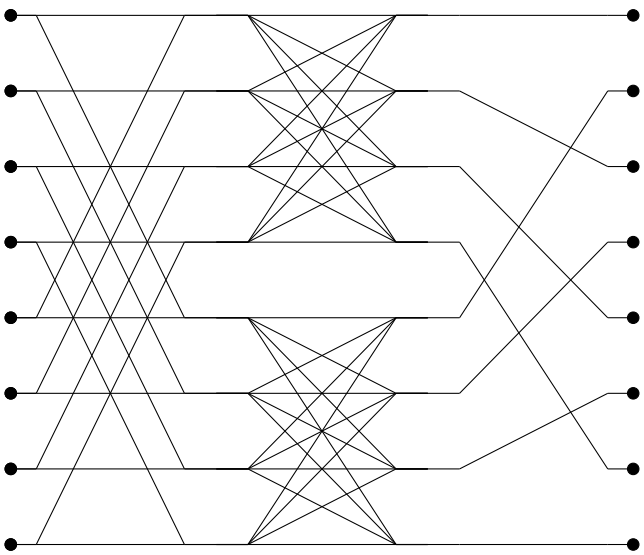
Computer Generation of Fourier Transform Libraries for Distributed Memory Architectures

SRINIVAS CHELLAPPA
ADVISOR: MARKUS PÜSCHEL
December 2010



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

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

Doctoral Dissertation Committee:
Professor FRANZ FRANCHETTI, Carnegie Mellon University
Professor JAMES HOE , Carnegie Mellon University
Professor JEREMY JOHNSON, Drexel University
Professor MARKUS PÜSCHEL (Advisor), Carnegie Mellon University
Professor P. SADAYAPPAN, Ohio State University

Abstract

High-performance discrete Fourier transform (DFT) libraries are an important requirement for many computing platforms. Unfortunately, developing and optimizing these libraries for modern, complex platforms has become extraordinarily difficult. To make things worse, performance often does not port, thus requiring permanent re-optimizations. Overcoming this problem has been the goal of SPIRAL, a library generation system that can produce highly optimal DFT code from a high level specification of algorithms and target platforms.

However, current techniques in SPIRAL cannot support all target platforms. In particular, several emerging target platforms incorporate a distributed memory parallel processing paradigm, where the cost of accessing non-local memories is relatively high, and handling data movement is exposed to the programmers. Traditionally used only in supercomputing environments, this paradigm is now also finding its way in the form of multicore processors into desktop computing.

The goal of this work is the computer generation of high-performance DFT libraries for a wide range of distributed memory parallel processing systems, given only a high-level description of a DFT algorithm and some platform parameters. The key challenges include generating code for multiple target programming paradigms that delivers load balanced parallelization across multiple layers of the compute hierarchy, orchestrates explicit memory management, and overlaps computation with communication.

We attack this problem by first developing a formal framework to describe parallelization, streaming, and data exchange in a domain-specific declarative mathematical language. Based on this framework, we develop a rewriting system that structurally manipulates DFT algorithms to "match" them to a distributed memory target architecture and hence extracts maximum performance. We implement this approach as a part of SPIRAL together with a backend that trans-

lates the derived algorithms into actual libraries. Experimental results show that the performance of our generated code is comparable to hand-tuned code in all cases, and exceeds the performance of hand-tuned code in some cases.

Table of Contents

List of Figures

List of Tables

“

Let the machine do the dirty work. ”

– KERNIGHAN AND RITCHIE
(Elements of Programming Style)

Acknowledgments

Numerous people have contributed in many ways to make this thesis possible. This is a meager attempt at thanking some of them.

The vision for this thesis originated with my advisor Markus Püschel. In addition to being an outstanding advisor and mentor, Markus was consistently supportive and dedicated a great deal of time and effort toward critiquing and improving my work throughout the course of my PhD. In addition, I thank Markus for teaching, and for showing by example, how to strive for quality in conducting research work, writing and presentation, and all else.

Franz Franchetti played a key role in this thesis through many initial discussions on parallelization which helped me lay the foundation for this work. In return, I "helped" Franz practice his whiteboard skills by allowing him to patiently bring me up to speed with SPIRAL as I commenced my graduate studies. In addition, I thank Franz for always being willing to entertain discussions both on and off-topic, and for always being up for having a good laugh discussing and making light of all things worldly and unworldly.

Having laid the foundation for SPIRAL decades ago, Jeremy Johnson supported and critiqued my work, and pointed me to valuable information along the way. I would like to thank him along with the other committee members James Hoe and P. Sadayappan for their invaluable inputs during and after the proposal of this thesis that have since guided my work.

Yevgen Voronenko, as the primary architect of SPIRAL, helped me in more ways than he was probably aware. Everything from his fantastic research ideas to feature additions to SPIRAL have been fun and inspiring to work with and think about.

My former advisor, Jonathan Simonson, was crucial in inspiring and deepening my interest in computer systems right from my undergraduate years. I owe a lot to his teaching and his

mentoring, which have proven to be highly valuable in the years since.

SPIRAL group members with their diverse backgrounds have been important in numerous ways in improving my work, including critiquing it and providing insightful suggestions. In particular, I would like to thank Frédéric de Mesmay[1], Peter Milder and Marek Telgarsky for all the time they spent helping me, whether by providing feedback on my writing or by helping me practice for a talk; and Volodymyr Arbatov for lending a helping hand with the code base whenever needed. Most importantly, thanks to them and to the other SPIRAL folks for having been a terrific group of people to have worked and spent time with, overall!

None of this would have been possible without my family: in particular, my uncle who sowed the seed of scientific curiosity and instilled in me an awe of the engineering approach which have both been boundless sources of passion and satisfaction; my parents who always pushed me to succeed, and never hesitated to give me whatever it took to do so; my sister and brother-in-law for their steadfast support; and finally, my nephew and niece, both of whom commenced their life's journey during the course of this graduate work, and have since unfailingly kept me entertained with their antics.

I thank the Electrical and Computer Engineering department at CMU for the fellowship received to work on this PhD, for having a great set of faculty, peers, and staff to work with, and for creating and maintaining a highly enriching environment for graduate work. Speaking of staff, a special thank you to Carol Patterson for having made my time in the department smoother than I could have ever asked for.

And speaking of environments, my time in Pittsburgh and at Carnegie Mellon has been great, to say the least, thanks to countless friends, old and new, nearby and far, EGO volunteers, the CMU quiz club folks (for the entertaining quizzing sessions, and for never giving up on questionable attempts to expand the boundaries of wit), social dancers, confronting philosophers, fellow skydivers, orthopedists[2], beer recommenders, and a ton of other folks who have contributed to my experience here.

This research would have been impossible to conduct without funding from various organizations. It was supported by the National Science Foundation (NSF) through awards 0325687, 0702386, by Defense Advance Research Projects Agency (DARPA) via Department of Interior (DOI) grant NBCH-1050009, Army Research Office (ARO) grant W911NF0710416, Mercury Computer Systems, and the National Science Foundation through TeraGrid resources provided by Pittsburgh Supercomputing Center under grant number CCR100026. In addition, I thank IBM, Georgia Institute of Technology's Sony-Toshiba-IBM Center of Competence, and Cray Inc. for

[1] I credit my association with Fred and Markus for my current level of expertise with diacritics in LaTeX.

[2] The close vicinity of these two is not a coincidence!

their equipment resources that made some of the experimental evaluation in this thesis possible.

And finally, a big thank you to the thousands of free and open source contributors for having made computers themselves an inexpensive, yet inexhaustible pleasure to learn from and work with!

Pittsburgh, PA, December 2010
SRINIVAS CHELLAPPA

CHAPTER 1

Introduction

Most compute intensive applications spend the bulk of their run time in basic numerical functions, called kernels. Examples of such kernels include matrix multiplication, linear system solvers, and linear transforms. The most important transform, and arguably one of the most important kernels overall, is the discrete Fourier transform (DFT). The DFT is used by applications as diverse as audio, image, and video processing, wireless communications, molecular simulations, and differential equation solvers. These applications are executed on various computing platforms, from large scale supercomputers to desktop computers to embedded systems.

Typically, software application developers use libraries, available for kernels such as the DFT, for two main reasons. First, they lower the programming burden by allowing code reuse. Second, libraries can abstract away the need for platform-specific performance optimizations: applications can make library calls while being oblivious to platform specific details, while libraries providing the same functionality can be tuned separately on various platforms for performance.

Since a significant part of an application's run time is spent in calls to libraries, it is critical for such libraries to be fast. However, due to the increasing complexity and diversity of hardware architectures, it is common for a DFT library that performs well on one platform to perform poorly on another. As mentioned above, it has thus become increasingly common to tune libraries to specific platforms for maximal performance. For this reason, several implementations of a library may exist optimized for different platforms.

We illustrate the problem for the DFT on a recent multicore processor (the Cell BE, with 9 cores) in Figure 1.1. The horizontal axis spans a range of input problem sizes, and the vertical axis is performance (inverse run time) in pseudo-giga-floating point operations per second (higher is better). "Pseudo" means that the operations count is estimated as $5N\log N$. Performance is

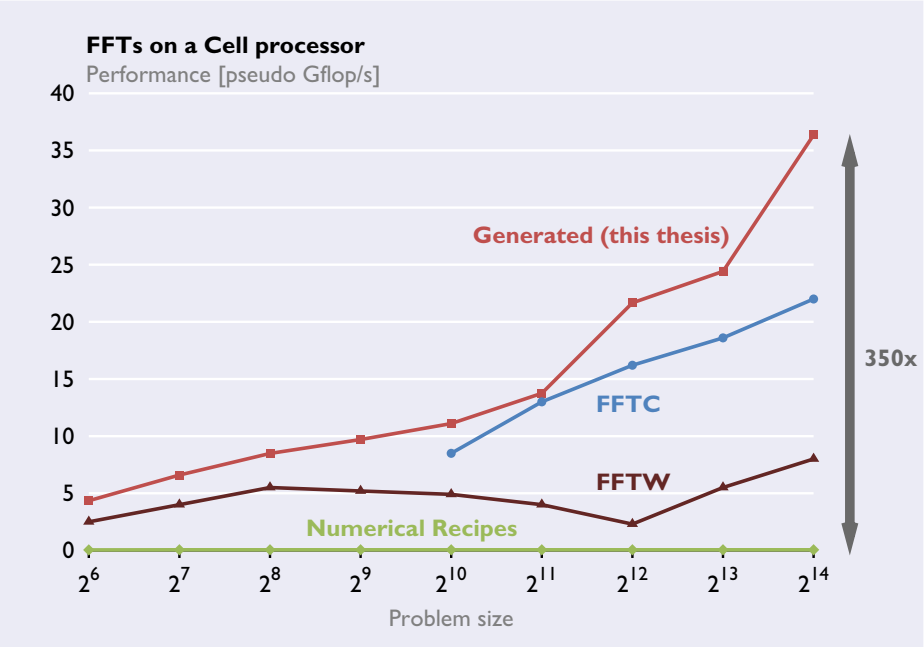


FIGURE 1.1: DFT performance on the IBM Cell BE (peak floating point performance is 204.8 Gflop/s). A naïve DFT implementation based on only minimizing the operations count under-performs by two orders of magnitude, and even expert libraries do not achieve the optimal performance.

hence inversely proportional to run time. The plot shows the performance of four difference DFT implementations, all based on fast algorithms, called fast Fourier transforms (FFTs). The bottom line, which performs worst is the C implementation from Numerical Recipes of the fast Fourier transform based directly on its mathematical definition [Press et al., 1992]. FFTW [Frigo and Johnson, 2005], a popular adaptive library with a Cell extension performs considerably better, but fails to achieve over 10 Gflop/s for the problem sizes shown. FFTC [Bader and Agarwal, 2007], a library written and manually tuned specifically for the Cell performs better, but is still suboptimal on the Cell. The topmost line shows the performance of code that was automatically generated by the work in this thesis. It is two orders of magnitude faster than the naïve Numerical Recipes implementation, and up to 1.6x faster than the fastest hand-tuned implementation.

Surprisingly, all the implementations shown in Figure 1.1 have roughly the same operations count, which dramatically illustrates the limitations of compilers on modern multicore platforms. Ideally, simply compiling the Numerical Recipes code would yield the performance of the topmost line. However, this is likely to be impossible since the most important optimizations involve restructuring the algorithms to match the features of the target platform. For distributed memory platforms like the Cell, this means generating parallelized code that is load balanced,

code that uses the interconnect between the cores efficiently, and code that overlaps computation with communication to minimize or hide communication costs.

Mapping a DFT to a complex platform like the Cell is difficult. There are various FFTs, each of which can be modified in numerous ways to better "fit" a given target platform. General purpose compilers are unable to perform this task because expressing an FFT in an imperative language like C fixes the algorithm being used, and the compiler lacks the domain knowledge to change it. Hence, the optimization task is left with the programmer who needs to have a profound understanding of the algorithm and platform to adapt to its parallelism, memory structure, and other architectural details.

Further compounding this problem, the number of platforms is also large, and they are continuously updated. Writing, tuning, and maintaining separate DFT libraries for each of these platforms is very expensive, and in practice, infeasible. The goal of this thesis is to overcome this problem for DFT libraries targeting the important class of parallel distributed memory platforms. The idea, hinted at in Figure 1.1, is to replace the human programmer by a program generation system that produces optimal DFT code from a high-level specification and a few platform parameters. Before we state the goal in more detail, we provide some background on distributed memory platforms and state-of-the-art DFT libraries to put our work in context.

1.1 Distributed Memory Platforms

Simply put, a distributed memory architecture is one in which multiple processors exist, each with its own private memory. Moving data between the processors, when required, is relatively expensive relative to computation.

The distributed memory architectural paradigm exists in both chip-based processors and node-based supercomputers. As Figure 1.2(c) shows, chip-based distributed memory architectures like the Cell BE are constructed from many single core processors (Figure 1.2(a)) with associated on-chip local memories, connected via an on-chip interconnect. Larger supercomputing platforms also follow the distributed memory paradigm: as Figure 1.2(d) shows, platforms like the Cray XT5 are constructed using many multicore shared memory chip processors (Figure 1.2(b)) connected via an external network like InfiniBand.

To see how the distributed memory paradigm evolved and why the paradigm is and will remain an important feature of computing systems, we look at these two broad classes of computing platforms—supercomputing platforms and desktop computing platforms—in greater detail below.

Parallelism in supercomputing platforms. Supercomputing platforms are designed to run

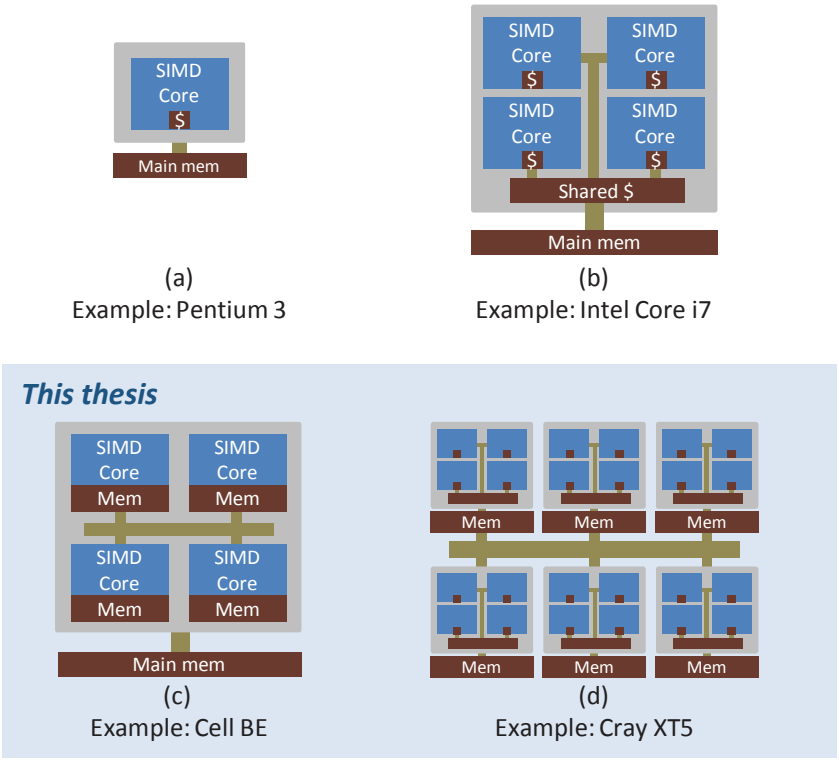


FIGURE 1.2: Architectures targeted by this thesis. (a) shows a single core SIMD CPU. (b) shows a cache based multicore CPU with shared cache and main memory. (c) shows a distributed memory scratchpad based multicore CPU with shared main memory. (d) shows a distributed memory multi-node supercomputer. In this thesis, we target architectures of the type shown in (c) and (d).

large scale compute intensive programs. In original supercomputing designs, processing was typically the bottleneck, and the focus was on designing faster scalar processors. As processor speeds scaled to their limits, the need for ever higher compute power made designers turn to parallelism, first in the form of vector architectures, then in the form of multi-node (4–16) architectures, and finally, in current massively parallel processing (MPP) designs with over 200,000 nodes. This is particularly evident in the ratio shift of distributed memory systems (clusters and MPP architectures) from 25% in 1993 to over 99% in 2010 among the top 500 fastest supercomputers [TOP500, 2010].

The need for parallelism stems primarily from the impracticality of scaling the amount of compute power in a single processor or chip due the power density problem and contention for off-chip resources including memory. The distributed memory design overcomes these problems by exploiting the data access locality of programs to reduce overall contention for memory, which in turn allows for scalability.

On the other hand, the main design constraint in distributed memory systems is the slow

speed of data movement relative to the processing speed. This means that the exchange of data often causes bottlenecks in computing. Another problem is that application programs executing on distributed memory architectures must explicitly manage data movements, which increases programming complexity. Even with languages like Unified Parallel C (UPC) that attempt to alleviate programming complexity by providing the application with a single shared partitioned global address space (PGAS), the underlying hardware constraints still exist, and algorithms must be designed to both minimize data movement, and minimize overhead costs by reading and writing non-local data in large packets. In many cases, algorithms must be redesigned and rewritten at a high level to achieve the required characteristics.

In summary, distributed memory is is an important feature of supercomputing systems, and is expected to remain so for the foreseeable future. We next look at this paradigm in desktop computing platforms.

Parallelism in desktop platforms. For several decades, the growth in performance of computing platforms, and in particular chip processors, primarily came from frequency scaling. In the last few years, frequency scaling has plateaued due to physical limitations. To keep increasing floating point performance in accordance with Moore's Law, architects have instead turned to parallelism. While this allows for growth in peak theoretical floating point performance, it causes new problems: programming complexity, and the increased contention for shared resources.

Programming for parallel architectures is difficult by nature of the problem. Software must be written to expose parallelism to take advantage of multicore processors. In addition, software may have to take into account the specifics of the hardware platform in order to run efficiently. As an example, false cache-line sharing is a problem introduced by multicore processors that affects performance. Software must be aware of the cache-line size of the processor to ensure that false-sharing is minimized. This, in combination with the increasing number of different chip processors leads to reduced performance portability, and further increases programming burden by forcing developers to create and maintain differently tuned versions of their software for various platforms.

The scaling of parallelism in chip processors also leads to increased contention for both on-chip and off-chip shared resources. Arguably the most important resource is access to memory, including both on-chip (shared coherent caches) and off-chip (main memory) types. Scaling the number of cores on a chip places an enormous burden on shared on-chip caches, and also on the off-chip data bandwidth, which does not scale with the number of cores per chip. This further aggravates the existing problem of the "memory wall," which is the problem where data intensive programs are unable use the available processing power due to inadequate off-chip latencies and bandwidths.

One possible solution is the distributed cache approach, where cache memories are placed physically close to the processing core that will primarily use them. This is reminiscent of, and very similar to the idea of distributed local memories used by supercomputing platforms.

Such cache systems may be coherent (as in the Intel Larrabee) and allow for a shared-memory programming paradigm, or non-coherent (like in the Cell BE), and require a distributed memory programming paradigm. Regardless of the nature of coherency or the programming paradigm supported, the underlying hardware tradeoffs are similar to distributed memory platforms in supercomputers: inter-core data transfers are expensive as they are subject to bandwidth and latency limitations, and also suffer overhead costs. From a performance perspective, software must minimize inter-core data transfers, and use algorithms designed to transfer memory using large packet sizes. In addition, applications must manage the reduced per-core memory bandwidth well. These factors further add to the programming burden, especially for performance critical software like DFT libraries.

The future of the distributed memory paradigm. For the reasons we have stated, the distributed memory paradigm is an important template in the architect's toolbox to design platforms that can scale in performance. It is currently a mainstay of supercomputing platforms, and is expected to remain so for the foreseeable future. In addition, it is rapidly finding its way into desktop platforms. Thus, from the perspective of writing performance-critical libraries, it is important to understand how to develop software for the distributed memory paradigm, and in addition, how to reduce the development and optimization effort.

1.2 High-Performance DFT Libraries

As demonstrated in Figure 1.1, writing high-performance DFT libraries is difficult. There have been several approaches to achieving platform-tuned high performance DFTs in the past. In this section we discuss approaches for non-distributed memory systems, followed by approaches for distributed memory systems. We also illustrate the deficiencies in current state-of-the-art of DFT libraries for distributed memory systems that this thesis addresses.

The problem. Computing the DFT using a straightforward implementation of an FFT that only minimizes operations count does not yield performance as Figure 1.1 shows. This is mainly because run time on modern architectures is dominated by architectural features such as parallelism and the memory hierarchy. This means that the "structure" of an algorithm matters as much as the operations count. As mentioned earlier, a general purpose compiler is unable to perform the necessary optimizations because it lacks domain knowledge. Parallelizing and vectorizing compilers are unsuccessful for the same reasons. Below we look at alternative ap-

proaches to tuning DFT libraries.

1.2.1 DFT Libraries for Non-Distributed Memory Platforms

Hand-tuned libraries. Several hand-tuned DFT libraries exist for single core and multicore platforms. Vendor libraries like Intel's IPP and MKL are hand-tuned specifically to the vendor's hardware products. FFTE [Takahashi, 2002] is a DFT library that includes support for a range of shared memory multicore processors. The GNU Scientific Library (GSL) [FSF, 2010], also manually written, supports FFTs and is meant to be used on a wide variety of platforms. Accordingly, the performance is suboptimal.

There are several issues with manual tuning: first, it requires programmers with combined algorithm and architecture expertise, which is expensive. Second, such a manually tuned library may not perform well on other hardware. Third, libraries may have to be manually re-tuned whenever new platforms are released.

Performance portable libraries. One successful approach to facilitate performance porting is the use of adaptive libraries like FFTW [Frigo and Johnson, 2005] and UHFFT [Mirković and Johnsson, 2001], which both support DFTs on multicore architectures. In this approach, the library has built-in degrees of freedom in how to perform the computation, which arise from the different possible ways to recursively decompose the DFT into smaller DFTs. A runtime planning routine (the *planner*) selects the best decomposition strategy based on timing information obtained from the platform, at library installation or program initialization time. Small DFTs are computed using pre-written code (called *codelets* in FFTW terminology), which are either handwritten, or in the case of FFTW, generated using a special-purpose compiler (*genfft*).

Although a degree of performance portability is achieved with these libraries, extensive work or a fundamental redesign may be required to allow these libraries to work on new architectural features or new programming paradigms. An example of this is the UHFFT library: though it supports multicore architectures, it is unable to support SIMD (Single Instruction Multiple Data) vectorization. In addition, these libraries are not able to easily exploit new algorithms, and cannot easily be extended to new compute functions.

Program generation and optimization. The approach taken by SPIRAL [Püschel et al., 2005] is to "teach" computers how to generate and optimize code for linear transforms. SPIRAL works by representing both the algorithm and the architecture at a high level of abstraction using Kronecker (tensor) products ([Johnson et al., 1990]), and manipulates the algorithm at this level to map it to the hardware.

This has several advantages over adaptive libraries. First, SPIRAL can generate code for any transform that can be represented using its internal abstraction. These may include transforms

other than the DFT, or even functions beyond transforms [Franchetti et al., 2009a]. Second, adding new algorithms for an existing transform is comparatively simple as long as they can be represented using the internal abstraction. New algorithms may yield increased performance for specific sizes, or may be useful in producing a desired tradeoff point between performance and accuracy. Third, SPIRAL is designed to be extensible to new architectural paradigms such as multithreading or vector instruction sets [Franchetti et al., 2005, 2006a]. Fourth, SPIRAL can also generate adaptive libraries of the type discussed above [Voronenko, 2008]. Finally, since SPIRAL works at a high level of abstraction, lower level details such as the programming paradigm, programming memory model (e.g., distributed or shared memory), and platform specific instructions (e.g., DMA, MPI) can be easily included as backends. We discuss SPIRAL in more detail in Section 2.4 and Section 2.4.4.

1.2.2 DFT Libraries for Distributed Memory Platforms

Hand-tuned libraries. Hand-tuned libraries that exist for distributed memory platforms include FFTE [Takahashi, 2002], IBM's eSSL [IBM, 2010], and also parallelized versions of GSL [Aliaga et al., 2009]. All these use fixed algorithms, and while they may perform well on certain target architectures, performance typically does not port over to a range of architectures.

Performance portable libraries. FFTW includes support for both the Cell BE and for MPI based systems. These are both separate components in FFTW. On most MPI based systems, FFTW performs reasonably well and is typically used as the standard DFT library unless a faster, vendor tuned version is available.

Program generation and optimization. Although SPIRAL can generate code for architectures with a memory hierarchy, SIMD vector architectures, and shared memory multicore processors, it is currently unable to generate code for distributed memory platforms. [Chen and Johnson, 2004] presented a self-adapting distributed memory package to compute the Walsh-Hadamard transform (WHT), which is similar to the FFT. [Bonelli et al., 2006] added some preliminary, basic support to SPIRAL for generating DFT code for MPI platforms. However, this support was for only a single parallelized FFT, was not compatible with SIMD vectorization, did not support flexible explicit data transfers, and did not support any form of memory streaming or overlapping computation with communication to minimize communication costs.

In summary, the problem of developing highly optimized DFT libraries is very well understood as shown by SPIRAL which completely automates the process. However, this excludes the important class of parallel distributed memory platforms. While the hand-written FFTW provides a good solution, its performance may be suboptimal (Figure 1.1). Hence, it is desirable to have a flexible method to automatically generate DFT libraries to produce a long-term solution

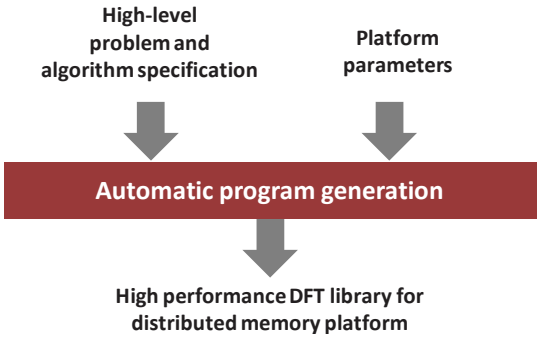


FIGURE 1.3: Goal of this thesis.

to the optimization and porting problem.

