{G}(f) \circ \mathrm{perm}(p) \to \mathrm{G}(p \circ f) \tag{3.18}$$

$$\mathrm{perm}(p) \circ \mathrm{S}(f) \to \mathrm{S}(p^{-1} \circ f) \tag{3.19}$$

TABLE 3.2: $\sum$-OL loop merging rules. Assume A and B are operators, C and D are operators which can be composed to A and B (they need to have compatible signatures), f, f_1, f_2 are index mapping functions, and p is a bijective index mapping function.

$$I_n \to \mathrm{perm}(\imath^{n\to n}) \tag{3.20}$$
$$L_k^{mk} \to \mathrm{perm}(\ell_k^{mk\to mk}) \tag{3.21}$$
$$(\ell_k^{mk\to mk})^{-1} \to \ell_m^{mk\to mk} \tag{3.22}$$
$$\ell_n^{mk\to mk} \circ h_{kj,1}^{k\to mk} \to h_{j,m}^{k\to mk} \tag{3.23}$$
$$h_{b',s'}^{N\to N'} \circ h_{b,s}^{n\to N} \to h_{b'+bs',ss'}^{n\to N'} \tag{3.24}$$
$$h_{0,1}^{n\to n} \to \imath^{n\to n} \tag{3.25}$$
$$f \circ \imath \to f \tag{3.26}$$
$$\imath \circ f \to f \tag{3.27}$$

TABLE 3.3: $\sum$-OL index simplification rules. f is an arbitrary index function.

of rewriting rules are applied to simplify the expressions, notably the loop merging rules (Table 3.2) and the index mapping simplification rules (Table 3.3). The rewriting system is designed to be confluent, which means that the order of application of the rules does not have an impact on the final outcome (see [Baader and Nipkow, 1998] for more information on rewriting systems).

As an example, we derive the optimized $\sum$-OL expression in Equation 3.2 from the initial naïve OL algorithm (in Equation 3.1). The steps are annotated with the rewriting rules used:

$$\begin{aligned}
&\big(R_k \otimes (K_{m\times 1} \otimes K_{1\times n})\big) \circ \big(L_k^{mk} \times I_{kn}\big) \\
&\to \Big(\sum_{0\le p<k} (K_{m\times 1} \otimes K_{1\times n}) \circ (\mathrm{G}(h_{pm,1}^{m\to km}) \times \mathrm{G}(h_{pn,1}^{n\to kn}))\Big) \circ \big(\mathrm{perm}(\ell_k^{mk\to mk}) \times \mathrm{perm}(\imath^{kn\to kn})\big) && \text{(3.7), (3.21)} \\
&\to \sum_{0\le p<k} (K_{m\times 1} \otimes K_{1\times n}) \circ (\mathrm{G}(\ell_k^{mk\to mk} \circ h_{pm,1}^{m\to km}) \times \mathrm{G}(\imath^{kn\to kn} \circ h_{pn,1}^{n\to kn})) && \text{(3.13), (3.15), (3.18)} \\
&\to \sum_{0\le p<k} (K_{m\times 1} \otimes K_{1\times n}) \circ (\mathrm{G}(h_{p,k}^{m\to km}) \times \mathrm{G}(h_{pn,1}^{n\to kn})) && \text{(3.23),(3.27)} \\
&\to \sum_{0\le p<k} \Big(\sum_{0\le i<m} \mathrm{S}(h_{in,1}^{n\to mn}) \circ K_{1\times n} \circ (\mathrm{G}(h_{i,1}^{1\to m}) \times \mathrm{G}(h_{0,1}^{n\to n}))\Big) \circ (\mathrm{G}(h_{p,k}^{m\to km}) \times \mathrm{G}(h_{pn,1}^{n\to kn})) && \text{(3.9)} \\
&\to \sum_{0\le p<k} \sum_{0\le i<m} \mathrm{S}(h_{in,1}^{n\to mn}) \circ K_{1\times n} \circ (\mathrm{G}(h_{p,k}^{m\to km} \circ h_{i,1}^{1\to m}) \times \mathrm{G}(h_{pn,1}^{n\to kn} \circ \imath^{n\to n})) && \text{(3.13), (3.15), (3.17), (3.25)} \\
&\to \sum_{0\le p<k} \sum_{0\le i<m} \sum_{0\le j<n} \mathrm{S}(h_{in,1}^{n\to mn} \circ h_{j,1}^{1\to n}) \circ P_1 \circ (\mathrm{G}(h_{ik+p,k}^{1\to km} \circ \imath^{1\to 1}) \times \mathrm{G}(h_{pn,1}^{n\to kn} \circ h_{j,1}^{1\to n})) \\
&\qquad \text{(3.11), (3.13), (3.15), (3.17), (3.24), (3.27), (3.25)} \\
&\to \sum_{0\le p<k} \sum_{0\le i<m} \sum_{0\le j<n} \mathrm{S}(h_{in+j,1}^{1\to mn}) \circ P_1 \circ (\mathrm{G}(h_{ik+p,k}^{1\to km}) \times \mathrm{G}(h_{pn+j,1}^{1\to kn})) && \text{(3.24), (3.27)}
\end{aligned}$$

The final expression shows various properties of $\sum$-OL: it is compact, mathematical, declarative and close to imperative code but it is still point-free (no references to the input and output arrays).

3.1.2 Recursion Step Closure

The computation of the recursion step closure is probably the most critical step of the library core generation. It was first introduced in the context of library generation for linear transforms by

[Voronenko, 2008] who recognized that $\sum$-SPL could be used not only to manipulate codelets but also purely symbolic expressions. In this thesis, we extend the concept and derivation to OL and will therefore present it using a matrix multiplication library as example.

The recursion step closure derivation is the set of mutually recursive functions necessary to "efficiently" implement the target interface. The recursion step closure is derived automatically using a rewriting system expressed in $\sum$-OL which identifies the auxiliary functions needed and also derives their implementations. The resulting set of mutually recursive functions depends on both the actual algorithms used and the platform parameters set. Efficient means that it minimizes the number of data reorderings needed to compute the target functionality. In many aspects what is really performed during the computation of the closure is *loop merging across function prototypes*. We note that in some cases (Prefetching, TLB optimizations), the literature suggests that some copying of the data actually boosts performance ([Goto and van de Geijn, 2008; Temam et al., 1993]). This could be modeled in our framework but is not implemented in this thesis.

3.1.2.1 Example

We first explain the closure computation with an example. Assume we again want to create a *recursive* matrix-multiplication library which performs $C = AB$ where the matrices dimensions are $m \times k$ and $k \times n$ and all matrices are tightly packed in a row-major order. The associated interface is:

```
MMM(int m, int k, int n, double* A, double* B, double* C);
```

The pseudocode of a blocked implementation is given on the left column of Figure 3.4a. Since matrices have to be tightly packed to fit the interface, explicit copies need to be introduced to "compact" blocks that are scattered in memory into contiguous buffers, so that the same function can be called recursively. These copies are a direct consequence of the chosen interface. To optimize this code, we start by extracting the body of the loop and "outlining" it; we will call this new auxiliary function `T` (left column of Figure 3.4b). While this intermediary step does not in itself remove inefficient copies, it reveals that `T` is actually a performing a matrix multiplication, only on matrices that are scattered in memory. Therefore, blocking could be applied again, yielding a self-contained recursive implementation of `T` (left column of Figure 3.4c). Note that there is not a single explicit copy left in this optimized version.

Using $\sum$-OL, the exact same optimization can be derived *automatically* as we sketch in the right columns of the figures. In all three figures, the $\sum$-OL rule in the right column precisely specifies the imperative code in the left column. In the right column of Figure 3.4a, we show the smallest base case for the matrix-matrix multiplication and a $\sum$-OL translation of the OL matrix blocking algorithm of Equation 2.19. The loop body is exactly the argument $\mathrm{S}_{\ldots} \circ \mathbf{MMM}_{\ldots} \circ \mathrm{G}_{\ldots} \times \mathrm{G}_{\ldots}$ of the nested $\sum$-OL sums. Note how in Figure 3.4a, the gathers and scatters are performed explicitly in the corresponding code. Pulling out the loop body as $\mathbf{T}$ yields Figure 3.4b. Using a simple

```
MMM(m, k, n, A, B, C){
   if m = k = n = 1
     C[0] = A[0]*B[0]
   else
     determine mb nb kb
     loop i = 0 ..  m/mb
       loop j = 0 ..  n/nb
         loop p = 0 ..  k/kb
           Ab <- submatrix(A, ...)
           Bb <- submatrix(B, ...)
           Cb <- 0
           MMM(mb,kb,nb,Ab,Bb,Cb)
           submatrix(C, ...)  += Cb}
```

$$\begin{cases} \mathbf{MMM}_{1,1,1} \rightarrow P_1 \\ \mathbf{MMM}_{m,k,n} \rightarrow \\ \quad \sum_{i=0}^{m/m_b} \sum_{j=0}^{n/n_b} \sum_{p=0}^{k/k_b} \mathrm{S}_{\ldots} \circ \mathbf{MMM}_{m_b,k_b,n_b} \circ \mathrm{G}_{\ldots} \times \mathrm{G}_{\ldots} \end{cases}$$

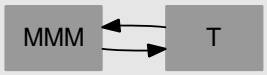
(a) Naïve version. Inefficient explicit copies are needed to collect blocks scattered in memory into contiguous buffers.

```
MMM(m, k, n, A, B, C){
   if m = k = n = 1
     C[0] = A[0]*B[0]
   else
     determine mb nb kb
     loop i = 0 ..  m/mb
       loop j = 0 ..  n/nb
         loop p = 0 ..  k/kb
           T(mb,kb,nb,A,B,C,...)}

T(mb, kb, nb, A, B, C, ...){
   Ab <- submatrix(A, ...)
   Bb <- submatrix(B, ...)
   Cb <- 0
   MMM(mb,kb,nb,Ab,Bb,Cb)
   submatrix(C, ...)  += Cb}
```

$$\begin{cases} \mathbf{MMM}_{1,1,1} \rightarrow P_1 \\ \mathbf{MMM}_{m,k,n} \rightarrow \sum_{i=0}^{m/m_b} \sum_{j=0}^{n/n_b} \sum_{p=0}^{k/k_b} \mathbf{T}_{m_b,k_b,n_b,\ldots} \end{cases}$$

$$\begin{cases} \mathbf{T}_{m_b,k_b,n_b,\ldots} \rightarrow \\ \quad \mathrm{S}_{\ldots} \circ \mathbf{MMM}_{m_b,k_b,n_b} \circ \mathrm{G}_{\ldots} \times \mathrm{G}_{\ldots} \end{cases}$$



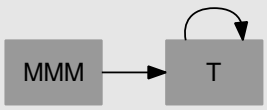
(b) Intermediate step. The body of the loop has been put in a different function that we call `T`.

```
MMM(m, k, n, A, B, C){
   if m = k = n = 1
     C[0] = A[0]*B[0]
   else
     determine mb nb kb
     loop i = 0 ..  m/mb
       loop j = 0 ..  n/nb
         loop p = 0 ..  k/kb
           T(mb,kb,nb,A,B,C, ...)}

T(mb, kb, nb, A, B, C, ...){
   if mb = kb = nb = 1
     C[i*n+j]+=A[i*k+p]*B[p*n+j];
   else
     determine mb2 nb2 kb2
     loop i2 = 0 ..  mb/mb2
       loop j2 = 0 ..  nb/nb2
         loop p2 = 0 ..  pp<kb/kb2
           T(mb2,kb2,nb2,A,B,C,...)}
```

$$\begin{cases} \mathbf{MMM}_{1,1,1} \rightarrow P_1 \\ \mathbf{MMM}_{m,k,n} \rightarrow \sum_{i=0}^{m/m_b} \sum_{j=0}^{n/n_b} \sum_{p=0}^{k/k_b} \mathbf{T}_{m_b,k_b,n_b,\ldots} \end{cases}$$

$$\begin{cases} \mathbf{T}_{1,1,1,\ldots} \rightarrow \mathrm{S}_{\ldots} \circ P_1 \circ \mathrm{G}_{\ldots} \times \mathrm{G}_{\ldots} \\ \mathbf{T}_{m_b,k_b,n_b,\ldots} \rightarrow \\ \quad \sum_{i_2=0}^{m_b/m_{b2}} \sum_{j_2=0}^{n_b/n_{b2}} \sum_{p_2=0}^{k_b/k_{b2}} \mathbf{T}_{m_{b2},k_{b2},n_{b2},\ldots} \end{cases}$$



(c) Optimized version. The former body of the loop (`T`) is recognized as a function that multiplies matrices scattered in memory (i.e., with optional leading dimension). Not a single explicit copy is needed anymore.

FIGURE 3.4: Recursion step closure. Three different recursive matrix multiplication implementations are displayed, simultaneously in imperative code (left) and in $\sum$-OL (top right). The corresponding static call graphs are given (bottom right). The *closure* derives the optimized version (c) from the naïve one (a).

derivation, one can now discover a new self-contained rule for $\mathbf{T}$ (all parameters are omitted for clarity):

$$\begin{aligned}\mathbf{T} &= \mathrm{S} \circ \mathbf{MMM} \circ \mathrm{G} \times \mathrm{G} \\ &\rightarrow \mathrm{S} \circ (\sum\sum\sum \mathrm{S} \circ \mathbf{MMM} \circ \mathrm{G} \times \mathrm{G}) \circ (\mathrm{G} \times \mathrm{G}) \\ &\rightarrow \sum\sum\sum (\mathrm{S} \circ \mathbf{MMM} \circ \mathrm{G} \times \mathrm{G}) \\ &\rightarrow \sum\sum\sum \mathbf{T}\end{aligned}$$

Indeed, applying the MMM rule of Figure 3.4a inside the definition of $\mathbf{T}$ and rewriting using Table 3.2 and Table 3.3 yields an optimized algorithm (right part of Figure 3.4c) for $\mathbf{T}$ that does not require explicit copies. The resulting closure consists of both the functions $\mathbf{MMM}$ and $\mathbf{T}$.

This example illustrates how the recursion step closure can be derived automatically so that the mutually recursive functions that compose the library avoid explicit data movements. This optimization should be of critical importance for a number of applications that are forced to reuse existing performance libraries because redevelopment is expensive. For instance, the quantum chemistry software NWChem [Bylaska et al., 2007] uses code that calls a general matrix-multiplication but precedes and succeeds the call by transpositions which seriously impacts the performance. Using the above procedure, the development effort would be reduced and these transpositions could be folded into the library for potentially major performance gains.

3.1.2.2 General Procedure

We provide now the general procedure to derive the recursion step closure. Extending the original recursion step closure algorithm [Voronenko, 2008] for OL is relatively straightforward. More precisely, the preceding and following layers of the compilation stack were engineered so that this step would be almost exactly compatible with the existing procedure in the linear transform library generator.

The input is a functionality $\mathbf{F}$ and a set of OL rules R. The output is a set of $\sum$-OL expressions that characterizes all the recursion steps required in the generated library and a set of $\sum$-OL rules for these recursion steps that capture all possible recursions.

Figure 3.5 presents the global procedure for computing the recursion step closure in detail: one needs to try all OL rules on each recursion step which can optionally produce new recursion steps. When no new recursion steps are found, the closure is reached.

We provide now some additional details for the application of rules and possible generation of new recursion steps. As depicted in Figure 3.5, it happens in a loop of three successive steps:

1. Application of a given OL breakdown rule to a recursion step.

2. Rewriting of the OL formula into a $\sum$-OL formula using Table 3.1, Table 3.2 and Table 3.3.

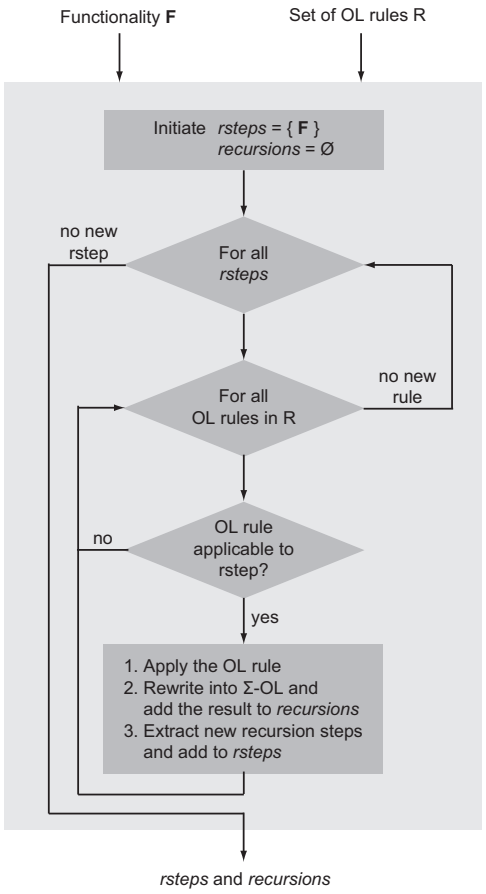


FIGURE 3.5: Derivation of the Recursion Step Closure.

3. Extraction of the recursion steps.

The last step is explained next.

3.1.2.3 Recursion Step Extraction

The recursion step extraction is responsible for the generation of new recursion steps directly from $\sum$-OL formulas. For example, in Figure 3.4, the additional recursion step that is automatically derived is $\mathbf{T}$. Since the overall goal is to avoid data permutations, new recursion steps are obtained by fusing existing recursion steps with adjacent data accesses (scatters and gathers). In $\sum$-OL, this process happens in two phases: first, the new recursion steps are *selected* using a rewriting system and second, the new recursion steps are *parametrized* so that they can be turned into proper operators.

$$\{\mathrm{A}\} \circ \mathrm{G} \to \{\mathrm{A} \circ \mathrm{G}\} \tag{3.28}$$
$$\{\mathrm{A}\} \circ (\mathrm{G} \times \mathrm{G}) \to \{\mathrm{A} \circ (\mathrm{G} \times \mathrm{G})\} \tag{3.29}$$
$$\mathrm{S} \circ \{\mathrm{A}\} \to \{\mathrm{S} \circ \mathrm{A}\} \tag{3.30}$$

TABLE 3.4: Recursion Step Selection Rules.

Recursion step selection. The first stage of the extraction marks the parts of the OL formula that are to be encapsulated into their own recursion steps. In practice, this is done by a simple rewriting system whose target is to locate clusters comprised of functionalities and their corresponding data access. To represent these clusters, we introduce special delimiters inside $\sum$-OL that we denote by curly brackets $\{\}$ and that mark the boundaries of the future recursion step.

For example, we consider the full $\sum$-OL expression corresponding to the rule displayed to the right of Figure 3.4a. We first initiate the rewriting process by enclosing all functionalities within the brackets and then rewrite using the rules of Table 3.4:

$$\sum_{i=0}^{m/m_b} \sum_{j=0}^{n/n_b} \sum_{p=0}^{k/k_b} \mathrm{S}(h_{im_b,1}^{m_b \to m} \otimes h_{jn_b,1}^{n_b \to n}) \circ \Big\{ \mathbf{MMM}_{m_b,k_b,n_b} \Big\} \circ \big(\mathrm{G}(h_{im_b,1}^{m_b \to m} \otimes h_{pk_b,1}^{k_b \to k}) \times \mathrm{G}(h_{pk_b,1}^{k_b \to k} \otimes h_{jn_b,1}^{n_b \to n}) \big)$$
$$\to \sum_{i=0}^{m/m_b} \sum_{j=0}^{n/n_b} \sum_{p=0}^{k/k_b} \Big\{ \mathrm{S}(h_{im_b,1}^{m_b \to m} \otimes h_{jn_b,1}^{n_b \to n}) \circ \mathbf{MMM}_{m_b,k_b,n_b} \circ \big(\mathrm{G}(h_{im_b,1}^{m_b \to m} \otimes h_{pk_b,1}^{k_b \to k}) \times \mathrm{G}(h_{pk_b,1}^{k_b \to k} \otimes h_{jn_b,1}^{n_b \to n}) \big) \Big\}.$$

As it can be seen, all nearby data permutations are pulled inside the newly delimited recursion step. Observe that it prefigures the recursion step that we called $\mathbf{T}$ in Figure 3.4b.

Recursion step parametrization. The recursion step selection marks a subexpression of a $\sum$-OL expression but does not define a proper recursion step yet since no binding information has been provided. In an imperative language, the difference between the two would be understood as the difference between a macro and a function.

The *parametrization* creates a proper recursion step out of a given $\sum$-OL formula. It does so by 1) replacing all expressions by new variables and 2) enforcing $\sum$-OL constraints between the new variables. We explain the procedure continuing with our previous example.

First, all existing variables except constants are replaced by fresh ones:

$$\Big\{ \mathrm{S}(h_{u_3,1}^{u_1 \to u_2} \otimes h_{u_6,1}^{u_4 \to u_5}) \circ \mathbf{MMM}_{u_7,u_8,u_9} \circ \big(\mathrm{G}(h_{u_{12},1}^{u_{10} \to u_{11}} \otimes h_{u_{15},1}^{u_{13} \to u_{14}}) \times \mathrm{G}(h_{u_{18},1}^{u_{16} \to u_{17}} \otimes h_{u_{21},1}^{u_{19} \to u_{20}}) \big) \Big\}.$$

Second, for the above expression to be a valid $\sum$-OL expression, compositions must have domains and ranges that match, which creates additional constraints. The first composition yields two equalities, one for each dimension: $u_1 = u_7$ and $u_4 = u_9$. The second composition yields four

equalities (two dimensions times two inputs): $u_7 = u_{10}$, $u_8 = u_{13}$, $u_8 = u_{16}$, and $u_9 = u_{19}$.

We resolve the system by substitution and we can therefore properly define the auxiliary recursion step $\mathbf{T}$:

$$\mathbf{T}_{u_2,u_3,u_5,u_6,u_7,u_8,u_9,u_{11},u_{12},u_{14},u_{15},u_{17},u_{18},u_{20},u_{21}} = \\ \mathrm{S}(h_{u_3,1}^{u_7\to u_2} \otimes h_{u_6,1}^{u_8\to u_5}) \circ \mathbf{MMM}_{u_7,u_8,u_9} \circ \big(\mathrm{G}(h_{u_{12},1}^{u_7\to u_{11}} \otimes h_{u_{15},1}^{u_8\to u_{14}}) \times \mathrm{G}(h_{u_{18},1}^{u_8\to u_{17}} \otimes h_{u_{21},1}^{u_9\to u_{20}})\big). \tag{3.31}$$

Observe that the recursion step $\mathbf{T}$ might seem complicated at first glance but it is nothing more than a more general matrix multiplication that operates on matrices scattered in memory. As seen in the above example, the advantage to using $\mathbf{T}$ is that it is naturally self-contained.

3.1.3 Hot and Cold Partitioning

The hot and cold partitioning of the parameters is the second block of the library structure derivation in Figure 3.1. It analyzes the closure and partitions the different recursion step parameters into two groups that each need to be initialized simultaneously.

Indeed, recursion steps are actual mathematical functions but traditional imperative languages like C are too restrictive to capture their functional nature. In particular, one could want to curry them so that two different recursion steps share some precomputations when some of their parameters are similar. Therefore, object languages such as C++ are more suitable to describe recursion steps since they directly map to classes.

In the context of adaptive library generation, one precomputation that is interesting to share is the *search*. Indeed, some parameters, called the *hot* parameters (following [Voronenko, 2008]), do not impact the performance so it is interesting to do the search without taking them into account. By opposition, the parameters that influence the search are called the *cold* parameters.

As illustrating example, we consider the recursion step $\mathbf{MMM}_{m,k,n}$ and its implementation `MMM(m, k, n, A, B, C)` from Figure 3.4a. Obviously, the algorithm choice can heavily depend on the sizes of the input matrices `m`, `k` and `n`, which therefore should be cold. Simultaneously, the performance is oblivious to the actual values of `A`, `B` and `C`, which are therefore hot[5]. From a functional language point of view, this observation means that `MMM(m, k, n)(A, B, C)` is the "natural" curried form of `MMM(m, k, n, A, B, C)` since it makes sense to search for the best matrix multiplication of some size regardless of the location of the matrices.

A possible C++ implementation of the curried form is given next (the actual implementation of the search mechanisms is delayed until Section 4.2).

[5]In fact, the pointers are so hot that they have been entirely implicit until now. Note that this is not exactly true if the target code is vectorized since proper alignment is required. However, under the assumption that the pointers are aligned, the above assumption is reasonable.

```
class MMM{
  int m, k, n;
  MMM(m, k, n);
  compute(double* A, double* B, double* C);
};
```

The hot/cold partitioning is responsible for automatically assigning all recursion step parameters a category, either cold or hot. We use the iterative algorithm proposed in [Voronenko, 2008], which also has a precise description. Basically, considering the closure as a whole, the algorithm maximizes the number of cold parameters, under the following constraints:

1. Pointers are hot,
2. Parameters that depend on loop variables are hot, and
3. Parameters that depend on hot parameters are hot.

Note that while hot/cold partitioning might have been seemed simple for $\mathbf{MMM}_{m,k,n}$ alone, other recursion steps are more challenging. For instance, one can verify that the correct partition for the recursion step $\mathbf{T}$ in Equation 3.31 according to the above rules is `T(u2, u5, u7, u8, u9, u11, u14, u17, u20)(u3, u6, u12, u15, u18, u21, A, B, C)`.

3.1.4 Library Plan

The last step in the library structure generation (Figure 3.1) is the generation of the *library plan* which is a data structure that entirely describes the library. All recursion steps are successively described in details: their formula, their parameters partition and their rules are specified. Rules are themselves characterized by their conditions of applicability, their degree of freedom and their associated formulas.

In Table 3.5, we show the library plan corresponding to our running example. Note that, for readability, we use the C function call notation to bind variables to recursion steps parameters but it is not exactly proper since it hides the hot and cold partial evaluation.

3.2 Library Implementation

The library plan contains all recursion steps and makes explicit how they recurse into each other using $\sum$-OL. This section explains how to turn the plan into code, which requires three main tasks to be performed for each recursion step (Figure 3.1): First, the code corresponding to the recursions needs to be generated (left block); second, one must determine and generate base cases to terminate the recursions (right block); finally, the recursions and the base cases need to be integrated into a complete library.

RS 1

Formula: $\mathrm{MMM}_{u_1,u_2,u_3}$

Cold Parameters: u_1, u_2, u_3

Hot Parameters: input and output pointers

Rule 1: MMM-Base

* *Applicability*: $u_1 = 1$, $u_2 = 1$, and $u_3 = 1$
* *Freedoms*: none
* *Formula*: P_1

Rule 2: MMM-Block

* *Applicability*: $u_1 > 1$, $u_2 > 1$, or $u_3 > 1$
* *Freedoms*: $f_1 \in \mathrm{divisors}(u_1)$, $f_2 \in \mathrm{divisors}(u_2)$, and $f_3 \in \mathrm{divisors}(u_3)$
* *Formula*: $\sum_{i=0}^{u_1/f_1} \sum_{j=0}^{u_3/f_3} \sum_{p=0}^{u_2/f_2} RS2(u_1, u_3, f_1, f_2, f_3, u_1, u_2, u_2, u_3, if_1, jf_3, if_1, pf_2, pf_2, jf_3)$

RS 2

Formula: $\mathrm{S}(h_{u_3,1}^{u_7 \to u_2} \otimes h_{u_6,1}^{u_8 \to u_5}) \circ \mathbf{MMM}_{u_7,u_8,u_9} \circ \big(\mathrm{G}(h_{u_{12},1}^{u_7 \to u_{11}} \otimes h_{u_{15},1}^{u_8 \to u_{14}}) \times \mathrm{G}(h_{u_{18},1}^{u_8 \to u_{17}} \otimes h_{u_{21},1}^{u_9 \to u_{20}}) \big)$

Cold Parameters: $u_2, u_5, u_7, u_8, u_9, u_{11}, u_{14}, u_{17}, u_{20}$

Hot Parameters: $u_3, u_6, u_{12}, u_{15}, u_{18}, u_{21}$, input and output pointers

Rule 1: MMM-Base

* *Applicability*: $u_7 = 1$, $u_8 = 1$, and $u_9 = 1$
* *Freedoms*: none
* *Formula*: $\mathrm{S}(h_{u_3,1}^{1 \to u_2} \otimes h_{u_6,1}^{1 \to u_5}) \circ P_1 \circ \big(\mathrm{G}(h_{u_{12},1}^{1 \to u_{11}} \otimes h_{u_{15},1}^{1 \to u_{14}}) \times \mathrm{G}(h_{u_{18},1}^{1 \to u_{17}} \otimes h_{u_{21},1}^{1 \to u_{20}}) \big)$

Rule 2: MMM-Block

* *Applicability*: $m > 1$, $k > 1$, or $n > 1$
* *Freedoms*: $f_1 \in \mathrm{divisors}(u_7)$, $f_2 \in \mathrm{divisors}(u_8)$, and $f_3 \in \mathrm{divisors}(u_9)$
* *Formula*: $\sum_{i=0}^{u_7/f_1} \sum_{j=0}^{u_9/f_3} \sum_{p=0}^{u_8/f_2} RS2(u_2, u_5, f_1, f_2, f_3, u_{11}, u_{14}, u_{17}, u_{20}, u_3 + if_1, u_6 + jf_3, u_{12} + if_1, u_{15} + pf_2, u_{18} + pf_2, u_{21} + jf_3)$

TABLE 3.5: Library plan corresponding to the closure in Figure 3.4c.