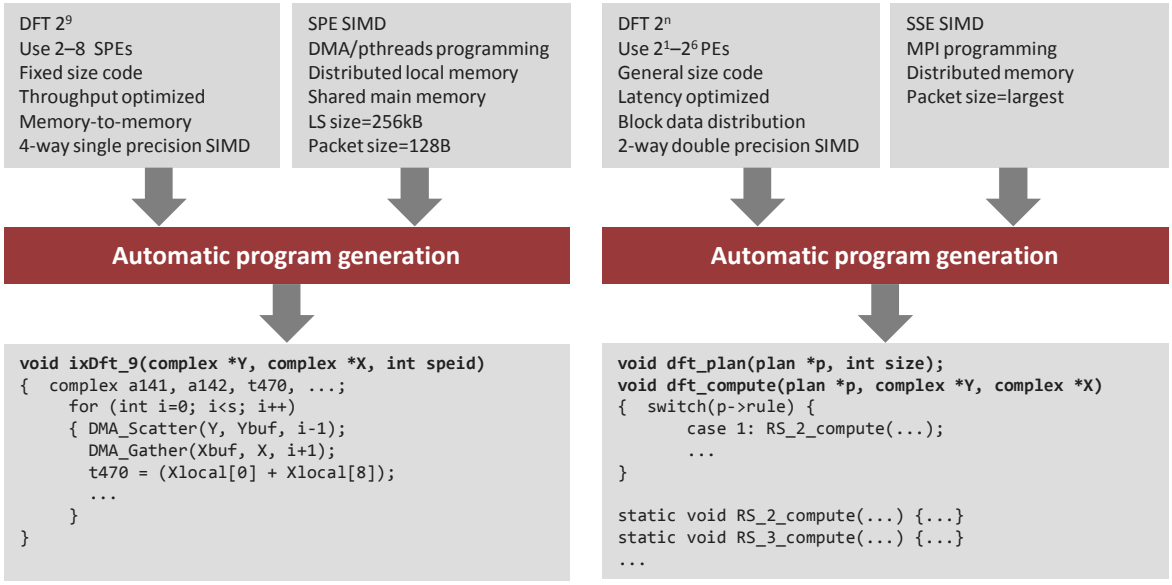
1.3 Problem Statement

Although the SPIRAL approach has been successful in automating library development for DFTs and other linear transforms, SPIRAL is unable to generate code for distributed memory platforms. We address this deficiency in this thesis, leveraging, but also considerably extending prior work:

> *The goal of this thesis is the computer generation of high-performance DFT libraries for distributed memory parallel processing systems, given only a high-level description of DFT algorithms and a set of platform and architecture parameters.*

The proposed generator is visualized in Figure 1.3. The problem specification is a functional description of the transform (we focus on the DFT) to generate and optimize code for. This includes transform parameters (e.g., transform size for fixed size code), code or library specification (e.g., block cyclic distributed data layout, and optimization for throughput). The algorithm specification, which is a part of our system's "knowledge," is a mathematical description of various DFT algorithms. The platform parameters provide a small set of relevant architectural details (e.g., number of processors, vector length, data transfer packet size, programming paradigm).

In Figure 1.4, we show two examples of libraries that we generate. In Figure 1.4(a), we specify that we want to generate fixed-size code for a DFT of size 2^9 using 2–8 processors, optimized for throughput, for an input vector stored in main memory, using the processor's 4-way single precision SIMD vectorization. We also specify the relevant platform parameters of the target platfom, which is the Cell BE in this case, including the SIMD ISA and the DMA/pthreads based programming paradigm, the memory paradigm, the size of the local store, and the minimum data packet size to use for communications. Correspondingly, the output is a single function in C that implements the DFT using Cell specific programming instructions.



(a) Fixed size throughput optimized code for the Cell BE

(b) General size latency optimized MPI library

FIGURE 1.4: Example libraries that can be generated using our system in Figure 1.3.

In Figure 1.4(b), we specify a different scenario: we want to generate a latency optimized general size library for the DFT that uses a number of PEs specified at run time, a block data distribution, and the architecture's 2-way double precision SIMD vectorization. Under platform parameters, we specify that we want to use an MPI based distributed programming model, Intel SSE SIMD instructions, and the largest packet size possible for any problem size. The output of our system in this case is a general size library [1] that is somewhat similar to FFTW: it includes a `dft_plan` routine to be called at library initialization time with a specified problem size. The routine finds the fastest computation strategy for this size and returns it as a "plan." This means that the library has an adaptation mechanism. The `dft_compute` routine then uses this plan to compute the DFT using a set of mutually recursive functions.

Though our system can potentially provide a wide range of functionality on many distributed memory platforms, we focus on the DFT and two representative platforms: the Cell BE and MPI based platforms. We capture the range of functionality that is used to evaluate this thesis in Table 1.1.

[1] To generate general size libraries in this thesis, we use infrastructure developed by Yevgen Voronenko and Volodymyr Arbatov, including search mechanisms developed by Frédéric de Mesmay.

Feature	Cell	MPI
Code type	Fixed input size DFT	General input size DFT
Memory paradigm	Distributed local memory, shared main memory	Distributed memory
Programming paradigm	DMA + pthreads	MPI
SIMD ISA	SPE 2-way double precision, 4-way single precision	SSE 2-way and Power 2-way double precision
Optimizations	Latency, throughput	Latency
Data layouts	Block distributed, block cyclic	Block distributed, block cyclic

TABLE 1.1: An overview of the DFT library features supported by our system for the two target platforms.

From a standard FFT, and the platform parameters supplied, our system formally derives FFT variants that are adapted to the target architecture. In many cases, this eliminates the need to search over a large space of variants. For platform parameters that are difficult to model, we define and search over a small space of potential algorithms.

The output of our system is high-performance platform-tuned code for the desired input transform(s) that includes platform-specific instructions (e.g., DMA code for the Cell or MPI code for supercomputing platforms). Our system is also designed to work with other architectural paradigms and optimizations that previous work with SPIRAL accomplished, including the SIMD vector paradigm and optimizations for the memory hierarchy. We evaluate our system on several distributed memory platforms, including one chip-based platform (the Cell BE) and a few supercomputing platforms. In this thesis, we focus on two-power input sizes for 1D DFTs, and also generate code for 2D DFTs.

1.4 Thesis Contributions

The following are the main contributions of this thesis:

- A formal, rigorous, mathematical framework and its implementation that rewrites a DFT problem specification into highly optimized code for a target distributed memory architecture. The implementation of this framework enables the computer generation of fast DFT code for a range of existing platforms, and also potentially enables the generation of fast code for future architectures and programming paradigms based on the distributed memory design.

- A set of rewrite rules to formally parallelize and stream code at a high level of abstraction.

- Working platform-tuned libraries for the Cell and for MPI-based supercomputing platforms, all automatically produced by our generator.

- And finally, a deeper understanding of the structure of FFTs needed for the parallelism and memory streaming features found in distributed memory architectures.

1.5 Related Work

We already discussed FFT libraries in Section 1.2.1 and Section 1.2.2. Here, we discuss other related work on automatic performance tuning.

ATLAS. [Whaley et al., 2001] Automatically Tuned Linear Algebra Software (ATLAS) is an offline adaptive library that provides basic linear algebra subroutine (BLAS) functionality. Its approach is based on the observation that the tuning of matrix operations can be accomplished by finding optimal values of certain parameters like loop block sizes and unrolling factors. Benchmarks are executed on the target platform to determine optimal values for these parameters at installation time. The corresponding basic kernel is then generated and used when the library is called for BLAS operations.

LAPACK [Anderson et al., 1999] and the distributed memory extension to it, ScaLAPACK [Blackford et al., 1997], are linear algebra libraries that use an underlying BLAS library such as ATLAS to provide higher level functionality. The basic idea was that ATLAS would update BLAS routines for new platforms, whereas LAPACK would stay unchanged over time. This approach was successful for many years, but could not easily incorporate optimizations for recent architectural features such as SIMD vectorization and multicore parallelization.

Tensor contraction engine (TCE). The TCE compiler [Baumgartner et al., 2005] is an example of a project, that like SPIRAL, uses a domain-specific language to transform a high-level specification of computation to an optimized implementation in Fortran code. It targets tensor contractions used in quantum chemistry. It too performs algebraic transformations to minimize the operations count, and then optimizes code based on abstractions of the target architecture. Like LAPACK, it relies on an external library for performing BLAS operations.

FLAME. Formal Linear Algebra Method Environment [Gunnels et al., 2001; Van Zee et al., 2009] is a project that is similar in principle to SPIRAL, except it targets the development of dense linear algebra libraries. It uses a domain-specific notation for expressing loop-based linear algebra algorithms, which makes systematic derivations for algorithms, and their optimized implementations possible. Recent work allows parallelization for multi-core architectures using OpenMP [Van Zee et al., 2008].

1.6 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 provides background on FFTs, parallelization, and the prior work on automatic program generation using SPIRAL. Chapter 3 discusses the formalism for the parallelization and streaming of tensor products. Parallelization and streaming are the two fundamental architectural paradigms considered in this thesis. Chapter 4 applies this formalism to the derivation of FFTs tuned for our target platforms. Optimizations for both latency and throughput are presented. Chapter 5 presents our program generation system, including ways to produce code for various programming paradigms and platform specific instructions. Chapter 6 presents experimental results that evaluate the performance of our generated DFT code and benchmarks it against existing libraries. Chapter 7 concludes this thesis.

CHAPTER 2

Background

In this chapter, we present background on the discrete Fourier transform and its fast algorithms. We first describe the matrix representation of transforms, and then SPL, a declarative representation for transform algorithms. Then, we present an overview of existing fast algorithms that compute the DFT, including algorithms for parallel and vector machines. Finally, we present SPIRAL, a high-performance automatic program and library generator for linear transforms. The rest of this thesis is built on SPL and the SPIRAL system presented in this chapter.

2.1 Transforms and SPL

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

$$y = Mx,$$

where x and y are the input and output vectors respectively, and M is the fixed transform matrix. A generic matrix-vector multiplication requires $\Theta(n^2)$ arithmetic operations. However, for most transforms used in signal processing and other areas, the structure of the matrix can be exploited to reduce the complexity to $O(n \log n)$ arithmetic operations. This is done by factorizing the transform matrix into a set of structured sparse matrices, and performing a sequence of matrix-vector multiplications on the input vector with each of these sparse matrices.

In this thesis, we use an abstraction called SPL (Signal Processing Language) [Xiong et al., 2001] to represent such factorizations. SPL is a domain specific language derived from matrix algebra and developed to describe transforms and their algorithms. SPL is an extension of the

Kronecker product (or tensor product) formalism described in [Van Loan, 1992], and [Johnson et al., 1990].

SPL as used in this thesis consists of:

- predefined matrices (e.g., F_2),
- structured matrices (e.g., identity, permutation matrices), and
- matrix operators (e.g., product, tensor product).

These are introduced in detail next.

Predefined matrices include F_2 which is also the DFT of size 2 (DFT_2):

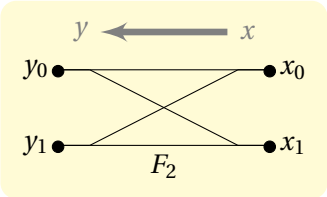
$$F_2 = DFT_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \tag{2.1}$$

and the *diagonal matrix*, which is filled with zeros except for its diagonal elements. For example, $diag(a,b,c,d)$ is a 4×4 diagonal matrix with the diagonal entries a, b, c, and d:

$$\text{diag}(a,b,c,d) = \begin{bmatrix} a & & & \\ & b & & \\ & & c & \\ & & & d \end{bmatrix}.$$

Before we continue, it is important to also view matrices from the dataflow perspective. Visualization is a powerful aid when understanding and developing algorithms, and dataflow graphs can help us intuitively visualize SPL expressions, algorithms, and program loops. As an example, we present the dataflow graph for the transform F_2 as defined in (2.1). The matrix-vector multiplication represented by (2.1) is presented below along with its corresponding dataflow graph. Note that the direction of flow in these dataflow graphs is from right to left, which corresponds with the flow of the input vector through the matrix to the output vector when written in matrix notation:

$$\begin{bmatrix} y_0 \\ y_1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} \tag{2.2}$$



The dataflow graph above expresses that y_0 is computed using x_0 and x_1 (their sum, in this case), and that similarly, y_1 uses x_1 and x_0. The actual addition and subtraction are abstracted away. The shape of the graph explains why F_2 is often called the butterfly matrix. We will use similar dataflow graphs later.

The dataflow graph above will be the general form of dataflow graphs we will use in the remainder of this thesis.

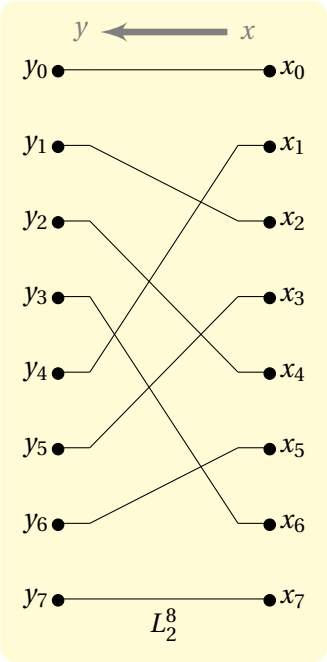
Structured matrices include the identity and permutation matrices. The *identity matrix* I_n is the square $n \times n$ matrix

$$I_n = \begin{bmatrix} 1 & & \\ & \ddots & \\ & & 1 \end{bmatrix}.$$

Permutation matrices have exactly one single 1 in each row or column, and hence permute the input vector. We only consider permutation matrices of the form L^{mn}_m, which transpose an input vector of size mn, when viewed as a $m \times k$ matrix stored in row-major order. This class of permutations is called "stride permutations" since the output of the permutation is equivalent to reading (or writing) the input (or output) at a stride. For example,

$$L^8_2 = \begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 \end{bmatrix},$$

and $y = L^8_2$ has the dataflow



The two **matrix operators** we will consider in this thesis are the product and the tensor product. The *product* is simply defined as $A \cdot B = AB$. This is equivalent to multiplying the input first with B, and the resulting vector by A:

$$(A \cdot B)x = A \cdot (Bx).$$

The *tensor (or Kronecker) product*, which is fundamental to the representation of fast Fourier transforms and to the approach used in this thesis is defined as:

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

We present a few special cases of tensor products that arise when one of the two matrices is the identity matrix. These are fundamental to this thesis for two important reasons: a) as we will see later, fast algorithms for the DFT consist mainly of such tensor products, and b) these tensor products expose the loop structure of such algorithms in a manner that allows us to map them to architectural features of the target hardware (e.g., distributed memory parallelism). We consider four cases below which differ only in the stride at which their inputs and outputs are accessed, as shown in Table 2.1. We present the four cases using both the matrix representations and as corresponding dataflow graphs:

- $I_n \otimes A_m$ is a block-diagonal matrix, where the transform A_m is applied n times to consec-

SPL Construct	Output stride	Input stride
$y = (I_n \otimes A_m)x$	1	1
$y = (A_m \otimes I_n)x$	n	n
$y = (I_n \otimes A_m)L^{mn}_n x$	1	n
$y = L^{mn}_m (I_n \otimes A_m)x$	n	1

TABLE 2.1: Basic SPL constructs. Strides refer to inputs and outputs of each of the A_m matrices.

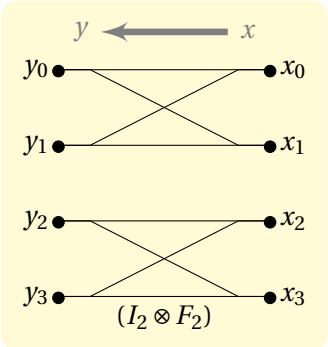
utive blocks (of size m) of the input vector:

$$I_n \otimes A = \begin{bmatrix} A_m & & & \\ & A_m & & \\ & & \ddots & \\ & & & A_m \end{bmatrix}$$

For example,

$$I_2 \otimes \mathrm{F}_2 = \begin{bmatrix} 1 & 1 & & \\ 1 & -1 & & \\ & & 1 & 1 \\ & & 1 & -1 \end{bmatrix},$$

and the corresponding dataflow graph of $y = (I_2 \otimes \mathrm{F}_2)$ is



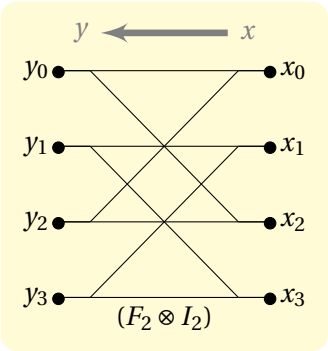
- $A_m \otimes I_n$ also involves applying A_m n times to the input vector, but it is applied to elements read and written at a stride of n (as also indicated in Table 2.1):

$$A \otimes I_n = \begin{bmatrix} a_{0,0} I_n & \dots & a_{0,m-1} I_n \\ \dots & \ddots & \dots \\ a_{m-1,0} I_n & \dots & a_{m-1,m-1} I_n \end{bmatrix}, A = [a_{k\ell}],$$

For example,

$$\mathrm{F}_2 \otimes I_2 = \begin{bmatrix} 1 & & 1 & \\ & 1 & & 1 \\ 1 & & -1 & \\ & 1 & & -1 \end{bmatrix},$$

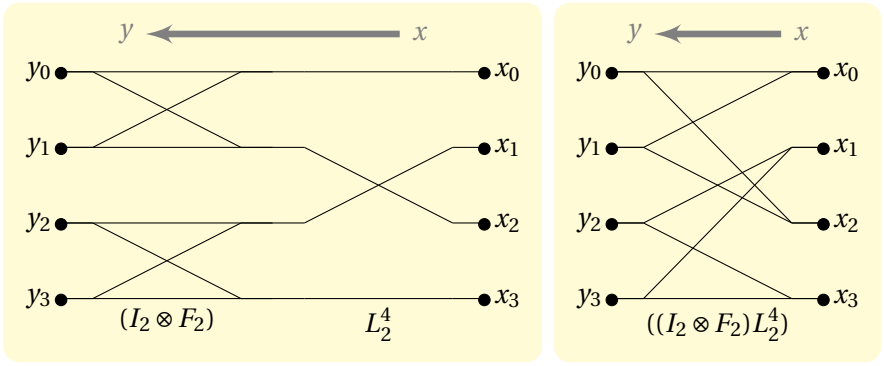
and the corresponding dataflow graph of $y = (\mathrm{F}_2 \otimes I_2)$ is



- $(I_n \otimes A_m) L^{nm}_n$ is a product of the first case and a permutation. This transform can be executed in two steps, with the permutation being applied first to the input vector followed by the $I_n \otimes A_m$ operation. However, to yield better performance, the whole transform can be applied in a single step, with the input vector being read at a stride of n, and the output vector being written at the regular unit stride. This makes it a combination of the two previous cases, as Table 2.1 shows. The example below illustrates both these methods:

$$(I_2 \otimes \mathrm{F}_2) L^4_2 = \begin{bmatrix} 1 & 1 & & \\ 1 & -1 & & \\ & & 1 & 1 \\ & & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 & & & \\ & & 1 & \\ & 1 & & \\ & & & 1 \end{bmatrix} = \begin{bmatrix} 1 & & 1 & \\ 1 & & -1 & \\ & 1 & & 1 \\ & 1 & & -1 \end{bmatrix}.$$

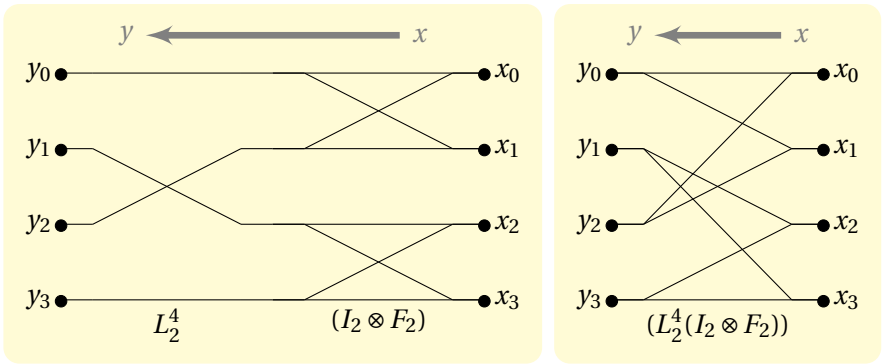
The two dataflow graphs corresponding to the first and the last matrices above are:



- $L^{nm}_m(I_n \otimes A_m)$ is a product of the second case and a permutation. It is a transposed version of the previous case. The input is read at a unit stride while the output is written at a stride of n. Below, we show an example of performing this transform both using two steps, and using a single step:

$$L^4_2(I_2 \otimes \mathrm{F}_2) = \begin{bmatrix} 1 & & & \\ & & 1 & \\ & 1 & & \\ & & & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & & \\ 1 & -1 & & \\ & & 1 & 1 \\ & & 1 & -1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & & \\ & & 1 & 1 \\ 1 & -1 & & \\ & & 1 & -1 \end{bmatrix}.$$

The two dataflow graphs corresponding to the first and the last matrices above are:



SPL expressions. SPL expressions or formulas used in this thesis follow the grammar shown in Table 2.2.

⟨formula⟩	::=	⟨matrix⟩ | ⟨transform⟩ |	
		⟨formula⟩⟨formula⟩ |	*product*
		⟨formula⟩ ⊗ ⟨formula⟩	*tensor product*
⟨matrix⟩	::=	$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \mid I_n \mid L^n_k \mid \ldots$	
⟨transform⟩	::=	$\mathrm{DFT}_n \mid \ldots$	

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

$$(BC)^\top = C^\top B^\top \tag{2.3}$$
$$(A \otimes B)^\top = A^\top \otimes B^\top \tag{2.4}$$
$$I_{mn} = I_m \otimes I_n \tag{2.5}$$
$$A \otimes B = (A \otimes I_m)(I_n \otimes B) \tag{2.6}$$
$$A \otimes (BC) = (A \otimes B)(A \otimes C) \tag{2.7}$$
$$A \otimes B = L^{mn}_n (B \otimes A) L^{mn}_m \tag{2.8}$$
$$\left(L^{mn}_m\right)^{-1} = L^{mn}_n \tag{2.9}$$
$$L^{kmn}_n = (L^{kn}_n \otimes I_m)(I_k \otimes L^{mn}_n) \tag{2.10}$$
$$L^{kmn}_{km} = (I_k \otimes L^{mn}_m)(L^{kn}_k \otimes I_m) \tag{2.11}$$

TABLE 2.3: SPL formula identities to manipulate SPL constructs. A is $n \times n$, and B and C are $m \times m$. $A^\top$ is the transpose of A.

2.1.1 SPL Identities

SPL expressions can be manipulated using mathematical identities. We present these identities in Table 2.3. We will use these and develop others in this thesis to manipulate fast algorithms expressed in SPL to better "fit" target architectures.

This section has introduced the matrix representation of transforms, and SPL. Next, we show how SPL is used to represent fast algorithms for the DFT.

2.2 Discrete and Fast Fourier Transforms

In this section, we first define the DFT and its most important fast algorithm: the general-radix Cooley-Tukey fast Fourier transform (FFT). Then we discuss variants of the DFT, and put terminology used in the literature in the context of this thesis.

DFT definition. The DFT of n input samples $x_0, \ldots, x_{n-1}$ is defined in summation form as

$$y_k = \sum_{0 \le \ell < n} \omega_n^{k\ell} x_\ell, \quad 0 \le k < n,$$

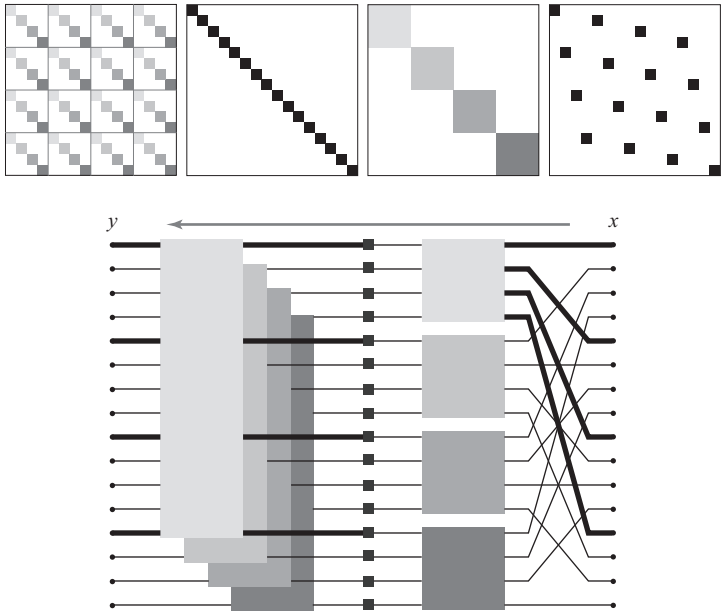


FIGURE 2.1: Cooley-Tukey FFT: matrix structure, dataflow visualization from [Franchetti et al., 2009b]. Each point on the dataflow visualization is a complex element. The dataflow visualization contains 4 stages, which correspond to the stages shown in the matrix structure visualization.

with $\omega_n = e^{-2\pi j/n}$ (a primitive n^{th} root of unity). In the matrix form, the DFT is written as:

$$y = \text{DFT}_n\, x, \quad \text{DFT}_n = [\omega_n^{k\ell}]_{0\le k,\ell<n}.$$

Cooley-Tukey FFT. The first "fast" algorithm for the DFT was discovered by Cooley and Tukey [Cooley and Tukey, 1965] [1], and is often referred to as "the" FFT [2]. It is expressed in SPL as:

$$\text{DFT}_{mn} = (\text{DFT}_m \otimes I_n) D_{n,m} (I_m \otimes \text{DFT}_n) L^{mn}_m, \tag{2.12}$$

where $D_{n,m}$ is the diagonal matrix containing the twiddle factors, defined in [Van Loan, 1992]. To aid an intuitive understanding, we show in Figure 2.1 the matrix and dataflow representations of a DFT of size 16 (16 input samples) performed using the FFT in (2.12) with $m = n = 4$.

Note that the FFT algorithm in (2.12) is recursive: it factorizes the DFT into a product of factors that contain smaller DFTs. For n a power of two, the base case is $\text{DFT}_2 = \text{F}_2$, defined in (2.2).

[1] The algorithm was already known around 1805 to Carl Friedrich Gauss, but his work was not widely recognized [Heideman et al., 1985].

[2] It is important to note the distinction between the terms "DFT" and "FFT." The former refers to the problem, while the latter refers to an algorithm. Literature often fails to make this distinction, which can be a source of confusion.

Several other FFTs exist. These include the Bluestein or Winograd FFT (for any input size n including non-2 power sizes), the Rader FFT (for prime n), and the prime-factor FFT (used when n is a product of coprime factors) [Van Loan, 1992]. These algorithms are typically used for small sizes (≤ 32) as the base-case for the FFT recursion, and are not relevant to this thesis. In this thesis, we focus on two-powers $n = 2^k$ and variants of the Cooley-Tukey FFT that work especially well with our target architectures.

DFTs of higher dimensions, especially 2 and 3-dimensions, are also used widely, and are naturally expressed within the Kronecker product formalism simply as the tensor product of two or more 1D DFTs:

$$\mathrm{DFT}_{m\times n} = \mathrm{DFT}_m \otimes \mathrm{DFT}_n, \tag{2.13}$$

$$\mathrm{DFT}_{k\times m\times n} = \mathrm{DFT}_k \otimes \mathrm{DFT}_m \otimes \mathrm{DFT}_n\,. \tag{2.14}$$

The row-column algorithm for breaking down multidimensional DFTs factorizes the tensor product of DFTs into tensor products of DFTs with identity matrices. For example,

$$\mathrm{DFT}_{m\times n} = \mathrm{DFT}_m \otimes \mathrm{DFT}_n = (\mathrm{DFT}_m \otimes I_n)(I_m \otimes \mathrm{DFT}_n). \tag{2.15}$$

Variants. In practice, applications may require *variants* of the DFT. Most of these are similar to the standard DFT, and hence, fast code for the standard DFT can usually be adapted with minor modifications. We present the most important variants here:

- **Forward/inverse.** A forward DFT goes converts the input in time domain to frequency domain, and vice versa. In terms of computation, the inverse DFT (with output scaled by $1/N$ where N is the problem size) is the same as the forward DFT with the exception of the twiddle factors used. Performance is typically the same or very close for the forward and inverse DFTs as the number of operations and the access patterns remain the same. Thus, program generation systems or DFT libraries can simply switch twiddle values to reverse direction. The performance numbers presented in this thesis apply to both forward and inverse DFTs.