We start this chapter by describing the $\sum$-OL compiler which is used to compile $\sum$-OL expressions into a C-like internal language that we simply call "code." Note that our target language, C, C++, or Java, is other compilers source language, which means that we leave other compilers carry the library the last mile, from source code to byte code[6]. We then explain in details the base case generation procedure and conclude with an overview of the source-to-source optimizations that are performed in the compiler.

3.2.1 Sigma-OL Compiler

The $\sum$-OL compiler is responsible for the transformation of $\sum$-OL expressions into pieces of code. One of the characteristics of the operation is that it requires to go from "point-free" to "point-

[6]Attempts have been made to actually generate assembly directly from Spiral but there is great hassle and actually little benefit in doing so since most local optimizations are already performed well by optimizing compilers. Global optimizations, on the other hand, are carried out more efficiently if they had it been done earlier, higher up inside Spiral.

wise"[7], which means that the data arrays that the functions were implicitly working on need to be explicited. The overall task is performed using parametrized code templates that gradually replace the mathematical expressions in the $\sum$-OL expression tree top-down. Some of these templates are presented in Table 3.6.

As example, consider the following $\sum$-OL expression from the rule in the right column of Figure 3.4a:

$$\sum_{i=0}^{m/m_b}\sum_{j=0}^{n/n_b}\sum_{p=0}^{k/k_b} \mathrm{S}(h_{im_b,1}^{m_b\to m}\otimes h_{jn_b,1}^{n_b\to n})\circ \mathbf{MMM}_{m_b,k_b,n_b}\circ\Big(\mathrm{G}(h_{im_b,1}^{m_b\to m}\otimes h_{pk_b,1}^{k_b\to k})\times \mathrm{G}(h_{pk_b,1}^{k_b\to k}\otimes h_{jn_b,1}^{n_b\to n})\Big).$$

When compiling it, the input and output arrays `A`, `B` and `C` are reintroduced and the following code is produced:

```
for (int i=0;i<m/mb;i++)
  for (int j=0;j<n/nb;j++)
    for (int p=0;p<k/kb;p++){
      for (int r=0;r<mb*kb;r++)
        Ap[r]=A[(i*mb*k)+p*kb+(r/mb)*k+(r%mb)];
      for (int r=0;r<kb*nb;r++)
        Bp[r]=B[p*kb*n+j*nb+(r/nb)*n+(r%nb)];
      for (int t=0;t<mb*nb;t++)
        Cp[t]=0;
      mmm(mb,kb,nb,Ap,Bp,Cp);
      for (int r=0;r<mb*nb;r++)
        C[i*mb*n+j*nb+(r/nb)*n+(r%nb)]+=Cp[r];
    }
```

Observe how the recursive call to $\mathbf{MMM}$ in $\sum$-OL is transformed into a function call to `mmm` in the imperative code[8].

3.2.2 Base Case Generation

Besides the implementation of recursions, the compiler is also responsible for selecting which base cases to implement and compiling them properly. We explain this process using the $\mathbf{MMM}_{m,k,n}$ recursion step Table 3.5 but the method is generally analogous to the one used in [Voronenko, 2008] for transforms.

A list of sizes is first specified by the user for the main functionality; for instance, $(m,k,n)\in\{1,2\}\times\{1,2\}\times\{1,2\}$. This triggers the offline search system to search for the best implementations of $\mathbf{MMM}_{1,1,1}$, $\mathbf{MMM}_{1,1,2}$, $\mathbf{MMM}_{1,2,1}$, $\mathbf{MMM}_{1,2,2}$, $\mathbf{MMM}_{2,1,1}$, $\mathbf{MMM}_{2,1,2}$, $\mathbf{MMM}_{2,2,1}$, and $\mathbf{MMM}_{2,2,2}$, using the set of rules that have been enabled inside the algorithmic pool.

[7] These expressions are quite common in the Haskell community.

[8] Again, we use a C call notation which is not entirely proper since `mb`, `kb` and `nb` are cold parameters that therefore need to be initialized before the hot parameters.

Operator S	Code
arity (1,1)	*code for* $y = \mathrm{S}(x)$
$\mathrm{G}(f^{d\to r})$	`for (i=0; i<d; i++)` `  y[i] = x[f(i)];`
$\mathrm{S}(f^{d\to r})$	`for (i=0; i<d; i++)` `  y[f(i)] += x[i];`
$\mathrm{A}\circ\mathrm{B}$	`<code for t = B x>` `<code for y = A t>`
$\sum_{0\le i<k}\mathrm{A}_i$	`for (i=0; i<k; i++)` `  {<code for y = A_i x>}`
arity (2,1)	*code for* $y = \mathrm{S}(x_1, x_2)$
P_1	`y[0] = x_1[0] * x_2[0];`
$\sum_{0\le i<k}\mathrm{A}_i$	`for (i=0; i<k; i++)` `  {<code for y = A_i (x_1, x_2)>}`
arity (2,2)	*code for* $(y_1, y_2) = \mathrm{S}(x_1, x_2)$
$\mathrm{A}\times\mathrm{B}$	`<code for y_1 = A x_1>` `<code for y_2 = B x_2>`

TABLE 3.6: Templates to translate $\sum$-OL to code. A and B are assumed to be generic operators.

In fact, these implementations are also inserted into all the other recursion steps as well, since all of them are variants of $\mathbf{MMM}$. In our example, it means that eight base cases would also be generated for $\mathbf{T}_{m_b,k_b,n_b,\ldots}$. Note that, in particular, if a recursion step is looped, then the implementations naturally become looped base cases.

For instance, assume that the compiler is building an implementation for $\mathbf{MMM}_{2,1,2}$. Depending on the available rules, one of the option is to break down to the naïve matrix multiplication from earlier:

$$\sum_{0\le p<1}\sum_{0\le i<2}\sum_{0\le j<2}\mathrm{S}(h^{1\to 4}_{2i+j,1})\circ P_1\circ(\mathrm{G}(h^{1\to 2}_{i+p,1})\times\mathrm{G}(h^{1\to 2}_{2p+j,1}))$$

After replacement by the parametrized templates from Table 3.6, the rewriting yields code that is functionally correct but is clearly not optimized (Figure 3.6).

This source code and the other alternatives to implement $\mathbf{MMM}_{2,1,2}$ are then compiled using an external optimizing compiler. They are then timed on the target platform and only the fastest implementation remains in the library[9].

In practice, external optimizing compilers do not provide the best performance for code such

[9]Actually, one is not forced to do this selection stage offline: the search mechanism can entirely support multiple implementations of the same base case and automatically search for the fastest at runtime.

```
for (p=0; p<1; p++)
   for (i=0; i<2; i++)
      for (j=0; j<2; j++) {
         for (q1=0; q1<1; q1++)
            t1[q1] = A[i+p+q1];
         for (q2=0; q2<1; q2++)
            t2[q2] = B[p*2+j+q2];
         t3[0] = t1[0] * t2[0];
         for (q3=0; q3<1; q3++)
            C[2i+j+q3] += t3[q3];
      }
```

FIGURE 3.6: A base case for $\mathbf{MMM}_{m,k,n}$ that has not been yet optimized. It implements $\mathbf{MMM}_{2,1,2}$. The seemingly "extra" `for` loops correspond to the gathers and the scatters (Table 3.6).

as the one that comes directly out of the rewriting stage. We therefore pipe the result into our own source-to-source optimizing compiler which is the topic of the next subsection.

3.2.3 Source-to-Source Optimizer

Traditionally, there has been two complementary strategies to obtain performance in computation kernels: on one hand, recursive algorithms designers tend to fully unroll their base cases in order to reduce the control flow overhead and increase the instruction level parallelism of the computation. On the other hand, experts writing iterative algorithms usually try to develop small kernels that function at peak performance inside a loop by carefully allocating the computing resources [Agner, 2010; Intel, 2009c].

The source-to-source optimizer is the component of the library generator that ensures that both types of code can be generated and are efficient. It produces better code directly from code by successively applying a series of optimizations. Note that the source-to-source optimizer does not need to perform the entire palette of optimizations since it is ultimately backed up by an external optimizing source-to-bytecode compiler. Therefore, it only makes sense to implement those optimizations where either we possess more information than compilers or where compilers do not handle properly the kind of code we generate.

The optimizer does not attempt to perform any advanced restructurings since doing so would defeat the purpose of having a separate algorithmic space search mechanism. In particular, optimizations such as vectorization or parallelization are expressed and performed in the algorithm space (see next section) and are therefore already performed during the $\sum$-OL compilation stage.

This section briefly overviews the various optimizations that are performed. These are fairly common in the compiler literature (see, for instance, [Allen and Kennedy, 2002]) and in prior work on library generation [Frigo, 1999; Rizzolo and Padua, 2004; Whaley and Dongarra, 1998; Xiong, 2001].

```
if ((m == 2) && (k == 1) && (n == 2)) {
    double a139, a140, a141, a142, s45, s46,
           s47, s49, s50, s52, s53, s55;
    s45 = A[0];
    s46 = B[0];
    s47 = (s45 * s46);
    a139 = C[0];
    s49 = A[1];
    s50 = (s49 * s46);
    a140 = C[2];
    s52 = B[1];
    s53 = (s45 * s52);
    a141 = C[1];
    s55 = (s49 * s52);
    a142 = C[3];
    C[0] = (a139 + s47);
    C[2] = (a140 + s50);
    C[1] = (a141 + s53);
    C[3] = (a142 + s55);
    return;
}
```

FIGURE 3.7: An unrolled base case for $\mathbf{MMM}_{m,k,n}$. It implements $\mathbf{MMM}_{2,1,2}$ and has been generated by Spiral (which explains the seemingly random naming of the variables). Note that the external `if` statement constrains the applicability conditions of the base case to what it can really handle.

Basic optimizations. The code presented above in Figure 3.6 has obvious shortcomings: single iteration loops and unscalarized arrays (which reduces the instruction level parallelism). The optimizer thus performs array scalarization on all internal arrays and unrolls all loops that have a constant number of iterations. The compiler also uses common subexpression elimination to minimize the number of index operations done. Whenever needed, it also converts the code to the single static assignment form (SSA). For our previous code, the final result can be seen in Figure 3.7.

Advanced optimizations. Basic optimizations are sufficient to reproduce the kind of code that is generated by unrolling recursions, that is, vanilla base cases. However, the next section will introduce a new operator that captures an entire loop nest. Plugging back base cases inside them therefore produces looped base cases, which is code that is traditionally thought of as being iterative. An additional set of optimizations are then required: loop induction variable manipulations (detection, fusion, associated strength reduction and rematerialization) and loop invariants hoisting (including index computation minimization).

Code generation. Ultimately, code is generated from the library core using a modular unparser. Efficient online search and precomputation for linear transforms suggest a language that supports function closures. For this reason, we target C++[10] as the original system for transforms. Further, we

[10]Object-oriented programming is equivalent to function closures.

support library generation in Java, which is widely used, mainly for its ease of access and security features.

Note that, despite that object-oriented languages are highly preferable for use with online search mechanisms, some search mechanisms have been ported to support simpler languages such as C.

3.3 Parallelism

One goal of the library generator is to leverage the on-chip parallelism which means that the generated code should be *threaded* and *vectorized*. We capture these optimizations at the OL level which is the "right" level of abstraction since it is upstream of the recursion step closure. The closure can then automatically discover and formalize the additional recursion steps that need to be introduced.

Note that this choice of capturing vectorization and parallelization as *just any other algorithm* in our system has an important consequence: since we start by producing additional recursion steps, the profitability of vectorization and parallelization cannot be evaluated during the library generation. The decision of applying a given algorithm or not is left as a choice that is decided later by the search mechanisms (which will be described in Chapter 4).

In order to capture parallelism in our framework, we introduce a new tool, the *generalized tensor* for the OL notation. It can capture and manipulate loops of OL objects and logically flows from the generalized tensor for the SPL notation introduced in [Voronenko, 2008]. Our notation is then used to provide both parallelization and vectorization in a way that earlier work is leveraged, particularly the vector and parallel compilers developed in [Franchetti et al., 2006a,c].

3.3.1 Generalized Tensor

The generalized tensor (GT) is an OL construct that serves the purpose of describing loop nests (loops, loops of loops and so on) in a way that is functional (index free) to allow for easier handling by the rewriting engine.

Rank-1 GT. A rank-1 GT captures a single $\sum$-OL loop:

$$\mathbf{GT}(\mathrm{A}, g, s, [k]) = \sum_{i=0}^{k-1} \mathrm{S}(s_i) \circ \mathrm{A} \circ \mathrm{G}(g_i) \tag{3.32}$$

In this expression, A is the kernel, k is the number of iterations and s_i and g_i are scatter and gather index mapping functions like the ones we defined before. The functions s and g are defined in the functional programming paradigm [Johnsson, 1985]: they are the λ-*lifted* versions of s_i and g_i which means, in classical terms, that while s_i and g_i are functions, s and g are functions of functions that map i to s_i and g_i. Since they have one unbound variable, we say that g and s are rank-1 functions. Analogously, g_i and s_i are called rank-0 functions. In essence, $\mathbf{GT}(\mathrm{A}, g, s, [k])$ in Equation 3.32 expresses the sum in an index-free form, i.e., without the index variable i.

Ranks and downranks. We define the rank-k GT as an OL operator that can capture k nested loops. The *downranking* rule converts a rank-k GT into a loop of rank-$(k-1)$ GTs; a rank-1 GT is transformed into a simple loop.

To perform this operation, we introduce the *bind* operator that simply binds a variable inside a function that has free variables. We write $bind(f, i = j)$ to symbolize that all instances of i inside f should be replaced by the expression j. To simplify further, we denote $bind(f, i = i)$ as $bind(f, i)$.

For example, downranking the rank-1 GT in Equation 3.32 yields:

$$\begin{aligned}\mathbf{GT}(\mathrm{A}, g, s, [k]) &\xrightarrow{\text{GT-Downrank(1)}} \sum_{i=0}^{k-1} \mathrm{bind}(\mathbf{GT}(\mathrm{A}, g, s, []), i) \\ &= \sum_{i=0}^{k-1} \mathrm{S}(\mathrm{bind}(s, i)) \circ \mathrm{bind}(\mathrm{A}, i) \circ \mathrm{G}(\mathrm{bind}(g, i)) \\ &= \sum_{i=0}^{k-1} \mathrm{S}(s_i) \circ \mathrm{A} \circ \mathrm{G}(g_i)\end{aligned}$$

A rank-2 GT has to be downranked twice before it is entirely expressed as loops:

$$\begin{array}{l}\mathbf{GT}(\mathrm{A}, g, s, [k, \ell]) \xrightarrow{\text{GT-Downrank(2)}} \sum_{j=0}^{\ell-1} \mathrm{bind}(\mathbf{GT}(\mathrm{A}, g, s, [k]), j) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \Big\downarrow \text{GT-Downrank(1)} \\ \sum_{j=0}^{\ell-1} \sum_{i=0}^{k-1} \mathrm{bind}(\mathrm{bind}(\mathbf{GT}(\mathrm{A}, g, s, []), i), j) \\ \qquad = \sum_{j=0}^{\ell-1} \sum_{i=0}^{k-1} \mathrm{S}(s_{i,j}) \circ \mathrm{A} \circ \mathrm{G}(g_{i,j})\end{array}$$

We specify the index of the loop variable to be downranked as an argument to the downranking rule.

Loop interchange. Since a GT represents a fully permutable loop nests, various loop transformations can be applied. We describe here how to do a simple loop interchange to introduce the reader to GT transformations.

Obviously, in our previous example, the loop interchange can be trivially achieved by downranking in the other order; first with respect to k and then, with respect to ℓ:

$$\begin{array}{l}\mathbf{GT}(\mathrm{A}, g, s, [k, \ell]) \xrightarrow{\text{GT-Downrank(1)}} \sum_{i=0}^{k-1} \mathrm{bind}(\mathbf{GT}(\mathrm{A}, g, s, [\ell]), i) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \Big\downarrow \text{GT-Downrank(2)} \\ \sum_{i=0}^{k-1} \sum_{j=0}^{\ell-1} \mathrm{bind}(\mathrm{bind}(\mathbf{GT}(\mathrm{A}, g, s, []), j), i) \\ \qquad = \sum_{i=0}^{k-1} \sum_{j=0}^{\ell-1} \mathrm{S}(s_{i,j}) \circ \mathrm{A} \circ \mathrm{G}(g_{i,j})\end{array}$$

The same transformation can be performed directly on the GT level with a direct loop interchange rule:

$$\mathbf{GT}(\mathrm{A}, g, s, [k, \ell]) \xrightarrow{\text{GT-Interchange}} \mathbf{GT}(\mathrm{A}, g, s, [\ell, k])$$

MMM using GT. While GT has been originally developed for the exclusive use with linear transforms [Voronenko, 2008][11], one insight of our research is that the GT theory is applicable to more general operators, provided that the considered loop nest remains fully permutable. We therefore present here the formulation of the matrix-multiplication blocking rules using GT.

To do this, we start by introducing the rank-k index mapping stride function $h_{b,s,s_1,\ldots,s_k}$ which is no more than a standard mapping with base b and stride s that is wrapped inside k loops with vector strides $s_1, \ldots, s_k$:

$$h^{n\to N}_{b,s,s_1,\ldots,s_k} : (i, j_1, \ldots, j_k) \mapsto b + si + s_1 j_1 + \cdots + s_k j_k.$$

Observe that:

$$\mathrm{bind}(h^{n\to N}_{b,s,s_1,\ldots,s_k}, j_p) = h^{n\to N}_{b+s_p j_p,s,s_1,\ldots,s_{p-1},s_{p+1},\ldots,s_k}$$

Using this function, the horizontal blocking of $\mathbf{MMM}_{m,k,n}$ into a many panels (captured in Equation 2.16) can be re-expressed as:

$$\mathbf{MMM}_{m,k,n} \to \mathbf{GT}(\mathbf{MMM}_{m/a,k,n}, (h_{0,1,m/a} \otimes h_{0,1}) \times (h_{0,1} \otimes h_{0,1}), h_{0,1,m/a} \otimes h_{0,1}, [a]).$$

Note that the notation exposes multiple mathematical properties of the matrix multiplication:

1. The operation is recursive in nature and requires a loop of a iterations.

2. Two inputs are coming in, as highlighted by the cross product $\times$.

3. The two inputs and the single output are two dimensional, as highlighted by the tensor products $\otimes$.

4. The first input and the output (otherwise denoted as the matrices A and C) are dependent on the loop since their index mapping functions are of rank-1. The block slice happens in the external dimension (otherwise denoted as the height of the matrices).

For reference, we provide here the two other MMM blocking rules, that respectively cut the

[11] The reader that actually goes to the original reference might be puzzled by the description of the GT there. Indeed, the formulation there is slightly different since it exposes the implementation of the lambda system inside the rewriting engine. Because it is not our focus here, we choose to simply abstract it away using our bind functions.

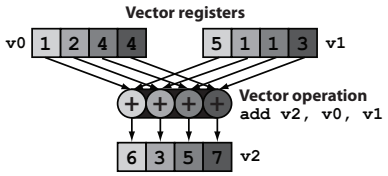


FIGURE 3.8: A vector addition in the SIMD vector paradigm.

matrices in b vertical panels and in c depth panels:

$$\mathbf{MMM}_{m,k,n} \to \mathbf{GT}(\mathbf{MMM}_{m,k/b,n}, (h_{0,1} \otimes h_{0,1,k/b}) \times (h_{0,1,k/b} \otimes h_{0,1}), h_{0,1} \otimes h_{0,1}, [b]),$$
$$\mathbf{MMM}_{m,k,n} \to \mathbf{GT}(\mathbf{MMM}_{m,k,n/c}, (h_{0,1} \otimes h_{0,1}) \times (h_{0,1} \otimes h_{0,1,n/c}), h_{0,1} \otimes h_{0,1,n/c}, [c]).$$

3.3.2 Vectorization

We choose to abstract complex computing platforms behind coarse structural descriptions of machines that we call *paradigms*. This subsection presents how the single instruction multiple data (SIMD) vector instruction paradigm can be harnessed using GT; the next subsection discusses the shared memory paradigm. Observe that paradigms can be hierarchically composed: for instance, an Intel Core2 Duo is characterized as a two-processor shared memory system (cache line is 64 bytes) where both CPUs are 4-way SIMD vector units (in single precision floating-point mode).

Background. The *SIMD vector paradigm* models a class of processors with the following characteristics:

- The processor implements a vector register file and standard vector operations that operate pointwise: addition (Figure 3.8), multiplication, and others. The use of vector operations usually results in a significant speed-up over scalar operations.
- The most efficient data movement between memory and the vector register file is through aligned vector loads and stores. Unaligned memory accesses or subvector memory accesses are more expensive.
- The processor implements shuffle instructions that rearrange data inside a vector register (intra-register moves).

Most important examples of vector instruction sets are Intel's SSE family, the newly announced Intel extensions AVX, AMD's 3DNow! family, Motorola's AltiVec family including the IBM-developed variants for the Cell and Power processors.

Automatic compiler vectorization is available in common optimizing compilers (e.g.: GCC [Naishlos, 2004], Intel Compiler [Bik et al., 2002]) but its applicability is limited to simple cases. The reasons are that:

1. The expressivity of many languages make it difficult to analyze the code and decide the legality of some transformations, and
2. The underlying variability in the architectures makes it difficult to provide performance models to assess the profitability of the transformations.

Therefore, most performance libraries are still vectorized by hand as of 2010. It is usually done either by directly programming in assembly or using vendor provided compiler intrinsics for C.

Vectorization in OL. Designing a mathematical framework for vectorization was one of the original motivations to come up with the Kronecker product formalism [Johnson et al., 1990a]. Since the tensor product simultaneously expresses structure and data independence, SPL has a very convenient way to describe perfectly vectorizable computation. The most important example is $\mathrm{A} \otimes I_\nu$ where A is any transform and ν is the vector length. Vectorized code for $y = (\mathrm{A} \otimes I_\nu)x$ can be obtained by replacing every scalar operation in the code for $y = \mathrm{A}\, x$ by its corresponding ν-way vector instruction [Franchetti et al., 2006c].

OL mathematically extends the tensor product to multiple inputs and outputs, hence it is no surprise that it can also describe vectorizable structures. The identity I_ν, however, has to be replaced with the point-wise product P_ν[12]. For instance, if A is an arity-(2,1) operator from $\mathbb{C}^k \times \mathbb{C}^m$ into $\mathbb{C}^n$, then $\mathrm{A} \otimes P_\nu$ denotes the operator from $\mathbb{C}^{\nu k} \times \mathbb{C}^{\nu m}$ into $\mathbb{C}^{\nu n}$ that performs ν interleaved As, i.e., the relationship between A and $\mathrm{A} \otimes P_\nu$ is as for transforms before.

Overview of the implementation. The vectorization process in our framework is fully integrated with the recursion step closure. It is performed in three steps:

1. Tagging and propagation of the loops that need to be vectorized,
2. Vectorization using either as-hoc rules or using the vector strip-mining GT rule that we introduce, and
3. Implementation of the vectorized loops using vector operations.

Tagging. The first step is simply a mean to indicate the intent to vectorize a given OL construct. For instance, if the target machine supports ν-way vector instructions, a target functionality (e.g. MMM) could be "tagged" with a simple mark:

$$\underbrace{\mathbf{MMM}_{m,k,n}}_{\mathrm{vec}(\nu)}.$$

Propagation of the vectorization tag is assured by rules such as

$$\underbrace{\mathrm{A} \circ \mathrm{B}}_{\mathrm{vec}(\nu)} \to \underbrace{\mathrm{A}}_{\mathrm{vec}(\nu)} \circ \underbrace{\mathrm{B}}_{\mathrm{vec}(\nu)}.$$

[12]The point-wise product is the only generalization of the identity that maintains the concept of matching slots which is at the core of the vector unit idea (see Equation 2.15 and followings).

Vectorization. The second step involves finding and locking in actual opportunities for vectorization. Indeed, we consider that some constructs are *naturally* vectorizable, such as loops of ν iterations with unit vector stride or some particular permutations [Franchetti and Püschel, 2008; Franchetti et al., 2006c]. These constructs are the only ones that are cast into vector code later on, so a formula has to be uniquely composed of naturally vectorizable components to be entirely vectorized. We denote such vectorizing rules by dropping the vectorization intent tag since the opportunity is actually locked in.

One easy way to do this is to introduce functionality specific rules, for example:

$$\underbrace{\mathbf{MMM}_{m,k,n}}_{\mathrm{vec}(\nu)} \xrightarrow{\text{MMM-Vec}} \mathbf{GT}(\mathbf{MMM}_{m,k,n/\nu}, (h_{0,1} \otimes \mathrm{dup}_\nu) \times (h_{0,1} \otimes h_{0,\nu,1}), (h_{0,1} \otimes h_{0,\nu,1}), [\bar{\nu}]).$$

One can verify that the above rule slices the B matrix into ν interleaved chunks, multiplies it with a A matrix that has been splatted ν times and that the C matrix is produced by reconstituting interleaved chunks. We bar the loop counter $\bar{\nu}$ to symbolize that the loop is a vector loop and should never be downranked.

Vectorization can also happen directly at the GT level using the *strip-mining* loop transformation. This operation is analogous to the standard vectorization mechanism inside optimizing compilers and transforms a loop into a doubly nested loop in which the inner loop performs as many iterations as the vector length so that it can be directly mapped to vector units.

In GT terms, strip-mining is the transformation of a rank-k GT into a rank-$(k+1)$ GT. In contrast with most compiler techniques, GT allows *any* loop to be strip-mined, that is, inner or outer loops, whether unit-stride or not. Of course, unit stride loops are expected to be more efficient since they naturally match the hardware capabilities but non-unit stride loops can also be handled using subvector loads and stores. This offers more search possibilities down the road and might lead to interesting niche performance gains.

The general rule requires the introduction of the strip function which is a specific lambda function. It breaks a given rank-k index mapping function into a rank-$(k+1)$ function, here is the example for a rank-1 function:

$$\mathrm{strip}((i, s_1) \mapsto f(i, s_1), \nu) = (i, s_1, s_2) \mapsto f(i, \nu s_2 + s_1).$$

The general rule can then be formulated:

$$\underbrace{\mathbf{GT}(\mathrm{A}, g, s, [n])}_{\mathrm{vec}(\nu)} \xrightarrow{\text{GT-Vec}} \mathbf{GT}(\mathrm{A}, \mathrm{strip}(g, \nu), \mathrm{strip}(s, \nu), [\bar{\nu}, n/\nu]).$$

Code generation. Finally, vector code has to be generated for the constructs that have been deemed naturally vectorizable. The strategy here varies: for some constructs (e.g. the stride per-

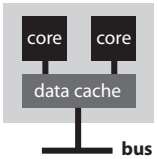


FIGURE 3.9: The shared memory paradigm.

mutations), a database look up picks up the right vector code straight away[13]. For vector loops, the entire code for the kernel is generated and scalar operations are then replaced by vector operations.

3.3.3 Parallelization

Background. The *shared memory paradigm* models a class of multiprocessor systems with the following characteristics (Figure 3.9):

- The system has multiple processors (or cores) that are all of the same type.
- The processors share a main, directly addressable memory.

Important processors modeled by the shared memory paradigm include Intel's and AMD's multicores and systems built with multiple of them.

While a general body of work on automatic parallelization does exist [Banerjee et al., 1993; Hiranandani et al., 1992] and is offered within some optimizing compilers, it is not yet considered as a serious alternative to manual implementation. Multiple root causes can be found:

1. Threading operates at a coarse grain which makes automatic analysis to discover it naturally costly.
2. The overhead to threading is important, so failures during automatic parallelization can seriously cripple performance. Compilers are thus very conservative in their attempts.

However, and in comparison to vectorization, good tools exist to help the developers to manually leverage the threading capabilities. In particular, language extensions and libraries such as OpenMP or pthreads are widely accepted standards to help manage threads in a way that is portable. Our backend supports both tools.

Overview of the implementation. Parallelization essentially follows the same principles as vectorization and is performed in three phases: the tagging of the loops, the actual parallelization and the code generation.

[13]While this might be simple, the actual challenge resides in automatically generating the said database, which is explained in depth in [Franchetti and Püschel, 2008].

Tagging. We denote by a tag an operator that has to be optimized for a system with p processors:

$$\underbrace{\mathrm{A}}_{\mathrm{smp}(p)}$$

Similarly to the vectorization, a set of rewriting rules directs the propagation of tags in a way that is compatible with the recursion step closure. This ensures that all possibilities are made available to the search mechanism which is responsible for taking the right decisions.

For instance, using a new SMP $\mathrm{barrier}$ operator, the rule for the propagation of the tag is expressed like this:

$$\underbrace{\mathrm{A} \circ \mathrm{B}}_{\mathrm{smp}(p)} \rightarrow \mathrm{barrier} \circ \underbrace{\mathrm{A}}_{\mathrm{smp}(p)} \circ \mathrm{barrier} \circ \underbrace{\mathrm{B}}_{\mathrm{smp}(p)}.$$

Parallelization. Loop parallelization is essentially a strip-mining operation. The general GT transformation rule can be written as:

$$\underbrace{\mathbf{GT}(\mathrm{A}, g, s, [n])}_{\mathrm{smp}(p)} \xrightarrow{\text{GT-Par}} \mathbf{GT}(\mathrm{A}, \mathrm{strip}(g, p), \mathrm{strip}(s, p), [\bar{p}, n/p]).$$

Note that the bar on top of the $\bar{p}$ tags this loop as being parallel and guarantees that it will not be downranked.

Code generation. After the recursion step closure, the backend implement parallel primitives using either OpenMP or pthreads. For instance, generated OpenMP code for the parallel loop nest above looks like this:

```
#pragma omp parallel for
for (i=0; i=n/p; i++)
    ...
```

3.4 Putting it all together

We conclude this chapter by detailing the generation of a complex MMM library that exhibits most of the features that we have presented in this chapter.

This library supports the commonly used DGEMM interface from [Dongarra et al., 1990] with the limitation that $\alpha = \beta = 1$. More precisely, this means that the library can perform the following computations:

$$C = A^{(\mathrm{T})} B^{(\mathrm{T})} + C$$

where A, B and C are column-major matrices, possibly transposed and possibly subject to a non trivial leading dimension (i.e., submatrices of packed matrices are captured).

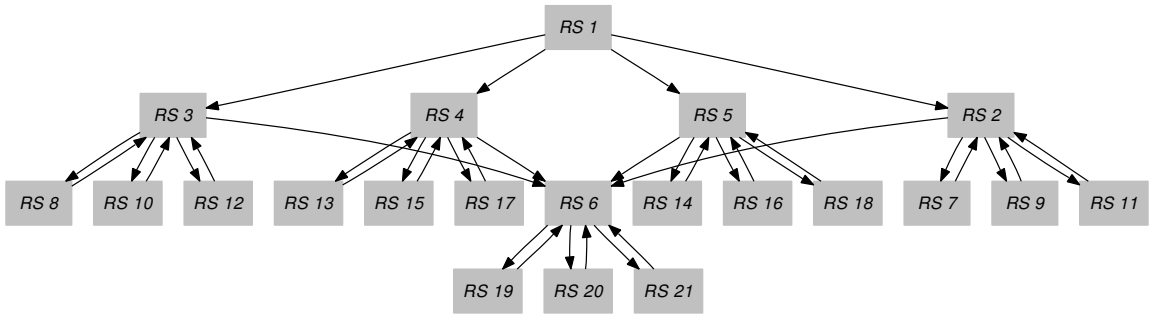


FIGURE 3.10: Static call graph of a vectorized matrix-matrix multiplication library with an interface close to the one of the GEMM. "RS" stands for recursion step.

We capture this interface by creating a DGEMM operator with four simple rules that dispatch to four different computations, depending if A is transposed and if B is transposed.

We then ask the system to generate a library core, based on this DGEMM functionality, on the four new dummy rules and on the three blocking rules. We also ask the system to vectorize the library and specify that base cases to be included for MMM are to be constrained within $[1, 8] \times [1, 8] \times [1, 8]$.

The library generator then computes for one hour and produces a library core that is composed of 105958 lines of code weighting 4MB overall and whose closure graph is presented in Figure 3.10.

On this graph, it can be seen that the GEMM interface (*RS 1*) is broken down in four different recursion steps (*RS 2*, *RS 3*, *RS 4*, and *RS 5*) which corresponds to the different transposition steps of the matrices A and B. We consider now *RS 3*. It can breakdown into *RS 8*, *RS 10* and *RS 12* which corresponds to the three looped blocking rules. It can also breakdown into *RS 6* which corresponds to a SIMD vectorized matrix multiplication. *RS 6* can itself be broken down into *RS 19*, *RS 20* and *RS 21* using the three looped blocking rules. Therefore, the base case code inside these three recursion steps corresponds to vectorized looped base cases.

We show the loop body of one of the base cases of *RS 19* in Figure 3.11. The performance of the library is shown in the experimental results (Chapter 5).

```
// loop initialization omitted

do {
    s41379 = *(b119682);
    s41380 = _mm_mul_pd(s41378, s41379);
    a174716 = *(b119684);
    *(b119684) = _mm_add_pd(a174716, s41380);
    a174717 = (1 + b119682);
    s41381 = *(a174717);
    s41382 = _mm_mul_pd(s41378, s41381);
    a174718 = (1 + b119684);
    a174719 = *(a174718);
    *(a174718) = _mm_add_pd(a174719, s41382);
    s41384 = _mm_mul_pd(s41383, s41379);
    a174723 = *(b119688);
    *(b119688) = _mm_add_pd(a174723, s41384);
    s41385 = _mm_mul_pd(s41383, s41381);
    a174724 = (1 + b119688);
    a174725 = *(a174724);
    *(a174724) = _mm_add_pd(a174725, s41385);
    b119682 = (b119682 + a174703);
    s41387 = *(b119682);
    s41388 = _mm_mul_pd(s41386, s41387);
    a174729 = *(b119684);
    *(b119684) = _mm_add_pd(a174729, s41388);
    a174730 = (1 + b119682);
    s41389 = *(a174730);
    s41390 = _mm_mul_pd(s41386, s41389);
    a174731 = *(a174718);
    *(a174718) = _mm_add_pd(a174731, s41390);
    s41392 = _mm_mul_pd(s41391, s41387);
    a174734 = *(b119688);
    *(b119688) = _mm_add_pd(a174734, s41392);
    s41393 = _mm_mul_pd(s41391, s41389);
    a174735 = *(a174724);
    *(a174724) = _mm_add_pd(a174735, s41393);
    b119684 = (b119684 + a174712);
    b119688 = (b119688 + a174712);
    b119682 = (b119682 + inc55);
} while( b119682<ubound55 );
```

FIGURE 3.11: Automatically generated vectorized loop body of one of the base cases implementing *RS 19* in Figure 3.10. Note that most of the index computation is not shown since it is hoisted out of the loop.

CHAPTER 4

Adaptation: Tuning the Library to the Hardware

Adaptive libraries have been advocated as a solution to cut down performance library development costs in a setting where micro-architectures tend to become increasingly complex and diverse. They have proved to be successful in a number of domains, including basic dense linear algebra (ATLAS [Whaley and Dongarra, 1998]), sparse linear algebra (OSKI [Vuduc et al., 2005]), sorting (Adaptive Sorting Library [Li et al., 2004]), and linear transforms (FFTW [Frigo and Johnson, 2005], UHFFT [Ali et al., 2007], Spiral-generated libraries [Voronenko et al., 2009]).

The key features that distinguish all these research projects from more "traditional" libraries can be summarized in two simple points:

1. Adaptive libraries provide many different alternative strategies to implement a given functionality. These implementations constitute the *exploration space* and, a priori, it is not known which one of them will perform the fastest on the user machine.

2. An *exploration mechanism* is supplied in order to select one of these alternatives according to some criterion, usually the speed of the code (but we will see that the length of the search can also play a role). The choice depends on the feedback that can be obtained through actual timings on the user machine.

This definition deliberately excludes libraries that provide multiple different optimized routines for the same function, one for each of the supported architectures (e.g. GotoBLAS [Goto, 2008], IPP [Intel, 2009a], or MKL [Intel, 2009b]). In such libraries, the exploration is performed by experts during implementation rather than being mechanized which unfortunately does not scale when the number of platforms grow.

In the previous chapters, we have shown how various recursive algorithms can be represented as rules (Chapter 2) and how these rules can be compiled into efficient code (Chapter 3). The handling

of degrees of freedom was not discussed and questions such as "which one of the rules should be applied?" or "what block size should be used during the horizontal blocking?" have been delayed until now.

This chapter covers adaptation by selecting among the available degrees of freedom with the goal of maximizing the performance on the target platform. Two different usage scenarios will be tackled and they will naturally lead to two different formulations of the adaptation problem that essentially differ by the time at which the exploration is performed: *Online* libraries supply the user with the best implementation for any given problem size provided he or she is willing to spend time searching for it whereas *offline* libraries concentrate the search at installation, providing reasonable support for all sizes without any exploration overhead.

In Section 4.1, we exhibit the structure of the exploration space for adaptive libraries and present in depth the two usage scenarios we consider. The infrastructure and the various methods to solve the online problem are presented in Section 4.2. The offline problem is finally tackled in Section 4.3.

4.1 Structure of the Exploration Space

This section introduces most of the concepts that will be commonly used in the following sections. We start by presenting the degrees of freedom inside a generated DFT library, show actual expert-written heuristics and introduce two different representations of the exploration space, namely the *augmented closure graphs* and the *decision graphs*. We finally present the two usage scenarios for adaptation that we will consider in this thesis: online and offline adaptation.

4.1.1 Motivation: Degrees of Freedom and Heuristics

Depending on the functionality, the degrees of freedom in the choice of algorithms combined with the recursive nature of the problem can yield a huge number of different alternatives, even without considering any choices arising from the implementation. Traditional libraries or Spiral-generated general-size libraries from [Voronenko et al., 2008a] include expert-derived *heuristics* to manually select among the alternatives. This process is not automatic and therefore prevents the full mechanization of the library generation.

Degrees of freedom. We focus here on the classic Cooley-Tukey algorithm for the discrete Fourier transform:

$$\mathbf{DFT}_n = (\mathbf{DFT}_k \otimes I_m) T^n_m (I_k \otimes \mathbf{DFT}_m) L^n_k, \quad n = km. \tag{4.1}$$

The degree of freedom is the choice of $k|n$ and impacts the factorization as shown in Figure 4.1. Observe that the subproblems (smaller DFTs) are of different size depending on the choice of the radix, which means that the choice actually has repercussions in the entire recursion is affected.

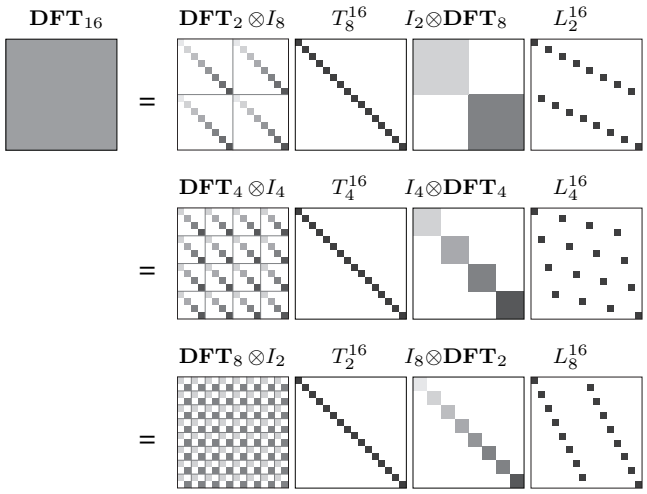


FIGURE 4.1: Visualization of the impact of radix choice inside the Cooley-Tukey algorithm. The non-zero values of the three different factorizations of $\mathbf{DFT}_{16}$ are displayed. In the tensor products, boxes of equal gray shade are part of the same $\mathbf{DFT}$.

FIGURE 4.2: Two fully expanded ruletrees for $\mathbf{DFT}_{16}$.

Note that an algorithm is not fully specified until it is fully expanded, or, equivalently, until it is broken down into base cases. In early Spiral papers [Püschel et al., 2005], such fully specified algorithms are represented using trees such as those presented in Figure 4.2 and are called *ruletrees*.

Recursively compounded this yields, in this case, and under certain assumptions, an algorithm space of $\Theta(5^t/t^{3/2})$ for $n = 2^t$ that this library covers (see Figure 4.3) [Johnson and Püschel, 2000]. We remind the reader that all of these are "fast Fourier transforms" and have roughly the same operations count; yet, the performance can differ widely due to cache misses and other effects. Albeit the number of different algorithms is already significant, we will see later that implementation optimizations adds further degrees of freedom, which exacerbates the problem.

Heuristics. To select an implementation among the different alternatives, traditional libraries use *heuristics*: functions designed by experts that restrict the search space to a single choice, that ideally is close to optimal.

To illustrate the difficulty of developing such heuristics, we show in Figure 4.4 the function used in [Voronenko et al., 2008a] to choose the radix as a function of the input size for DFTs. The choice,

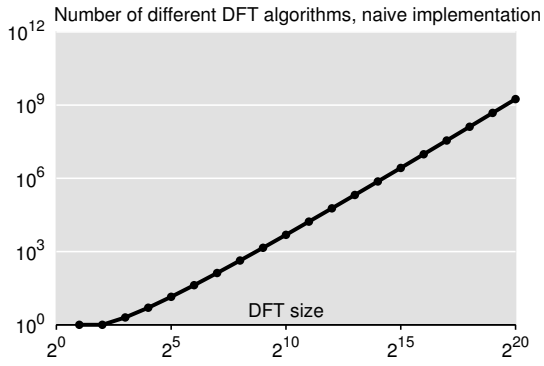


FIGURE 4.3: Number of DFT algorithms based on standard Cooley-Tukey FFT, implemented naïvely. All algorithms for a given DFT input size have roughly the same operations count.

encoded in a short decision tree is deceivingly simple: called recursively, it selects the radix in each step in Cooley-Tukey (Equation 4.1). The resulting performance is very good for scalar single-threaded implementations on new x86 platforms for sizes ranging from 2^1 to 2^{16}. The heuristic relies on the following assumptions:

1. The library only supports DFT sizes that are two-powers $n = 2^t$.
2. Other parameters than the size (such as strides for different recursion steps) are negligible.
3. Scaled DFT codelets[1] have been provided with the library for all sizes up to 32.
4. Right expanded DFT trees are known to be faster.
5. Large size codelets are known to be faster than small ones, up to some limit.
6. Small strided codelets[2] (size 2 and 4) are particularly bad and should be avoided.

Obviously, this simple piece of code raises many questions, such as how do we know that this heuristic is actually good, or how to extend it to odd sizes, onto different platforms, or to different functionalities or different algorithms. With the large number of parameters that can go inside a generated library (see for instance, Equation 3.31), it would be infeasible to derive heuristics by hand for every computer-generated library.

The goal of this chapter is to describe two mechanisms to replace expert-written heuristics. The first one, *online adaptation*, dynamically selects the degrees of freedom at runtime based on explicit trials on the target platforms. The second one, *offline adaptation*, harnesses machine learning tools to automatically generate such heuristics at installation time.

[1] Scaled codelets correspond to codelets that contain the twiddle factors. See [Voronenko, 2008] or [Frigo and Johnson, 2005] for additional information on the specifics of the DFT base cases.

[2] Strided codelets correspond to codelets that contain the final permutation. Again, see [Voronenko, 2008] or [Frigo and Johnson, 2005] for additional information's.

```
if (divisible(32, n) && (n/32) >= 8) {return 32;}
else if (divisible(16, n) && (n/16) >= 8) {return 16;}
else if (divisible(8, n)) {return 8;}
else if (divisible(4, n)) {return 4;}
else if (divisible(2, n)) {return 2;}
else { error(''no divisors'');}
```

FIGURE 4.4: The heuristic developed by Voronenko to choose the DFT radix in function of the input size in his doctoral thesis [Voronenko, 2008]. The only constraint on the output is that the radix must divide the input size.

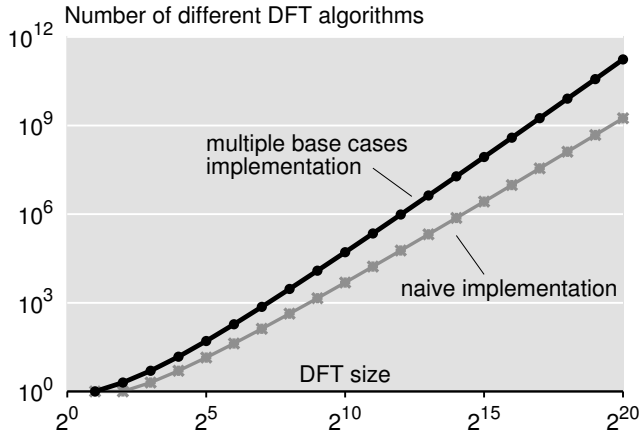


FIGURE 4.5: Difference in the number of DFT algorithms, if base case termination is implemented as a choice.

4.1.2 Structure of the Search Space

The above discussion of the exploration space assumed a naïve DFT implementation. In fact, the complex optimizations needed to implement such recursive algorithms into state-of-the-art libraries introduces additional degrees of freedom of quite different types.

Multiple base cases. All recursions require base cases to terminate. In the previous situation, we implicitly assumed that a single base case was available, $\mathbf{DFT}_2$, but it is well known that bigger base cases reduce the recursion overhead and enable better register level optimizations which results in faster execution for all sizes. As an example, for the FFT, usually two-powers up to 64 are provided as base cases inside a two-power DFT library.

In practice however, there is a tradeoff point where the reduction of the control and the increase in instruction level parallelism is over compensated by the increased pressure for registers and instruction cache. As a consequence recursion unrolling ends up being detrimental if pushed too far. To compensate for this effect, we choose to enable recursion on all sizes, *even those* that are directly supported by base cases, leaving the choice of early termination up to the search mechanism. This decision has a large impact on the size of the search space as can be observed in Figure 4.5.

```
void dft(int n, cpx *y, cpx *x) {
    if (use_dft_base_case(n))
        dft_bc(n, y, x);
    else {
        int k = choose_dft_radix(n);
        for (int i=0; i < k; ++i)
            dft_strided(m, k, t + m*i, x + m*i);
        for (int i=0; i < m; ++i)
            dft_scaled(k, m, precomp_d[i], y + i, t + i);
    }
}

void dft_strided(int n, int istr, cpx *y, cpx *x) { ... }
void dft_scaled(int n, int str, cpx *d, cpx *y, cpx *x) { ... }
```

FIGURE 4.6: FFTW 2.x-like implementation of the DFT. In this code, degrees of freedom are implemented by the heuristics `use_dft_base_case(n)` and `choose_dft_radix(n)`—in the FFTW code, these degrees of freedom would be searched online by the planning system. For brevity, auxiliary recursion steps `dft_strided` and `dft_scaled` are not detailed entirely.

Recursion step closure. Deriving the recursion step closure explained in Subsection 3.1.2, it is easy to see that the naïve four passes implementation can be actually reduced to only two passes, hence improving locality. Such an implementation is used, for instance, in FFTW 2.x. Namely, the explicit (and expensive) permutation L^n_k can be replaced with a readdressing in the subsequent smaller DFTs and the scaling by T^n_m can be fused with the subsequent DFTs. The generated recursion steps are designated as shown below:

$$\underbrace{\mathbf{DFT}_n}_{\texttt{dft}} = \underbrace{\left((\mathbf{DFT}_k \otimes I_m) T^{km}_m\right)}_{\texttt{dft\_scaled}} \underbrace{\left((I_k \otimes \mathbf{DFT}_m) L^{km}_k\right)}_{\texttt{dft\_strided}}, \quad n = km.$$

The pseudo-code for such an implementation is displayed on Figure 4.6. Since the recursion must eventually terminate, we assume that base cases are provided for all recursion steps in the form of unrolled codelets of fixed size (denoted with a `bc` for base case in the code). The two degrees of freedom we referred to earlier (choice of the radix and whether to use a codelet or not) are to be specified using the functions `choose_dft_radix(n)` and `use_dft_base_case(n)`[3].

The important thing to realize is that the two auxiliary recursion steps are nothing but DFTs with a different interface and therefore also possess their own degrees of freedom. These degrees of freedom depend on all cold parameters (see Subsection 3.1.3) of the recursion steps; so, in this case they do not only depend on the size `n` but also on the strides `str` and `istr`. These extra choices can be observed in Figure 4.7 which shows an *augmented closure graph* where the choices (white diamonds) are represented together with the recursion steps (gray boxes). The outgoing edges of

[3]For the sake of the explanation, we make the dynamic assumption here which implies that the performance of the recursion step only depends on the actual size n. The topic will be covered in depth later on.

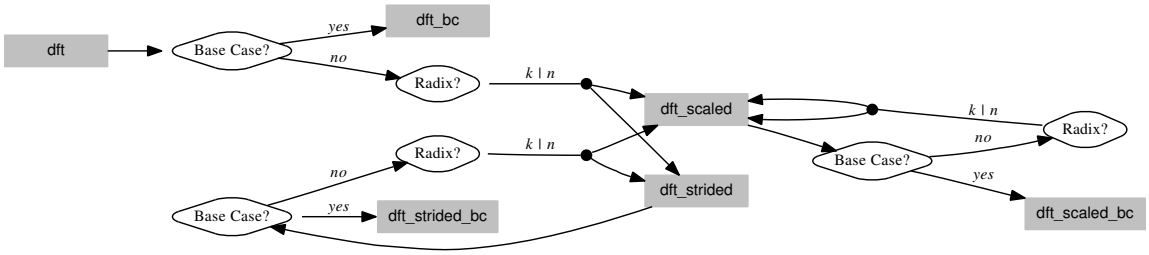


FIGURE 4.7: Augmented closure graph underlying the implementation from Figure 4.6. Note that, in the case of radix choices, both children actually need to be computed.

choices are labeled with decisions and connect to spawned recursion steps. The outgoing edge of a recursion step may connect to a choice or to another recursion step that it calls without choice.

The graph leads to further observations:

1. Choices of the same type (e.g., "base case?") may occur multiple times inside the graph since they correspond to different recursion steps.

2. The decision graph contains cycles due to the recursive nature of the search space. During the decision process, the same choice box can hence be encountered several times (e.g., the choice of radix for a strided DFT of size 128 and later the choice of radix for a strided DFT of size 16). What is not evident from the decision graph is that the decision procedure eventually terminates since the DFT sizes decrease.

3. A given recursion step is implemented using zero, one, or more recursion steps. Therefore, open choices are of different nature: a recursion step is *either* implemented as a base case or not, but an actual recursion always uses *both* recursion steps[4].

Artificial restrictions. The library we just described offers many different ways to compute a given functionality. The search space is in fact so large that some branches of the graph could be removed while still guaranteeing termination for all cases. The advantage is a reduction in code size (breakdowns and heuristics). In some cases, it may be possible to do without hurting performance. For instance, Figure 4.8 depicts FFTW 2.x's implementation of the DFT. It restricts the scaled DFT recursion step to being a codelet.

Advanced implementations. The search space gets considerably more complicated with state-of-the-art libraries. The reason is in the support for vectorization, multithreading, and advanced memory hierarchy optimizations [Voronenko, 2008; Voronenko et al., 2009]. The latter includes support for buffering and for on-the-fly twiddle factor computation. These optimizations require transformations of Equation 4.1 that produce additional recursion steps together with additional choices.

[4]In this particular example, there is no recursion step being implemented using a single other recursion step.

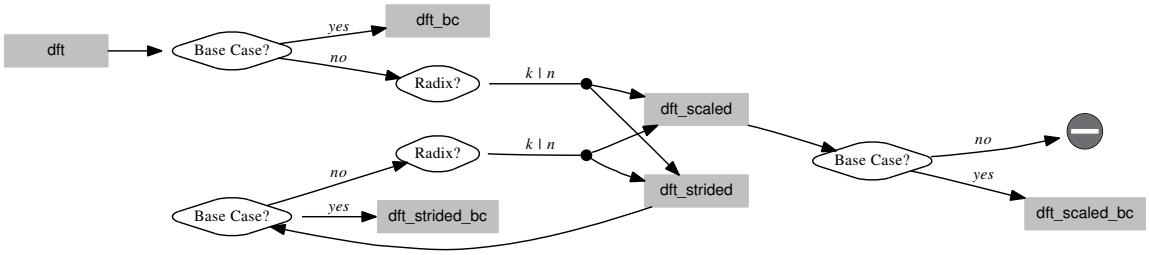


FIGURE 4.8: Decision graph underlying the implementation from Figure 4.6 where the scaled DFT recursion step has been restricted to being a codelet. FFTW 2.x functions exactly like this.

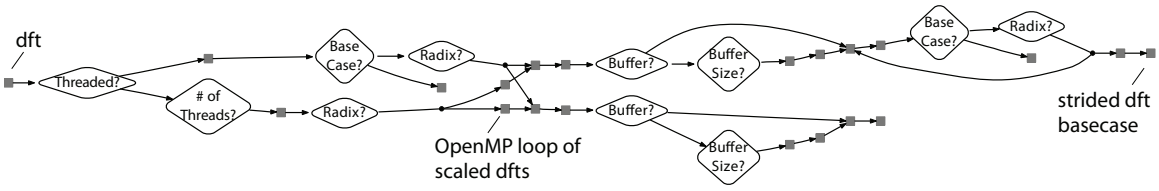


FIGURE 4.9: Decision graph underlying a vectorized, multi-threaded and buffered implementation of the DFT. Recursion step names are omitted for readability except for the initial `dft` recursion step and two examples.

Figure 4.9 shows the decision graph of such a library generated by Spiral. Observe the increase in the number of recursion steps and the associated increase in choices inside the library.

The above discussion holds not only for the DFT but also for most of the other functionalities that Spiral can generate libraries for. Further, not all algorithms decompose a functionality into functionalities of the same type. In this case the search space is further increased.

4.1.3 Online and Offline Adaptation

This chapter describes a mechanism to *automatically* determine the right decisions for a given machine. Within this adaptation goal, there are two different usage scenarios that make sense and we call them *online* and *offline* adaptation. We just present here the two alternatives but we cover them in detail in the coming sections.

Online adaptive libraries require search at time of use, namely every time the input specification (typically the input size) changes. An example is FFTW where this process is called *planning*. Because the exploration is done once for each problem, the user needs to compensate the search overhead with multiple computations for the same specification.

In offline adaptive libraries, in contrast, the search is done only once for all problem sizes, at installation time. An example of such a library is ATLAS: during installation, it automatically produces a tuned implementation of basic linear algebra subroutines (BLAS). The benefit of searching

Library type	Non-adaptive	Online adaptive	Offline adaptive
Prototype	IPP, FFTW-Estimate	FFTW-Measure	-
Generatable by Spiral	[Voronenko, 2008]	*this dissertation*	*this dissertation*
Interface	`d = dft(n)` `d(X,Y)`	`d = dft(n)` `d(X,Y)`	`d = dft(n)` `d(X,Y)`
Initialization cost	$O(n)$	$> O(n \log n)$	$O(n)$
Computation cost	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Adaptation mechanism	none	online (planner at runtime)	offline (at installation time)
User view			
When problem changes	-	replan	-
When platform changes	rebuy	replan	reinstall

TABLE 4.1: Different general size library types and their properties using the n point DFT as an example.

only once comes at a price: the decisions are not tuned to a specific problem size and therefore may be less efficient than those found by online adaptation.