- **Complex/real-input data.** The input to the DFT is a vector of complex numbers. Specialized algorithms exist to optimize DFTs for real input (where the imaginary parts are all 0), but are outside the scope of this thesis.

- **Interleaved/split-complex format.** Two major data formats exist for representing complex arrays. Arrays in the interleaved-complex format are comprised of alternating real

and imaginary values, where a complex number is represented by two successive values in the array. Arrays in the split-complex format are comprised of all real values grouped together, followed by the imaginary values. By default, we assume input/output using the interleaved-complex format. Split-complex format is handled by placing appropriate stride permutations at the beginning and the end of the transform, and pushing these permutations into inner loops to mitigate or avoid the cost of performing them explicitly.

- **Bit-reversed output.** When the output of a DFT is said to be "bit-reversed", it means the output array is permuted in such a fashion that their index bits are reversed (hence the term "bit-reversal permutation"). In some applications like a DFT-based convolution, when performing a forward DFT, followed by a set of operations, followed by an inverse DFT, the order of the output either does not matter, or the intervening operations can be modified at no cost to handle bit-reversed output. A permutation step may be saved if bit-reversed output is acceptable. This may provide significant runtime savings, especially in distributed memory systems where inter-node permutations are expensive. It is important to keep this in mind when comparing performance results across DFT libraries.

Relevant terms used in literature. The DFT-related terminology in literature can sometimes be ambiguous. We list relevant terminology here as a reference and to relate our presentation with the existing literature.

- **Decimation in time / decimation in frequency.** The DFT is symmetric: $\mathrm{DFT}_n{}^T = \mathrm{DFT}_n$. (2.12) is called the recursive *decimation-in-time FFT*, while its transpose (using (2.3),(2.4)) is called the recursive *decimation-in-frequency FFT*, and is shown below:

$$\mathrm{DFT}_{mn} = L^{mn}_n (I_m \otimes \mathrm{DFT}_n) D_{n,m} (\mathrm{DFT}_m \otimes I_n),$$

- **Radices, radix-r FFT.** The value of n in (2.12) is the radix of the Cooley-Tukey decomposition for that level. If the same radix is chosen recursively, the resulting algorithm is called a radix-r algorithm (necessarily, $n = r^t$ in this case). Mixed-radix algorithms are used for sizes that are not powers of a suitable radix or for performance reasons.

- **Corner turns, matrix transpositions.** A corner-turn (named so because it "turns" the matrix around a corner) or a matrix transposition is simply a stride permutation as defined in Section 2.1.

- **Transposed output.** This usually means bit-reversed output, as explained above.

2.3 FFTs for Parallel Architectures

So far, we have introduced the Cooley-Tukey FFT in SPL. In this section, we present other FFTs that were designed for parallel architectures. All these FFTs can be derived from (2.12).

Parallelism in platforms: a historical view. High-performance DFT libraries have long been used on a wide range of platforms. In the past, platforms could be broadly categorized into desktop and supercomputing platforms. Desktop platforms in the past had minimal or no support for parallelism (1970–1996) after which they were restricted to short-vector parallelism (1996–2004). Performance bottlenecks for the DFT on these platforms was primarily rooted in the memory hierarchy. However, the latest desktop architectures are based on cache-coherent multicore designs (e.g., Intel Quad-core, and AMD Opteron, Phenom). Since we build upon some ideas used on parallel DFTs on these machines, we present them here.

Supercomputing platforms have included various forms of parallelism including long-vector instructions and multi-node parallelism. Since these platforms most closely resemble the distributed memory parallel architectures this thesis considers, we look closely at FFTs that have been used on them in the past.

Below, we present several types of historical and existing parallel DFT algorithms, including their strengths and weaknesses.

Four-step FFT. The four-step algorithm [Hegland, 1994; Norton and Silberger, 1987; Van Loan, 1992] was developed for supercomputers which provided parallelism in the form of long vector instructions. It produces the longest possible unit stride vector operations at the cost of a transposition. It is a simple manipulation of the general Cooley-Tukey FFT (2.12) using (2.8):

$$\mathrm{DFT}_{mn} = (\mathrm{DFT}_m \otimes I_n) D_{n,m} L^{mn}_m (\mathrm{DFT}_n \otimes I_m). \tag{2.16}$$

The algorithm is named so because of the four factors seen in (2.16). This algorithm is primarily used on vector architectures, but is inefficient on multicore or multi-node architectures.

Six-step FFT. The six-step algorithm was developed for distributed memory machines. It is still used on such machines along with variants. The six-step is again a simple modification of the general Cooley-Tukey FFT (2.12) using (2.8) that yields six factors, three of which are global transpositions (which become global all-to-all data exchanges):

$$\mathrm{DFT}_{mn} = L^{mn}_m (I_n \otimes \mathrm{DFT}_m) L^{mn}_n D_{n,m} (I_m \otimes \mathrm{DFT}_n) L^{mn}_m. \tag{2.17}$$

The main issue with the six-step algorithm is the high implementation cost of the permutations. The algorithm is used, for example, in the FFTE library [Takahashi, 2002].

Iterative Algorithm: triple-loop. There also exist iterative algorithms for the DFT which can be derived by recursively expanding the Cooley-Tukey FFT and applying identities from Table 2.3. The simplest iterative FFT is the decimation in time *triple-loop* [Cooley and Tukey, 1965; Van Loan, 1992]:

$$\mathrm{DFT}_{r^k} = R_r^{r^k} \prod_{i=0}^{k-1} D_i^{r^k} \left(I_{r^{k-i-1}} \otimes \mathrm{DFT}_r \otimes I_{r^i}\right). \tag{2.18}$$

$R_r^{r^k}$ is the radix-r digit reversal permutation and $D_i^{r^k}$ contains the twiddle factors in the ith stage. A radix-2 version is implemented by Numerical Recipes [Press et al., 1992].

Iterative algorithm: Pease. A variant of (2.18) is the *Pease* FFT [Pease, 1968; Van Loan, 1992], which has constant geometry, i.e., the control flow is independent of the stage. It requires an explicit digit reversal permutation at the end, which may be expensive to implement. It was originally developed for parallel computers, and its regular structure makes it a good choice for field-programmable gate arrays (FPGAs) or ASICs:

Iterative algorithm: Stockham. The *Stockham* FFT [Schwarztrauber, 1987; Van Loan, 1992] is a self-sorting algorithm which means that it does not require a digit reversal permutation at the end. It was also originally developed for vector computers:

$$\mathrm{DFT}_{r^k} = \prod_{i=0}^{k-1} \left(\mathrm{DFT}_r \otimes I_{r^{k-1}}\right) D_i^{r^k} \left(L_r^{r^{k-i}} \otimes I_{r^i}\right).$$

Distributed memory FFT. Bonelli et al. [2006] derive an FFT for a distributed memory platform with p processors. First the stride permutation in (2.12) is factored using the following identity:

$$L_m^{mn} \rightarrow (I_p \otimes L_{m/p}^{mn/p})(L_p^{p^2} \otimes I_{mn/p^2})(I_p \otimes L_p^n \otimes I_{m/p}).$$

Then, (2.12) is manipulated to yield:

$$\begin{aligned}
\mathrm{DFT}_{mn} \rightarrow\ & L_m^{mn}(I_n \otimes \mathrm{DFT}_m) L_n^{mn} D_{n,m} (I_m \otimes \mathrm{DFT}_n) L_n^{mn} \\
\rightarrow\ & (I_p \otimes L_{m/p}^{mn/p})(L_p^{p^2} \otimes I_{mn/p^2}) \\
& (I_p \otimes L_p^n \otimes I_{m/p})(I_p \otimes I_{n/p} \otimes \mathrm{DFT}_m)(I_p \otimes L_{n/p}^{mn/p})(L_p^{p^2} \otimes I_{mn/p^2}) \\
& (I_p \otimes L_p^m \otimes I_{n/p}) D_{m,n} (I_p \otimes I_{m/p} \otimes \mathrm{DFT}_n)(I_p \otimes L_{m/p}^{mn/p})(L_p^{p^2} \otimes I_{mn/p^2}) \\
& (I_p \otimes L_p^n \otimes I_{m/p})
\end{aligned}$$

The resulting algorithm has four "macro-stages" (shown one per line), mutually separated by

three all to all data exchanges ($L^{p^2}_p \otimes I_{mn/p^2}$). The algorithm was intended primarily for MPI-based clusters, and is unable to take advantage of architectures that can reduce data transfer costs by performing communications in the background. It also does not allow for SIMD vectorization in its original form.

Multicore FFT. Franchetti et al. [2006a] derive an FFT for a shared memory multicore platform with p processors and cache block size μ:

$$\begin{aligned}\mathrm{DFT}_{mn} = \big((L^{mp}_m \otimes I_{n/p\mu}) \otimes I_\mu\big)\big(I_p \otimes (\mathrm{DFT}_m \otimes I_{n/p})\big)\big((L^{mp}_p \otimes I_{n/p\mu}) \otimes I_\mu\big) D_{n,m} \\ \times \big(I_p \otimes (I_{m/p} \otimes \mathrm{DFT}_n) L^{mn/p}_{m/p}\big)\big((L^{pn}_p \otimes I_{m/p\mu}) \otimes I_\mu\big). \quad (2.19)\end{aligned}$$

Inspection of (2.19) shows that all permutations have the form ($A \otimes I_\mu$ which always ensures that cache lines are accessed in blocks of size μ, and avoids false sharing. Further, all computation steps have the form $I_p \otimes A$, which ensures that these are load balanced.

In this section, we have seen FFTs for parallel architectures at a mathematical level of abstraction. We now present SPIRAL, a system that can generate high performance code based on such abstractions.

2.4 SPIRAL

Overview. We presented several fast algorithms for the DFT based on the FFT. We now discuss SPIRAL, a system that was developed to reap the practical benefits of algorithm exploration. This thesis builds upon and extends SPIRAL to generate fast code.

SPIRAL [Püschel et al., 2005] is a program generator that autonomously generates a high-performance software library for some selected functionality and target platform. SPIRAL originally focused on linear transforms, and in particular the DFT. However, latest developments expand SPIRAL beyond this domain to include libraries for coding, linear algebra (matrix-matrix-multiplication), and image processing [de Mesmay et al., 2010; Franchetti et al., 2008]. While SPIRAL still only supports restricted domains, the generated code is very fast and often rivals the performance of expertly hand-tuned implementations. We limit the further discussion on SPIRAL to the DFT.

At the heart of SPIRAL is the declarative representation of algorithms—SPL (discussed earlier), symbolic computation (rewriting systems), and high-level architecture models. SPIRAL represents algorithms and hardware models in SPL An intelligent search mechanism accounts for hardware details that are difficult to model (such as cache-based systems). Algorithms and program transformations are encoded as rewriting rules that operate on SPL formulas to produce

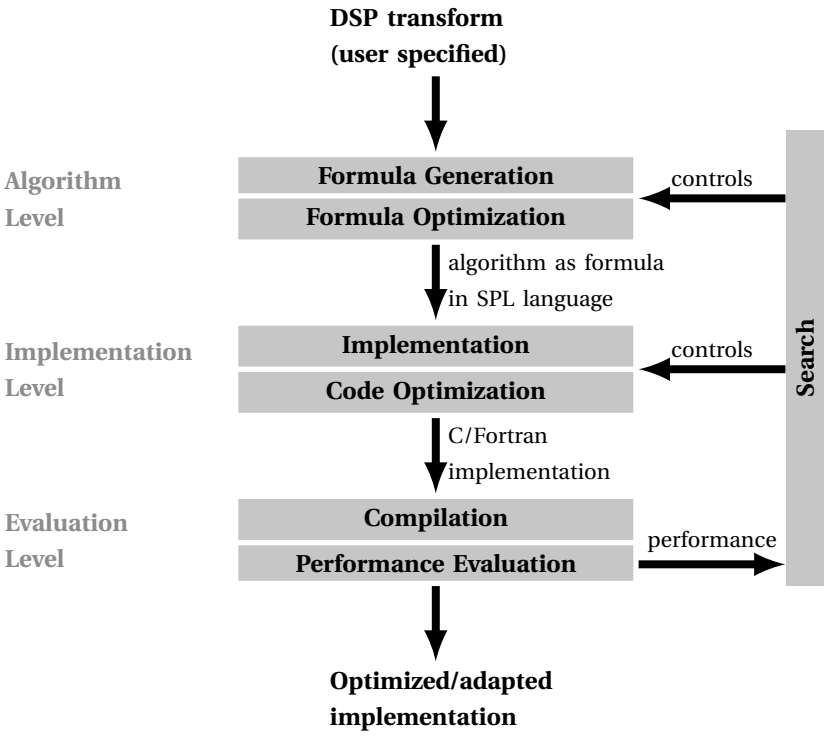


FIGURE 2.2: SPIRAL's program generation system.

code.

2.4.1 System Structure and Sequential Code Generation

SPIRAL's structure is shown in Figure 2.2, which we discuss in detail below.

Input. The input to SPIRAL is a problem specification containing the requested functionality and some target hardware properties, for instance "generate a 8,192-point DFT using 4 Cell SPEs," appropriately encoded in SPIRAL's internal language.

Algorithm Level. *Breakdown rules* in SPIRAL are used to represent algorithms for the DFT that break down larger transforms into smaller kernels. These smaller kernels are recursively broken down into even smaller kernels using breakdown rules at each level, until the recursion's base cases are reached. The final algorithm typically involves various DFT algorithms applied at each level of the recursion. This also means that a large space of algorithms (formulas) for a single transform may be obtained using these breakdown rules. Next, a rewriting system uses a set of *rewrite rules* to structurally optimize the formula thus obtained to match the target architecture.

Implementation Level. Algorithms in SPL can be directly translated into sequential C code,

discussed in more detail in Section 2.4.3. At the implementation level, an *SPL compiler* translates the formula output into optimized C code, possibly including intrinsics (for vectorized code) [Franchetti et al., 2006b] and threading instructions (for multicore parallelism) [Franchetti et al., 2006a].

Evaluation Level. The C code thus obtained is compiled by a traditional backend compiler (such as `gcc`). The performance of this implementation is measured and used in a feedback loop to search over algorithmic alternatives for the fastest one. Searching over an algorithm space is primarily useful for architectural components such as the memory hierarchy which are difficult to model.

Output. The final output is a high performance platform adapted implementation of the transform requested by the user.

2.4.2 Parallel and Vector Code Generation

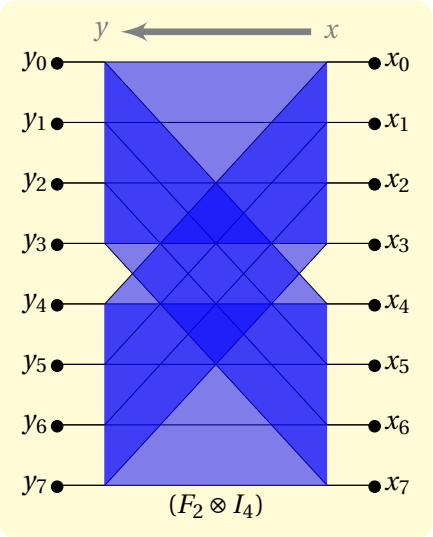
A key observation is that SPL's tensor product representation of the transform algorithms can be mapped to components of the target architecture. This is because the tensor product can be viewed as a program loop, where the structure of the loop is made explicit. Using well known formula identities recast as rewriting rules, SPIRAL manipulates algorithms in SPL to match the target architecture [Franchetti et al., 2006a,b], thus deriving high-performance implementations of the loop. We now discuss code generation for parallel and vector architectures in further detail.

Architectural paradigms. Target architectural features like multicore parallelism and SIMD vector units that require optimization of the algorithm structure are called architectural *paradigms* and are first identified. The main idea then is to find SPL expressions that map naturally well onto each of the paradigms of the target architecture. Thus, these formula constructs represent a high-level model of the target architecture. Remaining SPL expressions contained in the DFT are converted into these natural constructs using rewrite rules that must be identified. The conversion typically comes at a cost, which might be vector permutations, synchronization barriers, or additional index computations. Below, we look at two examples of SPL expressions that map well to architectural paradigms: SIMD vectorization and multicore parallelization.

SIMD vectorization. Single Instruction Multiple Data architectures can perform the same instruction on multiple data input in the same cycle. The main constraint in such architectures is that SIMD instructions are setup to work only on data that is consecutive in memory. Due to this constraint, the most natural fit for vectorization is the $A_m \otimes I_n$ construct which was presented in Section 2.1, assuming $n \geq \nu$ and ν divides n (where ν is the SIMD vector length). The pseudocode and dataflow graph below illustrate why this formula construct naturally fits the paradigm. The highlighted area in the dataflow graph represents a single 4-way vector instruc-

tion. The values x_0, x_1, x_2, x_3 are loaded in a vector register using a single load instruction. The values x_4, x_5, x_6, x_7 are loaded in a second register, and a single SIMD add instruction on these registers yields 4 output data points (y_0, y_1, y_2, y_3):

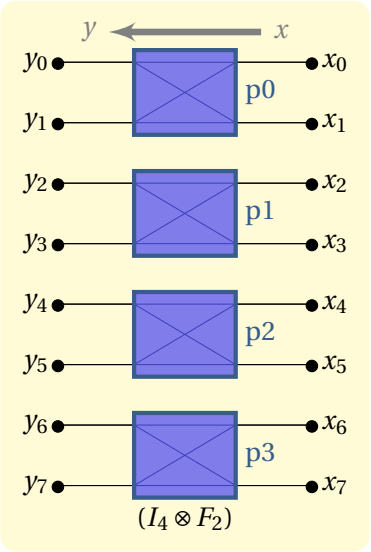
```
vec(y0, y1, y2, y3) = vec(x0, x1, x2, x3) + vec(x4, x5, x6, x7);
vec(y4, y5, y6, y7) = vec(x0, x1, x2, x3) - vec(x4, x5, x6, x7);
```



Other SPL constructs are rewritten to fit this construct. For example, $I_n \otimes A_m$ can be rewritten using (2.8). The cost of rewriting are the additional permutations which are implemented as a combination of vector-block-level re-indexing, and in-vector permutations [Franchetti et al., 2006b].

Multicore parallelism. Since this involves simultaneously performing computations on more than one processor core, the dataflow graph must be partitioned to allow for this. Any loop where iterations are completely independent of each other is a data-parallel loop that fits this paradigm. The most natural SPL construct that does this is the $I_n \otimes A_m$ construct which was presented in Section 2.1. Each core of p cores works on n/p iterations independent of other cores. The dataflow graph below shows an example of $I_4 \otimes \mathrm{F}_2$ parallelized across 4 processors, as highlighted by the boxes. The graph implements the following pseudocode, with each line executing on a different processor:

```
y0 = x0 + x1; y1 = x0 - x1; // Executes on p0
y2 = x2 + x3; y3 = x2 - x3; // Executes on p1
y4 = x4 + x5; y5 = x4 - x5; // Executes on p2
y6 = x6 + x7; y7 = x6 - x7; // Executes on p3
```



Other SPL constructs are rewritten to fit this construct. For example, $A_m \otimes I_n$ can be rewritten using (2.8). The cost is the set of additional permutations, which are implemented as re-indexing inside the parallel loop.

In reality, rewriting for parallelization is more involved because of cache memory considerations. False sharing would occur if data within the same cache line was processed by different cores. Thus, the high-level rewriting must account for this, as presented in detail in [Franchetti et al., 2006a]. This last observation illustrates an important point: rewriting for one architectural paradigm may produce a conflict for another paradigm on the same target architecture. We will see this phenomenon reoccur at various points in this thesis.

Rewriting for Target Architectures. A platform may have multiple architectural paradigms that our algorithm needs to take advantage of. For example, for a multicore Intel CPU like the Xeon Quadcore, the algorithm must be both parallelized and vectorized. Adapting the SPL formula to the target architecture is accomplished using a tagging system within SPIRAL. The input to SPIRAL discussed in Section 2.4.1 can be tagged with architectural parameters such as parallelization and vectorization (tags can be nested when required). Tags are removed as SPIRAL successively uses rewrite rules (and breakdown rules, as discussed in Section 2.4.1) to modify tagged SPL constructs into combinations of constructs that naturally fit the architecture, and additional constructs which are the cost of such conversions. At this point, the formula is ready to be converted into code. Below, we show an example of a rewrite rule that rewrites a tagged

SPL construct into its untagged form:

$$\underbrace{I_n \otimes A_m}_{\text{par}(p,\mu)} \rightarrow I_p \otimes_{\parallel} I_{n/p} \otimes A_m \tag{2.20}$$

A construct of the form $I_p \otimes_{\parallel} A$ specifies that A is executed in parallel across multiple processors. The actual implementation is not specified at this level, and is accomplished using a programming paradigm that is best suited to the target architecture (e.g., pthreads or OpenMP).

2.4.3 Looped Code Generation and Σ-SPL

So far, we have seen how SPIRAL generates simple, sequential code for uniprocessors and for vector and parallel processors. However, these techniques do not scale to larger transform sizes, for which looped code must be generated. The main idea in generating high performance FFT looped code is to fuse permutation constructs into adjoining kernel constructs so as to avoid explicit copying steps in the final code, which are expensive. However, this is difficult to accomplish using SPL. Specifically, this is because SPL does not distinguish between kernel constructs that perform computations, and permutation constructs that perform only memory loads and stores.

Σ-SPL, a system for formal loop merging, was developed to address this issue. Σ-SPL [Franchetti et al., 2005] is an extension of SPL that makes loops, and memory accesses as function of loop variables explicit, while retaining the matrix abstraction. It can be seen as an abstraction layer between SPL and actual C code. Σ-SPL adds the iterative sum of matrices and matrices parameterized by index mapping functions (gather and scatter matrices) to SPL. Thus, Σ-SPL is made up of constructs that explicitly represent compute kernels (called skeletons), and constructs that represent their associated loads and stores in the form of gather and scatter matrices (called decorations). This allows it to concisely describe loop-carried access patterns inside a matrix-based algorithm representation.

In addition to using it to produce looped code, in this thesis, we also extend Σ-SPL to expose and manipulate details about loads and stores in order to incorporate explicit memory access management. We first provide an intuitive introduction to Σ-SPL, followed by a formal definition. As an illustrating example, consider the transform $I_2 \otimes A_2$ for an arbitrary 2×2 matrix A,

that operates on an input vector of length 4. This construct can be written as:

$$I_2 \otimes A = \begin{bmatrix} A & \\ & A \end{bmatrix} = \begin{bmatrix} 1 & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot \end{bmatrix} A \begin{bmatrix} 1 & \cdot \\ \cdot & 1 \\ \cdot & \cdot \\ \cdot & \cdot \end{bmatrix} + \begin{bmatrix} \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & 1 \end{bmatrix} A \begin{bmatrix} \cdot & \cdot \\ \cdot & \cdot \\ 1 & \cdot \\ \cdot & 1 \end{bmatrix} = \sum_{i=0}^{1} \mathrm{S}_i \, A_m \, \mathrm{G}_i, \tag{2.21}$$

where the dots represent zero entries. In each of the summands, the two vertically long matrices, called the gather matrices, select the elements of the input vector that A_m works on, and the horizontally long matrices, called the scatter matrices, select the part of the output vector the summand writes to, and can thus be parameterized as shown in (2.21).

More formally, Σ-SPL defines matrices parameterized by index mapping functions, which map integer intervals to integer intervals. An index mapping function f with domain $\{0,\ldots,n-1\}$ and range $\{0,\ldots,N-1\}$ is written as $f^{n\to N} : i \mapsto f(i)$. For convenience, we omit the domain and range where it is clear from the context. We now introduce the two index mapping functions used in this thesis,

$$h_{b,s}^{n\to N} :\ i \mapsto b + is, \quad \text{and} \tag{2.22}$$

$$q_{b,s,\mu}^{n\to N} :\ i \mapsto (b + \lfloor i/\mu \rfloor\, s)\mu + (i \bmod \mu) \tag{2.23}$$

which abstract strided memory access at scalar and packet granularities, respectively (b, s, and μ correspond to base, stride, and packet size). Index mapping functions are used to parameterize gather and scatter matrices, which encode data movement (loads and stores). Let $e_k^n \in \mathbb{C}^{n\times 1}$ be the column basis vector with the 1 in k-th position and 0 elsewhere. The gather matrix for the index mapping $f^{n\to N}$ is

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

Gather matrices thus gather n elements from an array of N elements. Scatter matrices are simply transposed gather matrices, $\mathrm{S}(f^{n\to N}) = \mathrm{G}(f)^\top$.

The iterative matrix sum, which encodes loops, is conventionally defined. However, it does not incur operations since by design, each of the summands produce a unique part of the output vector. Based on our formal definitions, our previous example (2.21) is expressed in Σ-SPL using index mapping functions to parameterize gathers and scatters as $\sum_{i=0}^{1} \mathrm{S}(h_{2i,1}) A_m \mathrm{G}(h_{2i,1})$.

Code Generation. Since Σ-SPL formulas essentially represent loops, converting them into C code is straightforward. Gather matrices read from the input vector, which the kernel then

SPL	Σ-SPL	Code
$y = (A_n B_n)x$	$y = (A_n B_n)x$	`t[0:1:n-1] = B(x[0:1:n-1]);` `y[0:1:n-1] = A(t[0:1:n-1];)`

$y = (I_n \otimes A_m)x$	$y = \left(\sum_{j=0}^{n-1} \mathrm{S}(h_{jm,1}) A_m \mathrm{G}(h_{jm,1})\right)x$	`for(i=0;i<n;i++)`
		`  y[i*n:1:i*m+m-1] =`
		`  A(x[i*n:1:i*m+m-1]);`

$y = (A_m \otimes I_n)x$	$y = \left(\sum_{j=0}^{n-1} \mathrm{S}(h_{j,n}) A_m \mathrm{G}(h_{j,n})\right)x$	`for(i=0;i<n;i++)`
		`  y[i:n:i+m-1] =`
		`  A(x[i:n:i+m-1]);`

$y = (I_n \otimes A_m)L_n^{mn}x$	$y = \left(\sum_{j=0}^{n-1} \mathrm{S}(h_{jm,1}) A_m \mathrm{G}(h_{j,n})\right)x$	`for(i=0;i<n;i++)`
		`  y[i*n:1:i*m+m-1] =`
		`  A(x[i:n:i+m-1]);`

TABLE 2.4: Translating SPL to Σ-SPL, and then to code. x denotes the input and y the output vector. The subscripts of A and B specify the size of the (square) matrix. We use MATLAB-like notation: `x[b:s:e]` denotes the subvector of x starting at `b`, ending at `e` and extracted at stride `s`.

operates on. Scatter matrices then write to the output vector. The matrix sum becomes the actual loop. Table 2.4 shows how the fundamental Σ-SPL constructs are derived from SPL constructs, and then translated to C code. By applying these rules recursively, any Σ-SPL formula can be translated into C code.

2.4.4 General-size Library Code Generation: Autolib

So far, we have looked at automatic generation of code for fixed sizes only (the transform size is fixed at problem specification time). Library generation for general-size code, called "Autolib", is a new addition to SPIRAL that enables the automatic generation of transforms where the size parameter is known only at runtime [Voronenko, 2008; Voronenko et al., 2009].