DFT example. A summary of the different properties of the libraries is presented in Table 4.1 using the discrete Fourier transform (DFT) as example. We note that FFTW also possesses hand-written heuristics which provides mixed performance results as we will show later.

Observe that the interface in all libraries is the same, requiring two distinct calls in a way that is similar to currying (partial application) [Curry et al., 1958]. In the first phase, `d = dft(n)`, the libraries perform initialization, which includes precomputation of the trigonometric constants (called *twiddle factors*), and, in the case of online adaptation, the library searches for the fastest way to compute the DFT of the given size n. The actual computation is only performed in the second phase when the data is provided. The time for search is only amortized if many computations of the same type (here size n) are performed. The advantage of non-adaptive and offline adaptive libraries is that the initialization step is asymptotically faster than the computation step ($O(n)$ vs $O(n \log n)$), but in practice it is still slower than the computation step, until n is at least several thousand points. In an online adaptive library, the adaption process usually must perform at least one timing of the entire computation, and thus is guaranteed to take longer than just the computation step. In practice, for larger problem sizes, it may take several hours.

This interface conflicts with many legacy applications, which use a simpler one-call interface. Non-adaptive and offline adaptive libraries can emulate the one-call interface without substantial overhead. A caching system could be used to hide the online adaptation behind a legacy interface, but this complicates usage, potentially increases memory footprint, and does not resolve the problem of the high latency in applications with changing problem sizes.

Assuming a perfect implementation, offline adaptation enables the best of both worlds: low

initialization overhead (which enables easier emulation of a legacy interface) and automatic adaptation. Alas, the development is difficult since it requires the developer to come up with a framework to automatically find suitable heuristics that have to work well for all problem sizes. In ATLAS this was done successfully but the search space (which includes different blocking and unrolling sizes) is BLAS-specific and cannot be used for other libraries, e.g., for the DFT. In practice however, online adaptive libraries are tuned for any given size and are therefore expected to provide a better performance than offline adaptive libraries.

Formal problem statement. We consider an adaptive performance library that provides a computing function parametrized by one or multiple positive integers, typically the input size or sizes. For simplicity, we assume in our problem formulation a single parameter n; the formulation for several parameters is analogous. One parameter is sufficient for the DFT but not for all the subroutines needed to compute the DFT (as shown before).

The library has degrees of freedom in the computation. We call every degree of freedom a *choice* and model it as a set of positive integers $C \subset \mathbb{N} = \{0, 1, 2, ...\}$. Examples include binary choices $\{0, 1\}$ such as "threading or not," the choice of the number of threads, and the choice of radix. In the ATLAS generator, choices include various tile sizes and the degree of unrolling. Making a *decision* for a given choice C means choosing $d \in C$. The complete computation is specified by a finite sequence (list) of decisions $D = \{d_1, \ldots, d_k\}$. We denote the set of possible decision lists (which may have different lengths) for input size n with $\mathcal{D}(n)$. Hence, in the library $\mathcal{D}(n)$ is the search space for size n that is available for adaptation.

We now can formalize and distinguish the problems of online and offline adaptation for a given library. Given a list D of decisions, let $p(D)$ be the associated performance (higher is better) of the computation with the library *on the target platform*.

PROBLEM 1 (*Online adaptation)* Given the size n and the platform P, find

$$D_n^P = \operatorname*{argmax}_{D \in \mathcal{D}(n)} p(D)$$

PROBLEM 2 (*Offline adaptation)* Given the platform P, find a function

$$D^P : \; n \mapsto D^P(n) = D_n^P, \text{ where } D_n^P \text{ is defined as in above.}$$

Note that in both problems, the optimum can only be obtained by exhaustive search which is not feasible. Hence functions that approximate $D^P(n)$ will be called *heuristics*. For instance, FFTW provides a mode called FFT_ESTIMATE that features such a hand-written machine independent heuristic $D(n)$ which provides mixed performance results as we will see later.

In addition, FFTW can solve Problem 1 using a dynamic programming search and the ability to store the D_n^P (called "wisdom"). ATLAS solves Problem 2 by providing a generic heuristic $D^P(n)$ that is parametrized by several numeric values determined at installation time, all independent of

the input size[5]; hence, $D^P(n)$ is mostly known and can be inlined in the code.

Related work. In the domain of high-performance library development, [Singer and Veloso, 2002] is the only paper, to the best of our knowledge, that deals with automatically designing heuristics (which we denoted as Problem 2). Their heuristics are derived by predicting the runtime of different linear transform algorithms. The features that they use are collected directly in the transform algorithm SPL formula. Unfortunately, their work is not easily extensible to other domains since it heavily relies on a number of signal processing observations and is also not easily reproducible since it only handles the DFT and WHT libraries and does not cover general-size libraries which have more parameters than the radix.

In the larger domain of optimizing compilation, two different subjects directly relate to this topic. First, *iterative compilation* (also called adaptive or feedback-driven) describes the process of successively compiling and executing the code in order to find the best optimizations for *one* given application. This topic is analogous, in a more general setting, to our Problem 1. Compilation time issues have led to smart timing frameworks [Cooper et al., 2005] and various interesting search methods whether ad hoc [Pan and Eigenmann, 2006] or rooting in machine learning [Cooper et al., 1999] have been proposed.

The second trend in compiler research that relates to our subject is work on automatically tuning the heuristics themselves, which in turn allows the compiler to perform better on *all* programs (which is similar to our Problem 2). Early work on the subject [Moss et al., 1998] suggested to use reinforcement learning in order to improve the scheduling of straight-line code. Closer to our work, [Calder et al., 1997; Cavazos and Moss, 2004; Monsifrot et al., 2002] have suggested to learn whether or not an optimization should be triggered by learning decision trees using algorithms derived from C4.5. In contrast to our work, legality is never an issue for them and they only focus on dealing with a single heuristic whereas we generate all the heuristics of the library with this method. Other researcher have pursued the same goal with different search methods such as genetic programming [Stephenson et al., 2003]. In this area, research essentially focuses on predicting whether an optimization should be triggered but leaving the actual parameter choices to an heuristic. Predicting the actual parameters is needed for our work and there are few reports on the topic [Stephenson and Amarasinghe, 2005].

4.2 Online Adaptation

Online adaptation is only suitable for a user that computes multiple problems of the same size. It logically follows from this assumption that it might be beneficial to spend more time upfront, on the discovery of a faster implementation and amortize that one-time cost over the multiple computation instances.

[5] Actually, in the case of ATLAS, the equivalent of the input size would be the dimensions of the matrices.

In this sense, online adaptation should be understood as an *optimization problem*: The precise size of interest is known, the computing platform is available, so the only thing that remains to be done is to select the one alternative that maximizes the performance. This selection has to be done in a *reasonable* amount of time, where reasonable actually depends on the user: there is more time for adaptation if the function and size is supposed to run in an embedded system for the next decade compared to one that is used only 10 times.

This section starts with a short overview of the planning system, which will help with understanding the different issues associated with online adaptation. We then present how dynamic programming, the search strategy of choice from earlier versions of Spiral [Püschel et al., 2005], can be updated in the context of general size libraries. Finally, we introduce a "bandit-based" optimization algorithm, which was inspired by recent advances in computer Go.

4.2.1 Planning System

This section presents the architecture of the planning system which sits between the library user and the generated library core. Its purpose is to enable online adaption while hiding the complexity of the core. In particular, the system provides different search algorithms that works will all the different supported libraries and hides it behind a convenient user interface.

After the presentation of a *naïve online adaptation interface*, we describe the structure of the *planners*, which are a key component to simultaneously support different search strategies. Finally, we quickly present some *implementation subtleties* that also drove the design choices.

Naïve online adaptation interface. The operation performed during online adaptation is equivalent to the currying of the interface. For instance, we consider an $\mathrm{MMM}_{m,k,n}$ library. In the ML programming language [Milner et al., 1990], the type of such an interface would be

```
(int * int * int * double array * double array) -> double array
```

The exact addresses of the arrays play a minor role in the performance[6] of the multiplication; they are hot parameters (see Subsection 3.1.3). Therefore we curry the function in this way:

```
int * int * int -> (double array * double array -> double array)
```

Ideally, evaluating the function partially like above would allow a perfect compiler to get the first triplet of integers, compute for some time and return a function optimized for arrays of the fixed sizes that were specified in the first arguments. In practice however, the lack of perfect compilers forces us to do the said transformation ourselves. And, in fact, the lack of functional languages compilers suitable for high-performance computing even forces us to use imperative languages to implement the above closure. In an object oriented language such as C++, partial application can be emulated by first having the user build an object, the problem *descriptor*, and then, calling different

[6] This is a reasonable assumption if the addresses stay aligned to the vector length. A detailed study of the impact of the offsets on the performance can be found in [Jalby and Lemuet, 2002].

```
class Naive_Template_RStep {
    Naive_Template_RStep(cold parameters);
    void plan();
    void compute(hot parameters);
};
```

FIGURE 4.10: Naïve interface for recursion steps in order to do online adaptation. The cold and hot parameters vary for the different recursion steps (see Subsection 3.1.3).

methods of the said object. For instance, a matrix multiplication library could be called in the following way:

```
descr = new mmm(m,k,n);
descr->plan();
descr->compute(A,B,C);
descr->compute(A2,B2,C2);
```

In the example, the descriptor is first created with the dimension of the matrices. At this point, the planning system explores the many possible alternative implementations and selects just one. Finally, the selected implementation can be used and subsequently reused on different arrays.

The above intuition is generalized in Figure 4.10. It describes a first interface that needs to be implemented by all all recursion steps.

Planners. The interface we just described requires the search method to be directly coded inside every recursion step which means that the process is neither convenient (modifying the search strategy is complicated) nor robust (the same code is replicated needlessly). To solve this, we introduce *planners* that decouple the search strategies from the recursion steps.

The organization of the planners arises naturally from the observation that any search strategy works on two different levels:

- At the local level, one needs to provide all decisions that make up a plan. For this purpose, planners must provide a function `choose` that is passed a vector of integers (the choice) and simply returns one of the integers (the decision).

- At the global level, one needs to decide how to search and when to stop. It is necessary to evaluate previous decisions and eventually modify the local strategy in consequence. This is accomplished by a `plan` function whose purpose is to return an optimized recursion step.

To illustrate the need for both concepts, consider two different exploration strategies, the omniscient oracle and the standard Monte-Carlo. On one hand, the oracle logic is essentially local: every time a decision is needed the right answer is produced; so, at the global level, everything is done perfectly right from the first time. On the other hand, Monte-Carlo compensates a failing local decider, choosing randomly, by a better global strategy, repeating the former for as long as possible and finally returning the best seen answer.

```
class Planner {
    virtual void plan(RStep*)=0;
    virtual int choose(RStep*, vector<int>)=0;
};

class RStep {
    virtual void plan(Planner*)=0;
};

class Template_RStep: public RStep {
    Template_RStep(cold parameters);
    void compute(hot parameters);
    void plan(Planner*);
};
```

FIGURE 4.11: Planner interface for online adaptation. Planners encapsulate the search strategies which allows to easily swap from one to the other.

The interface in Figure 4.11 is based on these observations. It is designed for maximal flexibility and, as a matter of fact, the two competitive search strategies we will describe in the coming subsections are implemented based on it. Planners have also been developed by other members of the Spiral group such as Eric Lee Turner who reimplemented the STEER evolutionary search planner [Singer and Veloso, 2001] for general size libraries and Volodymyr Arbatov who ported some planners to the C language.

Implementation subtleties. One of the unexpected difficulties that the search mechanism needs to deal with is that, sometimes, the search might get trapped in a dead-end. This situation is usually triggered when a recursion step needs to be implemented but no rules actually apply. For instance, we already illustrated such a situation in Figure 4.8, where a recursion step was artificially constrained to be implemented as a base case. If the search ends up in a branch that requires this recursion step but its size is not provided as part of the codelets, the whole algorithm choice must be forfeited as there is no way to implement it.

In our planning system, exceptions are used to signify such dead-ends and the system backtracks. We will not provide deeper details in this document, but it is important to realize that the design and the quality of the planning is critical to the functioning of the adaptive library. In particular, the memory management during planning needs to be flawless since memory leaks tend to snowball in the exploration phase.

4.2.2 Dynamic Programming (DP)

Theoretically, a general-size problem is equivalent to a fixed-size one as soon as a size is provided. A simple idea to provide online adaptation to the general-size framework is therefore to port the methods that were effective in the fixed-size framework [Püschel et al., 2005]. This paragraph de-

scribes the most important search strategy, dynamic programming, as well as its specifics in the context of a general-size library.

Dynamic programming assumption. Dynamic programming designates a class of algorithms that efficiently solve complex problems by breaking them down into simpler steps. Such methods are only applicable to problems that exhibit a property called *optimal substructure*: it requires the best solution of a given problem to be only built out of optimal solutions to its subproblems.

The question of whether online adaptation presents an optimal substructure or not depends on whether the performance of a given piece of code can be seen as function or not. Intuitively, the fastest way to do a task containing multiple subtasks is to do each subtask as fast as possible — dynamic programming should therefore be applicable for online adaptation. In practice however, the performance of an algorithm is dependent on the context in which it is running because the content of the cache is a side effect. For instance, it would be beneficial to maximize the cache usage of an algorithm running on a single core but the same optimization would be detrimental if two instances of the same algorithm were running simultaneously on two cores contending for the same cache.

Implementation. It is possible to follow the dynamic programming guidelines even if the status of the assumption itself is not settled. By doing so, the result of the search is not guaranteed to be optimal but may still be good in practice.

Using the planner interface (Figure 4.11), the implementation of the dynamic programming search method is strictly local, mimicking that of an oracle. When a choice is necessary, it checks a hash-table to see if the same choice hasn't been made earlier. If not, it times each possible decision out of context by cloning the current recursion step and recursively planning it with the said decision. Finally, the fastest decision is applied to the original recursion step and stored in the hash table. One of the convenient features of the exploration mechanism is therefore that it can be interrupted and restarted later since one only has to save the hash table in the process.

Enabling dynamic programming therefore requires that the recursion steps support three new methods:

1. An `id()` method is necessary to recognize recurring recursion steps in the hash-table. Recursion steps that have different cold parameters must be issued different IDs since decisions involving, for instance, $\mathbf{DFT}_8$ and $\mathbf{DFT}_4$ might not the same.

2. All recursion steps must provide a `clone()` method that is used to test a strategy without committing to it.

3. Finally, a `time()` method is required for all recursion steps. Dummy hot parameters need to be created and fed in the computation during this process.

Results. In practice, our dynamic programming implementation has been shown to perform well while being entirely generic. It is therefore featured in all the Spiral-generated libraries

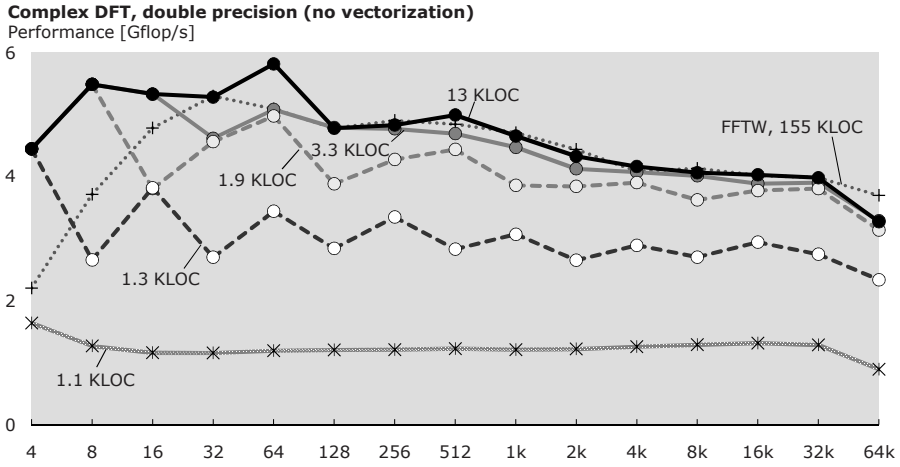


FIGURE 4.12: Dynamic programming powering five DFT libraries that differ by their code size (expressed in kilo-lines of code). The libraries are non-vectorized single-threaded Cooley-Tukey FFT. The platform is a 3GHz Intel Xeon 5160 and the libraries are compiled with `icc` 10.1. Since the libraries differ by the size of the base cases they embed, producing the same graph without online adaptation would have required to develop five different heuristics. FFTW code size is presented for comparison but includes other transforms and supports non two-power sizes, and is hence one order of magnitude longer.

[Voronenko et al., 2009] and constitutes the base line that all other search strategies need to compete with.

The advantage of having a generic search method is indirectly illustrated in Figure 4.12. The plot presents the possible tradeoffs between code-size and performance which arise by embedding more or less codelets with the library. Without online adaptation, five different heuristics would have been needed, one for each library, while dynamic programming provides a solution that exhibit a very good performance.

As we will see in later benchmarks, the performance achieved by DP is generally very good, except for very large transforms. DP terminates quickly for small and medium DFT sizes. However, for large sizes and certain libraries, the combinatorial explosion of choices slows DP as all subproblems have to be solved before any solution is returned. For example, in FFTW, which also implements DP, it can amount to hours in the case of large transforms on multicore systems. To mitigate this problem, the next subsection presents a different search strategy based on reinforcement learning.

4.2.3 Bandit-Based Algorithm

As the search space becomes bigger, because of larger problem sizes or because of more complicated libraries, two weaknesses of dynamic programming become apparent: the time to solution gets prohibitive and the quality of the solutions degrades. The latter may be a waste since the library

could have the ability to run faster. The former problem cannot easily be solved as DP needs to terminate to provide any solution.

Our goal in this subsection is to provide an alternative to DP that helps solving both problems. In particular, we want our search strategy to provide a continuum of solutions, that is, solutions that get increasingly better as time passes. For instance, a user interested in waiting little time would get a reasonable solution, while another user could wait a bit more and get a better library. An algorithm that presents this property is said to be *anytime*—it can be stopped at anytime, a solution being always available.

In this subsection, we describe a novel anytime algorithm that is inspired by recent advances in computer Go. It can cut down the search time by an an order of magnitude compared to DP and, in some cases, the performance of the implementations found is also increased by up to 10%.

4.2.3.1 Intuition

The key difficulty in developing good anytime search strategies for online adaptation is that it is extremely difficult to evaluate the impact of a single decision: the decision itself might open additional choices and the evaluation of the performance only comes at the end, when all choices have been decided. In other words, our situation is the one of a beginner in some board game: every decision matters but victory or defeat are only known at end. This realization is exactly what triggered our interest in finding connections with machine learning in general and reinforcement learning in particular. After investigation, it became clear that the recent advances in developing good computer Go players produced a class of algorithms that could be applied to our own online adaptation problem.

Go is an ancient Chinese board game considered much more difficult to solve than chess for computers [Müller, 2002]. The main reason of its difficulty is precisely that there is no known function that evaluates properly intermediary positions due to the high number of possible moves and the lack of simplifications as time passes, e.g. pieces are not removed from the board, but added instead.

Yet, advances in the field have boosted the computer level from total beginner to advanced amateur during these last years, and that even without proposing an evaluation function. These new strategies all derive from traditional Monte-Carlo (sampling) methods [Metropolis and Ulam, 1949], which have been heavily used in physics [Landau and Binder, 2005] and finance [Boyle et al., 1997]. The basic idea of Monte-Carlo is simple: since there are so many possible moves, one might as well play randomly and observe what works well [Bouzy and Helmstetter, 2003; Bruegmann, 1993]. More precisely, the move to be played starting from a given position is the move that leads to the highest percentage of victories if both players were playing randomly (see Figure 4.13). The underlying assumption is that the space explored by taking decisions uniformly is not biased in comparison to the space explored by taking good decisions.

Note that this simple method is directly applicable to our problem and probably constitutes the

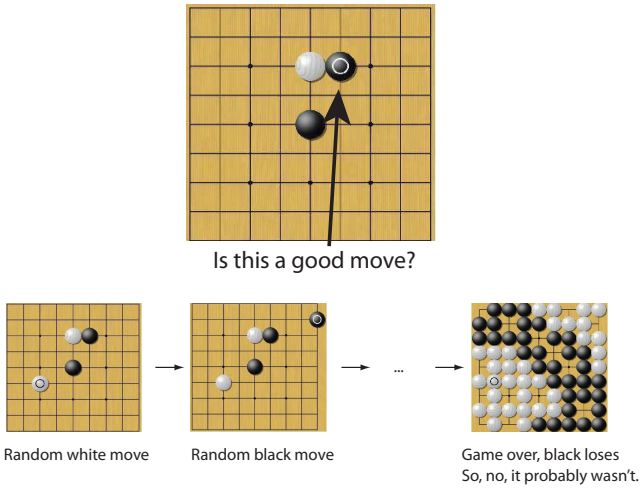


FIGURE 4.13: Monte-Carlo Go. Assessing the quality of a move is done by playing randomly until the game is settled. In practice, more than one simulation is done and the results are averaged.

simplest anytime strategy: each time there is a decision to take, one could choose according to a uniform distribution[7]. At the end of the descent, it evaluates (times) the library and then tries again. At any point in time the user can interrupt the search to retrieve the best known candidate. Actually, even if there seems to be little refinement in this method, such a sampling strategy works decently well, both in Go and with Spiral-generated libraries[8].

The big weakness of the traditional Monte-Carlo is that the uniform decision making means that a lot of trials are wasted in irrelevant areas of the search space. A breakthrough to overcome this limitation came in 2006 with an algorithm called Upper Confidence Bounds applied to Trees (UCT) [Kocsis and Szepesvári, 2006]. The idea was to bias the trials in a way that emphasizes the search along successful moves while still allowing for other moves to be explored. The balance is done by using a *bandit* algorithm jointly with Monte-Carlo simulation. The method proved very successful in Computer Go: all Go program that won the Go Computer Olympiads since 2007 have used derivatives of the technique [Coulom, 2007; Gelly and Silver, 2007; Wang and Gelly, 2007].

Introduction to bandits. Historically, slots machines have been known as one-armed bandits because they are operated by pulling a lever on their side and leave the gamers penniless. In statistics, the problem of maximizing the expected sum of rewards of k slot machines with different payout distributions is therefore designated as the "classical" k-armed bandit problem (Figure 4.14). A concrete example of the problem is somebody that has been given a certain amount of tokens to play slot machines in a casino and trying to maximize his or her gains. The key difficulty here is managing the tradeoff between *exploitation*, that is playing the machine with the highest payout so

[7]Since at each step, there is an equal chance for all branches to be picked but branches are not laid out uniformly, the overall online adaptation space is *not* sampled uniformly.

[8]It helps that many deterministic optimizations are already built-in all Spiral-generated libraries.

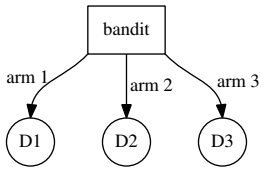


FIGURE 4.14: A 3-armed bandit. Arm i produces an output according to distribution D_i.

far, and *exploration*, that is assessing if there are any better machines around.

Maximizing the cumulative reward is equivalent to minimizing the so-called *regret* which is the loss since the algorithm does not always choose the best arm. Assuming that each arm i ($1 \leq i \leq k$) if the machine has an unknown expectation μ_i and that n_i is the number of times the arm i has been selected during the first n steps. Let $\mathbb{E}$ denote the expectation, the regret $R(n)$ is then defined by

$$R(n) = \left(n * \max_{1 \leq i \leq k} \mu_i \right) - \sum_{i=1}^{k} \mu_i \mathbb{E}[n_i].$$

The other terms of the formula being fixed, the regret can only be minimized by minimizing $\mathbb{E}[n_i]$. [Lai and Robbins, 1985] provides a lower bound for this expression proving that the best possible regret grows as $\Omega(ln(n))$.

A simple algorithm called *Upper Confidence Bound* (UCB) achieves this bound and is therefore asymptotically optimal [Auer et al., 2002]. Assuming $\bar{\mu_i}$ is the realized average reward for the arm i so far, the algorithm advises to pull the arm i_{best} given by

$$i_{\text{best}} = \underset{1 \leq i \leq k}{\operatorname{argmax}} \; h(n_i),$$
$$\text{with } h(n_i) = \begin{cases} \bar{\mu_i} + \sqrt{\frac{2\log(n)}{n_i}}, & \text{if } n_i > 0 \\ \infty, & \text{else} \end{cases}$$

In words, the left part of the sum ($\bar{\mu_i}$) has a high value only if the current arm has given good rewards so far—it represents the exploitation term which stresses that good arms should be favored. The right part (the square root) is the exploration term that makes sure that good opportunities are not lost—it gets bigger if the current arm has not been tested often (compared to the others). To guarantee that all arms have a chance, we also assign an infinite weight to arms that have not been tested at all.

4.2.3.2 Algorithm overview

Recent advances in computer Go have been mostly inspired by an algorithm called UCT whose idea is to use the bandit-algorithm UCB to bias the Monte-Carlo simulations towards the best games.

In this subsection, we introduce a novel online adaptation algorithm named *Threshold Ascend on Graph* (TAG) that follows the same inspiration: Similarly to UCT, it gradually builds up a structure by considering local bandit problems and by valuating the nodes with Monte-Carlo simulations. However, it differs from UCT in two aspects: it optimizes for the single maximum reward instead of shooting for the biggest accumulative reward and it does so over a graph instead of doing it over a tree. Before presenting the algorithm itself, we start by abstractly formulating the library online adaptation problem (Problem 1) as an optimization problem over the language generated by a large acyclic formal grammar. Each word in the language generated by the grammar corresponds to a recursion strategy the library can perform.

Grammar formulation. The online adaptation problem can be specified using a formal grammar $G = (T, N, P, S)$. The start symbol S is the functionality specification, including the problem size, as entered by the user. The terminals T are the *base cases*, the set of problems that can be directly solved by the library. The non-terminals N are the set of recursion steps, non base case subproblems that could be needed to solve the problem. The production rules P breakdown a problem from N into one or more subproblems by fixing a degree of freedom. The function to maximize, f, is the performance of the implementation. The acyclicity of the grammar is guaranteed by the underlying algorithms that provably finish. Note that the grammar itself changes with the problem size.

For instance, if a naïve DFT library based on Cooley-Tukey is used to compute $\mathbf{DFT}_8$, we would define

$$\begin{aligned} S &= \mathbf{DFT}_8 & P = \{&\mathbf{DFT}_8 \to (\mathbf{DFT}_2, \mathbf{DFT}_4),\\ T &= \mathbf{DFT}_2 & &\mathbf{DFT}_8 \to (\mathbf{DFT}_4, \mathbf{DFT}_2),\\ N &= \{\mathbf{DFT}_8, \mathbf{DFT}_4\} & &\mathbf{DFT}_4 \to (\mathbf{DFT}_2, \mathbf{DFT}_2)\} \end{aligned}$$

Formal problem statement. Below, we formally restate the online adaptation problem using formal grammars.

PROBLEM 3 (*Grammar Optimization)* Given is an acyclic formal grammar $F = (T, N, P, S)$ with T the set of terminals, N the set of nonterminals, P the set of production rules or simply rules, and S the starting symbol. $\mathcal{L}(F)$ is the associated language and f is an objective function from $\mathcal{L}(F)$ into the positive reals $\mathbb{R}^+$. We want to compute

$$w_{\text{best}} = \operatorname*{argmax}_{w \in \mathcal{L}(F)} f(w).$$

F has an associated *derivation graph* $G = G(F)$ which is directed, acyclic and weakly connected as shown in Figure 4.15: S is the root, the directed edges (arrows) correspond to applications of rules in P, the nodes are partially derived words in the language, and the sinks (outdegree = 0)

$$
\begin{aligned}
T &= \{\text{a, b, ac}\} \\
N &= \{\text{S, A, B}\} \\
P &= \{\text{S} \to \text{AB}, \\
&\quad \text{A} \to \text{a}, \\
&\quad \text{B} \to \text{b}, \\
&\quad \text{aB} \to \text{ac}\}
\end{aligned}
$$

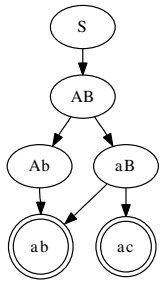


FIGURE 4.15: Formal grammar $F = (T, N, P, S)$ (*left*) and associated derivation graph $G(F)$ (*right*). S, A, B are nonterminals and a, b, c are terminals. The graph has two sinks (double circled), i.e., the language $\mathcal{L}(F)$ has two elements.

are precisely the elements of $\mathcal{L}(F)$. Hence we can reduce Problem 3 to:

PROBLEM 4 (*Graph Optimization)* Given a weakly connected, acyclic, directed graph $G = (V, E)$ and an objective function f (as above) on the sinks $S(G)$ of G. We want to compute

$$w_{\text{best}} = \operatorname*{argmax}_{w \in S(G)} f(w).$$

We assume the graph $G(F)$ to be large such that it is impossible to generate and evaluate all sinks in a reasonable time. Our goal is to find an algorithm that discovers a "very good" sink with a small number of evaluations.

4.2.3.3 The TAG Algorithm

TAG is an *anytime* algorithm that determines an approximate solution to Problem 4. Due to the size of the graph it is not meant to run until completion, in which case it would be equivalent to an exhaustive search.

TAG finds solutions by incrementally growing and exploiting the subgraph $\hat{G} = (\hat{V}, \hat{E})$ of $G = (V, E)$: $\hat{V} \subset V$, $\hat{E} \subset E$, starting with $\hat{G} = (\{S\}, \{\})$. Evaluations are used to direct the growth of $\hat{G}$ towards the expected bests sinks.

Assume the current subgraph is $\hat{G}$. Then TAG proceeds in three high level steps visualized in Figure 4.16:

1. *Descend:* G is traversed starting at its root. Each choice along the way is solved by a *bandit algorithm*. The descent stops when it uses an arrow e that is not in $\hat{E}$.

2. *Evaluate:* If e is incident with a vertex not in $\hat{V}$, this vertex is evaluated using a Monte-Carlo expansion.

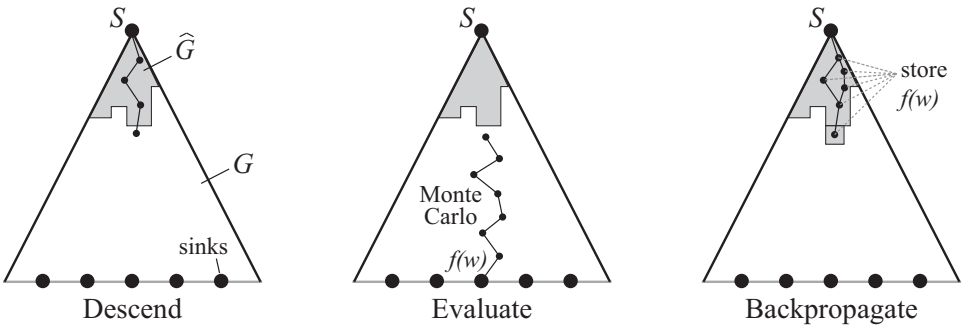


FIGURE 4.16: Visualization of the three main steps in Threshold Ascend on a Graph (TAG) [de Mesmay et al., 2009]. The problem of searching the best algorithm is equivalent to finding the best valuated sink in a connected directed acyclic graph G. This algorithm essentially proposes to grow a subgraph $\hat{G}$ in the direction of the best sinks. In a first phase, the most promising node right outside of $\hat{G}$ is added to $\hat{G}$ by descending from the root S and considering each choice as a *maximum k-armed bandit problem* [Cicirello and Smith, 2005; Streeter and Smith, 2006]. Starting from that node, a sink is selected by randomly walking the graph. Every node along the descent is finally back-propagated the value obtained by that sink so that the next descent step has more information when choosing a new node. Note that $\hat{G}$ (shaded area) and G are not trees (e.g., see Figure 4.15).

3. *Backpropagate:* The evaluation is stored in all ancestors of the vertex.

We proceed with introducing the bandit problem we use, describing the three steps in detail and providing the pseudocode.

Background: maximum bandit. The "classical" bandit problem we just described does not fit the context of online adaptation since we are interested in finding the fastest computation and not in minimizing the overall length of the computations. We therefore have to consider a different version of the problem called the *maximum* k-armed bandit in which the goal is to maximize the single best reward obtainable over $\bar{n}$ trials [Cicirello and Smith, 2005]. Formally, if each arm has distribution D_i and $R_j(D_i)$ denotes the j-th reward obtained on arm i, the goal is to solve

$$\max_{\sum_{i=1}^{k} \bar{n}_i = \bar{n}} \max_{1 \leq i \leq k} \max_{1 \leq j \leq \bar{n}_i} R_j(D_i).$$

In this document, we use a straightforward variation: an anytime version of the problem where the total number of pull $\bar{n}$ is not known in advance. Only the n previous pulls and their associated rewards are known.

Threshold Ascend (TA) is an algorithm that solves the maximum problem without making assumptions on the form of the distributions [Streeter and Smith, 2006]. The main idea is to track only the s best rewards and the arms they are coming from. Let s_i be the number of such rewards among the n_i rewards received by the arm i. Also, let δ be a positive real parameter. The algorithm

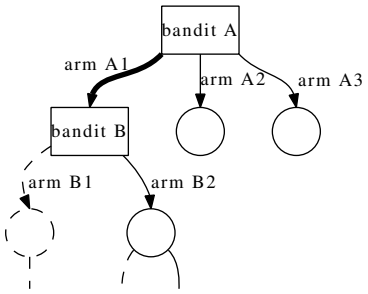


FIGURE 4.17: The descent in the graph is done as a cascade of multi-armed bandits. Solid arrows, circles and boxes are in $\hat{G}$, dashed arrows and circles are in $G \setminus \hat{G}$. For bandit A all arms had been played before, and A1 is chosen based on the stored rewards. Bandit B will now choose B1, since it is the only arrow not played before.

advises to pull the arm i_{best} given by

$$i_{\text{best}} = \operatorname*{argmax}_{1 \le i \le k} h(s_i, n_i),$$

$$\text{with } h(s_i, n_i) = \begin{cases} \frac{s_i + \alpha + \sqrt{2 s_i \alpha + \alpha^2}}{n_i}, & \text{if } n_i > 0 \\ \infty, & \text{else} \end{cases}$$

and $\alpha = \ln(2nk/\delta)$.

Descend. The goal of the *descent* step is to select the next edge to add to the subgraph $\hat{G} \subset G$, initially restricted to the root. The purpose of the descent is to select an arrow in $E \setminus \hat{E}$ that leads towards the best expected rewards. It does so by tracing a path starting from the root and considering each successor choice as a max k-armed bandit problem (Figure 4.17). For now, assume that a table of positive real rewards $R(v)$ has been maintained for each vertex $v \in \hat{V}$.

Let v denote the current vertex in the descent. Starting from v, there are multiple ways to continue the path since it can follow any of the arrows originating from v (we denote these with $E(v)$). The arrows in $E(v)$ that are also in $\hat{E}(v)$ lead to vertices of $\hat{V}$ corresponding to "arms" that have already been played (they have previous rewards attached to them). The other arrows lead to arms that have never been played. The bandit algorithm discussed above decides which arrow to follow, which has to be one that was not followed before if such an arrow exists (due to the infinite weight in $h(s_i, n_i)$). If the arrow belongs to $\hat{E}(v)$ and the successor is not a sink, the successor becomes the new descent vertex and the descent continues. If not, the descent ends.

Evaluate. Assume the descent ended on an arrow pointing to a vertex v that is not part of $\hat{V}$. The arrow and vertex are then immediately added to $\hat{G}$ and v is *evaluated*.

If v is a sink of G, then $f(v)$ can be directly computed. Otherwise a path to a sink of G is chosen by "Monte-Carlo," which means in each step a (uniformly drawn) random choice is made until a sink w is obtained. The evaluation $f(w)$ gives a value for v.

Also, if the evaluation is better than $f(w_{\text{best}})$, the current best sink is replaced.

Backpropagate. After v has been evaluated, the reward is added to its reward list $R(v)$ and to the reward lists of all its ancestors.

Note that if the descent ended on an arrow pointing to a vertex v that is already a part of $\hat{V}$, we just discovered a new way to connect to an already evaluated vertex. In this case, we add the new arc to $\hat{E}$ and propagate the rewards of v only to the vertices that would not be ancestors of v without the new arrow (since the other ancestors already have these rewards).

Pseudocode. Figure 4.18 summarizes the previous discussion by presenting the pseudocode of TAG. After initialization, the graph $\hat{G} = (\hat{V}, \hat{E})$ is grown one arc at a time until the user signifies an interruption. The vertex pointed by an arrow e is denoted *head(e)*. `BANDIT` refers to the *Threshold Ascend* algorithm summarized earlier. `RANDOM` refers to an uniform draw.

Remark. Practically, if the objective function is deterministic, it is useless to evaluate a sink twice. It is therefore possible to modify the algorithm to guarantee that it never returns in a branch where choices have been exhausted. This obfuscates the explanation without adding real benefits to the pseudocode so we choose not to present this version here.

4.3 Offline Adaptation

In Section 4.2, we have presented a mechanism to provide *online* adaptation to a generated library. That is, enabling the user of the library to tune to his own machine his size of interest (or set of sizes). This scenario is not be suitable to all users for two main reasons: First, online adaptation naturally complicates the library interface as the problem descriptor has to be exposed to the user. Therefore, pre-existing legacy interfaces simply cannot be supported. Second, the usage profile of some users simply might not fit the bill for online adaptation. In particular, if the user is changing sizes often, the time spent in the planner will not be recovered during the computations.

In this section, we optimize a library for a target computer *without* having to specialize it for a given size. This process is done at installation and automatically generates a set of specialized decision trees that support *all* problem sizes and are traversed at runtime with little overhead. Key advantages of the method include:

- The capability of factoring knowledge about some sizes to infer the general performance profile of the library, resulting in an adaptive library that does not need to search for all sizes. In related manner, it allows to concentrate search at installation time, making shared use much more efficient (e.g., one time deployment on a super-computer).

- The possibility of providing a much simpler interface to the user, reducing the possibility of misuses. This relaxation in the constraints makes it possible to match existing legacy interfaces.

TAG Algorithm

$\hat{G} \leftarrow S$
$w_{\text{best}} \leftarrow \emptyset$
$R(\hat{V}) \leftarrow \emptyset$
while not interrupted **do**
  $e \leftarrow$ BANDIT$(E(S))$ *Descend*
  while $e \in \hat{E}$ & $E(head(e)) \neq \emptyset$ **do**
    $e \leftarrow$ BANDIT$(E(head(e))$
  end while
  $v \leftarrow head(e)$
  if $v \notin \hat{G}$ or $e \in \hat{G}$ **then**
    add v and e to $\hat{G}$
    $e \leftarrow$ RANDOM$(E(v))$ *Evaluate*
    while $E(head(e)) \neq \emptyset$ **do**
      $e \leftarrow$ RANDOM$(E(head(e))$
    end while
    $w \leftarrow head(e)$
    if $f(w) > f(w_{\text{best}})$ **then**
      $w_{\text{best}} \leftarrow w$
    end if
    $r \leftarrow f(w)$
    add r to $R(v)$ *Backpropagate*
    for a ancestor of v in $\hat{G}$ **do**
      add r to $R(a)$
    end for
  else
    for a ancestor of v in $\hat{G}$ **do**
      mark a
    end for
    add e to $\hat{G}$
    for a ancestor of v in $\hat{G}$ **do**
      if a is marked **then**
        unmark a
      else
        add all $R(v)$ to $R(a)$
      end if
    end for
  end if
end while
return w_{best}

FIGURE 4.18: Pseudo-code for the Threshold Ascend on a Graph (TAG) Algorithm.

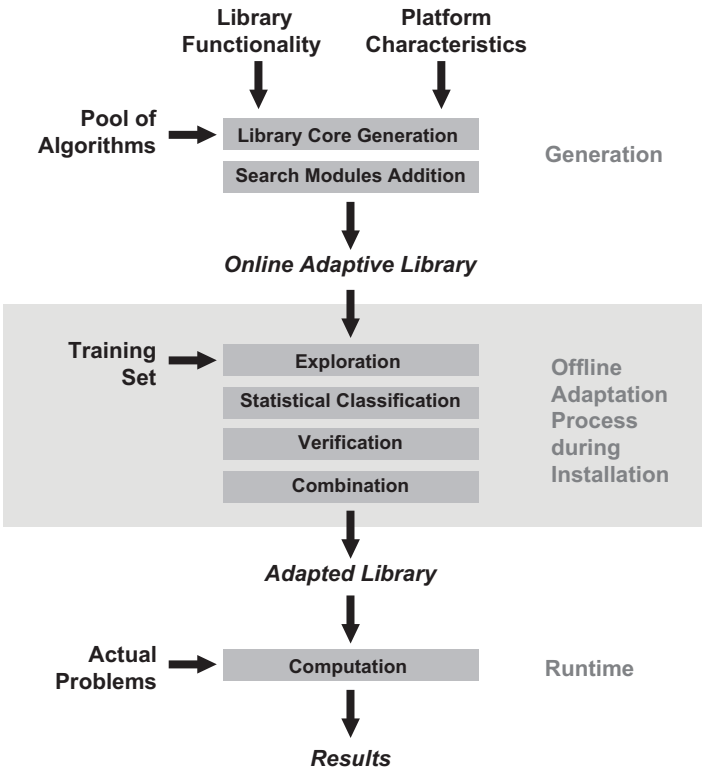


FIGURE 4.19: The offline adaptive library creation stack: generation, installation and utilization.

- The faculty of developing a low-latency library as no mechanisms are used for loading and interpreting plans. Incidentally, this provides much safer assurances on the robustness of the library.
- The possibility of pruning complex libraries into simpler ones by cutting rules that are sub-optimal.
- Finally, the generated decision tress provide a mechanism for the developer to understand when certain choices are taken.

Offline search is very different from previous searches in that the sizes that are interesting to the user are actually never specifically taken into account by the search mechanism. Therefore, the offline search relies on the *generalization* capabilities of a learning algorithm working with a restricted set of sizes that are assumed to be of general interest.

In practice, the generation of an offline adaptive library is characterized by two distinct phases:

1. An *online* adaptive library is generated as explained in Section 4.2. This library is then optimized on the *training set*, a set of sizes that are known to be useful to most users. Any of the search algorithms presented above can be used in this process.

2. A statistical classifier generalizes the knowledge learned during the optimization of the training set and produces decision trees that can tackle any size. These decision trees are then verified and, if needed, corrected, to enforce the applicability conditions of the rules. They are then combined with the library core that was generated earlier to produce the final adapted library.

To summarize the process, Figure 4.19 depicts how generation, installation and utilization play together.

Obviously, the statistical classifier plays here the key role so we will first review the concepts and mechanisms behind it before explaining how it can be used to derive a custom library.

4.3.1 Overview of the Approach

In this subsection we use a machine learning technique to solve Problem 2 while achieving similar performance as search-based solutions of Problem 1. The function $D^p(n)$ will be generated as a set of decision trees.

Several efficient methods to solve Problem 1 have been proposed in Section 4.2. Hence, we assume that Problem 1 is solved (even if time-consuming).

Our approach uses solutions D_n^p for several n as training set to solve Problem 2, i.e., to generate the heuristic $D^p(n)$. The approach is summarized in Figure 4.19. It decomposes in four phases:

1. *Exploration:* For a training set of sizes n, D_n^p are computed.

2. *Statistical classification:* We use the C4.5 classifier [Quinlan, 1993] to generalize the knowledge learned during the exploration to produce $D^p(n)$ in the form of a set of decision trees. If needed, "hints" based on the library functionality are provided to help with the generalization.

3. *Verification:* The decision trees are verified and, if needed, corrected, to enforce the constraints that are imposed by the library.

4. *Combination:* Finally, the generated decision trees are inserted into the library as heuristics and replace the parts where the online decision code was called. The final result is a library that is adapted to the target platform.

4.3.2 Background: Inducing Classification Models

The goal of decision tree learning is to create a model that classifies records based on a training set of already classified records. It is built by recursively partitioning the training set using well chosen tests. The interest is two-fold: On one hand, it helps *summarizing* and *understanding* the already known training data and on the other hand it gives a *simple* mechanism to *infer* a classification for new cases. The most famous algorithm for producing such decision trees is probably Quinlan's C4.5

Outlook	Temperature	Humidity	Windy	Decision
sunny	85	85	false	don't play
sunny	80	90	true	don't play
overcast	83	78	false	play
rain	70	96	false	play
rain	68	80	false	play
rain	65	70	true	don't play
overcast	64	65	true	play
sunny	72	95	false	don't play
sunny	69	70	false	play
rain	75	80	false	play
sunny	75	70	true	play
overcast	72	90	true	play
overcast	81	75	false	play
rain	71	80	true	don't play

TABLE 4.2: The famous "weather" machine learning data set.

[Quinlan, 1993] which is based on his earlier algorithm ID3 [Quinlan, 1986] but supports numerical features. Their behavior is best understood by walking through an example and we will use for this purpose the famous "weather" data set.

Example: golfing or not golfing. A golf manager has observed that the attendance varies greatly depending on the weather (Table 4.2) and is trying to understand the pattern behind it in order to manage his staff better. Before explaining the algorithm any further, we invite the reader to directly look at Figure 4.20, which is the decision tree produced by C4.5 for such a data set. Observe how the tree concisely captures the decisions in the table and also enables generalization for cases that not in the table. The main difficulty is to determine which question should be asked in order to partition the cases in the most meaningful way. Note that practical algorithms only consider local decisions since the problem of learning an optimal tree has been proved to be NP-complete [Hyafil and Rivest, 1976].

Entropy of an event. ID3 and C4.5 are both based on Occam's razor, preferring simpler explanations over complicated ones: the goal is to always maximize the information gain. To do this, the algorithms rely on the concept of *entropy*, which is a measure of the information content of a distribution, and was first introduced by Shannon [Shannon, 1948],

Formally, suppose that the final decision can take any of the values $\{d_1, \ldots, d_n\}$ (here: golfing or not) and that the feature a (e.g., temperature) can take any of the values $\{a_1, \ldots, a_m\}$. We denote by $P(d_i|a = a_j)$ the conditional probability that decision d_i is made given that a takes the value a_j. By definition, the quantity

$$H(a = a_j) = -\sum_{i=1}^{n} P(d_i|a = a_j) \log_2 P(d_i|a = a_j)$$

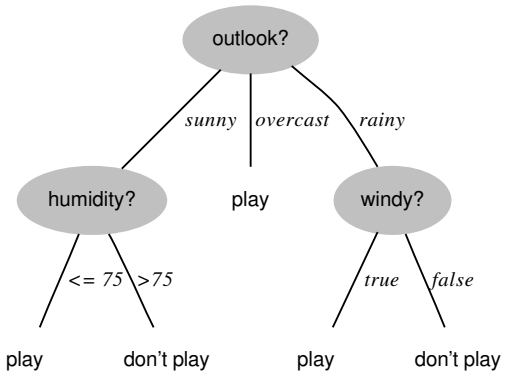


FIGURE 4.20: Decision tree generated by C4.5 for the "weather" data set (Table 4.2).

is called the *entropy* of the event "$a = a_j$" and measures the level of uncertainty that remains about the final decision if the said event happens. It is computed in *bits* (a fair coin has an entropy of one bit).

In our example, we can use Table 4.2 to first compute all conditional probabilities and then the entropy of events such as

$$\left.\begin{aligned} P(\text{play}|\text{outlook=overcast}) &= 4/4 = 1 \\ P(\text{don't play}|\text{outlook=overcast}) &= 0/4 = 0 \end{aligned}\right\} \Rightarrow H(\text{outlook=overcast}) = -(1\log_2 1 + 0\log_2 0) = 0$$

$$\left.\begin{aligned} P(\text{play}|\text{windy=false}) &= 6/8 \\ P(\text{don't play}|\text{windy=false}) &= 2/8 \end{aligned}\right\} \Rightarrow H(\text{windy=false}) = -(6/8\log_2 6/8 + 2/8\log_2 2/8) = 0.81$$

We see that, once golfers know that the outlook is overcast, their decision is already taken (they will definitely play). If it is not windy, uncertainty remains.

Entropy of a feature. Computing the weighted sum of the entropies over all the possible values a feature may take yields the entropy of the feature:

$$H(a) = \sum_{j=1}^{m} P(a = a_j)H(a = a_j)$$

The feature with the smallest entropy is the one that best partitions the training data set and is therefore the one that should be placed at the root of the decision tree. Recursively applying this process yields a full decision tree.

In our example, we observe that the "outlook" feature discriminates better than the "windy"

Temperature	64	65	68	69	70	71	72	72	75	75	80	81	83	85
Play?	Yes	No	Yes	Yes	Yes	No	No	Yes	Yes	Yes	No	Yes	Yes	No

TABLE 4.3: Possible split points (without merging) for the temperature feature of the weather data set.

feature, hence it becomes the root of the decision tree in Figure 4.20:

$$H(\text{outlook}) = 5/14 * 0.97 + 4/4 * 0 + 5/14 * 0.97 = 0.69$$
$$H(\text{windy}) = 6/14 * 1 + 8/14 * 0.81 = 0.89$$

Numerical features. In our example, "temperature" and "humidity" are both described by continuous ranges rather than by discrete classes and therefore, cannot be directly taken into account by the above algorithm. It is clear, however, that any numerical range can be split into two classes using a threshold: one containing values that are bigger and the other containing values that are lower or equal. This process is called discretization. The key question that remains is how to select such a threshold.

C4.5 selects the threshold that provides the most information. Furthermore, no knowledge exists for values between split points; hence, at worst only as many split points as there are values need to be considered. As a consequence, dealing with numerical features is more expensive than dealing with classes and the splitting may have to be repeated. However, there are methods to reduce the number of possible split points and, in particular, it has been proved that it is useless to consider a split point inside a range where the decision doesn't change [Fayyad and Irani, 1993]. As a consequence, we show on Table 4.3 the possible split points for the temperature feature. In practice, heuristics are also used to merge adjacent groups together, preventing any overfit caused by potential outliers.

Shortcomings. C4.5 is limited to classifying rectangular regions in the feature space. This is due to the conjunctive partitioning system that can only produce expressions of the type $(x > 16) \wedge (y > 3) \wedge (x \leq 70)$. Therefore, it fails at properly handling XOR and parity problems.

In the context of adaptive numerical libraries, C4.5 will therefore not be able, unless it is "hinted", to recognize a decision that is beneficial only if two features are equal (e.g., the input and output strides). Also it will not be able to recognize number theoretic properties that may be significant (e.g., divisibility for the radix choice).

4.3.3 Generating Decision Trees for Libraries

In this subsection, we first explain how statistical classification can be used to convert an online adaptive library into an offline one. Then we explain two techniques, *hinting* and *automatic correction* that enlarge the class of problems that the classifier can tackle.

4.3.3.1 Mapping of the Problem

The structure of the choices inside an adaptive library has been presented earlier in Figure 4.7, Figure 4.8 and Figure 4.9. We remind the reader that each of these choices is qualified which means that it depends on the parameters of the recursion steps: the questions read "should dft *of size 1024* be threaded?", "should the strided dft *of size 16 and stride 4* be implemented as a base case?," and so on.

Using online adaptation and a training set, it is fairly easy to collect a choice (e.g., "should dft *of size* n be threaded?") and the corresponding answers (e.g., for size 256 and 1024, no threading is required but it is needed for size 4096). Using C4.5, we then integrate all the answers in a single decision tree that constitutes a heuristic for the choice. More details are provided below.

Features. Selecting relevant features is usually the crucial problem for machine-learning based compilation. However, in our case, it is quite natural to assume that the best decisions only depend on the cold arguments of the recursions steps since it is what we already assumed for dynamic programming[9]. For example, in Figure 4.6, the only relevant feature for the heuristics of `dft` is the size `n`; for `dft_strided`, it is the size `n` and the input stride `istr`.

Decisions. The set of choices is fixed by the underlying online adaptive library. Since each choice is either binary or takes a numerical value, C4.5 is clearly applicable.

Training set. The training set for the main recursion step has to be selected by the developer of the library. As we will show in Section 5.5, it is interesting to choose a variety of sizes that encompasses the different performance regimes of the library. Note that training cases for the additional recursion steps are automatically derived, since they consist in the different possibilities that stem from the main recursion step.

4.3.3.2 Advanced Manipulation of the Decision Trees

C4.5 is limited from a learning point of view and might require good *hints* to produce good results. Due to the finite character of the training set, it might also generate trees that cannot be generalized as heuristics and therefore require an *automatic correction* pass.

Hinting. C4.5 can only cut the feature space into orthogonal rectangles and can only consider one dimension at a time. In some cases, these restrictions prevent a good generalization which in turn leads to a disappointing performance.

For instance, we have seen that, in DFT libraries, the choice of the radix must divide the size so the performance actually depends on the prime factorization of a number. Since multiples of 2 and multiples of 3 are interleaved and even mixed on the real axis, C4.5 is not able to discover by itself that both groups naturally exhibit a very different behavior. The tree C4.5 then produces treats individual cases, shows no global understanding of the problem and, ultimately, performs poorly.

[9]Note that TAG solutions can be forced to comply to the assumption by making sure that subtrees of the same algorithms are necessarily implemented in the same way.

However, it is possible to drastically improve the quality of the trees by providing "hints" to the classifier. A hint is a function that is directly computed from the features and fed to the classifier as if it was an extra feature. For instance, in the above case, hints that could be provided are the number of powers of 2 and of powers of 3 in the prime decomposition of the size. In practice, this creates two new dimensions, which enables the selection of meaningful groups using only rectangles. Finally, note that designing hints is not pushing the burden onto developers: anybody using DFTs knows that powers of two sizes are inherently very different from non-powers of two. Writing a hint is only giving this information to the classifier without explaining the implications of such a difference—the classifier will figure that out for itself.

Figure 4.21 shows a generated heuristic that chooses a radix as a function of the size. The functions `nfactor(f, n)`, that computes the number of times the factor `f` appears inside `n` were provided as hints to the classifier. The comments in the code display the effect of the automatic correction which is the topic of the following paragraph. It can be noticed that, for this recursion step, this library and this training set, the chosen radix is often, *but not always*, the largest integer of $\{2, 3, 4, 6, 12, 18\}$ that divides the size.

Automatic correction. The C4.5 classifier is, in some way, short-sighted: it can come up with decisions that are not correct in the general case but were true inside the finite training set. For instance, if the classifier trains only on the set $\{12, 14, 16, 18, 20, 22, 24\}$, it might conclude that the best radix for any number divisible by 3 is 6 (since all numbers of the set divisible by 3 are also divisible by 6), overlooking the fact that choosing such a radix also needs a factor 2 which might lead to unpleasant surprises at runtime.

Consider Figure 4.21. We want to make sure that the returned radix is always valid, that is, always divides the input size. As an example, we focus on the first `return 8`. It is easy to prove that 8 is always a correct radix because the above condition, `nfactor(2, n) <= 3` is false so `n` is necessarily a multiple of 8. If we focus on the `return 18` line now, we observe that `n` is only guaranteed to be divisible by 6. A correction needs to take place there.

In our system, this verification phase is mechanized by traversing each tree and automatically correcting decisions (leaves) that cannot be justified by the information contained in the internal nodes. Correcting means that decisions are either changed to more conservative ones or additional internal nodes are added. We also insert a errors in case all tests are failed which means that the situation was not learned during training.

```
choose_dft_radix(int n) {
   if ( nfactor(3, n) <= 0 ) {
      if ( nfactor(2, n) <= 2 ) {return 2;}    //Corrected to: error()
      else {
         if ( nfactor(2, n) <= 3 ) {return 4;}
         else {return 8;}
      }
   }
   else {
      if ( nfactor(2, n) <= 1 ) {
         if ( nfactor(2, n) <= 0 ) {return 3;}
         else {
            if ( nfactor(3, n) <= 1 ) {return 2;}
            else {return 6;}
         }
      }
      else {
         if ( nfactor(2, n) <= 3 ) {
            if ( nfactor(2, n) <= 2 ) {
               if ( nfactor(3, n) <= 1 ) {return 6;}
               else {return 12;}
            }
            else {return 18;}                  //Corrected to: return 6
         }
         else {return 12;}
      }
   }
}
```

FIGURE 4.21: A computer-generated heuristic for choosing the radix in the implementation from Figure 4.6. The function `nfactor(f,n)` returns the number of times the factor `f` appears inside `n` and constitutes an hint. After the generation of the heuristic, the automatic correcter enforces the divisibility policy (the effect of which is displayed in the comments).