Autolib can generate libraries similar to FFTW [Frigo and Johnson, 2005]. The main advantage of such a system is, it obviates the need to redesign or rewrite a library when major changes need to be made. For instance, when a target architecture's set of paradigms changes, or when a new paradigm needs to be targeted, or when a new algorithm needs to be added, one can simply make high-level changes to the library generator, and use it to produce a new library.

In addition, an automatic library generation system offers several other benefits. Appropriate libraries can be easily be generated for different platforms that use similar architectural paradigms but need different sets of instructions (e.g., Intel's SSE vector instruction intrinsics are slightly different from IBM's AltiVec instruction set, although both are SIMD vector paradigms). Autolib can also generate smaller libraries for platforms where code size limitations exist (trad-

ing off performance). It can also generate a single library for a given set of transforms (e.g., the discrete Fourier, Sine, and Cosine transforms), while reusing code across transforms. We will limit further discussion of Autolib to the aspects pertinent to this thesis.

Design. Autolib-generated libraries are composed of recursive functions that call themselves and each other in a recursive closure. Such a mutually recursive closure derives from the recursive nature of the transform algorithms. A major component of Autolib is to automatically determine this recursion step closure from a given set of transforms, algorithms, and rules.

A significant difference between fixed-size program generation and general size library generation is that, instead of applying (and searching over) breakdown rules at program generation time, the rules must be "compiled" into code which can then be applied (and searched over) at runtime. This is because rules have different applicability conditions based on the transform size, which is not known at library generation time.

Autolib and this thesis. It is important to note that the differences between the fixed size program generation system and the automatic general-sized library generation system are mostly orthogonal to the topic of this thesis: the techniques presented in this thesis are primarily of an algorithmic nature, and in concept, can be interfaced either with the original SPIRAL system to produce fixed size code, or with Autolib to produce general sized libraries.

The reason for this is the following. The fundamental assumption in this thesis is that FFTs can be manipulated using certain degrees of freedom and factorizations to yield high performance for a target architecture. In essence, this thesis establishes a set of rules that are used to rewrite a problem specification into high performance platform-tuned code. In their straightforward form, these rules can be used to produce fixed size code. However, Autolib can use the same set of rules to automatically derive the recursion step closure that is needed to build a general-sized library. In practice, some rules may have to be expressed in a manner that is more appropriate to the type of program generation system being used. We use the fixed-size program generation infrastructure for the Cell platform, and the general-size infrastructure for the cluster platform in this thesis.

2.5 Chapter summary

We presented SPL, a declarative representation that is at an abstraction level appropriate to several manipulations that we will perform later in this thesis. We presented the FFT algorithm and several existing parallelized versions which are all limited by their lack of support for minimizing data movement costs. Finally, we presented SPIRAL and Autolib which use SPL along with rewriting systems to automatically generate fixed and general-sized code from problem specification

using only a set of rewrite rules and some high-level information about the target architecture. SPIRAL rewrites SPL formulas to perform well on certain architectural paradigms. In the next chapter, we will look at the two fundamental architectural paradigms required for automatic program generation for distributed memory computing platforms.

CHAPTER 3

The Parallelization and Streaming Paradigms

Optimizing FFTs for distributed memory target architectures requires a thorough understanding of both the algorithm and the architecture. In this chapter, we first present an abstraction of our distributed memory target architectures, followed by a discussion of actual platforms used for evaluation in this thesis. We then present techniques to adapt SPL formula fragments found in the Cooley-Tukey FFT and other FFTs to two fundamental features that are crucial for performance on distributed memory architectures: parallelism and streaming. We use the techniques presented in this chapter as building blocks in the next chapter, where we build the actual FFTs.

3.1 Target Architectures

In this section, we first present abstractions of our target architectures. Then, we present details of the actual platforms based on our abstractions that were chosen to evaluate this thesis. Specifically, we look at the Cell Broadband Engine, and three supercomputers: the Axon and the Cray XT4 and XT5 systems.

Our target architectures can be broadly described as MIMD (Multiple Instruction stream, Multiple Data stream) [Flynn, 1972] and NUMA (Non-Uniform Memory Access). An abstraction is shown in Figure 3.1. As this figure shows, our target architectures consist of several identical PEs, each with its own private memory. Below, we expand this abstraction by examining its various components in better detail.

Terminology: nodes versus processing elements. To avoid the ambiguity of the conventional term "node" in the context of systems that include multicore processors and incorporate multiple processors in each motherboard, we instead use the term "PE" (PE) to refer to the small-

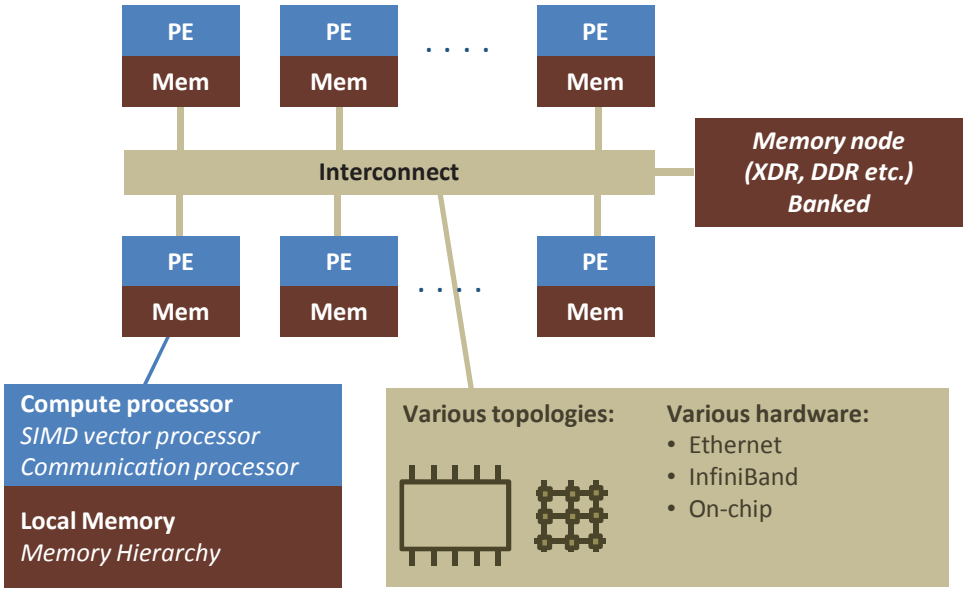


FIGURE 3.1: Abstraction of the target platform architecture used in this thesis. Optional components or features are *italicized*.

est unit of the system that can independently perform generic computations (such units can internally contain SIMD units). A PE is typically a single core of a multicore processor inside the system, or a single unicore processor inside the system.

3.1.1 Abstraction of Target Architectures

Multiple processing elements. Our model assumes a number of identical PEs that operate independently, as shown in Figure 3.1. Each PE consists of at the least a compute processor, which may optionally include SIMD vector units. Each PE may also optionally contain a communications processor, which assists in performing communications in the background without interrupting the compute processor.

Since we are interested in a platform's ability to perform numerical computations, we measure this ability using the platforms' peak floating point performance, which is the maximum number of floating point operations that it can retire per unit time, usually stated in giga floating point operations per second (Gflop/s). Floating point addition, subtraction, and multiplication instructions are counted as one operation each; fused multiply-add instructions count as two operations.

Distributed memory. Each PE possesses its own local memory. We assume a PE has faster access to its local memory than to another PE's local memory. Every PE is assumed to have its own independent memory address space. The last two are not strict requirements, and it is

straightforward to implement our ideas on systems that use a shared memory address space, or other schemes such as PGAS (partitioned global address space) [Stitt, 2010].

Explicit data transfer. We also assume the architecture requires explicit programming calls to transfer data across address spaces. Some architectures can overlap computation with communication. These architectures may use a one-sided or a two-sided communications model. We define a "one-sided push" model as one where the sender can send data asynchronously using a non-blocking call, and can later block on completion of the send (the receiver cannot natively block on the receipt of expected data). We define the "one-sided pull" model analogously. The Cell BE (discussed in Section 3.1.2) is based on DMA programming, and is capable of operating under both the push and pull models. We also include architectures with a two-sided communications model, in which the sender can use a synchronous non-blocking call to send data, and the receiver can use a blocking call to complete receiving expected data. The MPI 2.0 specifications [Forum, 1998b] include calls to operate under both the one-sided and the two-sided communications models.

Interconnection. The PEs are connected through an *interconnect*. We assume that, from a programming perspective, every node can communicate directly with every other node.

The interconnect network may use one of several *topologies*, including a ring (e.g., Cell BE), hypercube, fat-tree (e.g., Axon), 3D torus (e.g., BlueGene/P), mesh (Tilera TILE64) or others. The actual hardware connection may consist of an on-chip network (Cell BE) or other transport technologies including InfiniBand, Myrinet, NUMAlink, and Ethernet.

Since our FFTs require all-to-all data exchanges, both the *bandwidth* and the *latency* of the interconnect network will critically affect performance. Interconnect performance during an all-to-all exchange is also dependent on the topology. A good indicator of interconnect performance is its bisection bandwidth [Hennessy and Patterson, 2003].

Memory node. An optional global memory node may also exist in our model. For distributed memory architectures that reside entirely on a single chip or motherboard (e.g., the Cell BE), this is typically the main memory. Clusters may have a dedicated storage node. We assume the address space of the memory node is shared among the PEs.

Emerging multicore processors and the distributed memory paradigm. Although some of the mainstream multicore architectures (x86 Intel, AMD multicores) use the shared memory model, it is important to distinguish between the *programming model* and the design and constraints of the *underlying architecture*. Shared memory programming models allow the programmer to assume all PEs share a single unified memory space, and are typically implemented using cache coherence protocols. As the number of cores per processor scales, so does the penalty to be paid in accessing data cached on a different PE. Consequently, some of the techniques pre-

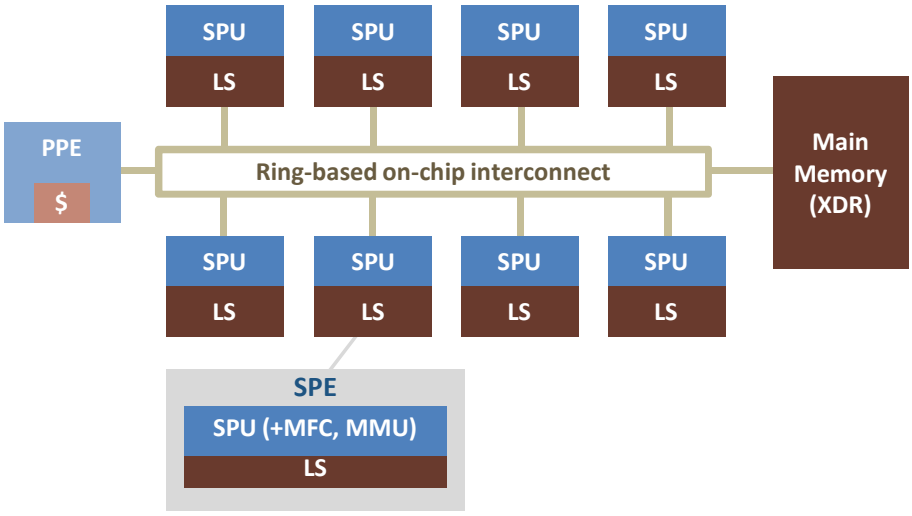


FIGURE 3.2: Cell processor architecture. The PPE is dimmed out in this figure since it is not used in this thesis. Each SPE consists of an SPU, an MFC, a memory management unit (MMU), and a local store (LS).

sented in this thesis should also be applicable to shared memory manycore processors.

The ideas presented in this thesis apply to any platform which can be modeled as an instance of our abstraction. Actual architectures may implement this abstraction in different ways. We consider several different actual platforms for this thesis, including the Cell processor (an example of a single-chip implementation of this abstraction, which is becoming an increasingly common form of multicores), and several clusters. Next, we show how each of our chosen architecture targets implements this abstraction.

3.1.2 Cell Broadband Engine

Introduction and architecture. The Cell Broadband Engine (Cell BE, or simply, the Cell) is a multicore processor that is designed for high-density floating-point computation required in multimedia, visualization, and other science and engineering applications. As shown in Figure 3.2, its design includes several cores called synergistic processing elements, or SPEs, and a main core, the power processing element (PPE). The PPE is designed to be the "master" core where the operating system and other main application threads run. Numerically intensive computations are performed by the SPEs.

Each SPE consists of a compute processor (called synergistic processing unit or SPU) with SIMD vector units, a memory flow controller (MFC), and a fast local memory (256kB in current versions), called the *local store* (LS). An SPU can access its own local store much faster than it can access the local stores of other SPUs, or the main memory. The memory flow controller is

designed to handle multiple memory requests in the background without affecting the computations on the SPU, thus parallelizing memory operations with numeric computation. The SPEs, PPE, and the main memory controller are connected via an on-chip interconnect called the element interconnect bus (EIB), whic is ring-based in current instantiations.

Performance characteristics. We consider two versions of the Cell architecture: the original Cell BE, and the newer PowerXCell 8i. Both include a single PPE and 8 SPEs with 256kB local stores. Each SPE can retire one 4-way vectorized single precision FMA (fused multiply-add) floating point instruction per cycle. Therefore, up to 8 single-precision operations can be performed per cycle. At a clock rate of 3.2GHz, this translates into a peak performance of 25.6 Gflop/s per SPE, or 204.8 Gflop/s across all 8 SPEs. The Cell BE provides limited, non-pipelined support for 2-way vectorized double precision computation, thus restricted to a low peak performance of 12.8 Gflop/s. The PowerXCell 8i does not have this limitation and is capable of a double-precision peak performance of 102.4 Gflop/s.

The MFC has a DMA queue that can service up to 16 outstanding DMA requests. It can also work on "DMA-list" requests: the SPU can submit a list of up to 2,048 DMA requests by storing them in an array in local memory, which the MFC can then process in the background.

The ring-based EIB contains 4 stepwise unidirectional channels, providing an aggregate bandwidth of 204.8 GB/s. Each channel that emerges from the ring connects to either an SPE, or the PPE, or the memory or I/O controllers, and can handle up to 25.6 GB/s of traffic. Rambus XDR memory is used for the main memory, whose address space is striped across 16 banks.

Programming model. The Cell can be programmed using the pthreads library. Main memory is accessed by the PPE and SPE threads using the shared virtual memory address space. The memory addressing scheme also allows each SPE to provide other SPEs with access to its local store. The Cell architecture uses a DMA (direct memory access) based memory access model. It requires the programmer to explicitly orchestrate all memory and inter-core data movement operations. The Cell's explicit memory access programming model allows for finer control of the memory system compared to a conventional cache-based processor, and allows for potentially better exploitation of memory bandwidth. However, it also drastically increases the programming burden and expertise required by the software developer. An automatic code generation and optimization system is thus particularly valuable for such an architecture.

To automatically generate high-performance code for the Cell we must address three architectural characteristics: 1) within an SPE we need to produce SIMD vector code, 2) executing code across multiple SPEs requires parallelization, and 3) hiding the latency of data transfer between local stores and the main memory requires streaming, which is a technique similar to software pipelining.

The use of several architectural paradigms and the programming complexity due to the need for explicit memory orchestration makes the Cell a challenging and therefore an excellent choice of platform for evaluating the work in this thesis. The Cell is also one of the first single-chip platforms to be designed with the distributed memory paradigm, which has traditionally been reserved to cluster-based supercomputing platforms. As the number of cores on a chip scales in the future, due to the scaling limitations of shared on-chip cache designs, the distributed memory paradigm is likely to find its way into an increasing fraction of chip based multiprocessors. In this sense, the Cell is a likely representative of a class of future platforms.

3.1.3 Cluster: Axon

Distributed memory computing architectures are traditionally associated with clusters computers. A cluster is essentially a group of linked computers that work together as one system. We primarily consider "compute clusters," by which we mean clusters used for scientific computing, simulations, and other applications involving intensive numerical computations. Compute clusters are typically tightly coupled, which means their design includes a low latency, high performance dedicated interconnect to handle frequent communications between nodes. Clusters are primarily defined by the number and type of PEs, and the topology and type of the interconnect.

Introduction and architecture. The Axon is a supercomputer with 256 Intel Xeon (Intel's Core architecture) based PEs and an InfiniBand network with a tree topology. As shown in Figure 3.3, the Axon consists of 32 nodes connected via two InfiniBand switches, with one motherboard per node, two chips per motherboard, and 4 cores per chip, for a total of 256 PEs.

Performance characteristics. Each PE on the Axon provides a peak performance of 3 Gflop/s (double precision). With 256 PEs, the double precision floating point peak performance is thus 768 Gflop/s.

The interconnect performance varies depending on the level as Figure 3.3 shows. At the top most level, each node is connected to 15 other units through an InfiniBand 4x SDR switch, with each link capable of a bandwidth of 8 Gb/s. The two switches are connected via 4 similar links. The two CPUs on each motherboard are connected via the motherboard's fast interconnect, while the cores within each CPU use the faster on-chip interconnect.

Communication tasks must also be handled by the computing cores, i.e., the Axon has no dedicated cores or other processing units exclusively for handling communications in the background.

Programming model. The Axon can be programmed using an MPI [Forum, 1998a] library, which is typical for supercomputing clusters. The address space is distributed.

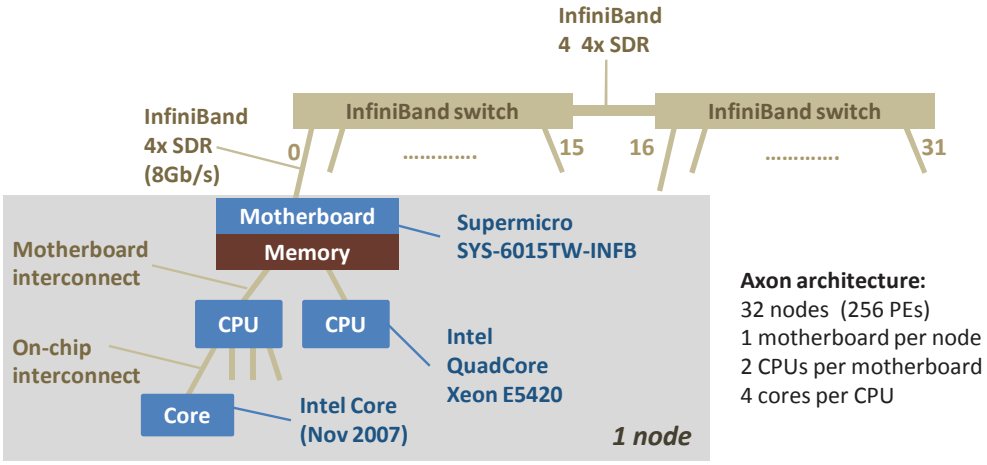


FIGURE 3.3: Architecture of the Axon cluster supercomputer.

3.1.4 Cluster: Cray XT4 and XT5

Introduction and architecture. The Cray XT4 and its upgraded version, the XT5, are commercially used supercomputing platforms. They use AMD Opteron processors, and Cray SeaStar2 or SeaStar2+ interconnects. Our XT4 evaluation platform provides 16 4-core compute nodes (64 PEs), and our XT5 evaluation platform provides 16 8-core compute nodes (128 PEs).

Performance characteristics. Each XT4 node contains an AMD Opteron QuadCore processor clocked at 2.2GHz (64 PEs at 8.8 Gflop/s each), and each XT5 node contains two AMD QuadCore Shanghai 2.7GHz processors (128 PEs at 10.8 Gflop/s each).

The SeaStar2 interconnect on the XT4 provides a 6.4 GB/s connection between the processors in a node, and 7.6 GB/s links to six neighboring PEs in a 3D Torus topology. The SeaStar2+ on the XT5 is similar, except that it offers 9.6 GB/s of intra-node bandwidth.

Programming model. The Cray XT4 and XT5 systems can be programmed using an MPI [Forum, 1998a] library. The address space is distributed.

3.2 Framework for Adaptation Through Formula Rewriting

We have thus far examined features of our target architecture. In this section, we present the framework we use for adapting the dataflow of FFTs to our target architecture through formula manipulation and rewriting. We will use this framework in the rest of this chapter to define the parallelization and streaming paradigms. The basic approach that we use is similar to previous work on mapping FFTs to SIMD vector architectures [Franchetti et al., 2006b] and cache-based SMP multicore processors [Franchetti et al., 2006a].

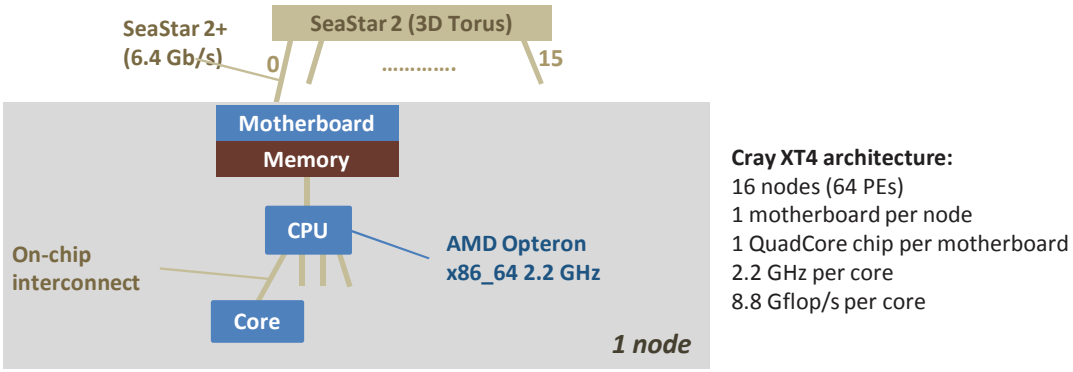


FIGURE 3.4: Architecture of the Cray XT4 supercomputer.

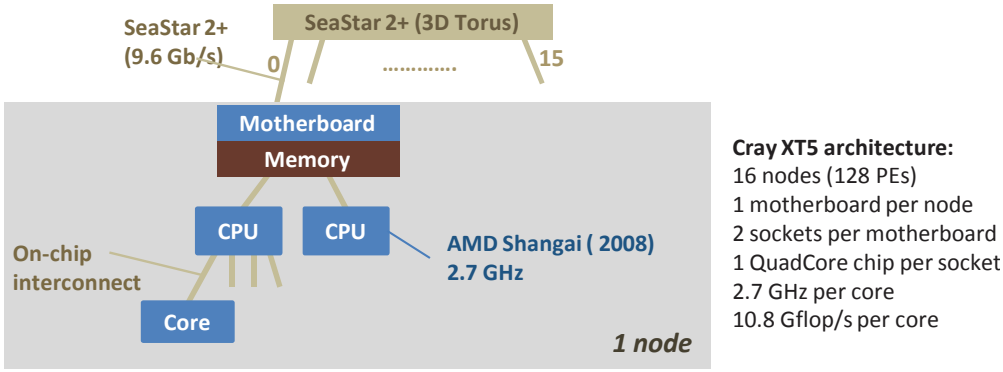


FIGURE 3.5: Architecture of the Cray XT5 supercomputer.

Architectural paradigms: parallelization and streaming. A *paradigm*, as defined in Chapter 2, is an architectural feature that is the target for structural optimizations of the FFT. Our target architectures may include several features including cache based memory hierarchies, SIMD vectorization, multiple forms of parallelism, and support for explicit background memory management. Previous work covered code generation for cache based systems and for SIMD vector units, but did not address the *distributed memory parallelism* and *memory streaming*, which is the focus of this thesis. In most cases, the previous work composes well with our work, and is reused. Before we discuss parallelization and streaming in detail, we sketch the basic idea.

Basic idea: rewriting. We consider the Cooley-Tukey FFT (2.12) for 1D DFTs, and the row-column FFT (2.15) for 2D DFTs. Both inherently contain data parallelism represented by the occurring tensor products of the form $(I \otimes A)$ and $(A \otimes I)$ as we explained in Section 2.4.2. However, the data parallelism as is may not be suited for an efficient implementation on our target platform. To solve this problem, we will formally manipulate the FFT to obtain the appropriate structure. The design of the rewriting system depends on the paradigm and is done in three

steps. We explain these steps using a simplified view of parallelism as an example:

Step 1: Architecture abstraction and specification. We determine and abstract the main architectural feature that is the target of our optimization, and identify relevant parameters. For parallelization, this may be the number of processors, p. The paradigm is then specified by its name and the parameters (par(p) in our example). We tag an SPL expression indicate that we want to adapt the expression to that paradigm. For example, to parallelize A over p PEs, we write:

$$\underbrace{A}_{\text{par}(p)}$$

Step 2: Identification of mappable constructs. We determine the set of SPL constructs whose dataflows map naturally well to the paradigm. These constructs can then be directly translated into corresponding C code. Our goal then, is to rewrite our initial FFT algorithm to be composed only of these mappable constructs. Mappable constructs are written using special tensor products to identify them.

For example, as mentioned in Section 2.4.2, constructs of the form $I_p \otimes A_m$ naturally map to p PEs. In this case, we represent the tensor product using the $\otimes_\parallel$ symbol:

$$I_p \otimes_\parallel A_m \tag{3.1}$$

Step 3: Rewriting via formula manipulation. Our goal is to rewrite a given SPL expression into a form that is directly mappable in the sense discussed above. To do so, we determine rewrite rules that convert SPL expressions into mappable expressions. We remove our specification tags (shown in step 1) once the rewriting is complete, and we have arrived at an algorithm consisting of mappable constructs (shown in step 2). An example of a rewrite rule is:

$$\underbrace{I_n \otimes A_m}_{\text{par}(p,\mu)} \rightarrow I_p \otimes_\parallel (I_{n/p} \otimes A_m) \tag{3.2}$$

The rule parallelizes $(I_n \otimes A_m)$ for p processors. Note how the right-hand side of (3.2) matches (3.1).

Next, we expand this framework to the two main architectural paradigms in this thesis: parallelization and streaming.

3.3 Parallelization

This section presents the parallel paradigm. We target "coarse-grained" platform parallelism available in the form of multiple PEs. Load balancing the computation is our primary goal, which is necessary to achieve good performance. We include explicit memory movement, which is typically required by distributed memory platforms. Our methods expand the prior work in [Bonelli et al., 2006; Franchetti et al., 2006a].

3.3.1 Architecture Abstraction and Specification

Our main target architectural component is parallelism. We define a new tag, called par() to specify that we want to parallelize an SPL expression To generate high performance parallelized FFT code, we have the following requirements:

- **Load balancing.** The computation load should be divided equally among all PEs, and sequential code should be minimized to improve speed-up (Amdahl's law). We specify the number of PEs using the tag parameter p.
- **Explicit distributed memory access.** Since our data is distributed among multiple PEs, generated code must include explicit memory instructions (message passing or DMA) to transfer data between PEs. Input and output data is assumed to be distributed in a block fashion among the PEs. Non-standard initial and final data distributions are discussed in Section 4.2.2.
- **Size of communication packets.** We design our algorithm to produce data exchanges, when needed, in packets of a size that is optimal for our target architecture, to minimize communication costs. Typically, this means producing large packet sizes. The optimal packet size can be determined either using architecture specifications, or by a simple experiment that measures achieved bandwidth for a range of packet sizes. For most architectures, achieved bandwidth increases with increasing packet sizes until a certain point, which is the architecture's optimal packet size.[1] We specify our packet size using the tag parameter μ.
- **Computation/communication overlap.** Optionally, if the architecture supports it, we aim to produce algorithms that are capable of overlapping computation with communication to minimize communication cost.

[1] Note that the transform size determines the maximum possible packet size

- **Minimizing barriers.** We minimize the number of global synchronization points or barriers to maximize performance. Optionally, when supported by the architecture, we produce code that does not use global barriers, but instead uses cheaper local barriers.

3.3.2 Identifying Mappable Constructs

We now identify constructs that map naturally well to parallel architectures, keeping in mind the goals we identified above.

Load balancing. As explained in Section 2.4.2, constructs of the form $I_p \otimes A_m$ are embarrassingly parallel, map naturally to p PEs. As in [Franchetti et al., 2006a], we use a tagged version of the tensor product, $\otimes_\parallel$, to signify a "parallel loop":

$$I_p \otimes_\parallel A.$$

As illustrated, we show the C pseudo-code equivalent of the SPL expression $y = (I_p \otimes_\parallel A_m)x$. We assume a distributed memory model that is programmed using an MPI library, and assume that the input and output vectors are divided into p blocks of size m each. As an example, the contiguous elements of the x vector (shown in the SPL expression above) beginning at the $m * i^{\text{th}}$ element and ending at the $(m * (i+1)) - 1^{\text{th}}$ element are resident in array X of processor i:

```
void A(double *out, double *in) {
  // Code that applies A to in to produce out
}

int main(int argc, char **argv) {
   double *Y, *X;

   // Y: portion of the y vector assigned to this PE
   // X: portion of the x vector assigned to this PE

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &p);
   // Allocate and initialize X, Y here

   A(X, Y);  // Computes (globally): y = (Ip x A)

   MPI_Barrier(MPI_COMM_WORLD);
   MPI_Finalize();
}
```

As seen in the code above, each PE simply applies A to its chunk of the input vector.

Explicit memory access. Explicit memory accesses are difficult to express in SPL, and are better expressed at the lower level using Σ-SPL. We introduce new constructs in Σ-SPL to ex-

press explicit memory access, and provide an an example of converting SPL code into Σ-SPL in Section 3.3.3.

Size of communication packets. Permutations of the form $P \otimes I_\mu$ (where P is any permutation) access memory in chunks of size μ. We use a tagged tensor product $\bar{\otimes}$ to recognize these constructs:

$$P \bar{\otimes} I_\mu.$$

This means they are in their final form required for accessing memory in packets of size μ.

Minimizing barriers. Synchronization barriers must occur only when a PE uses data that has been previously processed by another PE. We discuss this in the context of building complete FFT algorithms in Chapter 4.

Vectorization. Our rewrite rules should be compatible with the previous work on SIMD vectorization [Franchetti et al., 2006b].

3.3.3 Rewriting via Formula Manipulation

Constructs of the form $I_n \otimes A$ are easily parallelized when $p|n$ [2] using perform simple loop splitting. As a rewriting rule, this is expressed as:

$$\underbrace{I_n \otimes A_m}_{\text{par}(p,\mu)} \rightarrow I_p \otimes_\parallel (I_{n/p} \otimes A_m), \quad mn/p \geq \mu \tag{3.3}$$

Note that the rule also requires that the block size mn/p is effectively the packet size, which must be at least as large as the requested packet size μ.

In (3.3), iterations in the parallel loop $(I_p \otimes_\parallel A_m)$ are executed simultaneously on p PEs. The input vector is divided into p blocks of size mn/p each, and each block is read in by a PE in packets of size μ. The largest possible packet size is hence $\mu = mn/p$. We show the corresponding MPI code below:

```
void A(double *out, double *in) {
  // Code that applies A to in to produce out
}

int main(int argc, char **argv) {
   double *Y, *X;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &p);
```

[2] In this thesis, we only target sizes where $p|n$ (where p is the number of PEs). Extension to cases were p does not divide n is accomplished by executing the left-over parallel iterations on a subset of the PEs

```
    // Allocate and initialize X here

    // Loop performs (I(n/p) x Am) on this node:
    for(i=0;i<n/p;i++)
        A(Y+(i*m), X+(i*m));

    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Finalize();
}
```

Next, we rewrite the construct $A_m \otimes I_n$, which can be converted into our parallel mappable form using (2.8) and (3.3):

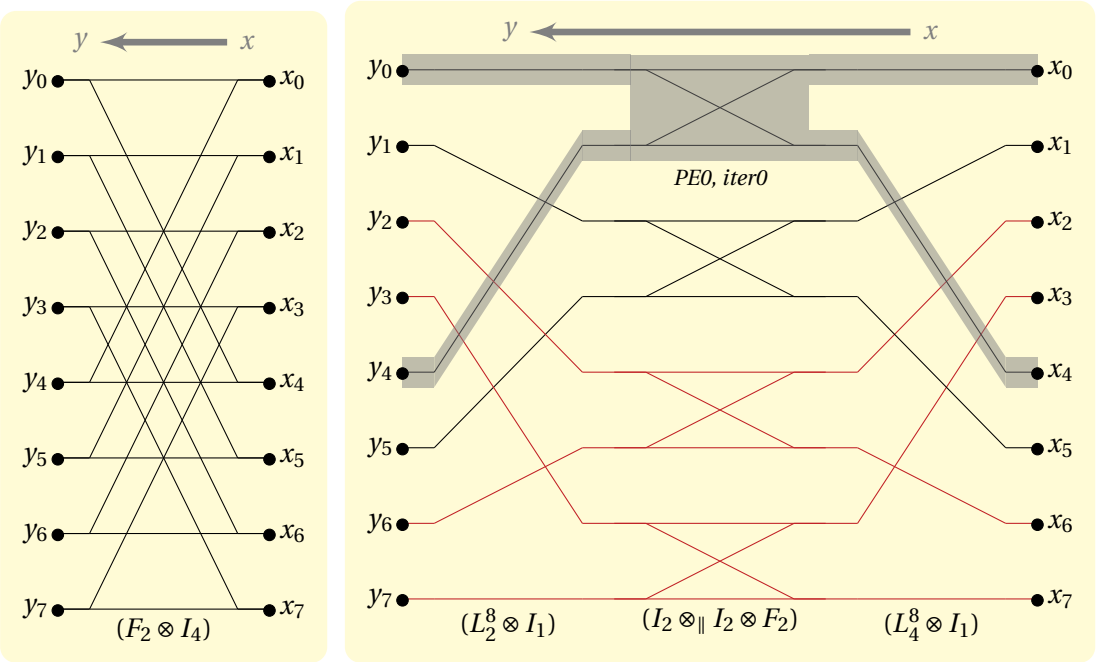
$$\underbrace{A_m \otimes I_n}_{\text{par}(p,\mu)} \rightarrow \underbrace{L^{mn}_m}_{\text{par}(p,\mu)} \underbrace{(I_n \otimes A_m)}_{\text{par}(p,\mu)} \underbrace{L^{mn}_n}_{\text{par}(p,\mu)} \rightarrow (L^{mn}_m \bar{\otimes} I_1)(I_p \otimes_{\parallel} (I_{n/p} \otimes A_m))(L^{mn}_n \bar{\otimes} I_1)$$

However, this leads to packet sizes of $\mu = 1$ for $(L^{mn}_m \bar{\otimes} I_1)$ and $(L^{mn}_n \bar{\otimes} I_1)$ in the equation above. This is because each PE works on a single kernel during each iteration of the loop over n/p. As an example, we first provide a formula, followed by its dataflow graph:

$$\underbrace{F_2 \otimes I_4}_{\text{par}(p,\mu)} \rightarrow (L^8_2 \bar{\otimes} I_1)(I_2 \otimes_{\parallel} (I_2 \otimes F_2))(L^8_4 \bar{\otimes} I_1)$$

.

In the dataflow graph below, data points and kernels in the dataflow graph assigned to the first PE are shown in black, while those assigned to the second PE are shown in red:



As seen in the right side dataflow graph, in its first iteration, PE 0 first loads x_0 and x_4, computes a single F_2 kernel, and stores the results to y_0 and y_4. Then, in its second iteration, PE 0 loads x_1 and x_5, computes a single F_2 kernel, and stores the results to y_1 and y_5.

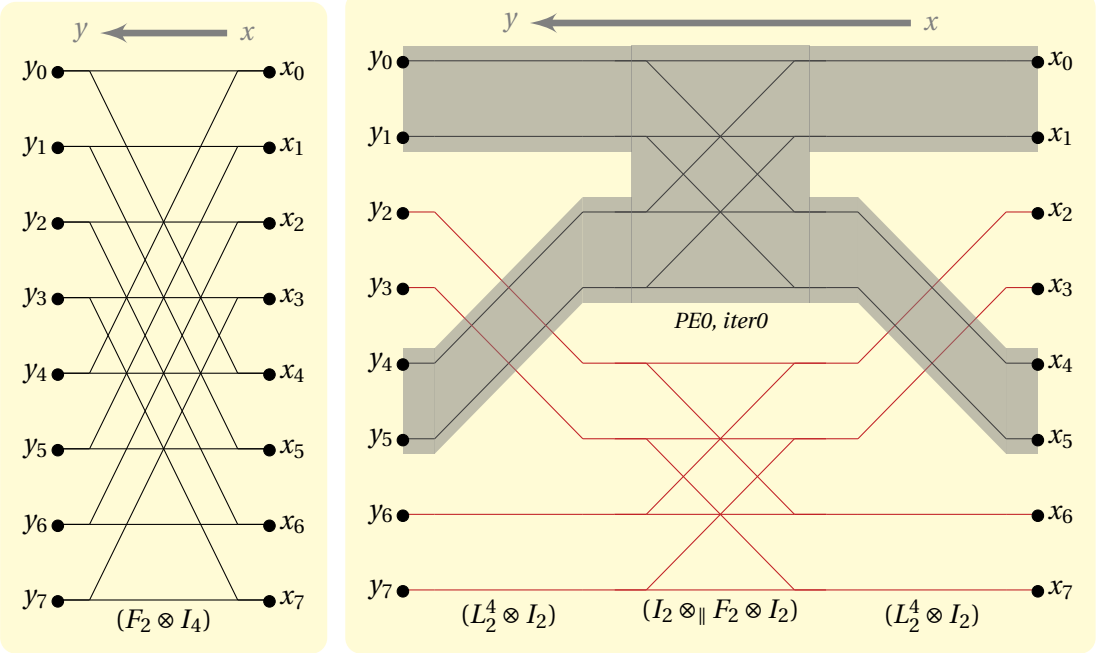
To increase the packet size, each PE can load and store data for multiple successive kernels (at least μ many) jointly. This can be done in several ways. We use the formula below which is one way to capture the partitioning of the dataflow graph such that μ successive data elements in memory are always loaded or stored at the same time:

$$\underbrace{A_m \otimes I_n}_{\text{par}(p,\mu)} \rightarrow (L^{mp}_n \bar{\otimes} I_\mu)(I_p \otimes_\parallel (A_m \otimes I_\mu))(L^{mp}_m \bar{\otimes} I_\mu), \mu = n/p$$

Note that our load and store packet sizes appear as μ in the formula above. Using the same example as earlier, we now parallelize $F_2 \otimes I_4$ to load and store in packets of size 2:

$$\underbrace{F_2 \otimes I_4}_{\text{par}(p,\mu)} \rightarrow (L^4_2 \bar{\otimes} I_2)(I_2 \otimes_\parallel (F_2 \otimes I_2))(L^4_4 \bar{\otimes} I_2)$$

As illustrated in the following dataflow graph (first processor in black, second processor in red), PE 0 now loads both x_0 and its neighbor x_1 together, and also x_4 and x_5 (packet size in each case is therefore now 2). It then computes two F_2 kernels before storing the results again using 2 packets of size 2 each.



Since our SPL construct now requires explicit memory access, we first convert it to Σ-SPL to extract the required memory accesses before generating C code.

Explicit memory access. As explained in Section 2.4.3, Σ-SPL distinguishes explicitly between skeletons (compute kernels) and decorations (loads and stores, including permutations). We extend this framework to mark up scatters and gathers in Σ-SPL that represent explicit data transfers instead of (implicit) loads and stores. We denote a sum to be parallelized such that every iteration runs on a different PE by $\sum\!\!\!\!\!\!\parallel$. We introduce a specially marked gather $G_{\mathrm{DT}}(q)$ and scatter $S_{\mathrm{DT}}(q)$ (DT stands for "data transfer") to express explicit communication to access non-local data. These are similar to the scatter and gather matrices defined in Section 2.4.3, except for two differences: a) we use the data transfer scatter and gather only with the mapping function (2.23), which means that these scatter or gather data in specified packet sizes, and b) these get converted into target-specific explicit memory transfer instructions like DMA or MPI code.

To illustrate the process of generating explicit memory accesses using Σ-SPL, we provide an example below:

$$\underbrace{A_m \otimes I_n}_{\text{par}(p,\mu)} \rightarrow (L^{mn}_n \bar{\otimes} I_\mu)(I_p \otimes_\parallel (A_m \otimes I_\mu))(L^{mn}_m \bar{\otimes} I_\mu)$$

$$\rightarrow \sum_{k=0}^{p-1}{}_\parallel S_{\text{DT}}(q_{k,n/\mu,\mu}) \left(\sum_{j=0}^{\mu-1} \text{S}(h_{j,\mu}) A_m \text{G}(h_{j,\mu}) \right) G_{\text{DT}}(q_{k,n/\mu,\mu}),$$

$$\mu = n/p \quad (3.4)$$

The second step is a standard SPL to Σ-SPL conversion except for the first and last SPL factors. These are permutations, and are converted into explicit memory scatters and gathers in the outer parallel sum. Also, note how the explicit memory scatter and gather access memory in chunks of size μ. The explicit memory scatter and gather operations are easily converted into code. We provide an example below where we convert the following scatter into DMA-based code:

$$\sum_{k=0}^{p-1}{}_\parallel S_{\text{DT}}(q_{k,2,4})$$

```
void DMA_SCATTER(source*, dest*, pk_size)
{ spu_mfcdma64(source, dest, size*8, 1, MFC_PUT_CMD)
}

/* Code runs in parallel on 2 processing elements */
int main()
{ for(i:=0; i<=7; i++) // Left most scatter
   { DMA_SCATTER(scat_func(X,i), scat_func(Y,i), 4)
   }
}
```

The other remaining basic SPL constructs are rewritten to produce explicit memory accesses similarly, and summarized in Table 3.1.

Increasing communication packet size. Most interconnects perform poorly when transferring small data packets, since the per-packet overhead may not amortize, or the underlying hardware may not be utilized effectively for packet sizes below a threshold. This is illustrated in the simple experiment in Figure 3.6, which shows the achieved interconnect bandwidth as a function of packet size for the same amount of data transferred.

Unfortunately, the data exchange packet sizes in the parallelization approach shown so far do not scale well with the number of PEs. This is because the number of available iterations of the kernel (given the dimension of the identity matrix) is split between the packet size and number of PEs (in other words, $n = p\mu$). For a given n, as p increases, μ decreases. Since the

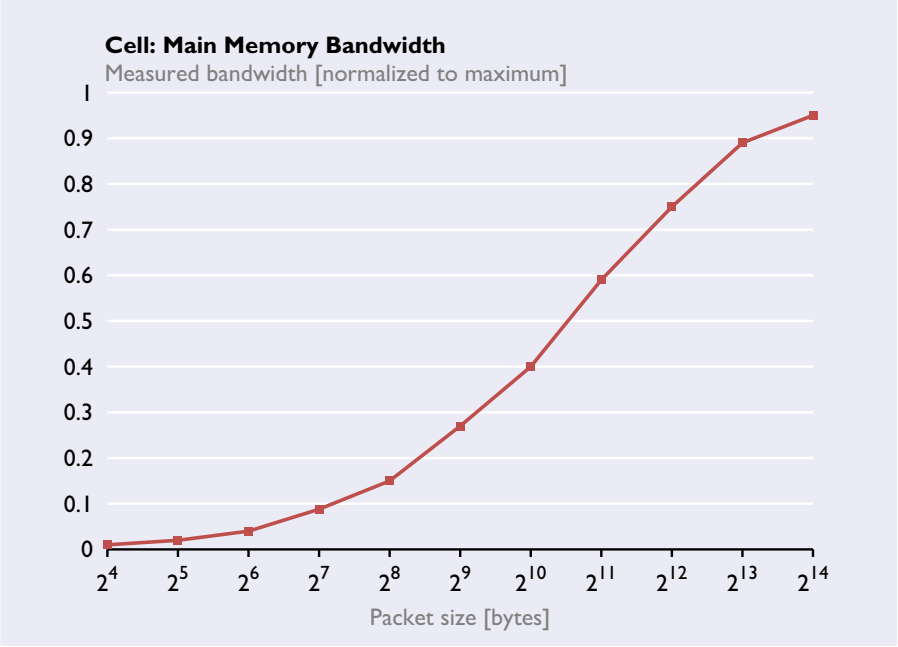


FIGURE 3.6: Cell Main memory bandwidth measured for a single read and write performed on 2^{20} bytes of contiguous data at various packet sizes. Measured bandwidth is normalized to the theoretical maximum bandwidth of 25.6 GB/s.

CT-FFT algorithm consists of two factors, the minimum packet size scales as a square root of the size of DFT, as shown in Figure 3.7.

With the previous parallelization method each PE sends multiple packets to every other PE. [Bonelli et al., 2006] and [Chen and Johnson, 2004] present methods that factorize the stride permutations into local and global parts: each processor p_i first locally collects all packets to be sent to processor p_j, assembles them, and sends a single huge packet to p_j, which disassembles them upon receipt. The cost of the local assembly and disassembly are largely hidden by being implemented as readdressing inside the computation kernels, using loop merging techniques developed in [Franchetti et al., 2005]. The factorization is shown below. We use the tag $\underbrace{A}_{\text{parBig}(p)}$ to indicate parallelization of A using the large packet algorithm (note that this tag is not parameterized by μ, because it is meant to parallelize with the largest packet size possible):

$$\underbrace{L^{mn}_{m}}_{\text{parBig}(p)} \rightarrow (I_p \otimes_{\parallel} L^{mn/p}_{m/p})(L^{p^2}_{p} \bar{\otimes} I_{mn/p^2})(I_p \otimes_{\parallel} (L^{n}_{p} \otimes I_{m/p})) \tag{3.5}$$

Using this factorization, we can convert, for example, $A_m \otimes I_n$ into its parallelized equivalent:

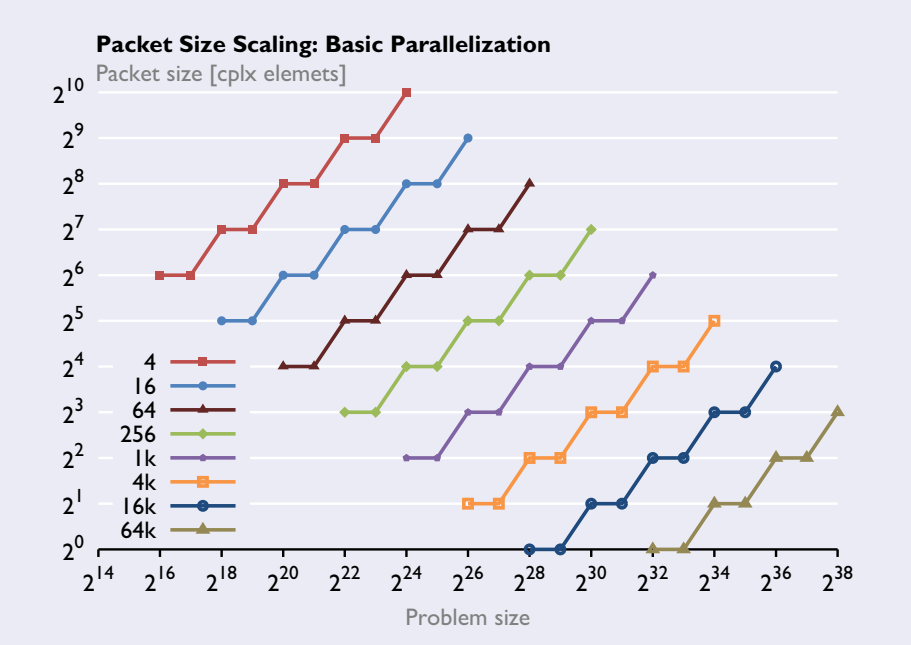


FIGURE 3.7: Packet size scaling using the basic parallelization algorithm. Each line corresponds to a specified number of PEs. The minimum packet size μ is given by $2^{(\frac{\log N}{2} - \log p)}$, where N is the problem size, and p is the number of PEs.

$$
\begin{aligned}
\underbrace{A_m \otimes I_n}_{\text{parBig}(p)} \rightarrow\ & \underbrace{L^{mn}_m}_{\text{parBig}(p)}\ \underbrace{(I_n \otimes A_m)}_{\text{parBig}(p)}\ \underbrace{L^{mn}_n}_{\text{parBig}(p)} \\
& \rightarrow (I_p \otimes_\parallel L^{nm/p}_{n/p})(L^{p^2}_p \bar{\otimes} I_{nm/p^2})(I_p \otimes_\parallel (L^m_p \otimes I_{n/p})) \\
& (I_p \otimes_\parallel (I_{n/p} \otimes A_m)) \\
& (I_p \otimes_\parallel L^{mn/p}_{m/p})(L^{p^2}_p \bar{\otimes} I_{mn/p^2})(I_p \otimes_\parallel (L^n_p \otimes I_{m/p}))
\end{aligned} \tag{3.6}
$$

As an additional benefit, because each processor sends exactly one packet to each other processor during an all-to-all exchange, this method can use the `mpi_alltoall` call (or its equivalent) when available, which is typically optimized for a given platform based on its topology and interconnect characteristics. In this thesis, we assume the `mpi_alltoall` call is the fastest way to perform an all-to-all exchange, and do not explore alternatives, as done in [Chen and Johnson, 2004] or FFTW [Frigo and Johnson, 2005].

When using the factorization shown in (3.6), the packet size in the data exchanges is always performed at N/P^2, where N is the size of the transform ($N = mn$ in (3.6)). This scales better than

the previous method, as shown in Figure 3.8. Note that because the packet size is always fixed at N/P^2, the parallelization tag for this methods does not contain a packet size specification.

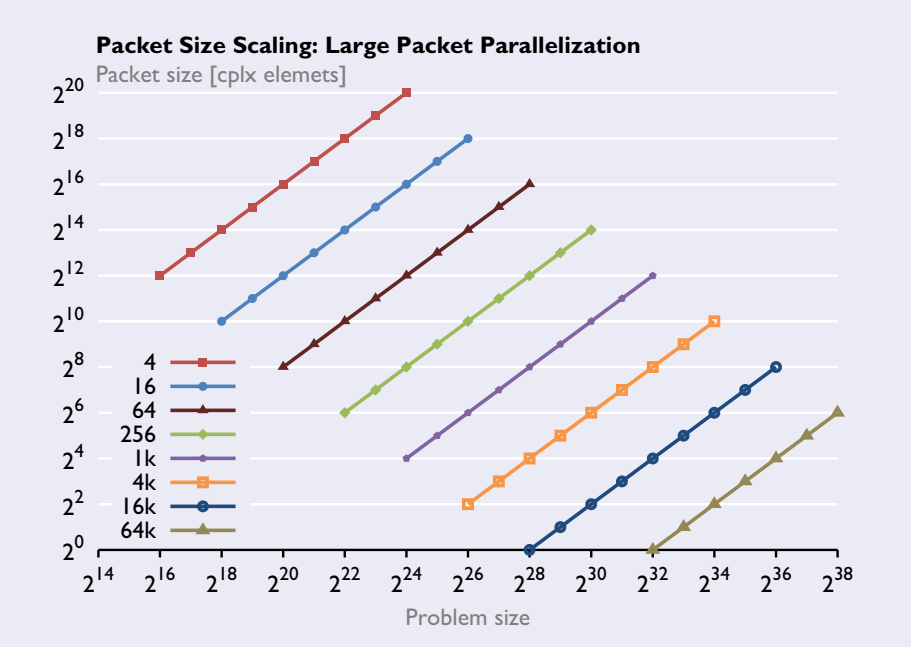


FIGURE 3.8: Packet size scaling using the large packet algorithm. Each line corresponds to a certain number of processors. The minimum packet size μ is given by N/p^2, where N is the problem size, and p is the number of PEs.

Vectorization. However, the method above, as taken from [Bonelli et al., 2006], does not lend itself easily to SIMD vectorization of the inner kernel. This is because restructuring the formula above for vectorization would cause the introduction of additional permutations which are superfluous and expensive. Note that 2-way vectorization can be performed using the fact that the DFT involves complex operations. For architectures that use 4-way vectorization or higher (either current single precision architectures, or future architectures like the Larrabbe which are expected to provide 4-way vectorization support for double-precision computation), we develop a variant that can be vectorized:

$$\underbrace{(A_m \otimes I_n)}_{\text{parBig}(p)} \rightarrow \underbrace{(L^{mn/v}_{m/v} \otimes I_v)}_{\text{parBig}(p)} \underbrace{(I_{n/v} \otimes A_m \otimes I_v)}_{\text{parBig}(p)} \underbrace{(L^{mn/v}_{n/v} \otimes I_v)}_{\text{parBig}(p)} \tag{3.7}$$

$$\rightarrow (I_p \otimes_{\parallel} L^{(mn/v)/p}_{(m/v)/p} \otimes I_v)(L^{p^2}_p \otimes I_{mn/p^2})(I_p \otimes_{\parallel} L^n_p \otimes I_{m/p}) \tag{3.8}$$

$$(I_p \otimes_{\parallel} I_{n/pv} \otimes A_m \otimes I_v) \tag{3.9}$$

$$(I_p \otimes_{\parallel} L^{(mn/v)/p}_{(n/v)/p} \otimes I_v)(L^{p^2}_p \otimes I_{mn/p^2})(I_p \otimes_{\parallel} L^m_p \otimes I_{n/p}) \tag{3.10}$$

The factors in (3.10) are now compatible with vectorizable constructs as presented in [Franchetti et al., 2006b].

Degree of freedom in factorizing the stride permutation. The factorization shown in (3.5) has a degree of freedom that can be exploited in certain situations. If $m = n$, The stride permutation can also be factorized as:

$$\begin{aligned} L^{mn}_m &\to ((L^{mn}_m)^{-1})^{-1} = ((L^{mn}_m)^\top)^\top = (L^{mn}_n)^\top \\ &\to \left((I_p \otimes_\parallel L^{mn/p}_{n/p})(L^{p^2}_p \otimes I_{mn/p^2})(I_p \otimes_\parallel L^m_p \otimes I_{n/p})\right)^\top \\ &\to (I_p \otimes_\parallel L^m_p \otimes I_{n/p})^\top (L^{p^2}_p \otimes I_{mn/p^2})^\top (I_p \otimes_\parallel L^{mn/p}_{n/p})^\top \end{aligned}$$

Therefore,

$$\underbrace{L^{mn}_m}_{\text{parBig}(p)} \to (I_p \otimes_\parallel L^m_{m/p} \otimes I_{n/p})(L^{p^2}_p \otimes I_{mn/p^2})(I_p \otimes_\parallel L^{mn/p}_{m/p}). \tag{3.11}$$

We will use the alternative factorization in (3.11) to our advantage when developing DFT algorithms in Chapter 4.

3.3.4 Summary

In summary, our parallel paradigm optimizes the structure of tensor products for load balanced execution across multiple PEs, with inter-processor data exchanges being made in specified packet sizes. Our basic parallel tag is parameterized by $\text{par}(p, \mu)$, where p is the number of PEs and μ, the minimum packet size. We rewrite all SPL constructs into the two basic mappable constructs $I_p \otimes_\parallel A_m$ (load balanced execution), and $P \bar{\otimes} I_\mu$ (data reads/writes with a specified size). Our large packet parallelization tag is parameterized by $\text{parBig}(p)$, where p is the number of PEs. We rewrite all SPL constructs into the two basic mappable constructs $I_p \otimes_\parallel A_m$ (load balanced execution), and $L^{p^2}_p \otimes I_{n/p^2}$ (data exchanges with size n/p^2 where n is the problem size. Our rewrite rules are summarized in Table 3.1.