CHAPTER 5

Experimental Results

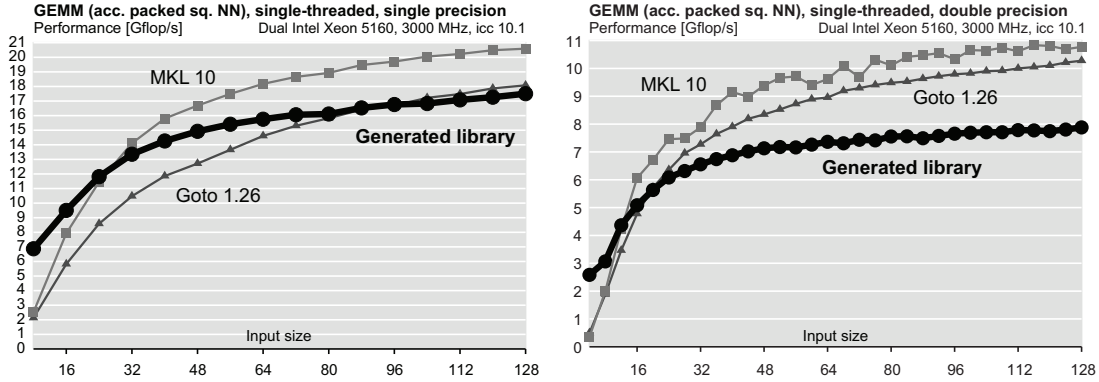
In this chapter, we show experimental results that we obtained with our library generation and adaptation work. We start with performance results for the OL specific functionalities we presented in Chapter 2 and Chapter 3: matrix-matrix multiplication and Viterbi decoding. We then present some of the capabilities of the Java back-end that we developed using linear transforms as example. Finally, we conclude with the evaluation of both the online and the offline adaptation mechanisms from Chapter 4.

5.1 Matrix-matrix multiplication

Experiments on x86. We generate a single and a double precision GEMM libraries on the model that was described in Section 3.4. We perform different series of experiments but in all cases, the operation count is $2mnk$.

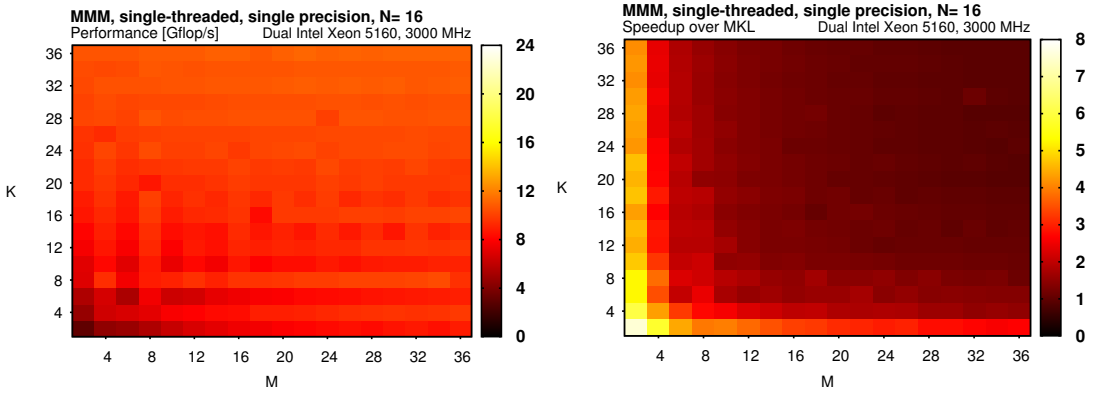
The libraries are compiled using the Intel Compiler 10.1 and benchmarked on a 64-bit Linux platform using a 3 GHz Intel Xeon 5160 processors. We compare our generated libraries to Goto BLAS 1.26 and MKL 10.0. The input to our system are the three blocking rules and the general-size library generated is depicted in Figure 3.10. Online search is conducted with the dynamic programming algorithm. Timing is done by repeating the functionalities without emptying the cache between runs. We compile the library using the Intel compiler (`icc`) 10.1.

In the first test scenario, Figure 5.1, we measure the performance of tightly packed square multiplications ($m = n = k$) focusing on smaller sizes. We observe that we reach two thirds of the peak, at equality with GotoBLAS in the single precision case. However, both competitors are still ramping up in the end whereas our generated libraries plateau. This is most likely due to buffering optimizations (copying intermediate matrix blocks into contiguous memory) that are not yet supported in our framework.

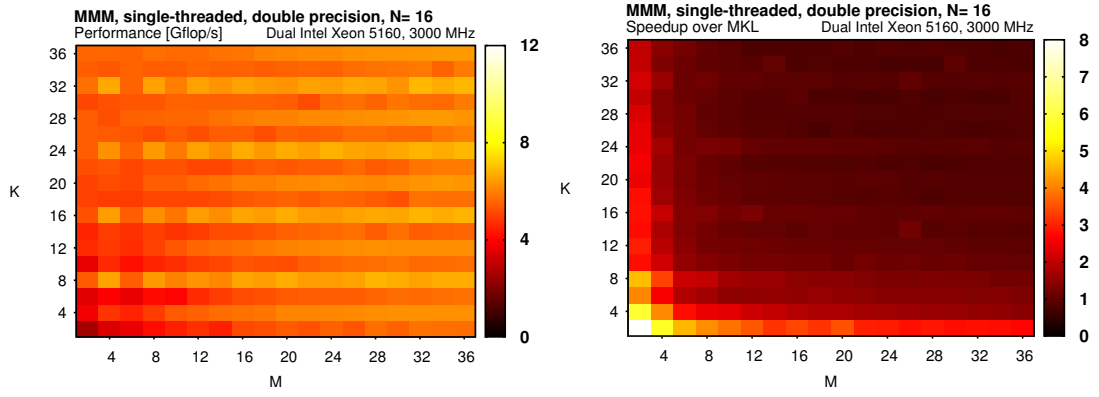


(a) (b)

FIGURE 5.1: GEMM library performance on x86, square case, single-threaded. (a) single precision (the peak performance is 24 GFlop/s). (b) double precision (the peak performance is 12 GFlop/s).

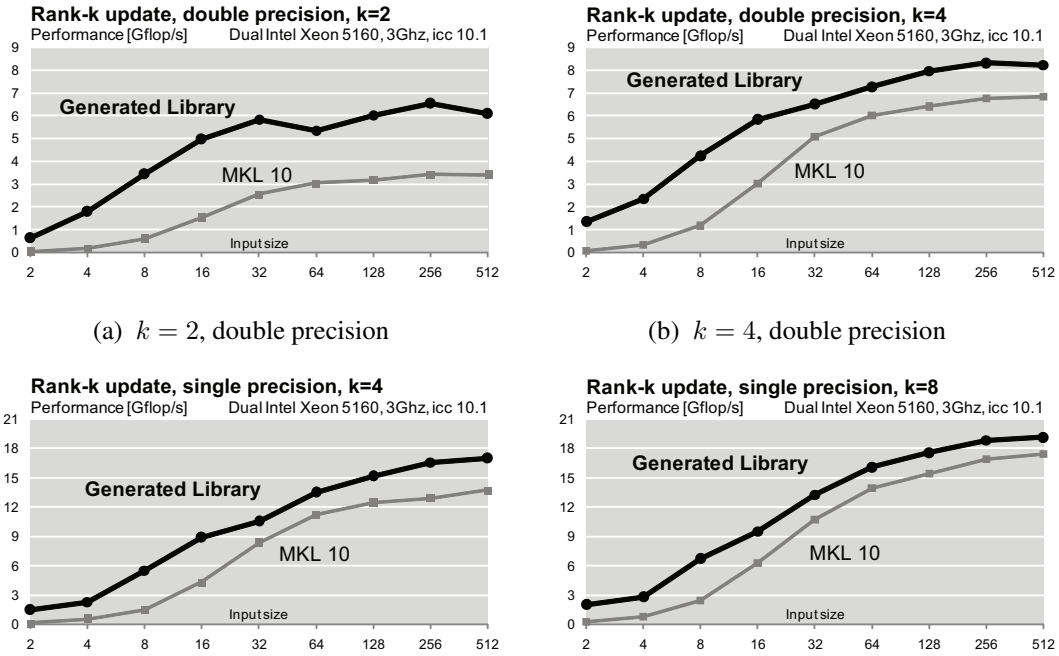


(a) single precision performance (b) single precision speedup over MKL



(c) double precision performance (d) double precision speedup over MKL

FIGURE 5.2: GEMM library performance on x86, non-square heatmaps for k and m with $n = 16$, single precision (a) and double precision (c). The corresponding speedup against MKL 10 is also shown (b) and (d).



(a) $k = 2$, double precision

(b) $k = 4$, double precision

(c) $k = 4$, single precision

(d) $k = 8$, single precision

FIGURE 5.3: Performance of rank-k updates on x86 for $k = 2, 4$ (double precision) and $k = 4, 8$ (single precision).

In the second test scenario, Figure 5.2, we present heatmaps corresponding to variable m and k dimensions while n is fixed to 16. We observe that performance is globally regular across the slice; the regular line and dots patterns that we can distinguish on the double precision performance figure are due to the divisibility by the vector length. The comparisons with MKL 10 shows that our libraries are relatively better when one of the dimensions is very small. In fact, the hyperbolic shape of the isolines of the speedup graph reveals that MKL routines often require a larger number of operations to be efficient. The reason here may be that the MKL developers focused on optimizing the most common cases while the automatic search adapts to these cases without discrimination.

In the third test scenario, Figure 5.3, we focus on *rank-k updates* which are standard matrix multiplications where k is small. These are important building blocks in various linear algebra algorithms such as LU factorization. Because of their special shape, library vendors such as MKL have to develop specific kernels to handle them efficiently. As we see in the plot, our generated libraries outperform these routines, for small values of k.

Performance on an IBM Cell SPU. We evaluate now the generator on a different platform, a Synergistic Processing Unit (SPU) of the IBM Cell Broadband Engine (BE). This time, we generate fixed sizes libraries as the amount of memory available in the local stores of the SPU is very limited. The input to our generator are the same blocking rules as before and we restrict the library to the

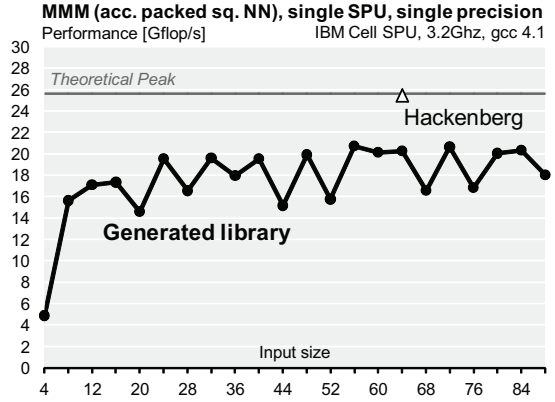


FIGURE 5.4: GEMM library performance on an IBM Cell SPU, square case. The white triangle represents the performance reached by the assembly routine from [Hackenberg, 2007] that exclusively supports the multiplication of 64x64 matrices.

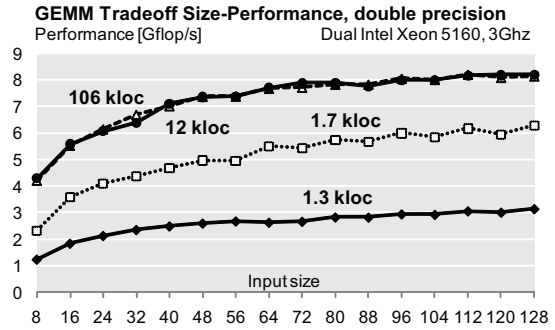


FIGURE 5.5: Size-performance tradeoff. Adding additional base cases into the library improves the performance but increases the size of the library (KLOC stands for kilo-lines of codes). Observe that returns are diminishing and there is a point where growing the library does not yield any additional improvements.

tightly packed interface. Vectorization is supported through the Altivec intrinsics and the library is compiled using the GNU compiler (`gcc`) 4.1. Only single precision libraries are generated since the SPUs are not competitive in double precision [Williams et al., 2006]. This work has been done in collaboration with Srinivas Chellappa who developed the Cell backend for Spiral [Chellappa et al., 2009].

We compare our generated library with the hand-coded routine from [Hackenberg, 2007] that multiplies 64x64 matrices nearly at peak performance. While the generated library achieves only 70% of the peak, observe that many more sizes are supported. The plot does not show bigger sizes since we reach the upper limit of the SPU local memory.

Size-performance tradeoff. Our generator allows us to add additional base cases which improves performance but increases the code size. In another experiment on the x86 platform, we gen-

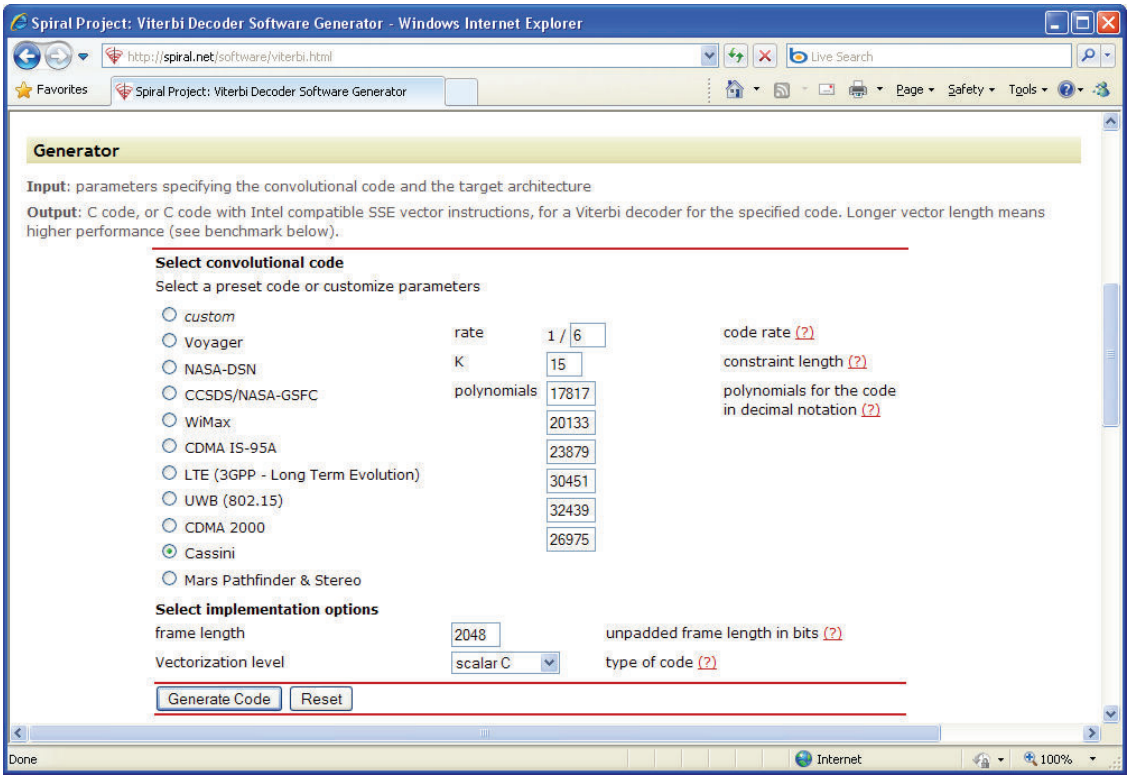


FIGURE 5.6: Web Interface to the Viterbi Decoder Software Generator, directly available at `http://www.spiral.net/software/viterbi.html`. The user either chooses a preset or enters a custom convolutional code. The target decoder is then live generated and returned to the user. At the time of this dissertation, more than 5000 decoders have been generated.

erate four different libraries with base cases chosen inside $[1,n]\times[1,n]\times[1,n]$ where $n = 2, 4, 8, 16$. Figure 5.5 shows the impact of adding more base cases on the overall code size Not surprisingly, there is a saturation point after which growing the number of base cases makes no sense. The reason is that the best base cases for this type of computations are very tightly scheduled loops and therefore, there is no need for having larger loop bodies.

5.2 Viterbi decoding

The performance critical forward pass of the Viterbi algorithm is captured in the Operator Language in Equation 2.29, from which we generate fixed input size Viterbi decoders. Our generator supports any valid combination of rate, polynomials, quantization, frame length, and constraint length. Vectorization is available for all convolutional codes and for processors that are SSE-compatible through 4-way (32 bits), 8-way (16 bits), and 16-way (8 bits) intrinsics. It is directly accessible on the Internet (see [de Mesmay, 2008]) through the web interface shown in Figure 5.6.

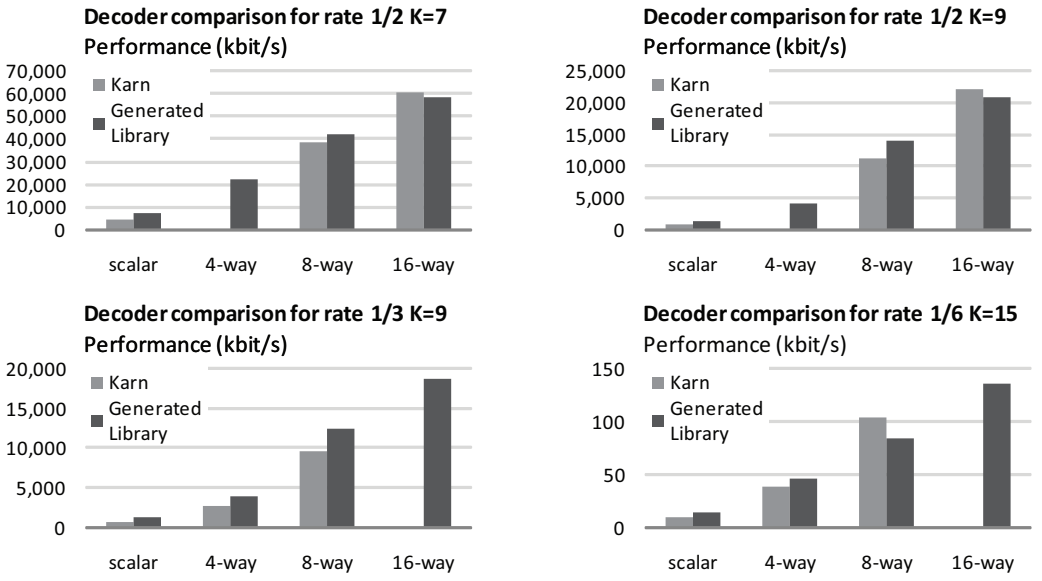


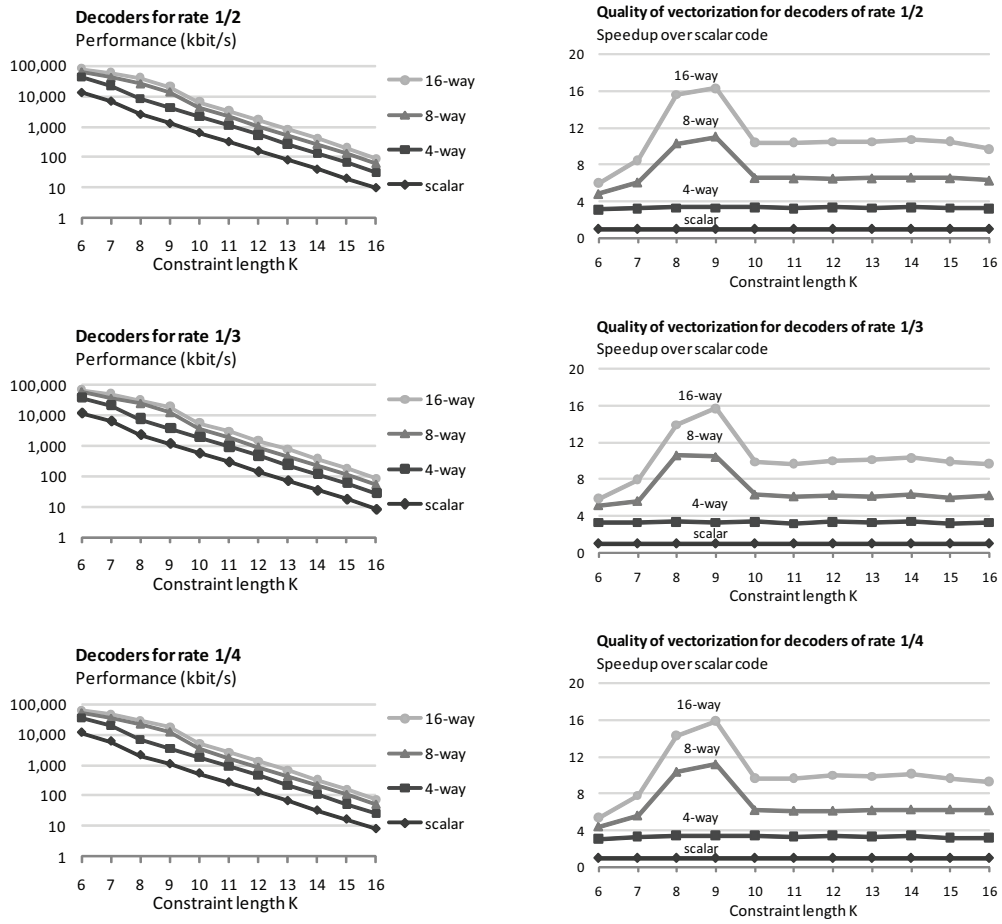
FIGURE 5.7: Performance comparison between the generated and hand-optimized decoders. A missing light-gray bar means that the hand-optimized implementation is not provided by [Karn, 2007].

Experimental setup. All experiments are performed on an Intel Core 2 Extreme X9650. All code is compiled using the Intel Compiler (`icc`) Version 10.1 and the performance in each case is measured by entirely decoding (forward pass and traceback) multiple frames. Initialization and precomputation (one time costs) are excluded.

Performance comparison. We first compare our generated decoders against Karn's hand-written decoders [Karn, 2007]. The forward error correction package provides decoders for four codes, available for different SSE vector lengths. The codes are $r = 1/2,\ K = 7$ nicknamed "Voyager," $r = 1/2,\ K = 9$, $r = 1/3,\ K = 9$, and $r = 1/6,\ K = 15$ nicknamed "Cassini." Karn does not support all vector lengths for all codes. The forward pass in [Karn, 2007] is written separately in assembly for each combination of code and vector length.

In Figure 5.7, we show the performance results for the four codes and for all vector lengths, including scalar code. The performance is reported in kbit/s so higher is better. Karn's decoders are displayed in light-gray, our generated decoders in dark gray. A missing light gray bar signifies that the implementation is not provided by Karn. Analysis of the plots shows that our generated decoders have roughly equal performance compared to [Karn, 2007] while supporting all vector lengths. However, our generator is not restricted to these four codes, as discussed next.

Performance of supported codes. To show the generality of our generator and the consistent performance, we generated decoders for the "good" convolutional codes of various rates collected in [Larsen, 1973] and [Chambers, 1992] and for all four vector lengths 1, 4, 8, 16. Figure 5.8a



(a) Performance

(b) Normalized performance over scalar code

FIGURE 5.8: Performance of generated decoders for rates 1/2, 1/3 and 1/4. (a) Performance with respect to the constraint length. (b) Speedup of vectorized decoders compared to their scalar counterparts.

shows the performance results for $N = 2, 3, 4$ and $K = 6$–16.

As expected, the lines show an exponential decay in performance with increasing constraint length. In contrast, changing the rate does not noticeably change performance. The speedup obtained through vectorization is consistent and investigated next.

Quality of vectorization. Figure 5.8b shows the speedup achieved by the vectorization of Figure 5.8a. The baselines are the non-vectorized scalar decoders.

We observe a consistent speedup of about 3.5 for 4-way, 6 for 8-way, and 10 for 16-way vectorization. This should be close to the achievable optimum, given that longer vectors require more shuffle operations $L_\nu^{2\nu}$ that consume time without performing computations. The peak for both 16-

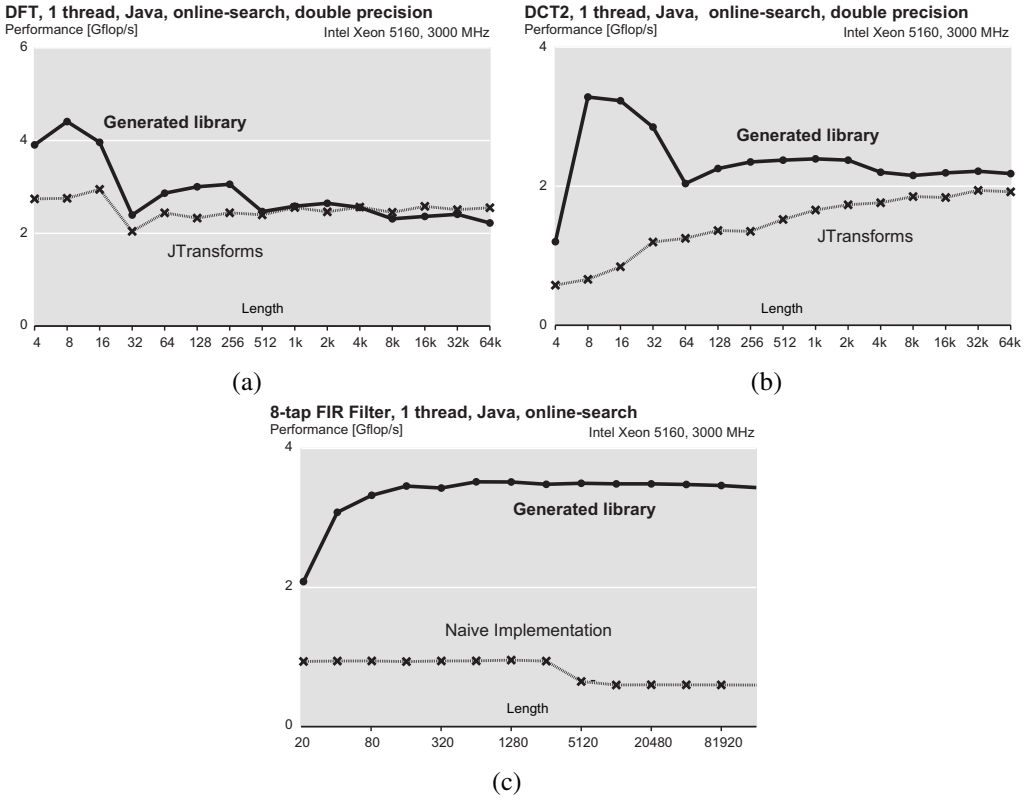


FIGURE 5.9: Performance of generated Java libraries. (a) Comparison with JTransforms on DFT. (b) Comparison with JTransforms on DCT2. (c) Comparison with a naïve implementation on FIR filter. Graphs are taken from [Voronenko et al., 2009].

way and 8-way with short constraint length is caused by the reduction of the memory footprint due to the use of shorter data types.

5.3 Generated Java Libraries

To show the versatility of the generative approach, we developed a Java backend and evaluated it with general-size linear transforms. Note that Java does not offer any mechanism to access SIMD vector units and that we did not take the time to implement support for Java threading pools. However, we did provide online search through a dynamic programming planner.

Clearly, interpreted performance libraries do not have a mainstream use but they are sometimes required, mainly for security and portability reasons (e.g., inside smartphones user-land). Using a library generator and an automatic search mechanism enables the retargeting of Java libraries for these niche uses at a very low cost. On the other hand, porting an entire C or C++ library by hand would be not only time consuming but would also require new development since the

performance tradeoffs and choices that were made for native code can be wildly different from those for interpreted code.

Experimental setup. Three different libraries are generated: a $\mathbf{DFT}$ library based on the Cooley-Tukey FFT (Equation 2.6), a $\mathbf{DCT\text{-}2}$ library based on Equation 2.13 and Equation 2.14 and a Finite Impulse Response (FIR) filter library based on blocking rules not stated in this thesis (see [Voronenko, 2008]).

All generated libraries are run with Java HotSpot 1.6.0 which includes a just-in-time compiler. The adaptive timing routines work around the just-in-time compilation by "warming" up the code path to make sure that the code is compiled when it is measured. The benchmark is done on a 64-bit Linux platform using two dual core 3 GHz Intel Xeon 5160 processors. The operation count for the complex DFT on n points is assumed to be $5n \log_2 n$ (standard practice), for the DCT2 on n points $2.5n \log_2 n$, and for the k-tap FIR filter on n points $(2k-1)n$.

We compare our generated libraries to JTransforms which is an open-source high-performance linear transform library [Wendykier, 2008]. Filters are compared to a naïve double loop implementation since we did not find competitors.

Experiments. Interestingly, JTransforms is very fast on the DFT and, as can be observed in Figure 5.9a, the generated library only has a slight advantage, mainly due to the inline base cases. On less common functionalities such as DCT-2 , Figure 5.9b, the generated library shows a clear advantage, likely since the human developer did not spend the same time optimizing this less used functionality. A generator is oblivious to such considerations. Finally, we show FIR performance results in Figure 5.9c: a 3–4x speedup over naïve code.

5.4 Online Adaptation

In this section, we evaluate the performance of three different online search algorithms: Monte-Carlo (MC), Dynamic Programming (DP) and our novel Threshold Ascend on a Graph (TAG). As a target, we use a generated complex DFT C++ library. The library is vectorized using intrinsics and threaded using OpenMP. It is compiled using the Intel Compiler 10.1 and benchmarked on a 64-bit Linux platform using two dual core 3 GHz Intel Xeon 5160 processors.

We display performance using pseudo mega floating-point operations per second (MFlop/s); the complex DFT operation count is again assumed to be $5n \log_2 n$.

Parameter tuning for TAG. As explained in Subsection 4.2.3, TAG requires two parameters, δ and s. We tune the s parameter on a specific problem: $\mathbf{DFT}_{2^{19}}$. The sensitivity of the algorithm to variations in the s parameter of the bandit algorithm is relatively minor as shown in Figure 5.10, which plots performance of solution found against search time. Since s is the size of the best rewards vector, a low s tweaks the bandit towards exploitation of previous good branches, while a bigger s leads to the exploration of new promising branches. We find that $\delta = 0.1$ and $s = 30$ work best and we use them for *all* following experiments.

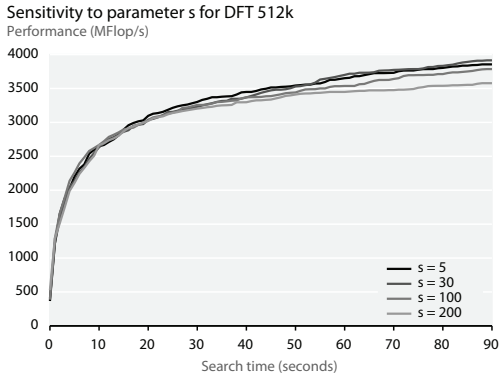
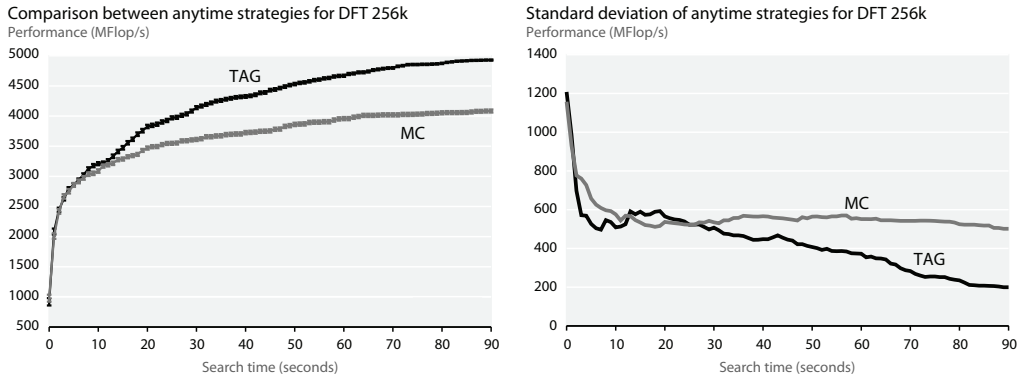


FIGURE 5.10: Sensitivity of TAG with respect to the parameter s. Data is averaged over 100 runs with $\mathbf{DFT}_{2^{19}}$. The parameter $s = 30$ is best in this particular example (and others not shown), we fix it for all further experiments.



(a) (b)

FIGURE 5.11: Comparison between TAG and Monte-Carlo: (a) Mean performance and standard error of the mean (which is so small it is barely visible), and (b) standard deviation for DP and Monte-Carlo on $\mathbf{DFT}_{2^{18}}$. The data is an averaged over 100 runs.

Comparison with Monte-Carlo. We compare the performance of TAG and MC on $\mathbf{DFT}_{2^{18}}$. Figure 5.11a and Figure 5.11b show that TAG performs better (higher performance found in the same search time) and more reliably (lower standard deviation) than Monte-Carlo. Note that the plots are done with respect to a fixed "wall clock" time and not with respect to a fixed number of simulations. This is a realistic assumption because the simpler MC algorithm performs more simulations than the more complex algorithm in the same time frame. Also it is worth remembering that, asymptotically, TAG and MC will converge since they both eventually explore the entire search space.

Comparison with dynamic programming. Figure 5.12a shows the comparison of TAG with DP for $\mathbf{DFT}_{2^{16}}$. We observe that TAG finds the same performance as DP faster than DP and finds a 10% better performance in the same search time as DP. Performing this experiment for several sizes,

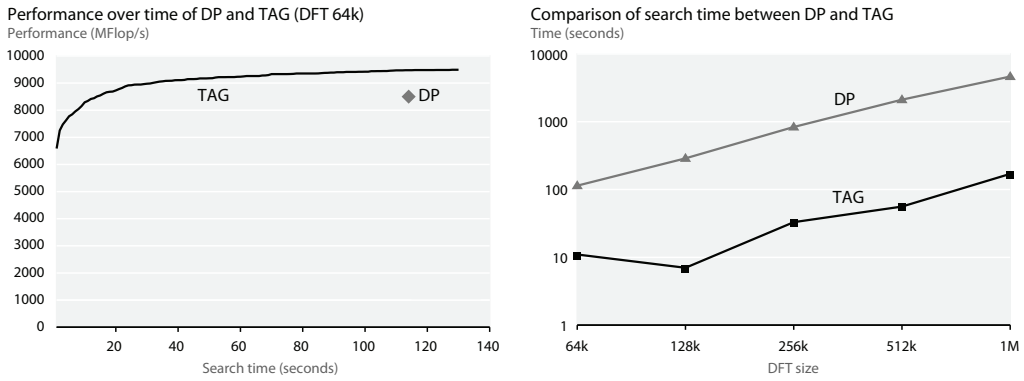


FIGURE 5.12: Comparison between TAG and dynamic programming. (a)Average performance of TAG compared with DP on a single problem size. (b)Search time of TAG and DP to achieve the same performance on different libraries.

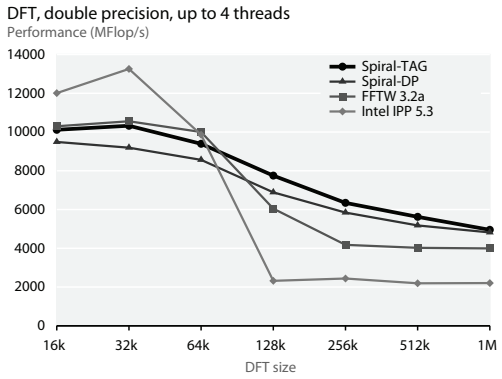


FIGURE 5.13: Comparison of DP and TAG together with concurrent FFT libraries. Data is averaged over 100 runs.

we show in Figure 5.12b the time it takes TAG to get results of the same quality as DP. We observe that TAG finds solutions of equal performance an order of magnitude faster than DP in these cases.

Comparison with other FFT libraries. Figure 5.13 shows the best performance achieved by the Spiral-generated library (with TAG and DP) and its competitors. We compare against FFTW 3.2 alpha 2 and Intel IPP 5.3. We use FFTW in the mode that does platform adaptation by dynamic programming. As far as we know, IPP does not use search and branches out to a specialized implementation for each platform. The plot shows the competitiveness of the generated library that we used for the experiments in this section. TAG performs slightly better than DP in all cases.

5.5 Offline Adaptation

In this section, we analyze the performance of the offline adaptation mechanism that we proposed in Section 4.3. As a target, we use different generated online adaptive complex DFT C++ libraries. Using our mechanism, these are then directly converted into offline adaptive libraries.

Experimental setup. Our benchmark platform has two dual core 3 GHz Intel Xeon 5160 processors (server version of Core 2 Duo) with 4 MB of shared L2 cache per processor, running Linux in 64-bit mode. Libraries are compiled using the Intel C/C++ Compiler 10.1. We compare against FFTW 3.2 alpha 2 and Intel IPP 5.3. We generate two libraries based off the Cooley-Tukey FFT: one that is vectorized and threaded and the other that is neither. The decision graph in Figure 4.9 corresponds to the multi-threaded library.

All libraries were timed out of the box. In particular, slightly better performance for FFTW and Spiral-generated libraries could have been achieved in the mid-range by ensuring that only two threads are used and pinning them properly to processors. However, these choices are not handled automatically by the library, hence we did not consider them.

As before, we show performance in pseudo GFlop/s assuming a (real) operations count of $5n \log_2 n$ for the complex DFT of size n.

Comparison with existing heuristics. In Figure 4.4, we presented the hand-written heuristics developed in [Voronenko et al., 2008a] to choose the radix as a function of the input size for DFTs. Figure 5.14 shows the automatically produced heuristic for the same choice in the non-threaded non-vectorized library. The radix choices of both heuristics as a function of the DFT size are contrasted in Figure 5.15b. Both strategies produce similar performance profiles (Figure 5.15a) except for mid-range sizes where the generated heuristic even performs slightly better.

Clothesline experiments. We want to demonstrate that heuristics can be learned. To do this, we show that the larger the training set becomes, the better the heuristics perform on the test set (which is not part of the training set). Such an experiment is presented in Figure 5.16 using the vectorized and threaded DFT library whose decision graph with 11 choices is shown in Figure 4.9. The competitiveness of this library is shown in Figure 5.16d by comparing to FFTW and IPP.

In Figure 5.16a, the training set consists of only two small sizes (2^4 and 2^8, marked by a circle). For these sizes, threading is irrelevant so the generated heuristics avoid it, which yields bad performance for bigger sizes. In Figure 5.16b, the training set includes four sizes that are well distributed so all regimes (in- and out-of-cache, threaded or not) of performance can be captured by the heuristics. In Figure 5.16c, the training set is so large that the offline library is almost as fast as the online one in all sizes. The nature of these plots motivate our notion of "clothesline experiments": a larger set of clothes pins (training set) yields a garment that hangs closer to the rope (performance obtained by search). Note that one given training size contains much more than one piece of information since there are multiple heuristics and they can be used multiple times due to the recursion.

```
if ( n <= 65536 ) {
   if ( n <= 32 ) {
      if ( n <= 4 ) {return 2;}
      else {return 4;}
   }
   else {
      if ( n <= 1024 ) {
         if ( n <= 256 ) {return 8;}
         else {return 32;}
      }
      else {
         if ( n <= 4096 ) {
            if ( n <= 2048 ) {return 8;}
            else {return 4;}
         }
         else {return 8;}
      }
   }
}
else {
   if ( n <= 262144 ) {
      if ( n <= 131072 ) {return 16;}
      else {return 32;}
   }
   else {return 16;}
}
```

FIGURE 5.14: A heuristic automatically crafted to replace the expert-written heuristic in Figure 4.4.

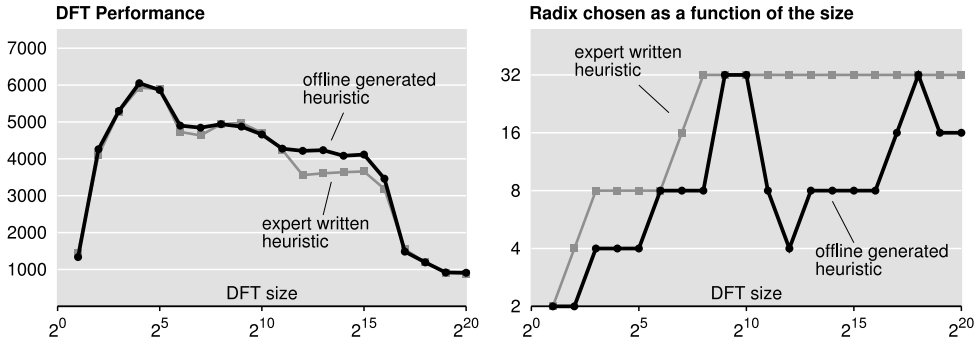
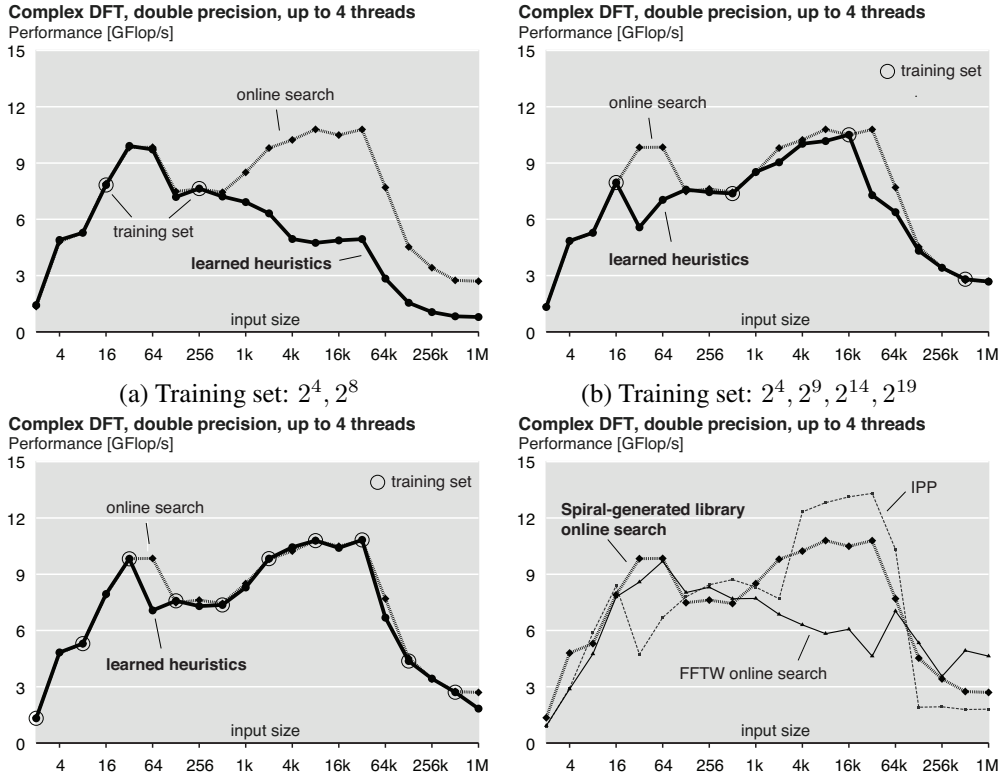


FIGURE 5.15: (a) Performance and (b) radix choices of two different DFT heuristics. The expert written heuristic is from Figure 4.4 and the automatically generated one is from Figure 5.14.

Mixed sizes experiments. The learning approach becomes particularly relevant if we consider a much larger set of sizes by including non-two-powers. We do so in the second experiment which considers all sizes up to 256K that decompose into prime factors smaller or equal than 19. We



(a) Training set: $2^4, 2^8$

(b) Training set: $2^4, 2^9, 2^{14}, 2^{19}$

(c) Training set: 2^k, k odd

(d) The library is competitive

FIGURE 5.16: (a), (b), and (c) are all *clothesline* experiments, demonstrating the effectiveness of heuristic generation. An online adaptive library first searches to find the best decisions for all powers of two (up to size 1M)—this line is the ground truth and plays the role of the *rope*. A subset of sizes is then picked to training set to generate the heuristics in an offline adaptive library—these sizes constitute the *clothes pins*. Finally, the performance of this new library is evaluated on all sizes—this line is the *cloth*. We simply want to verify that, the more clips are added, the straighter the cloth hangs, which shows that heuristics can pick up and generalize the knowledge provided by the nearby training sizes, even, (d), in the context of a highly optimized competitive library that is threaded, vectorized and buffered.

generate two offline adaptive libraries that differ in the size of their training set which respectively amounts for 1% and 6% of all 2^{18} sizes. Figure 5.17a shows the performance of the latter one and the performance of IPP on all 2^{18} sizes. The performance varies greatly depending on the prime factorization of the size. However, it can be seen that the performance of our offline adapted library provides a somewhat more stable performance, i.e., the cloud is less scattered than IPP's.

Next we measured the performance for all sizes up to 256K for all considered libraries that are not online-adaptive: IPP, FFTW with the heuristic activated by the FFT_ESTIMATE flag, and our generated offline adapted libraries with 1% and 6% training set. To reason about the performance across all sizes, we computed a logarithmic regression of order 6. The resulting four lines are

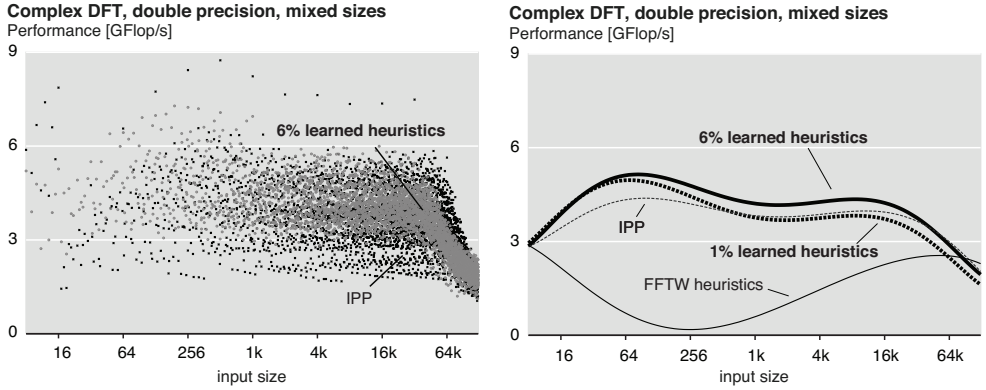


FIGURE 5.17: Both (a) and (b) present the DFT performance of different libraries on all numbers smaller than 256K that decompose in prime factors smaller or equal than 19. As can be seen on (a), the performance is not continuous with respect to the input size since fast algorithms depend on the prime factor decomposition of the size, which makes it difficult to read. For better readability, sixth order logarithmic trendlines of the same performance are presented in (b). Four libraries are evaluated: IPP, FFTW (only the heuristics were timed, online search would have performed better but it would have been too long to train), and too libraries whose heuristics were trained on 1% and 6% of the whole space.

plotted in Figure 5.17b. First, we observe that the library that is trained on the larger training set (6%) performs better than the one with 1%. Second, we observe that the 6% generated-heuristics library performs better than IPP and is the overall fastest for these sizes. Precise computation shows that the average performance gain is 10.7%. Finally, the poor performance of the hand-written heuristic mode of FFTW shows that writing heuristics is no simple task. We note that FFTW's heuristic does work reasonably well for 2-power sizes.

CHAPTER 6

Conclusions

We restate from introduction:

> The goal of the thesis is to extend Spiral's high-performance library generation framework in two orthogonal ways:
>
> 1. We extend the formal framework and the generator to new functionalities *beyond linear transforms*, notably matrix-matrix multiplication and Viterbi decoding.
>
> 2. We enable the automatic generation of *adaptive general input size* libraries. We provide both novel *online* (at runtime) and *offline* (at installation time) adaptation mechanisms that can be inserted into Spiral-generated libraries.

In this thesis work, we have presented the framework and developed a prototype of a library generator that achieves the above goals. The key to achieving the first goal is the new operator language (OL) that we introduced. As we have shown, this language can at least capture algorithm knowledge and necessary transformations for three computational domains: basic linear algebra, convolutional decoding, and linear transforms. For the latter it reduces to the prior signal processing language (SPL). All difficult tasks in the library production, including vectorization, base case generation, and the computation of the recursion step closure, are automated through carefully designed OL rewriting systems. We achieved the second goal through a modular backend that enables the insertion of different online adaptation (search) mechanism into our generated libraries. Further, we presented a method that converts an online adaptive library into an offline adaptive library.

We implemented all the contributions into one flexible library generation infrastructure that encapsulates and extends prior components developed for linear transforms such as the general-size library generation infrastructure, parts of the original SPL compiler, and existing SPL rewriting systems. The infrastructure supports fixed and general input size functionalities, different target languages, and various other options.

Across the functionalities that we support, we showed that our libraries have a performance that is roughly comparable with existing state-of-the-art libraries that are written by experts who carefully optimize their code.

In summary, our prototype demonstrates that "push-button" generation of high-performance adaptive domain-specific libraries is not only possible but also that the general infrastructure of such a system can be actually used for different domains. And while it can seem redundant at first to *generate* libraries that are *adaptive*, the generative and adaptive approaches actually complement each other. The former greatly increases the speed at which developers can modify their libraries while the later offers increased flexibility to users and augments the reliability and thus extends the lifetime of the library.

Limitations. Like any work, ours exhibits a number of limitations at this point. We briefly discuss the most important ones.

One of the main limitations of the library generation framework is the absence of support for tile copies, which are necessary to achieve optimal performance for large matrix multiplications. Indeed, the entire purpose of the recursion step closure is to eliminate explicit copies which are usually detrimental to performance. However, in some rare cases, a very limited amount of copying can actually help by contiguously packing data to reduce cache or page misses. It is clear to us that another mechanism is needed to introduce them but we were not able to unite this specification with the termination of the closure and the possibility of online search.

We have introduced the general tensor definition for two operators but have only implemented support for it when one of the operators is "unitary" which means that it is either an identity, a point-wise multiplication, a Kronecker product, or a scalar product. Adding a more general support would enable the capture of a larger space of algorithms as we already hinted in Equation 2.24. However, one of the reasons this is not done yet is that implementing such tensors would require implementing some meta-programming into Spiral which is not supported at the moment.

Finally, the last major limitation of the current framework is that not the entire cross-product of functionality, algorithms, platforms, languages, and search methods is actually available. The reason here is more one of development time issue than a theoretical limitation.

Future directions. Besides removing the mentioned limitations, there are many avenues for future research. One obvious direction is to gradually capture an increasing number of domains and an increasing number of platforms into the library generator since the general infrastructure is reusable. Both directions are already pursued in the Spiral project. Examples include extensions of OL to capture the domains of synthetic aperture radar imaging and the physical layers of software-

defined radios.

Another route might be equally or even more promising: the development of a specialized iterative compiler for generic recursive functionalities. The intuition there is that the occurrence of the concept of recursion step closure in both the transform and linear algebra domain hints that it may be a more general concept. It could therefore be ported to a generic loop manipulation framework such as the polyhedral model or others. This way, this crucial optimization in the domain considered here (and in the prior work on transforms) could possibly be extended into the "general purpose" compiler world.

Bibliography

Agner, Fog. Optimization manuals
`http://www.agner.org/optimize/`, 2010. 54

Ahmed, Nawaaz; Mateev, Nikolay; Pingali, Keshav. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. In *ICS '00: Proceedings of the 14th international conference on Supercomputing*, pages 141–152, New York, NY, USA, 2000. ACM. ISBN 1-58113-270-0. 11

Ali, Ayaz; Johnsson, Lennart; Mirkovic, Dragan. Empirical auto-tuning code generator for FFT and trigonometric transforms. In *ODES: 5th Workshop on Optimizations for DSP and Embedded Systems, in conjunction with International Symposium on Code Generation and Optimization (CGO)*, San Jose, CA, 2007. 2, 67

Allen, Randy; Kennedy, Ken. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers, 2002. 54

Auer, Peter; Cesa-Bianchi, Nicolò; Fischer, Paul. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2-3):235–256, 2002. 85

Baader, Franz; Nipkow, Tobias. *Term rewriting and all that*. Cambridge University Press, New York, NY, USA, 1998. ISBN 0-521-77920-0. 43

Banerjee, Utpal; Eigenmann, Rudolf; Nicolau, Alexandru; Padua, David A. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243, 1993. 62

Barthou, Denis; Donadio, Sébastien; Carribault, Patrick; Duchateau, Alexandre; Jalby, William. Loop optimization using hierarchical compilation and kernel decomposition. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '07)*, pages 170–184, March 2007. 10

Bastoul, Cédric. Code generation in the polyhedral model is easier than you think. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 7–16, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2229-7. 11

Baumgartner, Gerald; Auer, A.; Bernholdt, D.E.; Bibireata, A.; Choppella, V.; Cociorva, D.; Gao, X; Harrison, R.J.; Hirata, S.; Krishnamoorthy, S.; Krishnan, S.; Lam, C.; Lu, Q; Nooijen, M.; Pitzer, R.M.; Ramanujam, J.; Sadayappan,

P.; Sibiryakov, A. Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. *Proceedings of the IEEE*, 93(2):276–292, Feb. 2005. ISSN 0018-9219. 3, 12

Beauchamp, Ken G. *Applications of Walsh and related functions*. Academic Press, 1984. ISBN 978-0120841806. 14

Beckmann, Olav; Houghton, Alastair; Mellor, Michael; Kelly, Paul H. J. Run-time code generation in C++ as a foundation for domain-specific optimisation. In *Lecture Notes in Computer Science*, pages 291–306. Springer, 2004. 10

Bergland, Glenn D. Numerical analysis: A fast fourier transform algorithm for real-valued series. *Communications of the ACM*, 11(10):703–710, 1968. ISSN 0001-0782. 13

Bientinesi, Paolo. *Mechanical derivation and systematic analysis of correct linear algebra algorithms*. PhD thesis, University of Texas at Austin, Austin, TX, USA, 2006. 2, 12

Bik, Aart J. C.; Girkar, Milind; Grey, Paul M.; Tian, Xinmin. Automatic intra-register vectorization for the Intel® architecture. *Int. J. Parallel Program.*, 30(2):65–98, 2002. ISSN 0885-7458. 59

Black, Peter J.; Meng, Teresa H.-Y. A 140-mb/s, 32-state, radix-4 viterbi decoder. *IEEE Journal of Solid-State Circuits*, 27(12):1877–1885, Dec 1992. ISSN 0018-9200. 26

Black, Peter J.; Meng, Teresa H.-Y. A 1-gb/s, four-state, sliding block viterbi decoder. *IEEE Journal of Solid-State Circuits*, 32(6):797–805, Jun 1997. ISSN 0018-9200. 26

Bluestein, L. A linear filtering approach to the computation of discrete fourier transform. *Audio and Electroacoustics, IEEE Transactions on*, 18(4):451–455, Dec 1970. ISSN 0018-9278. 17

Bodin, François; Kisuki, Toru; Knijnenburg, Peter M. W.; O'Boyle, Michael F. P.; Rohou, Erven. Iterative compilation in a non-linear optimisation space. In *Proceedings of the Workshop on Profile and Feedback Directed Compilation*, 1998. 10

Bonelli, Andreas; Franchetti, Franz; Lorenz, Juergen; Püschel, Markus; Ueberhuber, Christoph W. Automatic performance optimization of the discrete Fourier transform on distributed memory computers. In *International Symposium on Parallel and Distributed Processing and Application (ISPA)*, volume 4330 of *Lecture Notes In Computer Science*, pages 818–832. Springer, 2006. 11

Bouzy, Bruno; Helmstetter, Bernard. Monte-carlo go developments. In *Proceedings of the 10th Advances in Computer Games Conference (ACG-10)*, pages 159–174. Kluwer Academic, 2003. 83

Boyle, Phelim; Broadie, Mark; Glasserman, Paul. Monte carlo methods for security pricing. *Journal of Economic Dynamics and Control*, 21(8-9):1267 – 1321, 1997. 83

Bruegmann, Bernd. Monte-carlo go,
`http://www.cgl.ucsf.edu/go/Programs/Gobble.html`, 1993. 83

Burgisser, Peter; Clausen, Michael; Shokrollahi, M. Amin. *Algebraic Complexity Theory*. Springer, 1997. 14