3.4 Streaming

This section presents the streaming paradigm. Streaming, also known as double-buffering or multi-buffering [3] is a technique to reduce or eliminate the cost of data movement by overlapping

[3] The name arises from the fact that multiple buffers must be maintained to implement the technique.

$$\underbrace{AB}_{\text{par}(p,\mu)} \rightarrow \underbrace{A}_{\text{par}(p,\mu)} \text{ barrier } \underbrace{B}_{\text{par}(p,\mu)} \tag{3.12}$$

$$\underbrace{I_n \otimes A_m}_{\text{par}(p,\mu)} \rightarrow I_p \otimes_{\parallel} \left(I_{n/p} \otimes A_m\right) \tag{3.13}$$

$$\underbrace{A_m \otimes I_n}_{\text{par}(p,\mu)} \rightarrow \underbrace{\left(L^{mp}_m \otimes I_{n/p}\right)\left(I_p \otimes (A_m \otimes I_{n/p})\right)\left(L^{mp}_p \otimes I_{n/p}\right)}_{\text{par}(p,\mu)} \tag{3.14}$$

$$\underbrace{L^{mn}_m}_{\text{par}(p,\mu)} \rightarrow \begin{cases} \underbrace{\left(I_p \otimes L^{mn/p}_{m/p}\right)}_{\text{par}(p,\mu)} \underbrace{\left(L^{pn}_p \otimes I_{m/p}\right)}_{\text{par}(p,\mu)} \\ \underbrace{\left(L^{pm}_m \otimes I_{n/p}\right)}_{\text{par}(p,\mu)} \underbrace{\left(I_p \otimes L^{mn/p}_m\right)}_{\text{par}(p,\mu)} \end{cases} \tag{3.15}$$

$$\underbrace{(P \otimes I_n)}_{\text{par}(p,\mu)} \rightarrow \left(P \otimes I_{n/\mu}\right) \bar{\otimes} I_\mu \tag{3.16}$$

$$\underbrace{D}_{\text{par}(p,\mu)} \rightarrow \bigoplus_{i=0}^{p-1}{}_{\parallel} D_i \tag{3.17}$$

$$\underbrace{I_n \otimes A_m}_{\text{par}(p,\mu)} \rightarrow \sum_{k=0}^{p-1}{}_{\parallel} \mathrm{S}(h_{knm/p,1}) \left(\sum_{j=0}^{n/p-1} \mathrm{S}(h_{jm,1}) A_m \mathrm{G}(h_{jm,1}) \right) \mathrm{G}(h_{knm/p,1}) \tag{3.18}$$

$$\underbrace{A_m \otimes I_n}_{\text{par}(p,\mu)} \rightarrow \sum_{k=0}^{p-1}{}_{\parallel} S_{\mathrm{DT}}(q_{k,n/\mu,\mu}) \left(\sum_{j=0}^{\mu-1} \mathrm{S}(h_{j,\mu}) A_m \mathrm{G}(h_{j,\mu}) \right) G_{\mathrm{DT}}(q_{k,n/\mu,\mu}) \tag{3.19}$$

$$\underbrace{AB}_{\text{parBig}(p)} \rightarrow \underbrace{A}_{\text{parBig}(p)} \text{ barrier } \underbrace{B}_{\text{parBig}(p)} \tag{3.20}$$

$$\underbrace{I_n \otimes A_m}_{\text{parBig}(p)} \rightarrow I_p \otimes_{\parallel} \left(I_{n/p} \otimes A_m\right) \tag{3.21}$$

$$\underbrace{L^{mn}_m}_{\text{parBig}(p)} \rightarrow (I_p \otimes_{\parallel} L^{mn/p}_{m/p})(L^{p^2}_p \bar{\otimes} I_{mn/p^2})(I_p \otimes_{\parallel} (L^n_p \otimes I_{m/p})) \tag{3.22}$$

$$\underbrace{L^{mn}_m}_{\text{parBig}(p)} \rightarrow (I_p \otimes_{\parallel} L^m_{m/p} \otimes I_{n/p})(L^{p^2}_p \otimes I_{mn/p^2})(I_p \otimes_{\parallel} L^{mn/p}_{m/p}) \tag{3.23}$$

TABLE 3.1: Parallel paradigm rewrite rules. P is any permutation, D, D_i are diagonal matrices. The tag $\text{par}(p,\mu)$ specifies parallelization across p PEs, using a minimum data packet size of μ for inter-processor data exchanges. We assume $n/p \geq \mu$ for all the above. The tag $\text{parBig}(p)$ specifics parallelization across p PEs using our large packet algorithm. The construct $(L^{p^2}_p \otimes I_{mn/p^2})$ is a global all to all data exchange to be implemented appropriately using explicit memory access.

All the basic SPL constructs shown in Table 2.1 can be parallelized using these rules. For the basic parallelization algorithm, note that rules (3.13)–(3.17) are shown in SPL to aid the reader in understanding the loop transformations that occur. In reality, our system converts SPL constructs directly into Σ-SPL to create parallel loops and explicit scatters and gathers as required, as shown in (3.18)–(3.19).

it with computation. Streaming requires hardware support.

Data movement: background. Data access in computer systems is typically expensive: an un-cached load from memory can cost hundreds or thousands of cycles, while a general purpose processor can typically retire several floating point computations in each cycle. The memory hierarchy was designed to alleviate data access costs in uniprocessor systems where data must be moved between the processor and main memory. SPIRAL's initial efforts were focused on designing recursive FFT algorithms that took maximal advantage of the memory hierarchy.

FFT algorithms in distributed memory systems require data exchange amongst the PEs. Moving data to its destination quickly and before it is needed is important, and may significantly affect performance. Chip-based distributed memory systems like the Cell processor have relatively fast on-chip interconnects. To scale well, on-chip interconnects may have a ring (e.g., Cell BE), mesh (e.g., Tilera TILE64), or other scalable topologies, and may use packet-based switching networks. Contention on these topologies may impact interconnect performance. Cluster based distributed memory systems use inter-node interconnects like InfiniBand or Ethernet, where data movement costs are much higher, and may dominate FFT performance on these systems.

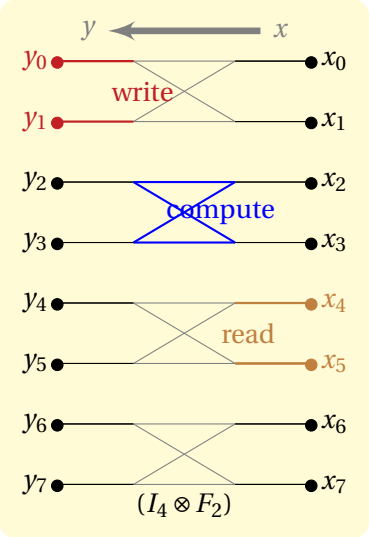
Streaming on modern hardware. To alleviate communication costs, modern distributed memory architectures may allow programs to programs to set up explicit data transfer requests (amongst the PEs and between the PEs and main memory) that will proceed in the background, in parallel with computation. For example, the Cell BE supports background DMA communications; the BlueGene/P includes processors that can be dedicated to servicing communication requests in the background. The MPI standard includes non-blocking MPI calls for this purpose. However, designing algorithms and writing programs to take advantage of streaming may not be trivial. We use our streaming paradigm to build FFT algorithms that take maximum advantage of such architectures.

Our main goal is to generate code that overlaps computation with communication for the four basic SPL constructs. We use our rewriting framework presented earlier to build our streaming paradigm.

3.4.1 Architecture Abstraction and Specification

Each of the basic SPL constructs involves a loop over a kernel. The computation consists of the kernel itself, and the communication consists of the required loads from the input vector and stores to the output vector. We assume the source and destination for the loads and stores is main memory, although it could also be other PEs. We define a new tag, called $stream()$ to specify that we want to adapt an SPL expression to the streaming paradigm. The following are our requirements:

- **Overlap of computation and communication.** The idea of streaming is similar to software pipelining, and involves computing the i'th iteration of tensor product loop while simultaneously storing the results of the $(i-1)^{\text{th}}$ iteration, and loading data for processing the $(i+1)^{\text{th}}$ iteration. This is illustrated in the following dataflow graph, which shows iteration 2 of $I_4 \otimes F_2$ being executed. Writes from iteration 1 and reads for iteration 3 are performed simultaneously.



- **Explicit memory access.** Generated code must include explicit memory instructions (message passing, DMA, etc.), similar to our parallel paradigm.

- **Size of communication packets.** Similar to the parallel paradigm, we must design our algorithm to produce exchanges of data in packets of a size that is optimal to our target architecture to minimize communication costs and maximize performance. Typically, this means producing large packet sizes. We specify our packet size using the tag parameter ψ.[4]

3.4.2 Identifying Mappable Constructs

We now identify constructs that map naturally well to streaming architectures, keeping in mind the goals we identified above.

The trivial case is $I_n \otimes A_m$. Normally, this involves computing the kernel A_m n times on the input vector. In our streamed case, we tag the tensor product using the symbol $\otimes_{\equiv}$ to show that

[4] We use μ to denote packet sizes used for inter-PE data transfers, and ψ to denote packet sizes used for data transfers between PEs and main memory.

we interpret it (at program generation time) as a loop where we compute the i'th iteration while simultaneously writing and reading the vectors for the previous and next iterations. Thus, constructs of the form:

$$I_s \otimes_{\equiv} A_m$$

are in their final form required for overlapped computation and communication. The following pseudo-code, written with a DMA-based programming paradigm (for an architecture such as the Cell BE) illustrates how this loop is implemented:

```
#define SWAP(x, y) {double* temp=x; x=y; y=temp;}

/* DMA_GET(dest, source, n) is a non-blocking DMA call to get n data
elements from source (in main memory) to dest (in local memory).

DMA_PUT(source, dest, n) is a non-blocking DMA call to put n data
elements from source (in local memory) to dest (in main memory).
*/

void A(double *y, double *x) { // Code that applies A to x to produce y }

void transform(double *Y, double *X) {
   // Header:
   i = 0;
   DMA_GET(altBufX, X[i*m], m);
   BLOCK_ON_DMA();
   SWAP(localX, altBufX);

   DMA_GET(altBufX, X[(i+1)*m], m);
   A(localY[0], localX[0]);
   BLOCK_ON_DMA();

   // Main body
   for(i=1; i<s-2; i++) {
     // Start non-blocking write of previous iteration's data
     SWAP(localY, altBufY);
     DMA_PUT(altBufY, Y[(i-1)*m], m);

     // Start non-blocking read of next iteration's data
     SWAP(localX, altBufX);
     DMA_GET(altBufX, X[(i+1)*m], m);

     // Compute current iteration
     A(localY[0], localX[0]);

     BLOCK_ON_DMA();
   }
```

```
    // Footer: (put s-2, compute s-1, put s-1)
    SWAP(localY, altBufY);
    DMA_PUT(altBufY, Y[(s-2)*m], m);

    SWAP(localX, altBufX);
    A(localY[0], localX[0]);

    BLOCK_ON_DMA();

    SWAP(localY, altBufY);
    DMA_PUT(altBufY, Y[(s-1)*m], m);
    BLOCK_ON_DMA();
}
```

Minimum iterations required. Note that to implement a streamed loop, the first and last iterations cannot involve overlap, as they are the setup and teardown stages, respectively. Thus, we need a minimum of 3 iterations in any streamed loop, giving us the constraint $s \geq 3$. When $s = 3$, only the middle iteration can be overlapped, while the first and last are not, which means only a third of the entire involves overlap. Therefore, in practice, we need more iterations for effective hiding of memory costs. Streamed loops with with 4 and 8 iterations can effectively overlap at most 50% and 75% of its iterations respectively.

Explicit memory access. To generate code with explicit memory accesses, we use the same technique as discussed earlier in Section 3.3.3. We only state the differences here. First, instead of the parallel iterative sum, we use a new iterative sum specially tagged for streaming, the $\sum^{\equiv}$, which corresponds to SPL's streaming tensor product (the $\otimes_{\equiv}$). Second, when necessary, we ensure that the scatters and gathers for explicit memory access are marked to communicate with main memory rather than with other PEs.

Size of communication packets. Permutations of the form $L \otimes I_\psi$ access memory in chunks of size ψ. Similar to our parallelization case, we use a tagged tensor product $\otimes_{\equiv}$, and recognize that constructs of the form:

$$P \bar{\otimes} I_\psi$$

are in their final form required for accessing memory in packets of size ψ.

3.4.3 Rewriting via Formula Manipulation

As shown above, constructs of the form $I_n \otimes A_m$ constitute our naturally mappable case, and are trivial to rewrite:

$$\underbrace{I_n \otimes A_m}_{\text{stream}(\psi)} \rightarrow I_s \otimes_{\equiv} (I_{n/s} \otimes A_m) \tag{3.24}$$

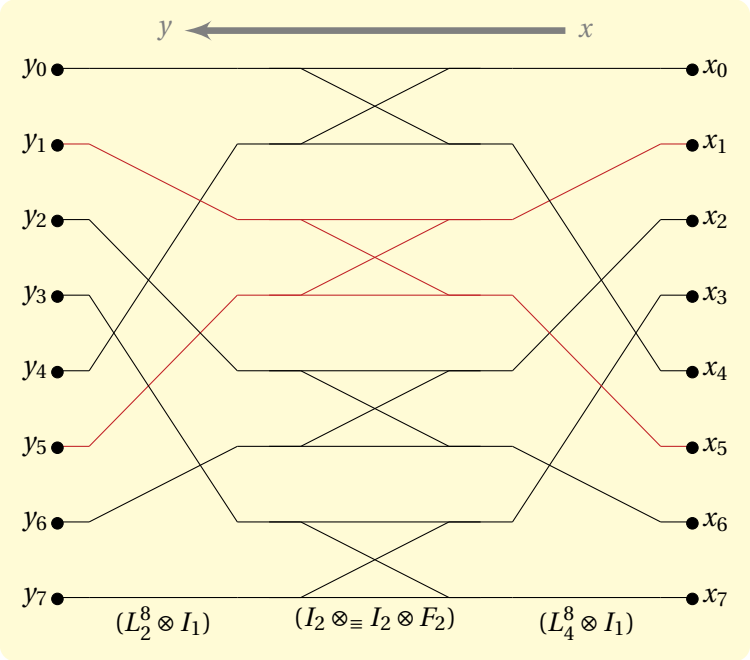
Here, we read and write in packets of size mn/s. A kernel of this size (mn/s) must also fit on-chip (in the local store or scratchpad or cache). Kernels smaller than a certain size (dependent on the architecture) might perform poorly.

To convert the remaining three basic constructs (as shown in Table 2.1) into the $I \otimes A$ form, we first examine the construct $A_m \otimes I_n$. If performed naively, for each iteration of the tensor product loop, we end up reading m packets of size 1 (complex element) each:

$$\underbrace{A_m \otimes I_n}_{\text{stream}(\psi)} \rightarrow L^{mn}_m (I_n \otimes_{\equiv} A_m) L^{mn}_n$$

The following dataflow graph example shows two reads and two writes of size 1 each to compute iteration 2 (highlighted in red) of $F_2 \otimes I_4$, as seen in the formula below:

$$\underbrace{F_2 \otimes I_4}_{\text{stream}(\psi)} \rightarrow L^8_2 (I_4 \otimes_{\equiv} F_2) L^8_4$$



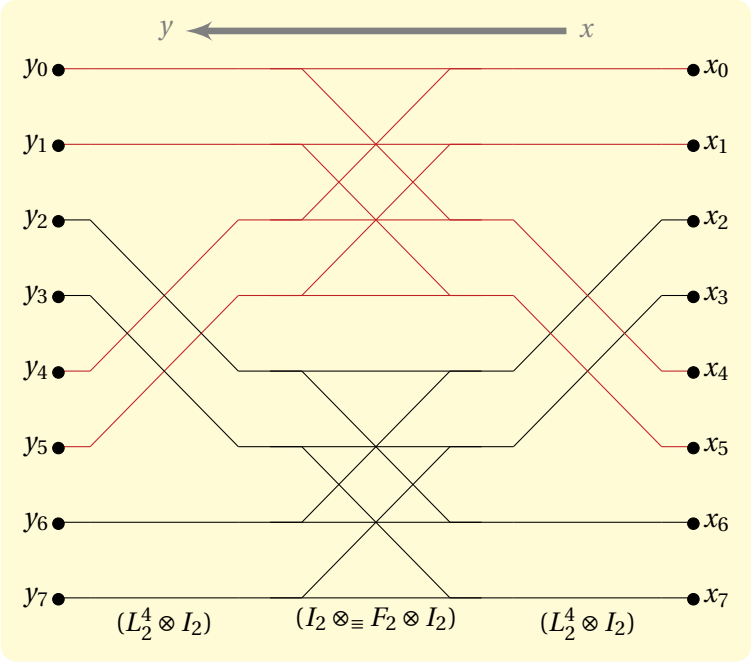
To achieve larger packet sizes, as done in the parallel paradigm, we process ψ iterations together, where ψ is the desired read/write packet size (note that ψ kernels must now fit in the local memory). When doing so, we bring in m single elements from the input vector which are spaced

at a stride of n:

$$\underbrace{A_m \otimes I_n}_{\text{stream}(\psi)} \rightarrow (L^{ms}_m \otimes_{\equiv} I_\psi)\big(I_s \otimes_{\equiv} (A_m \otimes I_\psi)\big)(L^{ms}_s \otimes_{\equiv} I_\psi) \tag{3.25}$$

where $\psi = n/s$. The leftmost and rightmost factors above are permutations that gather and scatter data from/to memory in packets of size ψ. Our previous example now contains two reads and two writes of size 2 each, to compute iteration 1 (which now computes two F_2 butterflies) of $F_2 \otimes I_4$, as seen in the formula below:

$$\underbrace{F_2 \otimes I_4}_{\text{stream}(\psi)} \rightarrow (L^4_2 \otimes_{\equiv} I_2)\big(I_2 \otimes_{\equiv} (F_2 \otimes I_2)\big)(L^4_2 \otimes_{\equiv} I_2)$$




```
void transform(double *Y, double *X) {
   // Header:
   i = 0;
   for(int j=0; j<m; j++)
     DMA_GET(altBufX, X[i*m], u);
   BLOCK_ON_DMA();
   SWAP(localX, altBufX);

   for(int j=0; j<m; j++)
     DMA_GET(altBufX, X[(i+1)*m], u);
   A(localY[0], localX[0]);
   BLOCK_ON_DMA();
```

```
  // Main body
  for(i=1; i<s-2; i++) {
    // Start non-blocking write of previous iteration's data
    SWAP(localY, altBufY);
    for(int j=0; j<m; j++)
      DMA_PUT(altBufY, Y[(i-1)*m], u);

    // Start non-blocking read of next iteration's data
    SWAP(localX, altBufX);
    for(int j=0; j<m; j++)
      DMA_GET(altBufX, X[(i+1)*m], u);

    // Compute current iteration
    A(localY[0], localX[0]);

    BLOCK_ON_DMA();
  }

  // Footer: (put s-2, compute s-1, put s-1)
  SWAP(localY, altBufY);
  for(int j=0; j<m; j++)
    DMA_PUT(altBufY, Y[(s-2)*m], u);

  SWAP(localX, altBufX);
  A(localY[0], localX[0]);

  BLOCK_ON_DMA();

  SWAP(localY, altBufY);
  for(int j=0; j<m; j++)
    DMA_PUT(altBufY, Y[(s-1)*m], u);
  BLOCK_ON_DMA();
}
```

The other two remaining basic SPL constructs are streamed similarly, using the technique of combining multiple kernels to increase packet size.

Interaction with parallelization and vectorization. Streamed algorithms may also have to be parallelized and/or vectorized. Streaming combines well with the parallelization paradigm, as seen by the similarity of the rewrite rules (in Table 3.1 and Table 3.2). Performing both paradigms on an SPL construct involves splitting loops between streaming iterations and parallel iterations. The splitting order determines the type of code generated, and is discussed in detail in Chapter 4. Streaming also combines well with the vectorization paradigm: vectorization and streaming transform the construct in an orthogonal manner, with the streaming tag always being the

outer one:

$$\underbrace{\underbrace{I_n \otimes A_m}_{\text{vec}(\nu)}}_{\text{stream}(\psi)} \rightarrow I_s \otimes_{\equiv} \underbrace{I_{n/s} \otimes A_m}_{\text{vec}(\nu)}$$

$$\underbrace{\underbrace{A_m \otimes I_n}_{\text{vec}(\nu)}}_{\text{stream}(\psi)} \rightarrow (L^{ms}_m \bar{\otimes} I_\psi)\Big(I_s \otimes_{\equiv} \underbrace{(A_m \otimes I_\psi)}_{\text{vec}(\nu)}\Big)(L^{ms}_s \bar{\otimes} I_\psi)$$

The standard vectorization rules apply to remove the vectorization tag on the right hand side of both the above rules. In the first case, unavoidable permutations are created to perform vectorization, while the second case is already a naturally vectorized construct as long as $\psi \geq \nu$, (where ψ is the streaming packet size, and ν is the SIMD vector length), which typically is the case.

3.4.4 Summary

In summary, our streaming paradigm optimizes the structure of tensor products for streamed execution, with data being read and written with a minimum specified packet size. Our stream tag is parameterized by stream(ψ), where ψ is the minimum packet size used by loads and stores.

We rewrite all SPL constructs into the two basic mappable constructs $I_s \otimes_{\equiv} A_m$ (streamed loop, interpreted accordingly at program generation time), and $(L^{mn}_m \otimes_{\equiv} I_\psi)$, which reads or writes data with a packet size of ψ. Our rewrite rules are summarized in Table 3.2.

3.5 Chapter Summary

In this chapter, we presented an abstraction of our target architecture, followed by details of actual platforms used to evaluate the ideas in this thesis. We then identified the two most fundamental architectural paradigms that we optimize for, which are the parallelism and the streaming paradigms. Finally, we presented SPE constructs that naturally map well to each of the paradigms, and showed how to rewrite all of the other basic SPL constructs into these natural paradigms using rewrite rules. In the next chapter, we will build directly upon the ideas presented in this chapter to construct algorithms for the DFT that are optimized for our target architectures.

$$\underbrace{AB}_{\text{stream}(\psi)} \rightarrow \underbrace{A}_{\text{stream}(\psi)} \text{ barrier } \underbrace{B}_{\text{stream}(\psi)} \tag{3.26}$$

$$\underbrace{I_n \otimes A_m}_{\text{stream}(\psi)} \rightarrow I_b \otimes_{\equiv} \left(I_{n/b} \otimes A_m\right), nm/b \geq \psi \tag{3.27}$$

$$\underbrace{A_m \otimes I_n}_{\text{stream}(\psi)} \rightarrow \underbrace{\left(L^{mb}_m \otimes I_{n/b}\right)\left(I_b \otimes (A_m \otimes I_{n/b})\right)\left(L^{mb}_b \otimes I_{n/b}\right)}_{\text{stream}(\psi)}, n/b \geq \psi \tag{3.28}$$

$$\underbrace{L^{mn}_m}_{\text{stream}(\psi)} \rightarrow \begin{cases} \underbrace{\left(I_b \otimes L^{mn/b}_{m/b}\right)}_{\text{stream}(\psi)} \underbrace{\left(L^{bn}_b \otimes I_{m/b}\right)}_{\text{stream}(\psi)} \\ \underbrace{\left(L^{bm}_m \otimes I_{n/b}\right)}_{\text{stream}(\psi)} \underbrace{\left(I_b \otimes L^{mn/b}_m\right)}_{\text{stream}(\psi)} \end{cases} \tag{3.29}$$

$$\underbrace{(P \otimes I_n)}_{\text{stream}(\psi)} \rightarrow \left(P \otimes I_{n/\psi}\right) \bar{\otimes} I_\psi, \tag{3.30}$$

$$\underbrace{D}_{\text{stream}(\psi)} \rightarrow \bigoplus_{i=0}^{p-1}{}_{\parallel} D_i \tag{3.31}$$

$$\underbrace{I_n \otimes A_m}_{\text{stream}(\psi)} \rightarrow \sum_{k=0}^{p-1}{}_{\equiv} S_{\mathrm{DT}}(q_{knm/p,1}) \left(\sum_{j=0}^{n/p-1} \mathrm{S}(h_{jm,1}) A_m \mathrm{G}(h_{jm,1}) \right) G_{\mathrm{DT}}(q_{knm/p,1}) \tag{3.32}$$

$$\underbrace{A_m \otimes I_n}_{\text{stream}(\psi)} \rightarrow \sum_{k=0}^{p-1}{}_{\equiv} S_{\mathrm{DT}}(q_{k,n/\psi,\mu}) \left(\sum_{j=0}^{\psi-1} \mathrm{S}(h_{j,\mu}) A_m \mathrm{G}(h_{j,\mu}) \right) G_{\mathrm{DT}}(q_{k,n/\psi,\mu}) \tag{3.33}$$

$$\tag{3.34}$$

TABLE 3.2: Streaming paradigm rules. P is any permutation, D, D_i are diagonal matrices. The tag stream(ψ) specifies streaming using a minimum data packet size of ψ for loads and stores. We assume $n/p \geq \psi$ for all the above.

CHAPTER 4

Designing Platform-Optimized FFTs

In the previous chapter, we developed rules to rewrite SPL expressions to fit two architectural paradigms fundamental to this thesis: parallelization and streaming. In this chapter, we use these rewrite rules to design FFTs that are tailored to our target platforms. We first present some usage scenarios to show the types of code we want to generate. Then, we discuss FFT code optimized for latency, followed by FFT code optimized for throughput.

Base algorithms. The streaming and parallel FFTs designed in this chapter are derived from the generalized Cooley-Tukey FFT (2.12) for 1D DFTs, and on the row-column algorithm (2.15) for 2D DFTs. Both are reproduced here for convenience.

Cooley-Tukey 1D DFT breakdown rule:

$$\mathrm{DFT}_{mn} \rightarrow (\mathrm{DFT}_n \otimes I_m) D_{n,m} (I_n \otimes \mathrm{DFT}_m) L^{nm}_n. \tag{4.1}$$

Row-column 2D DFT breakdown rule:

$$\mathrm{DFT}_{m \times n} \rightarrow \mathrm{DFT}_m \otimes \mathrm{DFT}_n \rightarrow (\mathrm{DFT}_m \otimes I_n)(I_m \otimes \mathrm{DFT}_n). \tag{4.2}$$

Problem sizes and constraints. In this thesis, we focus on two-power sizes. Since these are composite sizes, the Cooley-Tukey FFT is applicable. We also assume that the number of PEs divides the problem size. In addition, when SIMD vectorization is required, we assume that the problem size is compatible with the underlying vectorization technique used. Usually, this means that the size is a multiple of the square of the SIMD vector length, as discussed in [Franchetti et al., 2006b].

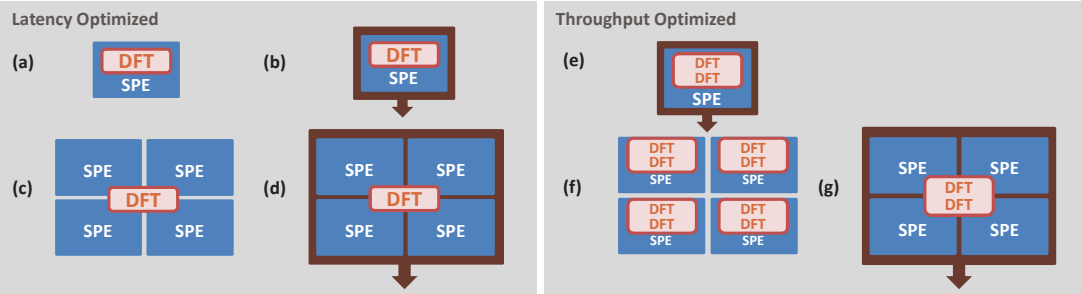


FIGURE 4.1: Usage scenarios: (a)–(d) have to be optimized for latency (notice that only a single DFT kernel is performed in these cases); (e)–(g) have to be optimized for throughput (multiple DFTs are performed). The box with the arrow around the PEs denotes operations on vectors resident in the memory node.

4.1 Usage Scenarios

When designing a high-performance implementation of the DFT, application demands have to be considered. In this section, we classify the typical usage scenarios to help us guide our algorithm construction in the rest of the chapter.

An application might require a library that takes into account the following options:

- *Latency versus throughput optimization:* An application might require either a latency-optimized version to compute a single DFT that is in the critical path of operations to be performed on the input data, or might require a throughput-optimized version to compute a large number of DFTs on a stream of input data.
- *Input and output data resident in local memories versus data resident in main memory:* This distinction typically exists in chip-based instantiations of our distributed memory architecture abstraction.
- *A specified number of PEs to use versus determining and using an optimal number of PEs:* An application may demand a specific number of PEs to be used regardless of the PEs available on the platform.

The combination of such requirements and other specifications such as the size or data format of the DFT and the number of PEs allocated gives rise to a large number of possible usage scenarios as depicted in Figure 4.1. A small latency-optimized DFT kernel can work on vectors resident on a single PE as shown (a). Medium sizes DFTs must be parallelized, and can work on vectors resident either across multiple PEs or stored in the memory node as shown in (c) and (d). Large DFTs must be streamed in and out of the PEs and computed in parts as shown in

(b). Throughput-optimized versions can use streaming to hide memory access costs, see (e), or execute small DFTs independently on multiple PEs as shown in (f), or stream larger DFTs parallelized across multiple PEs as shown in (g).

In Section 4.2, we construct latency optimized FFTs for input vectors distributed across the memories of the PEs. In Section 4.3, we consider input vectors resident on the memory node. In Section 4.4, we construct throughput-optimized distributed memory FFTs. Finally, in Section 4.5, we show how the specifications for the various usage scenarios shown in Figure 4.1 are implemented in our program generation system.

4.2 Latency Optimized Distributed DFTs

In this section, we parallelize our FFTs for distributed memory systems, where the input and output vectors for the DFT are distributed across the PEs on which they will be performed.

We first construct a parallelized FFT using our rewrite rules built in Chapter 3. We then discuss three ways to improve the performance of this algorithm, including different data distributions, large packet sizes, and overlapped communication.

4.2.1 Basic Parallelization

We use rewrite rules from Section 3.3 to design our parallelized 1D and 2D FFTs. We assume that the input data is distributed across p PEs. This means the input array of size n is divided into p equal chunks of size n/p each (we assume p divides n), and the i'th PE contains the i'th chunk of the array in its local memory. In some cases, computing using a smaller or larger number of PEs than the number of PEs the array is distributed on may be beneficial. Such rescaling techniques were addressed in [Bonelli et al., 2006], but are not considered in this thesis.

In the following sections, we first demonstrate our techniques on the 1D DFT, and then show how they can also be applied to the 2D DFT.

Notation for twiddle factors. Since twiddle factors in the DFT only introduce a multiplication by a constant, they do not affect our optimizations. Hence, we use the following abuse of notation for convenience:

$$\begin{aligned} \mathrm{DFT}_n^{\circ} &= (\mathrm{DFT}_n^{(i)} \otimes_i I_m) \\ &= (\mathrm{DFT}_n \otimes I_m) D_{n,m}. \end{aligned}$$

Representation of context. In the rest of this chapter, we frequently point to parts of an SPL expression to show what they mean. To aid in this process, we provide contextual information of

an SPL expression by using braces *above* of the expression. Braces *under* an expression always refer to tags. For example:

$$\overbrace{\underbrace{A}_{\text{tag()}}}^{\text{context}}$$

Basic parallelization. We begin by parallelizing the Cooley-Tukey 1D DFT shown in (4.1). We tag the left hand side of (4.1) to specify that we want to parallelize it across p PEs with data exchanges performed at a minimum packet size of μ. We apply (4.1) and use (3.12) to distribute the tags among the SPL constructs, and then remove the tags by using the rules from Table 3.1.

$$\begin{aligned}
\underbrace{\mathrm{DFT}_{mn}}_{\mathrm{par}(p,\mu)} &\to \underbrace{(\mathrm{DFT}_n \otimes I_m)}_{\mathrm{par}(p,\mu)}\ \underbrace{D_{n,m}}_{\mathrm{par}(p,\mu)}\ \underbrace{(I_n \otimes \mathrm{DFT}_m) L^{nm}_n}_{\mathrm{par}(p,\mu)} \\
&\to \underbrace{(\mathrm{DFT}^{\circ}_n \otimes I_m)}_{\mathrm{par}(p,\mu)}\ \underbrace{(I_n \otimes \mathrm{DFT}_m) L^{nm}_n}_{\mathrm{par}(p,\mu)}, \\
&\to \Big(\overbrace{(L^{np}_n \bar{\otimes} I_{m/p})}^{\text{comm}}\ \overbrace{(I_p \otimes_{\parallel} (\mathrm{DFT}^{\circ}_n \otimes I_{m/p}))}^{\text{comp}}\ \overbrace{(L^{np}_p \bar{\otimes} I_{m/p})}^{\text{comm}}\Big) \\
&\quad \overbrace{\Big((I_p \otimes_{\parallel} (I_{n/p} \otimes \mathrm{DFT}_m) L^{nm/p}_{n/p})}^{\text{comp}}\ \overbrace{(L^{pm}_p \bar{\otimes} I_{n/p})\Big)}^{\text{comm}}
\end{aligned} \tag{4.3}$$

Parallel stages. The algorithm in (4.3) is composed of alternating stages of computation and communication (global data exchanges), and thus has the form of bulk synchronous processing parallel programming model described in [Valiant, 1990]. The first step in (4.3), which is the rightmost $(L^{pm}_p \bar{\otimes} I_{n/p})$ performs an all-to-all communication to redistribute data amongst the PEs. (4.3) then performs four alternating stages of computation and communication. This is visualized in the final line of (4.3), where the overbraces mark the stages with "comp" for computation and "comm" for communication.

Observe that in (4.3), all the computation stages are p-way parallel, as described in Section 2.4.2, and the communication stages exchange data packets in blocks of size m/p (last two) or n/p (first one), as explained in Section 3.3.2.

Composing parallelized SPL constructs. When multiple parallel SPL constructs are composed, the following issues arise:

- **Barriers.** As (3.12) shows, barriers are required between constructs, due to data dependencies.
- **Address space.** We use the following rules to generate code for (4.3) that uses a distributed

memory address space:

1. A construct of the form $I_p \otimes_{\parallel} A$ is performed in parallel across p PEs, with A being performed in the local address space of each PE.
2. a DFT° that appears as a part of A in constructs of the form $I_p \otimes_{\parallel} A$ must be parameterized by p (not shown), so as to use the appropriate portion of the twiddle factors, as explained earlier. The entire set of twiddle factors are distributed among the PEs.
3. A construct of the form $P\bar{\otimes} I_\mu$ is a global permutation, and thus involves an all to all communication. We interpret the construct to read and write to a virtual global shared address space that is partitioned into p chunks and assigned in a block distribution to each of the p PEs.

- **Combining data exchanges.** If multiple data exchange stages appear between two computation stages, they are optimally implemented by first combining them using standard tensor product rules into a single communication stage. For example, assuming A and B are computation stages, P,Q permutations, and $\mu_2 \leq \mu_1$:

$$A \cdot (P\bar{\otimes} I_{\mu_1}) \cdot (Q\bar{\otimes} I_{\mu_2}) \cdot B \rightarrow A \cdot \left((P \otimes I_{\mu_1/\mu_2}) Q\bar{\otimes} I_{\mu_2}\right) \cdot B$$

  (A, B are compute kernels, P, Q are stride permutations) combines to a single exchange with the minimum packet size of the two.

- **Pushing versus pulling data exchanges.** In "push model" architectures, a single communication stage between two computation stages is attached to the earlier stage as its store operation, while in a "pull model architecture", it is attached to the later stage as its load operation

Increasing performance. To further increase the performance of algorithm (4.3), we must eliminate bottlenecks in each of the computation, and each of the communication stages. We use previous work ([Franchetti et al., 2006b]) to produce fast vectorized kernels for the computation stages, and therefore assume these are optimized. We are thus left with the task of optimizing the communication stages in this thesis. Communication can be sped up by three possible factors:

- One or more the *communication stages can be eliminated* by using an initial and final data distribution optimal for performance. This is covered next in Section 4.2.2.

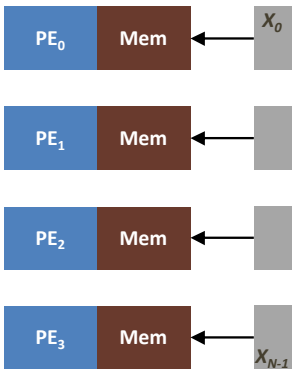


FIGURE 4.2: Block data distribution example: 4 PEs.

- On most platforms, achieved bandwidth is increased when *optimal packet sizes* are used. Typically, this means using large packets, which amortized packet overhead costs. We derive an FFT that cheaply assembles small packets to form large packets before sending them. This is covered in Section 4.2.3.
- Communication costs can also be reduced by hiding them partially or fully by *overlapping them with computation.* This is covered in Section 4.3 and Section 4.4.

4.2.2 Data Distribution

So far, we have assumed that the input and output vectors follows a block distribution. An example of a block distribution for four PEs is shown in Figure 4.2.

Applications may either demand different data distributions, or may accept other data distributions in exchange for higher performance. We can modify our parallel algorithm to handle other data distributions simply by adding permutations to to the beginning and to the end as necessary, and using loop merging techniques from [Franchetti et al., 2005] to minimize the cost of such permutations. Changing the data layout is equivalent to composing the input and the output of an SPL formula with permutations. Therefore, instead of computing the DFT, we instead compute:

$$\mathrm{DFT}^1 = P^{-1} \times \mathrm{DFT} \times Q \tag{4.4}$$

Based on this, we note that when we choose P and Q appropriately, we can effectively cancel out the initial and final permutations (with the general structure $L \otimes \mathrm{DFT}$) in (4.3), and thus increasing performance by eliminating two all-to-all data exchanges. Such a distribution is called the *block-cyclic data distribution*, discussed below.

Block cyclic data distribution. In this type of distribution, visualized below, a vector of size

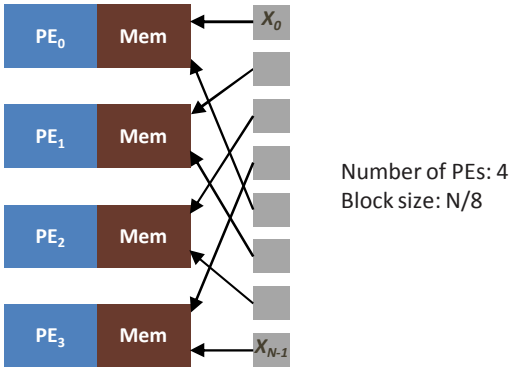


FIGURE 4.3: Block-cyclic data distribution example: 4 PEs, block size of $N/8$. This means that the vector of size N is divided into 8 blocks of size $N/8$ each, and each block is assigned in a cyclic fashion to the PEs.

N is divided into k-sized blocks, and distributed in a cyclic fashion among p PEs. An example with a block size of $N/8$ distributed in a block cyclic fashion among 4 PEs is shown in Figure 4.3.

To use this data distribution, we conjugate our FFT with $(L^{N/k}_p \otimes I_k)$. (Conjugating A with a permutation P means composing A with the permutation and its inverse to achieve $P^{-1}AP$). In the special case where $N/k = mp$, and $k = m/p$ in (4.3), this results in the first and last communication stages of (4.3) being cancelled out, which can significantly improve performance:

$$\underbrace{\mathrm{DFT}_{m^2}}_{\mathrm{par}(p,\mu)} \rightarrow$$

$$\left((L^{mp}_m \bar{\otimes} I_{m/p})(I_p \otimes_\parallel (\mathrm{DFT}^\circ_m \otimes I_{m/p}))(L^{mp}_p \bar{\otimes} I_{m/p})(I_p \otimes_\parallel (I_{m/p} \otimes \mathrm{DFT}_m) L^{m^2/p}_{m/p})(L^{pm}_p \bar{\otimes} I_{m/p})\right)^{(L^{pm}_m \otimes I_{m/p})}$$
$$\rightarrow \left((I_p \otimes_\parallel (\mathrm{DFT}^\circ_m \otimes I_{m/p}))(L^{mp}_p \bar{\otimes} I_{m/p})(I_p \otimes_\parallel (I_{m/p} \otimes \mathrm{DFT}_m) L^{m^2/p}_{m/p})\right)$$

Generalized data distributions. The same general idea applies when we want to use other data distributions, which can be integrated into our framework. We now consider different types of data distributions based on stride permutations. Specifically, assuming L is a stride permutation:

- $(I_p \otimes I_{n/p}) = I_n$ is a block distribution (default),
- P is a cyclic distribution,
- $(P \otimes I_\mu)$ is a block-cyclic distribution of block size μ (a common distribution),
- $(I_p \otimes_\parallel P)$ is a locally-cyclic distribution,

- $(I_p \otimes_{\parallel} (P \otimes I))$ is a locally-block-cyclic distribution

Using the above, we can generate code for various data distributions. Matching input and output distributions, provided by a conjugate pair of permutations, are called symmetric distributions. We can also easily generate code for asymmetric distributions. Note that asymmetric distributions can also include the same *type* of distribution for both the input and output, but with different *parameters* (e.g., a block-cyclic distribution with different block sizes for the input and output).

To summarize, we generate code for various data distributions by applying the appropriate permutations, and using loop merging (from [Franchetti et al., 2005]) to minimize the cost of the permutations. Further, we can also identify distributions that will enhance performance by eliminating the initial, final, or both communication stages.

4.2.3 Increasing Data Exchange Packet Size

In (4.3), the first communication state uses a packet size of n/p, while the second and the third use a size of m/p. Since $mn = N$ where N is the problem size of the DFT, for a given p, increasing m results in a decrease of n and vice versa. To achieve packet sizes that are approximately equal in all the communication stages, we must follow a square root decomposition (exact or approximate) of N into m and n. This still results in packet sizes for the 1D DFT that do not scale very well with increasing p, as shown in Figure 3.7.

On many platforms, the overhead of sending a packet is very high until a minimum threshold size is reached, as shown in Figure 3.6. In this section, we provide an alternate parallelized algorithm that produces larger packets. To do so, we assemble all packets to be sent from one PE to another into one large packet, send it, and disassemble it on the receiving end, as was shown in (3.5). We first present some permutation identities required to build our algorithm, and then present the scalar and SIMD vectorized versions of the large packet algorithm.

Permutations identities. We will use the following identities in our large packet algorithm:

For matrices A,B,C and a stride permutation P,

$$(ABC)^\top = C^\top B^\top A^\top \tag{4.5}$$

$$(A \otimes B)^\top = A^\top \otimes B^\top \tag{4.6}$$

$$(A \otimes B)^{-1} = A^{-1} \otimes B^{-1} \tag{4.7}$$

$$P^{-1} = P^\top \tag{4.8}$$

$$L^{mn}_m = (I_p \otimes_\parallel L^{mn/p}_{m/p})(L^{p^2}_p \otimes I_{mn/p^2})(I_p \otimes_\parallel L^n_p \otimes I_{m/p}) \tag{4.9}$$

$$\begin{aligned} L^{mn}_m &= ((L^{mn}_m)^{-1})^{-1} = ((L^{mn}_m)^\top)^\top = (L^{mn}_n)^\top \\ &= \left((I_p \otimes_\parallel L^{mn/p}_{n/p})(L^{p^2}_p \otimes I_{mn/p^2})(I_p \otimes_\parallel L^m_p \otimes I_{n/p})\right)^\top \\ &= (I_p \otimes_\parallel L^m_p \otimes I_{n/p})^\top (L^{p^2}_p \otimes I_{mn/p^2})^\top (I_p \otimes_\parallel L^{mn/p}_{n/p})^\top \end{aligned}$$

$$L^{mn}_m = (I_p \otimes_\parallel (L^m_{(m)/p} \otimes I_{(n)/p}))(L^{p^2}_p \otimes I_{((m)n)/p^2})(I_p \otimes_\parallel L^{((m)n)/p}_m) \tag{4.10}$$

Large packet algorithm. We begin by reproducing the basic large packet algorithm from [Bonelli et al., 2006]. As in Chapter 3, we use the tag $\underbrace{A}_{\text{parBig}(p)}$ to indicate parallelization of A using the large packet algorithm:

$$\begin{aligned} \underbrace{\text{DFT}_{mn}}_{\text{parBig}(p)} &\rightarrow \underbrace{(\text{DFT}_m \otimes I_n)}_{\text{parBig}(p)} \underbrace{D_{m,n}}_{\text{parBig}(p)} \underbrace{(I_m \otimes DFT_n) L^{mn}_m}_{\text{parBig}(p)} \\ &\rightarrow \underbrace{L^{mn}_m}_{\text{parBig}(p)} \underbrace{(I_n \otimes \text{DFT}_m)}_{\text{parBig}(p)} \underbrace{L^{mn}_n}_{\text{parBig}(p)} \underbrace{D_{m,n}}_{\text{parBig}(p)} \underbrace{(I_m \otimes \text{DFT}_n)}_{\text{parBig}(p)} \underbrace{L^{mn}_m}_{\text{parBig}(p)} \\ &\rightarrow \overbrace{(I_p \otimes_\parallel L^{((m)n)/p}_{(m)/p})}^{\text{local perm}} \overbrace{(L^{p^2}_p \otimes I_{((m)n)/p^2})}^{\text{comm}} \overbrace{(I_p \otimes_\parallel (L^n_p \otimes I_{(m)/p}))}^{\text{local perm}} \\ &\quad \overbrace{(I_p \otimes_\parallel I_{n/p} \otimes \text{DFT}^\circ_m)}^{\text{comp}} \\ &\quad \overbrace{(I_p \otimes_\parallel L^{((n)m)/p}_{(n)/p})}^{\text{local perm}} \overbrace{(L^{p^2}_p \otimes I_{((n)m)/p^2})}^{\text{comm}} \overbrace{(I_p \otimes_\parallel (L^m_p \otimes I_{(n)/p}))}^{\text{local perm}} \\ &\quad \overbrace{(I_p \otimes_\parallel I_{m/p} \otimes \text{DFT}_n)}^{\text{comp}} \\ &\quad \overbrace{(I_p \otimes_\parallel L^{((m)n)/p}_{(m)/p})}^{\text{local perm}} \overbrace{(L^{p^2}_p \otimes I_{((m)n)/p^2})}^{\text{comm}} \overbrace{(I_p \otimes_\parallel (L^n_p \otimes I_{(m)/p}))}^{\text{local perm}} \end{aligned}$$

In the above, in addition to marking the communication and computation parts, we mark expressions with "local perm" to indicate that an SPL expression is a local permutation, which means each PE locally permutes its own data.

Rearranging, and using (3.11) for the first factor, we get:

$$
\begin{aligned}
&\rightarrow (I_p \otimes_\parallel (L^m_{(m)/p} \otimes I_{(n)/p}))\overbrace{(L^{p^2}_p \otimes I_{mn/p^2})}^{\text{comm}} \\
&(I_p \otimes_\parallel L^{((m)n)/p}_m)(I_p \otimes_\parallel I_{n/p} \otimes \mathrm{DFT}^\circ_m)(I_p \otimes_\parallel L^{((n)m)/p}_{(n)/p})\overbrace{(L^{p^2}_p \otimes I_{mn/p^2})}^{\text{comm}} \\
&(I_p \otimes_\parallel (L^m_p \otimes I_{(n)/p}))(I_p \otimes_\parallel I_{m/p} \otimes \mathrm{DFT}_n)(I_p \otimes_\parallel L^{((m)n)/p}_{(m)/p})\overbrace{(L^{p^2}_p \otimes I_{mn/p^2})}^{\text{comm}} \\
&(I_p \otimes_\parallel (L^n_p \otimes I_{(m)/p})) \qquad (4.11) \\
&\rightarrow \overbrace{(I_p \otimes_\parallel (L^m_{(m)/p} \otimes I_{(n)/p}))}^{\text{local perm}}\overbrace{(L^{p^2}_p \otimes I_{mn/p^2})}^{\text{comm}} \\
&\overbrace{(I_p \otimes_\parallel (\mathrm{DFT}^\circ_m \otimes I_{n/p}))}^{\text{comp}}\overbrace{(L^{p^2}_p \otimes I_{mn/p^2})}^{\text{comm}} \\
&\overbrace{(I_p \otimes_\parallel ((L^m_p \otimes I_{(n)/p})(I_{m/p} \otimes \mathrm{DFT}_n)(L^{((m)n)/p}_{(m)/p}))}^{\text{comp}}\overbrace{(L^{p^2}_p \otimes I_{mn/p^2})}^{\text{comm}} \\
&\overbrace{(I_p \otimes_\parallel (L^n_p \otimes I_{(m)/p}))}^{\text{local perm}} \qquad (4.12)
\end{aligned}
$$

Notice in the above that we have now merged in the local permutations with adjoining computation stages whenever possible. We use loop merging techniques from [Franchetti et al., 2005] to accomplish this merging in practice. Local permutations that do not have an adjoining computation block (the first and the last local permutations) must be performed separately. Also notice that during each communication stage, each PE sends one single packet of size mn/p^2 to every other PE, as explained in Chapter 3.

Vectorized large packet algorithm. Below, we derive a ν-way SIMD vectorized variant of (4.12):

$$
\begin{aligned}
\underbrace{\mathrm{DFT}_{mn}}_{\mathrm{parBig}(p,v)} &\to \underbrace{(\mathrm{DFT}_m \otimes I_n)}_{\mathrm{parBig}(p,v)} \underbrace{D_{m,n}}_{\mathrm{parBig}(p,v)} \underbrace{(I_m \otimes DFT_n) L^{mn}_m}_{\mathrm{parBig}(p,v)} \\
&\to \underbrace{(L^{mn/v}_m \otimes I_v)}_{\mathrm{parBig}(p,v)} \underbrace{(I_{n/v} \otimes \mathrm{DFT}^\circ_m \otimes I_v)}_{\mathrm{parBig}(p,v)} \underbrace{(L^{mn/v}_{n/v} \otimes I_v)}_{\mathrm{parBig}(p,v)} \underbrace{(I_{m/v} \otimes (I_v \otimes \mathrm{DFT}_n) L^{nv}_v)}_{\mathrm{parBig}(p,v)} \underbrace{(L^{mn/v}_{n/v} \otimes I_v)}_{\mathrm{parBig}(p,v)} \\
&\to \underbrace{(I_p \otimes_\parallel (L^{((n/v)m)/p}_{n/v/p} \otimes I_v))(L^{p^2}_p \otimes I_{(n/v)m/p^2} \otimes I_v)(I_p \otimes_\parallel (L^m_p \otimes I_{n/v/p} \otimes I_v))}_{\mathrm{vec}(v)} \\
&\quad \underbrace{(I_p \otimes_\parallel I_{n/pv} \otimes \mathrm{DFT}^\circ_m \otimes I_v)}_{\mathrm{vec}(v)} \\
&\quad \underbrace{(I_p \otimes_\parallel (L^{((m)n/v)/p}_{m/p} \otimes I_v))(L^{p^2}_p \otimes I_{(m)n/v/p^2} \otimes I_v)(I_p \otimes_\parallel (L^{n/v}_p \otimes I_{m/p} \otimes I_v))}_{\mathrm{vec}(v)} \\
&\quad \underbrace{(I_p \otimes_\parallel I_{m/pv} \otimes (I_v \otimes \mathrm{DFT}_n) L^{nv}_v)}_{\mathrm{vec}(v)} \\
&\quad \underbrace{(I_p \otimes_\parallel (L^{((n)m/v)/p}_{n/p} \otimes I_v))(L^{p^2}_p \otimes I_{(n)m/v/p^2} \otimes I_v)(I_p \otimes_\parallel (L^{m/v}_p \otimes I_{n/p} \otimes I_v))}_{\mathrm{vec}(v)} \\
&\to \overbrace{\underbrace{(I_p \otimes_\parallel (L^{((n/v)m)/p}_{(n/v)/p} \otimes I_v))}_{\mathrm{vec}(v)}}^{\text{local perm}} \overbrace{(L^{p^2}_p \otimes I_{mn/p^2})}^{\text{comm}} \\
&\quad \overbrace{\underbrace{(I_p \otimes_\parallel (L^m_p \otimes I_{(n/v)/p} \otimes I_v))(I_p \otimes_\parallel I_{n/pv} \otimes \mathrm{DFT}^\circ_m \otimes I_v)(I_p \otimes_\parallel (L^{((m)n/v)/p}_{(m)/p} \otimes I_v))}_{\mathrm{vec}(v)}}^{\text{comp}} \\
&\quad \overbrace{(L^{p^2}_p \otimes I_{mn/p^2})}^{\text{comm}} \\
&\quad \overbrace{\underbrace{(I_p \otimes_\parallel (L^{n/v}_p \otimes I_{(m)/p} \otimes I_v))(I_p \otimes_\parallel I_{m/pv} \otimes (I_v \otimes \mathrm{DFT}_n) L^{nv}_v)(I_p \otimes_\parallel (L^{((n)m/v)/p}_{(n)/p} \otimes I_v))}_{\mathrm{vec}(v)}}^{\text{comp}} \\
&\quad \overbrace{(L^{p^2}_p \otimes I_{mn/p^2})}^{\text{comm}} \\
&\quad \overbrace{\underbrace{(I_p \otimes_\parallel (L^{m/v}_p \otimes I_{(n)/p} \otimes I_v))}_{\mathrm{vec}(v)}}^{\text{local perm}}
\end{aligned}
$$

This is a SIMD vectorizable version of (4.12) based on our rules developed and presented in

Table 3.1. This can be seen by all SPL expressions above (except the second computation stage, which must be vectorized as is) having the general form $A \otimes I_v$, which is naturally vectorizable as discussed in Section 2.4.2. SIMD vectorization must be taken into account when parallelizing our algorithm. If SIMD vectorization is performed separately at a later stage, additional unnecessary permutations may be introduced, which would affect performance. Note that our final algorithm above includes a vector tag that is removed using rules from previous work in [Franchetti et al., 2006b].

4.2.4 2D DFTs

So far, we have looked at parallelization of 1D FFTs. Parallelization of 2D FFTs is similar, except for the two following differences. First, we use the row-column algorithm (4.2) as the basic algorithm. Second, because of the structure of the row column algorithm, we only have two global permutations, which means we only have two global exchanges of data as opposed to three in the 1D case. We can use either our basic parallelization algorithm presented in Section 4.2.1 or use our large packet algorithm presented in Section 4.2.3 to perform our parallelization. We show each of these below.