Bylaska, Eric J.; de Jong, W. A.; Govind, N.; Kowalski, K.; Straatsma, T. P.; Valiev, M.; Wang, D.; Apra, E.; Windus, T. L.; Hammond, J.; Nichols, P.; Hirata, S.; Hackler, M. T.; Zhao, Y.; Fan, P.-D.; Harrison, R. J.; Dupuis, M.; Smith, D. M. A.; Nieplocha, J.; Tipparaju, V.; Krishnan, M.; Wu, Q.; Voorhis, T. Van; Auer, A. A.; Nooijen, M.; Brown, E.; Cisneros, G.; Fann, G. I.; Fruchtl, H.; Garza, J.; Hirao, K.; Kendall, R.; Nichols, J. A.; Tsemekhman, K.; Wolinski, K.; Anchell, J.; Bernholdt, D.; Borowski, P.; Clark, T.; Clerc, D.; Dachsel, H.; Deegan, M.; Dyall, K.; Elwood, D.;

Glendening, E.; Gutowski, M.; Hess, A.; Jaffe, J.; Johnson, B.; Ju, J.; Kobayashi, R.; Kutteh, R.; Lin, Z.; Littlefield, R.; Long, X.; Meng, B.; Nakajima, T.; Niu, S.; Pollack, L.; Rosing, M.; Sandrone, G.; Stave, M.; Taylor, H.; Thomas, G.; van Lenthe, J.; Wong, A.; Zhang, Z. NWChem, a computational chemistry package for parallel computers, version 5.1, 2007. 46

Calder, Brad; Grunwald, Dirk; Jones, Michael; Lindsay, Donald; Martin, James; Mozer, Michael; Zorn, Benjamin. Evidence-based static branch prediction using machine learning. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(1):188–222, 1997. ISSN 0164-0925. 77

Cavazos, John; Moss, J. Eliot B. Inducing heuristics to decide whether to schedule. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation (PLDI'04)*, pages 183–194, New York, NY, USA, 2004. ACM. ISBN 1-58113-807-5. 77

Cavazos, John; O'Boyle, Michael F. P. Method-specific dynamic compilation using logistic regression. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 229–240, New York, NY, USA, 2006. ACM. ISBN 1-59593-348-4. 10

Chambers, W. G. On good convolutional codes of rate 1/2, 1/3, and 1/4. *Singapore ICCS/ISITA '92. 'Communications on the Move'*, 2:750–754 vol, Nov 1992. 106

Chellappa, Srinivas; Franchetti, Franz; Püschel, Markus. Computer generation of fast Fourier transforms for the cell broadband engine. In *International Conference on Supercomputing (ICS)*, pages 26–35, 2009. 104

Cicirello, Vincent A.; Smith, Stephen F. The max k-armed bandit: a new model of exploration applied to search heuristic selection. In *AAAI'05: Proceedings of the 20th national conference on Artificial intelligence*, pages 1355–1361. AAAI Press, 2005. ISBN 1-57735-236-x. 88

Cohen, Albert; Sigler, Marc; Girbal, Sylvain; Temam, Olivier; Parello, David; Vasilache, Nicolas. Facilitating the search for compositions of program transformations. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 151–160, New York, NY, USA, 2005. ACM. ISBN 1-59593-167-8. 11

Cohen, Albert; Donadio, Sébastien; Garzarán, María J.; Herrmann, Christoph; Kiselyov, Oleg; Padua, David. In search of a program generator to implement generic transformations for high-performance computing. *Science of Computer Programming*, 62(1):25–46, 2006. ISSN 0167-6423. 10

Cooley, James W.; Tukey, John W. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965. 5, 17

Cooper, Keith D.; Schielke, Philip J.; Subramanian, Devika. Optimizing for reduced code space using genetic algorithms. In *LCTES '99: Proceedings of the ACM SIGPLAN 1999 workshop on Languages, compilers, and tools for embedded systems*, pages 1–9, New York, NY, USA, 1999. ACM. ISBN 1-58113-136-4. 77

Cooper, Keith D.; Subramanian, Devika; Torczon, Linda. Adaptive optimizing compilers for the 21st century. *J. Supercomput.*, 23(1):7–22, 2002. ISSN 0920-8542. 10

Cooper, Keith D.; Grosul, Alexander; Harvey, Timothy J.; Reeves, Steven; Subramanian, Devika; Torczon, Linda; Waterman, Todd. ACME: adaptive compilation made efficient. In *LCTES '05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 69–77, New York, NY, USA, 2005. ACM. ISBN 1-59593-018-3. 77

Coulom, Rémi. Efficient selectivity and backup operators in monte-carlo tree search. In *Computers and Games: 5th International Conference, CG 2006*, volume 4630 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 2007. 84

Curry, Haskell B.; Feys, Robert; Craig, William. *Combinatory Logic*, volume 1. North-Holland Publishing Company, Amsterdam, Holland, 1958. 75

D'Alberto, Paolo; Nicolau, Alexandru. Adaptive Strassen's matrix multiplication. In *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*, pages 284–292, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-768-1. 25

de Mesmay, Frédéric. Online software Viterbi decoder generator, `http://www.spiral.net/software/viterbi.html`, 2008. 105

de Mesmay, Frédéric; Rimmel, Arpad; Voronenko, Yevgen; Püschel, Markus. Bandit-based optimization on graphs with application to library performance tuning. In *International Conference on Machine Learning (ICML)*, pages 729–736, 2009. 9, 88

de Mesmay, Frédéric; Chellappa, Srinivas; Franchetti, Franz; Püschel, Markus. Computer generation of efficient software Viterbi decoders. In *International Conference on High Performance Embedded Architectures and Compilers (HiPEAC)*, volume 5952 of *Lecture Notes in Computer Science*, pages 353–368. Springer, 2010a. 6, 8

de Mesmay, Frédéric; Voronenko, Yevgen; Püschel, Markus. Offline library adaptation using automatically generated heuristics. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2010b. 9

Dijkstra, Edsger W. The humble programmer. *Communications of the ACM*, 15(10):859–866, 1972. ISSN 0001-0782. 3

Donadio, Sébastien; Brodman, James; Roeder, Thomas; Yotov, Kamen; Cohen, Albert; Garzarán, María J.; Padua, David; Pingali, Keshav. A language for the compact representation of multiple program versions. In *Languages and Compilers for Parallel Computers (LCPC.05)*, volume 4339 of *Lecture Notes in Computer Science*, pages 136–151. Springer, 2006. 10

Dongarra, Jack J.; Du Croz, Jeremy; Hammarling, Sven; Duff, Iain S. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)*, 16(1):1–17, 1990. ISSN 0098-3500. 4, 20, 37, 63

Elliott, Douglas F.; Rao, K. Ramamohan. *Fast Transforms: Algorithms, Analyses, Applications*. Academic Press, Inc., Orlando, FL, USA, 1983. ISBN 0122370805. 17

Fayyad, ; Irani, . Multi-interval discretization of continuous-valued attributes for classification learning. In *Proceedings of the International Joint Conference on Uncertainty in AI*, pages 1022–1027, 1993. 96

Fleming, Chip. A tutorial on convolutional coding with viterbi decoding `http://home.netcom.com/~chip.f/viterbi/tutorial.html`, Nov 2006. 30

Forney, George D. Jr. The Viterbi algorithm. *Proceedings of the IEEE*, 61(3):268–278, March 1973. ISSN 0018-9219. 33

Franchetti, Franz; Püschel, Markus. Generating SIMD vectorized permutations. In *International Conference on Compiler Construction (CC)*, volume 4959 of *Lecture Notes in Computer Science*, pages 116–131. Springer, 2008. 61, 62

Franchetti, Franz; Voronenko, Yevgen; Püschel, Markus. Formal loop merging for signal transforms. In *Programming Languages Design and Implementation (PLDI)*, pages 315–326, 2005. 36, 39, 40, 41

Franchetti, Franz; Bonelli, Andreas; Chuangsuwanich, Ekapol; Lee, Yu-Chiang; Lorenz, Juergen; Peter, Thomas; Shen, Hao; Telgarsky, Marek; Voronenko, Yevgen; Püschel, Markus; Moura, José M. F.; Ueberhuber, Christoph W. Parallelism in Spiral. In *Workshop on Programming Models for Ubiquitous Parallelism (PMUP)*, 2006a. 56

Franchetti, Franz; Voronenko, Yevgen; Püschel, Markus. FFT program generation for shared memory: SMP and multicore. In *Supercomputing (SC)*, 2006b. 11

Franchetti, Franz; Voronenko, Yevgen; Püschel, Markus. A rewriting system for the vectorization of signal transforms. In *High Performance Computing for Computational Science (VECPAR)*, volume 4395 of *Lecture Notes in Computer Science*, pages 363–377. Springer, 2006c. 11, 56, 60, 61

Franchetti, Franz; de Mesmay, Frédéric; McFarlin, Daniel; Püschel, Markus. Operator language: A program generation framework for fast kernels. In *IFIP Working Conference on Domain Specific Languages (DSL WC)*, volume 5658 of *Lecture Notes in Computer Science*, pages 385–410. Springer, 2009. 8, 25

Frigo, Matteo. A fast fourier transform compiler. *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation (PLDI'99)*, 34(5):169–180, 1999. ISSN 0362-1340. 7, 54

Frigo, Matteo; Johnson, Steven G. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. special issue on "Program Generation, Optimization, and Adaptation". 2, 6, 8, 11, 35, 67, 70

Fursin, Grigori; Cohen, Albert; O'Boyle, Michael F. P.; Temam, Olivier. A practical method for quickly evaluating program optimizations. In *HiPEAC'05: Proceeding of the International Conference on High Performance Embedded Architectures and Compilers*, volume 3793 of *Lecture Notes in Computer Science*, pages 29–46. Springer, 2005a. 10

Fursin, Grigori; O'Boyle, Michael F. P.; Knijnenburg, Peter M. W. Evaluating iterative compilation. In *Languages and compilers for parallel computing: 15th workshop, LCPC 2002*, volume 2481 of *Lecture Notes in Computer Science*, pages 362–367. Springer, 2005b. ISBN 978-3-540-30781-5. 10, 18

Gao, Xiaoyang; Sahoo, Swarup Kumar; Lam, Chi-Chung; Ramanujam, J.; Lu, Qingda; Baumgartner, Gerald; Sadayappan, P. Performance modeling and optimization of parallel out-of-core tensor contractions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 266–276, New York, NY, USA, 2005. ACM. ISBN 1-59593-080-9. 12

Gelly, Sylvain; Silver, David. Combining online and offline knowledge in UCT. In *ICML '07: Proceedings of the 24th international conference on Machine learning*, pages 273–280. ACM Press, 2007. ISBN 978-1-59593-793-3. 84

Gibbons, Jeremy. A pointless derivation of radixsort. *Journal of Functional Programming*, 9(3):339–346, 1999. 25

Gilhousen, Klein S.; Heller, Jerry A.; Jacobs, Irwin M.; Viterbi, Andrew J. Coding systems study for high data rate telemetry links. Technical report, NASA, 1971. 33

Good, I. J. The interaction algorithm and practical fourier analysis. *Journal of the Royal Statistical Society. Series B (Methodological)*, 20(2):361–372, 1958. ISSN 00359246. 17

Goto, Kazushige. GotoBLAS 1.26, `http://www.tacc.utexas.edu/resources/software/#blas`, 2008. 2, 6, 67

Goto, Kazushige; van de Geijn, Robert A. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software (TOMS)*, 34(3):1–25, 2008. ISSN 0098-3500. 44

Gunnels, John A.; Gustavson, Fred G.; Henry, Greg M.; van de Geijn, Robert A. FLAME: Formal Linear Algebra Methods Environment. *ACM Transactions on Mathematical Software (TOMS)*, 27(4):422–455, December 2001. ISSN 0098-3500. 12

Hackenberg, Daniel. Fast matrix multiplication on cell (smp) systems, `http://www.tu-dresden.de/zih/cell/matmul`, 2007. 104

Harrison, Michael A. *Introduction to Formal Language Theory*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1978. ISBN 0201029553. 17

Hiranandani, Seema; Kennedy, Ken; Tseng, Chau-Wen. Evaluation of compiler optimizations for fortran d on mimd distributed memory machines. In *ICS '92: Proceedings of the 6th international conference on Supercomputing*, pages 1–14, New York, NY, USA, 1992. ACM. ISBN 0-89791-485-6. 62

Hyafil, Laurent; Rivest, Ronald L. Constructing optimal binary decision trees is NP-complete. *Information Processing Letters*, 5(1):15–17, 1976. 94

Intel, Reference Manual. Integrated Performance Primitives (IPP) 6.0, 2009a. 1, 3, 6, 7, 67

Intel, Reference Manual. Math Kernel Library (MKL) 10.0, 2009b. 2, 7, 67

Intel, Reference Manual. Optimization guide, 2009c. 54

Intel, Reference Manual. VTune perfomance analyzer 9.1, 2009d. 25

ISO, Standard. ISO/IEC 15444-1 - information technology – JPEG 2000 image coding system: Core coding system, 2004. 25

Jain, Raj K. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley New York, 1991. ISBN 978-0-471-50336-1. 25

Jalby, William; Lemuet, Christophe. Exploring and optimizing Itanium 2^{TM}cache(s) performance for scientific computing. In *2nd Workshop on Explicitly Parallel Instruction Computing Architecture and Compilers (EPIC-2)*, pages 4–18, 2002. 78

Johnson, Jeremy R.; Püschel, Markus. In search of the optimal walsh-hadamard transform. In *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, volume 6, pages 3347–3350, 2000. 69

Johnson, Jeremy R.; Johnson, Robert W.; Rodriguez, Domingo; Tolimieri, Richard. A methodology for designing, modifying, and implementing Fourier transform algorithms on various architectures. *IEEE Transactions on Circuits, Systems, and Signal Processing*, 9(4):449–500, 1990a. ISSN 0278-081X. 15, 60

Johnson, Robert W.; Huang, C.-H.; Johnson, Jeremy R. Multilinear algebra and parallel programming. In *Supercomputing '90: Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, pages 20–31, Los Alamitos, CA, USA, 1990b. IEEE Computer Society Press. ISBN 0-89791-412-0. 23

Johnsson, Thomas. Lambda lifting: transforming programs to recursive equations. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 190–203, New York, NY, USA, 1985. Springer-Verlag New York, Inc. ISBN 3-387-15975-4. 56

Kågström, Bo; Ling, Per; van Loan, Charles. GEMM-based level 3 BLAS: high-performance model implementations and performance evaluation benchmark. *ACM Transactions on Mathematical Software (TOMS)*, 24(3):268–302, 1998. ISSN 0098-3500. 11

Kang, Inyup; Willson, Alan N. Jr. Low-power viterbi decoder for CDMA mobile terminals. *IEEE Journal of Solid-State Circuits*, 33(3):473–482, Mar 1998. ISSN 0018-9200. 26

Karn, Phil. FEC library version 3.0.1, `http://www.ka9q.net/code/fec/`, Aug 2007. 106

Kennedy, Ken; McKinley, Kathryn S. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 301–320, London, UK, 1994. Springer-Verlag. ISBN 3-540-57659-2. 39

Kisuki, Toru; Knijnenburg, Peter M. W.; O'Boyle, Michael F. P. Combined selection of tile sizes and unroll factors using iterative compilation. In *PACT '00: Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*, page 237, Washington, DC, USA, 2000a. IEEE Computer Society. ISBN 0-7695-0622-4. 10

Kisuki, Toru; Knijnenburg, Peter M. W.; O'Boyle, Michael F. P.; Wijshoff, Harry A. G. Iterative compilation in program optimization. In *Proceedings of CPC'10 (Compilers for Parallel Computers)*, pages 35–44, 2000b. 18

Kocsis, Levente; Szepesvári, Csaba. Bandit based Monte-Carlo planning. In *European Conference on Machine Learning (ECML)*, volume 4212 of *Lecture Notes in Computer Science*, pages 282–293. Springer, 2006. 84

Lai, Tze L.; Robbins, H. Asymptotically efficient adaptive allocation rules. *Advances in Applied Mathematics*, 6(1): 4–22, 1985. 85

Landau, David; Binder, Kurt. *A Guide to Monte Carlo Simulations in Statistical Physics*. Cambridge University Press, New York, NY, USA, 2005. ISBN 0521842387. 83

Larsen, Knud J. Short convolutional codes with maximal free distance for rates 1/2, 1/3, and 1/4. *IEEE Transactions on Information Theory*, 19(3):371–372, May 1973. ISSN 0018-9448. 106

Lau, Jeremy; Schoenmackers, Stefan; Calder, Brad. Transition phase classification and prediction. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 278–289, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2275-0. 10

Lawrie, D. H. Access and alignment of data in an array processor. *IEEE Transactions on Computers*, C-24(12):1145–1155, Dec. 1975. ISSN 0018-9340. 33

Leather, Hugh; Bonilla, Edwin; O'Boyle, Michael F. P. Automatic feature generation for machine learning based optimizing compilation. In *CGO '09: Proceedings of the International Symposium on Code Generation and Optimization*, pages 81–91, 2009. 10

Li, Xiaoming; Garzarán, María J.; Padua, David. A dynamically tuned sorting library. In *International Symposium on Code Generation and Optimization (CGO)*, pages 111–124, 2004. 2, 67

Lin, Chien-Ching; Shih, Yen-Hsu; Chang, Hsie-Chia; Lee, Chen-Yi. Design of a power-reduction viterbi decoder for wlan applications. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 52(6):1148–1156, June 2005. ISSN 1549-8328. 26

Malvar, Henrique S. *Signal Processing with Lapped Transforms*. Artech House, Inc., Norwood, MA, USA, 1992. ISBN 0890064679. 14

McFarlin, Daniel; Franchetti, Franz; Moura, José M. F.; Püschel, Markus. High performance synthetic aperture radar image formation on commodity architectures. In *SPIE Conference on Defense, Security, and Sensing*, volume 7337, page 733708. Proceedings of SPIE, 2009. 25

Metropolis, Nicholas; Ulam, Stanislaw. The monte carlo method. *Journal of the American Statistical Association*, 44 (247):335–341, 1949. ISSN 01621459. 83

Milder, Peter A.; Franchetti, Franz; Hoe, James C.; Püschel, Markus. Formal datapath representation and manipulation for implementing DSP transforms. In *Design Automation Conference (DAC)*, pages 385–390, 2008. 11, 33

Milner, Robin; Tofte, Mads; Harper, Robert. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1990. ISBN 0262132559. 78

Mirković, Dragan. Automatic performance tuning in the UHFFT library. In *ICCS '01: Proceedings of the International Conference on Computational Sciences*, volume 2073 of *Lecture Notes in Computer Science*, pages 71–80, London, UK, 2001. Springer. 11

Mitola, Joseph III. *Software Radio Architecture: Object-Oriented Approaches to Wireless Systems Engineering*. John Wiley & Sons, 2002. ISBN 978-0-471-38492-2. 26

Monsifrot, Antoine; Bodin, François; Quiniou, René. A machine learning approach to automatic production of compiler heuristics. In *AIMSA '02: Proceedings of the 10th International Conference on Artificial Intelligence: Methodology, Systems, and Applications*, volume 2443 of *Lecture Notes in Computer Science*, pages 41–50, London, UK, 2002. Springer. ISBN 3-540-44127-1. 10, 77

Moss, Eliot; Utgoff, Paul; Cavazos, John; Brodley, Carla; Scheeff, David; Precup, Doina; Stefanović, Darko. Learning to schedule straight-line code. In *NIPS '97: Proceedings of the 1997 conference on Advances in neural information processing systems 10*, pages 929–935, Cambridge, MA, USA, 1998. MIT Press. ISBN 0-262-10076-2. 77

Müller, Martin. Computer go. *Artificial Intelligence*, 134(1-2):145–179, 2002. ISSN 0004-3702. 83

Naishlos, Dorit. Autovectorization in GCC. In *Proceedings of the 2004 GCC Developers Summit*, pages 105–118, 2004. 59

Nussbaumer, Henry J. *Fast Fourier Transformation and Convolution Algorithms*. Springer, 2nd edition, 1982. 17

Pan, Zhelong; Eigenmann, Rudolf. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 319–332, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2499-0. 77

Pease, Marshall C. An adaptation of the fast fourier transform for parallel processing. *J. ACM*, 15(2):252–264, 1968. ISSN 0004-5411. 33

Püschel, Markus; Moura, José M. F. Algebraic signal processing theory: Cooley-Tukey type algorithms for DCTs and DSTs. *IEEE Transactions on Signal Processing*, 56(4):1502–1521, 2008. 14, 17

Püschel, Markus; Moura, José M. F.; Johnson, Jeremy R.; Padua, David; Veloso, Manuela; Singer, Bryan; Xiong, Jianxin; Franchetti, Franz; Gacic, Aca; Voronenko, Yevgen; Chen, Kang; Johnson, Robert W.; Rizzolo, Nicholas. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005. 3, 6, 7, 11, 13, 15, 17, 37, 69, 78, 80

Quinlan, J. Ross. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986. ISSN 0885-6125. 94

Quinlan, J. Ross. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993. ISBN 1-55860-238-0. 93, 94

Rader, Charles M. Discrete fourier transforms when the number of data samples is prime. *Proceedings of the IEEE*, 56 (6):1107–1108, June 1968. ISSN 0018-9219. 17

Rader, Charles M. Memory management in a viterbi decoder. *IEEE Transactions on Communications*, 29(9):1399–1401, Sep 1981. ISSN 0096-2244. 33

Rao, K. Ramamohan; Yip, Patrick. *Discrete cosine transform: algorithms, advantages, applications*. Academic Press Professional, Inc., San Diego, CA, USA, 1990. ISBN 0-12-580203-X. 14

Rizzolo, Nicholas; Padua, David. Hilo: High level optimization of FFTs. In *International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2004. 54

Shannon, Claude E. A mathematical theory of communication. *Bell system technical journal*, 27:379–423, 1948. 94

Sheard, Tim; Jones, Simon Peyton. Template meta-programming for haskell. *SIGPLAN Not.*, 37(12):60–75, 2002. ISSN 0362-1340. 35

Shen, Hao. Generation of a fast JPEG 2000 encoder using Spiral, Technical University of Denmark, Master's thesis, 2008. 25

Sherwood, Timothy; Perelman, Erez; Hamerly, Greg; Calder, Brad. Automatically characterizing large scale program behavior. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 45–57, New York, NY, USA, 2002. ACM. ISBN 1-58113-574-2. 10

Singer, Bryan; Veloso, Manuela. Stochastic search for signal processing algorithm optimization. In *Supercomputing (SC)*, page 22, 2001. 11, 80

Singer, Bryan; Veloso, Manuela. Learning to construct fast signal processing implementations. *Journal of Machine Learning Research, special issue on "the Eighteenth International Conference on Machine Learning (ICML 2001)"*, 3:887–919, 2002. 11, 77

Stephenson, Mark; Amarasinghe, Saman. Predicting unroll factors using supervised classification. In *CGO '05: Proceedings of the international symposium on Code generation and optimization*, pages 123–134, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2298-X. 10, 77

Stephenson, Mark; Amarasinghe, Saman; Martin, Martin; O'Reilly, Una-May. Meta optimization: improving compiler heuristics with machine learning. *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation (PLDI'03)*, 38(5):77–90, 2003. ISSN 0362-1340. 77

Strassen, Volker. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, 1969. 25

Streeter, Matthew J.; Smith, Stephen F. A simple distribution-free approach to the max k-armed bandit problem. In *Principles and Practice of Constraint Programming (CP 2006)*, pages 560–574, 2006. 88

Taipale, Dana. Implementing viterbi decoders using the VSL instruction on DSP families DSP56300 and DSP56600. Technical report, Freescale Semiconductor, Inc., 2004. 31

Temam, Olivier; Granston, Elana D.; Jalby, William. To copy or not to copy: a compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *Supercomputing '93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 410–419, New York, NY, USA, 1993. ACM. ISBN 0-8186-4340-4. 44

Tolimieri, Richard; An, Myoung; Lu, Chao. *Algorithms for Discrete Fourier Transforms and Convolution*. Springer, 2nd edition, 1997. ISBN 978-0-387-98261-8. 13, 17

van de Geijn, Robert A.; Quintana-Ortí, Enrique S. *The Science of Programming Matrix Computations*. Lulu, 2008. 12

Van Loan, Charles. *Computational frameworks for the fast Fourier transform*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1992. ISBN 0-89871-285-8. 11, 13, 15

Viterbi, Andrew J. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, 13(2):260–269, Apr 1967. ISSN 0018-9448. 25

Viterbi, Andrew J. *CDMA: principles of spread spectrum communication*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1995. ISBN 0-201-63374-4. 6

Viterbi, Andrew J. A personal history of the viterbi algorithm. *IEEE Signal Processing Magazine*, 23(4):120–142, July 2006. ISSN 1053-5888. 26

Voronenko, Yevgen. *Library Generation for Linear Transforms*. PhD thesis, Electrical and Computer Engineering, Carnegie Mellon University, 2008. 6, 7, 8, 9, 11, 12, 36, 37, 39, 44, 46, 49, 50, 52, 56, 58, 70, 71, 73, 75, 109

Voronenko, Yevgen; Püschel, Markus. Algebraic signal processing theory: Cooley-Tukey type algorithms for real DFTs. *IEEE Transactions on Signal Processing*, 57(1):205–222, 2009. 13, 17

Voronenko, Yevgen; Franchetti, Franz; de Mesmay, Frédéric; Püschel, Markus. Generating high-performance general size linear transform libraries using Spiral. In *High Performance Embedded Computing (HPEC)*, 2008a. 68, 69, 112

Voronenko, Yevgen; Franchetti, Franz; de Mesmay, Frédéric; Püschel, Markus. System demonstration of Spiral: Generator for high-performance linear transform libraries. In *Algebraic Methodology and Software Technology (AMAST)*, 2008b. 3

Voronenko, Yevgen; de Mesmay, Frédéric; Püschel, Markus. Computer generation of general size linear transform libraries. In *International Symposium on Code Generation and Optimization (CGO)*, pages 102–113, 2009. 9, 11, 17, 67, 73, 82, 108

Vuduc, Richard; Demmel, James W.; Yelick, Katherine A. OSKI: A library of automatically tuned sparse matrix kernels. In *Journal of Physics: Conference Series*, volume 16, pages 521–530, San Francisco, CA, USA, 2005. Institute of Physics Publishing. 2, 67

Wang, Yizao; Gelly, Sylvain. Modifications of UCT and sequence-like simulations for Monte-Carlo Go. In *IEEE Symposium on Computational Intelligence and Games, 2007. CIG 2007*, pages 175–182, 2007. 84

Wendykier, Piotr. JTransforms 1.2, `http://piotr.wendykier.googlepages.com/jtransforms`, April 2008. 109

Whaley, R. Clint. User contribution to ATLAS. Technical report, University of Tennessee, 2001. 11

Whaley, R. Clint; Dongarra, Jack J. Automatically tuned linear algebra software (ATLAS). In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-89791-984-X. 2, 5, 8, 11, 35, 54, 67

Whaley, R. Clint; Petitet, Antoine; Dongarra, Jack J. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001. 7, 11, 22

Williams, Samuel; Shalf, John; Oliker, Leonid; Kamil, Shoaib; Husbands, Parry; Yelick, Katherine. The potential of the cell processor for scientific computing. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 9–20, New York, NY, USA, 2006. ACM. ISBN 1-59593-302-6. 104

Xiong, Jianxin. *Automatic Optimization of DSP Algorithms*. PhD thesis, Computer Science, University of Illinois at Urbana-Champaign, 2001. 54

Xiong, Jianxin; Johnson, Jeremy R.; Johnson, Robert W.; Padua, David. SPL: A language and compiler for DSP algorithms. In *Programming Languages Design and Implementation (PLDI)*, pages 298–308, 2001. 8, 11, 13, 15

Yotov, Kamen; Li, Xiaoming; Ren, Gang; Garzarán, María J.; Padua, David; Pingali, Keshav; Stodghill, Paul. Is search really necessary to generate high-performance BLAS? *Proceedings of the IEEE*, 93(2):358–386, Feb. 2005. ISSN 0018-9219. 22

Yotov, Kamen; Roeder, Tom; Pingali, Keshav; Gunnels, John; Gustavson, Fred. An experimental comparison of cache-oblivious and cache-conscious programs. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 93–104, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-667-7. 25