Basic parallelization of the 2D DFT:

$$\underbrace{\mathrm{DFT}_{m\times n}}_{\mathrm{par}(p,\mu)} \rightarrow \underbrace{(\mathrm{DFT}_m \otimes I_n)}_{\mathrm{par}(p,\mu)}\underbrace{(I_m \otimes \mathrm{DFT}_n)}_{\mathrm{par}(p,\mu)}$$
$$\rightarrow \overbrace{\left((L^{np}_n \bar{\otimes} I_{m/p})\right.}^{\text{comm}}\overbrace{(I_p \otimes_\parallel (\mathrm{DFT}^\circ_n \otimes I_{m/p}))}^{\text{comp}}\overbrace{(L^{np}_p \bar{\otimes} I_{m/p})}^{\text{comm}}\overbrace{\left.(I_p \otimes_\parallel (I_{n/p} \otimes \mathrm{DFT}_m)\right)}^{\text{comp}}$$

Large packet parallelization:

$$
\begin{aligned}
&\underbrace{\mathrm{DFT}_{m\times n}}_{\mathrm{parBig}(p)} \rightarrow \underbrace{(\mathrm{DFT}_m \otimes I_n)}_{\mathrm{parBig}(p)} \underbrace{(I_m \otimes DFT_n)}_{\mathrm{parBig}(p)} \\
&\quad \rightarrow \underbrace{L^{mn}_m}_{\mathrm{parBig}(p)} \underbrace{(I_n \otimes \mathrm{DFT}_m)}_{\mathrm{parBig}(p)} \underbrace{L^{mn}_n}_{\mathrm{parBig}(p)} \underbrace{(I_m \otimes \mathrm{DFT}_n)}_{\mathrm{parBig}(p)} \\
&\quad \rightarrow \overbrace{(I_p \otimes_\parallel L^{((m)n)/p}_{(m)/p})}^{\text{local perm}} \overbrace{(L^{p^2}_p \otimes I_{((m)n)/p^2})}^{\text{comm}} \overbrace{(I_p \otimes_\parallel (L^n_p \otimes I_{(m)/p}))}^{\text{local perm}} \\
&\qquad \overbrace{(I_p \otimes_\parallel I_{n/p} \otimes \mathrm{DFT}_m)}^{\text{comm}} \\
&\qquad \overbrace{(I_p \otimes_\parallel L^{((n)m)/p}_{(n)/p})}^{\text{local perm}} \overbrace{(L^{p^2}_p \otimes I_{((n)m)/p^2})}^{\text{comm}} \overbrace{(I_p \otimes_\parallel (L^m_p \otimes I_{(n)/p}))}^{\text{local perm}} \\
&\qquad \overbrace{(I_p \otimes_\parallel I_{m/p} \otimes \mathrm{DFT}_n)}^{\text{comm}}
\end{aligned}
$$

Large packet vectorization and overlapped communication apply in a straightforward manner to 2D DFTs, and we do not discuss those here further.

4.2.5 Communication Bound on Distributed DFTs

Since performing a distributed FFT involves a significant amount of inter-PE communication, the interconnect topology and characteristics will have a major impact on performance. In this section, we present a model to evaluate and bound the performance of distributed parallel 1D and 2D FFTs.

We introduce the following notation:

- B = Bisection bandwidth (effective bandwidth we get when doing an all-to-all communication across all nodes), measured in GB/s.
- C = Number of bytes per complex element.
- N = Problem size of the DFT.
- S = Number of communication stages (depends on the DFT type and data distribution; 1D DFTs with block distribution involve 3 stages ((4.19)), and with block cyclic distribution with the appropriate block size involve 1 stage (discussed in Section 4.2.2). 2D DFTs with block distribution involve 2 stages ((4.25)).

- T_c = Runtime of all computation stages involved. Essentially, the time taken by the entire DFT computation without the communication cost.

- T_m = Time taken by the communication of each stage (assuming all stages take the same time).

- H = Overlap factor, defined thus: if we can overlap by H%, this means H% of the minimum time of the computation and communication can be hidden, and thus eliminated from runtime.

Performance bound. We first find a minimum bound on time, and use that to find a maximum bound on performance:

$$\text{Total runtime} = T_c + (S \times T_m)$$

$$T_m = \frac{\text{Total bytes transferred}}{\text{Bandwidth}}$$

$$\text{Total bytes transferred} = N \times C$$

$$T_m \geq \frac{NC}{B}$$

We use $5N\log N$ as the number of operations in a DFT of size N[1]. Based on this number, the performance is given by:

$$\text{Performance} = \frac{\text{Total number of operations}}{\text{Total runtime}}$$

$$= \frac{5N\log N}{T_c + S \cdot T_m}$$
$$\leq \frac{5N\log N}{T_c + \frac{SNC}{B}}$$

[1] This is an approximate and slight overestimate of the exact operations count which depends on the exact FFT used, as presented in [Frigo and Johnson, 2005].

We can use the above to estimate the performance of our DFTs, and as a reference to ensure that our performance is as expected.

4.2.6 Summary

In this section, we presented algorithms for latency optimized DFTs where the input and output data are distributed across the PEs. We summarize the algorithms, paradigms, and tags presented in this section:

- **Algorithms used:** Cooley-Tukey FFT (for 1D DFT), row-column (for 2D DFT).
- **Algorithms built:** Basic (small-packet) 1D and 2D FFT, large packet and vectorized 1D and 2D FFT.
- **Paradigms used:** Parallel paradigm.
- **Tags used:** $\underbrace{A}_{\text{par}(p,\mu)}$ tag to parallelize using the basic (small-packet) parallelization algorithm, and $\underbrace{A}_{\text{parBig}(p)}$ to parallelize using our large-packet algorithm.

4.3 Latency Optimized DFTs Streamed from Memory

Our distributed FFTs were based on the assumption that the input and output data for the DFT fits into the combined local memories of the PEs. When the target platform also contains a memory node, we can also compute larger size DFTs that only fit on the memory node. An example for this case is the Cell processor where we may want to compute DFTs that do not fit on-chip, but only in main memory. In this case, we must compute the DFT by working in steps on smaller chunks of the problem that fit across local memories of the PEs. Strategies for doing so while still achieving high performance are not trivial, for a few reasons. First, the FFT intrinsically requires multiple passes through the entire data for computation. This means algorithms that reduce the number of passes and make most use of data locality are required. Second, most memory systems require data access in chunks to achieve high bandwidth. Hence, algorithms must satisfy this constraint too. Third, algorithms should minimize the costs of shipping data back and forth (called streaming) between the PEs and main memory by overlapping them with computation. We address these challenges in this section.

We first show how to perform DFTs streamed from the storage node to a single PE, followed by strategies to use multiple PEs (all connected to a shared storage node). Finally, we present a

model for estimating performance limits on systems where performance is limited by memory bandwidth. Throughout this section, we use the Cell processor as an example target architecture to illustrate architectural features and tradeoffs that we can expect to find in other similar distributed memory processors.

4.3.1 Streamed FFTs

We discuss how to factorize the DFT so we can compute large DFTs using our streaming technique of interpreting tensor products as streamed loops as presented in Chapter 3. Since we target sizes where the entire input vector does not fit into the combined PEs' memory (henceforth called "local memory"), we must bring the vector in parts from the storage node (henceforth called "main memory") to the local memory. However, because every every point on the DFT's output vector depends on every point on the input vector, at least one intermediate write and read from main memory are required. We define a *stage* as a load of the entire input vector (in parts) from main memory into local memory, followed by computation, followed by a store of the entire output vector to main memory.

The Cooley-Tukey (4.1) and the row-column (4.2) algorithm can both be easily streamed in two stages, since they consist of two tensor products, each of which is a basic SPL construct that can be streamed. We discuss two-stage streaming first, and then motivate the need for using additional stages.

4.3.1.1 1D DFT: Streaming in Two Stages

As mentioned above, our approach of streaming the 1D DFT in two stages is straightforward, and consists of streaming each of the factors of the Cooley-Tukey algorithm. We show 1D DFT streaming in two stages below:

$$\begin{aligned}
&\underbrace{\mathrm{DFT}_{mn}}_{\mathrm{stream}(\psi)} \rightarrow \underbrace{(\mathrm{DFT}^{\circ}_{m} \otimes I_n)}_{\mathrm{stream}(\psi)} \underbrace{(I_m \otimes \mathrm{DFT}_n)}_{\mathrm{stream}(\psi)} \underbrace{L^{mn}_{m}}_{\mathrm{stream}(\psi)} \\
&\quad \rightarrow (L^{ms}_{m} \bar{\otimes} I_{n/s})(I_s \otimes_{\equiv} \mathrm{DFT}^{\circ}_{m} \otimes I_{n/s})(L^{ms}_{s} \bar{\otimes} I_{n/s}) \; (I_t \otimes_{\equiv} I_{m/t} \otimes \mathrm{DFT}_n)(I_t \otimes_{\equiv} L^{mn/t}_{m/t})(L^{tn}_{t} \bar{\otimes} I_{m/t}) \\
&\quad \rightarrow \overbrace{(L^{ms}_{m} \bar{\otimes} I_{n/s})(I_s \otimes_{\equiv} \mathrm{DFT}^{\circ}_{m} \otimes I_{n/s})(L^{ms}_{s} \bar{\otimes} I_{n/s})}^{\text{stage 2}} \; \overbrace{(I_t \otimes_{\equiv} (I_{m/t} \otimes \mathrm{DFT}_n) L^{mn/t}_{m/t})(L^{tn}_{t} \bar{\otimes} I_{m/t})}^{\text{stage 1}},
\end{aligned} \tag{4.13}$$

We visualize two-stage streaming in Figure 4.4. In the first stage, data is read at a stride from the input vector in parts, and written to a temporary intermediate vector at a unit stride. In the second stage, data is read at a stride from this intermediate vector in parts and written to the

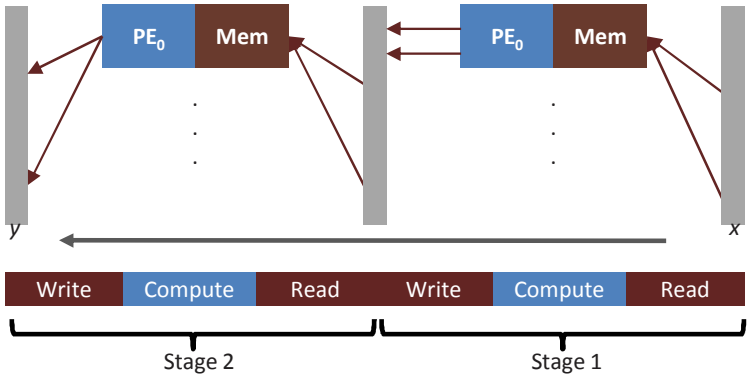


FIGURE 4.4: 1D DFT streamed from memory in two stages.

final output vector at a stride.

Packet size constraints. (4.13) has to satisfy:

$$n/s \geq \psi, \quad m/t \geq \psi \tag{4.14}$$

we assume ψ is the minimum packet size used for loads and stores to main memory required by our streaming paradigm.

Note that we could have chosen different packet sizes for each of the factors. For instance, because first stage writes back data to main memory at very large block sizes and thus has a low cost, we could have allowed it to read its data at even smaller packet sizes. For the Cell, the performance advantage gained by doing so is minimal (empirically determined), which means we use the same packet size constraints for all SPL constructs. This may not hold true for other platforms.

Kernel size constraints. The maximum problem size for which the DFT can be streamed using two stages is dictated by the maximum size of the kernel that will fit into the PE's local memory. Each of the chunks streamed in must fit into the local memory. If C is the maximum DFT kernel size that will fit in local memory, this means:

$$mn/s \leq C, \quad mn/t \leq C \tag{4.15}$$

in (4.13). These constraints ensure that the first and second factor in (4.13), respectively, fit into local memory.

Streaming loop constraints. As discussed in Section 3.4.2, we need at least 8 iterations to get

reasonable overlap in practice, which yields the following additional constraints:

$$s \geq 8, \quad t \geq 8 \tag{4.16}$$

Constraints on the DFT input size. The packet size, kernel size, and streaming loop constraints impose lower and upper bounds on DFT problem sizes that can be performed using the two stage streaming algorithm. We derive these bound below.

Lower bound on N, the DFT size, using (4.14), (4.15), and (4.16):

$$\begin{aligned} N &= mn \\ &\geq \psi t \psi s = \psi^2 st \\ N &\geq 64\psi^2 \end{aligned} \tag{4.17}$$

Upper bound on N:

$$\begin{aligned} s, t &\geq \frac{N}{C} \\ \Rightarrow \frac{n}{\psi}, \frac{m}{\psi} &\geq \frac{N}{C} \\ \Rightarrow \frac{nm}{l}\psi^2 &\geq \frac{N^2}{C^2} \\ \Rightarrow \frac{N}{l}\psi^2 &\geq \frac{N^2}{C^2} \\ \Rightarrow N &\leq \frac{C^2}{\psi^2} \end{aligned} \tag{4.18}$$

LEMMA 1 The lower and upper bounds of the two stage FFT (4.13) on input size N are given by:

$$64\psi^2 \leq N \leq C^2/\psi^2$$

To obtain bounds on the input size for the Cell, we substitute C and ψ for the Cell into Lemma 1. For the Cell, ψ must be 128 bytes, or 16 single precision complex elements, as demonstrated in Figure 3.6. The SPE's local memory is 256kB (2^{18} bytes), which can fit 4 single precision complex element vectors (for input, output, twiddles, double-buffer) of size 2^{13} each. Since we also need space for code, the largest two-power DFT problem size we can perform using one SPE

and the two-stage streaming algorithm is $C = 2^{11}$. Using these values in Lemma 1, we obtain:

$$2^{14} \leq N \leq 2^{14}$$

As we can see, the two-stage algorithm in this case is useful for only a very narrow range of DFT problem sizes.

The need for additional stages. As shown above, the combination of packet size and kernel size constraints, along with the number of streaming loop iterations required limits the problem size of the DFTs that can me streamed using the two-stage FFT in (4.13).

We solve this problem by further breaking down and streaming the smaller DFTs in the Cooley-Tukey FFT, which allow streaming larger DFTs at the cost of additional stages. On platforms where the achieved memory bandwidth has a linear relationship with packet size, there may be a degree of freedom in the number of stages that can be used for certain DFT problem size ranges. The number of stages used must thus be optimized: using too few stages may result poor bandwidth due to small packet sizes, while too many stages may result in poor performance due to the overhead of each stage.

Next, we show an algorithm for streaming the DFT using three stages.

4.3.1.2 1D DFT: Streaming in 3-stages Using Vector Recursion

We can factorize the CT-FFT into 3 stages using vector recursion, as done in [Frigo and Johnson, 2005]. We show 3-stage vector recursion, followed by how it can be used with streaming:

$$\begin{aligned}
\mathrm{DFT}_{kmn} &\rightarrow (\mathrm{DFT}_k \otimes I_{mn}) D_{k,mn} (I_k \otimes \mathrm{DFT}_{mn}) L^{kmn}_k \\
&\rightarrow (\mathrm{DFT}_k \otimes I_{mn}) D_{k,mn} \left(I_k \otimes (\mathrm{DFT}_m \otimes I_n) D_{m,n} (I_m \otimes \mathrm{DFT}_n) L^{mn}_m\right) L^{kmn}_k \\
&\rightarrow (\mathrm{DFT}_k \otimes I_{mn}) D_{k,mn} \left(I_k \otimes \mathrm{DFT}_m \otimes I_n) D_{m,n}\right) \left(I_k \otimes \left(I_m \otimes \mathrm{DFT}_n) L^{mn}_m\right) L^{kmn}_k \\
&\rightarrow (\mathrm{DFT}^\circ_k \otimes I_{mn}) \left(I_k \otimes \mathrm{DFT}^\circ_m \otimes I_n)\right) \left(I_k \otimes \left(I_m \otimes \mathrm{DFT}_n) L^{mn}_m\right) L^{kmn}_k \\
&\rightarrow (\mathrm{DFT}^\circ_k \otimes I_{mn}) \left(I_k \otimes \mathrm{DFT}^\circ_m \otimes I_n\right) (L^{km}_k \otimes I_n) \left(I_m \otimes (I_k \otimes \mathrm{DFT}_n) L^{kn}_k\right) (L^{mn}_m \otimes I_k)
\end{aligned}$$

Using vector recursion above, we can now stream in three stages:

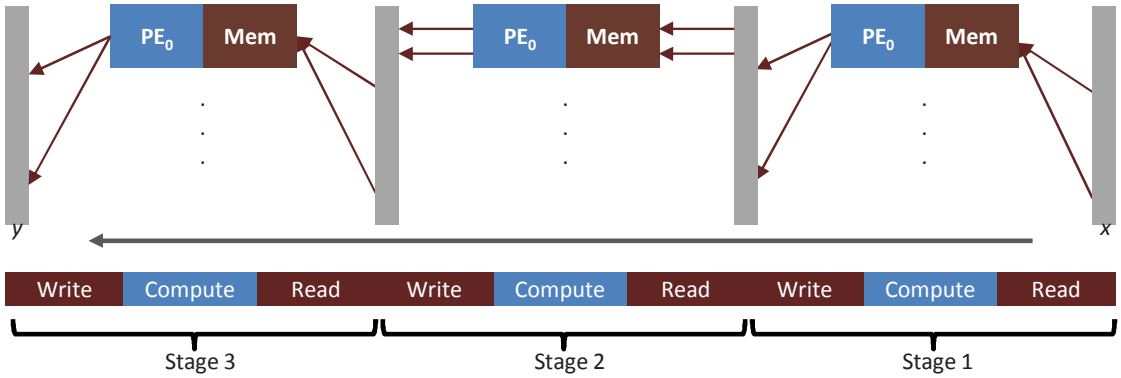


FIGURE 4.5: 1D DFT streamed from memory in three stages.

$$\underbrace{\mathrm{DFT}_{kmn}}_{\mathrm{stream}(\psi)} \to \overbrace{\underbrace{(\mathrm{DFT}_k^\circ \otimes I_{mn})}_{\mathrm{stream}(\psi)}}^{\text{stage 3}} \overbrace{\underbrace{(I_k \otimes \mathrm{DFT}_m^\circ \otimes I_n)}_{\mathrm{stream}(\psi)}}^{\text{stage 2}} \overbrace{\underbrace{(L_k^{km} \otimes I_n)\left(I_m \otimes (I_k \otimes \mathrm{DFT}_n) L_k^{kn}\right)(L_m^{mn} \otimes I_k)}_{\mathrm{stream}(\psi)}}^{\text{stage 1}}$$

$$\to \overbrace{(L_k^{kb} \bar{\otimes} I_{mn/b})\left(I_b \otimes_{\equiv} (\mathrm{DFT}_k \otimes I_{mn/b})\right)(L_b^{kb} \bar{\otimes} I_{mn/b})}^{\text{stage 3}} \overbrace{I_k \otimes_{\equiv} (\mathrm{DFT}_m \otimes I_n)}^{\text{stage 2}}$$

$$\overbrace{(L_k^{km} \bar{\otimes} I_n)\left(I_m \otimes_{\equiv} (I_k \otimes \mathrm{DFT}_n) L_k^{kn}\right)(L_m^{mn} \bar{\otimes} I_k)}^{\text{stage 1}} \tag{4.19}$$

Similar to two-stage streaming, we visualize three-stage streaming in Figure 4.5.

Constraints. Again, we show constraints for each stage based on the maximum kernel size that will fit into local memory (shown using C) and packet sizes used for moving data, and streaming loop requirements:

Constraints for stage 1 in (4.19):

$$\begin{aligned} kn &\le C && \text{(Kernel size restriction)} \\ n \ge \psi, \quad k &\ge \psi && \text{(Minimum packet size restriction)} \\ m &\ge 8 && \text{(Minimum streaming iterations)} \end{aligned} \tag{4.20}$$

Constraints for stage 2 in (4.19):

$$\begin{aligned} mn &\leq C && \text{(Kernel size restriction)} \\ mn &\geq \psi && \text{(Minimum packet size restriction)} \\ k &\geq 8 && \text{(Minimum streaming iterations)} \end{aligned} \tag{4.21}$$

Constraints for stage 3 in (4.19):

$$\begin{aligned} kmn/b &\leq C && \text{(Kernel size restriction)} \\ mn/b &\geq \psi && \text{(Minimum packet size restriction)} \\ b &\geq 8 && \text{(Minimum streaming iterations)} \end{aligned} \tag{4.22}$$

Constraints on the DFT input size. Again, we derive lower and upper bounds on the DFT problem size N based on the packet size, kernel size, and the streaming loop constraints for the three stage algorithm (4.19) based on (4.20), (4.21), and (4.22):

Lower bound on N, the DFT size:

$$\begin{aligned} N &= kmn \\ \frac{N}{b}, \frac{N}{k}, \frac{N}{m} &\leq C \\ mn &\geq 8\psi \\ k &\geq \psi \\ \Rightarrow N &\geq 8\psi^2 \end{aligned} \tag{4.23}$$

Upper bound on N:

$$\begin{aligned} b, k, m &\geq \frac{N}{C} \\ \frac{mn}{\psi} &\geq b \\ \Rightarrow \frac{mnkm}{\psi} &\geq \frac{N^3}{C^3} \\ \Rightarrow \frac{Nm}{\psi} &\geq \frac{N^3}{C^3} \end{aligned} \tag{4.24}$$

LEMMA 2 The lower and upper bounds of the three stage FFT (4.19) on input size N are given by:

$$8\psi^2 \leq N \leq \frac{C^3}{\psi^3}$$

To obtain bounds on the input size for the Cell, we substitute C and ψ for the Cell into Lemma 2:

$$2^{11} \leq N \leq 2^{21}$$

As we can see, the three-stage FFT ((4.19)) can stream a wider range of input sizes than the two-stage version ((4.13)).

4.3.1.3 2D DFT Streaming

We can use the Row Column algorithm in (4.2) to stream 2D DFTs in two and three stages. Two-stage streaming is straightforward:

$$\underbrace{\text{DFT}_{km\times nr}}_{\text{stream}(\psi)} \rightarrow \underbrace{\left(\text{DFT}_{km}\otimes I_{nr}\right)}_{\text{stream}(\psi)}\underbrace{\left(I_{km}\otimes \text{DFT}_{nr}\right)}_{\text{stream}(\psi)} \tag{4.25}$$

A similar set of constraints can be constructed, as we did the for 1D case. Assuming constraints are met, the factors above can be streamed using standard streaming rules. For larger sizes, when $km\psi$ does not fit into the local memories, we can further factorize the left factor by applying the Cooley-Tukey breakdown to the DFT inside it:

$$\begin{aligned}
\underbrace{\text{DFT}_{km\times nr}}_{\text{stream}(\psi)} &\rightarrow \underbrace{\left(\text{DFT}_{km}\otimes I_{nr}\right)}_{\text{stream}(\psi)}\underbrace{\left(I_{km}\otimes \text{DFT}_{nr}\right)}_{\text{stream}(\psi)} \\
&\rightarrow \underbrace{\left(\left((\text{DFT}^{\circ}_{k}\otimes I_m)(I_k\otimes \text{DFT}_m)L^{km}_{k}\right)\otimes I_{nr}\right)}_{\text{stream}(\psi)}\underbrace{\left(I_{km}\otimes \text{DFT}_{nr}\right)}_{\text{stream}(\psi)} \\
&\rightarrow \underbrace{\left(\text{DFT}^{\circ}_{k}\otimes I_{mn}\right)}_{\text{stream}(\psi)}\underbrace{\left((I_k\otimes \text{DFT}_m\otimes I_{nr})(L^{km}_{k}\otimes I_n)\right)}_{\text{stream}(\psi)}\underbrace{\left(I_{km}\otimes \text{DFT}_{nr}\right)}_{\text{stream}(\psi)}
\end{aligned}$$

Now, the constraints are more relaxed, and 2D DFTs up to a size where $k\psi$ and $m\psi$ fit on the

PEs can be streamed. For even larger sizes, or rectangular sizes where kmp fits, but nr does not fit, we can similarly split the DFT on the right factor:

$$\begin{aligned}
\underbrace{\mathrm{DFT}_{km\times nr}}_{\mathrm{stream}(\psi)} &\to \underbrace{\left(\mathrm{DFT}_{km}\otimes I_{nr}\right)}_{\mathrm{stream}(\psi)}\underbrace{\left(I_{km}\otimes \mathrm{DFT}_{nr}\right)}_{\mathrm{stream}(\psi)}\\
&\to \underbrace{\left(\mathrm{DFT}_{km}\otimes I_{nr}\right)}_{\mathrm{stream}(\psi)}\underbrace{\left(I_{km}\otimes ((\mathrm{DFT}^{\circ}_{n}\otimes I_r)(I_n\otimes \mathrm{DFT}_r)L^{nr}_n)\right)}_{\mathrm{stream}(\psi)}\\
&\to \underbrace{\left(\mathrm{DFT}_{km}\otimes I_{nr}\right)}_{\mathrm{stream}(\psi)}\underbrace{\left(I_{km}\otimes \mathrm{DFT}^{\circ}_{n}\otimes I_r)\right)}_{\mathrm{stream}(\psi)}\underbrace{\left(I_{km}\otimes ((I_n\otimes \mathrm{DFT}_r)L^{nr}_n))\right)}_{\mathrm{stream}(\psi)}\\
&\to \underbrace{\left(\mathrm{DFT}_{km}\otimes I_{nr}\right)}_{\mathrm{stream}(\psi)}\left(I_{km}\otimes \underbrace{\mathrm{DFT}^{\circ}_{n}\otimes I_r)\right)}_{\mathrm{stream}(\psi)}\left(I_{km}\otimes \underbrace{(I_n\otimes \mathrm{DFT}_r)L^{nr}_n}_{\mathrm{stream}(\psi)}\right)
\end{aligned}$$

Again, the constraints are relaxed, and nr does not have to fit on the PEs. When needed, we can perform both the above, which yields a 4-stage 2D DFT:

$$\begin{aligned}
\underbrace{\mathrm{DFT}_{km\times nr}}_{\mathrm{stream}(\psi)} \to \underbrace{\left(\mathrm{DFT}^{\circ}_{k}\otimes I_{mn}\right)}_{\mathrm{stream}(\psi)}\underbrace{\left((I_k\otimes \mathrm{DFT}_m\otimes I_{nr})(L^{km}_k\otimes I_n)\right)}_{\mathrm{stream}(\psi)}\\
\left(I_{km}\otimes \underbrace{\mathrm{DFT}^{\circ}_{n}\otimes I_r)\right)}_{\mathrm{stream}(\psi)}\left(I_{km}\otimes \underbrace{(I_n\otimes \mathrm{DFT}_r)L^{nr}_n}_{\mathrm{stream}(\psi)}\right)
\end{aligned}$$

4.3.2 Combining Streaming with Parallelism

So far, our streaming techniques were restricted to using a single PE. In this section, we show how to combine our streaming techniques with our parallelization techniques for distributed memory, so that we can stream using multiple PEs.

Each of our basic SPL constructs consists of a loop over a kernel. In essence, when combining streaming with parallelism, we must redistribute the iterations of this loop to either different PEs (parallelization), or to streamed loops. The order in which we redistribute loop iterations gives rise to two methods of combining these paradigms. We present two methods here, called ParStreamCore, and StreamParChip.

First, we note that when performing both streaming and parallelism, the rewrite rules for each must have knowledge of the other paradigm being applied. Thus, the rewrite rules for applying each of these paradigms are not perfectly orthogonal. An example demonstrates this issue:

$$\underbrace{\underbrace{I_n \otimes A_m}_{\text{par}(p,\mu)}}_{\text{stream}(\psi)} \rightarrow I_n \otimes_{\equiv} \underbrace{A}_{\text{par}(p,\mu)}$$

In the case above, if A is not parallelizable, then parallelization fails. If A is parallelizable, we might still not achieve maximum performance. This is because the streaming rewrite rule "consumed" all available iterations, leaving none for the parallelization rules to act on. On the other hand, if the streaming rule was aware that parallelization across p PEs would be performed on its result, the process would have proceeded thus:

$$\underbrace{\underbrace{I_n \otimes A_m}_{\text{par}(p,\mu)}}_{\text{stream}(\psi)} \rightarrow I_{n/p} \otimes_{\equiv} \underbrace{I_p \otimes A}_{\text{par}(p,\mu)} \rightarrow I_{n/p} \otimes_{\equiv} I_p \otimes_{\parallel} A$$

Thus, we assume that rewrite rules for each paradigm is aware of tags and tag parameters from the other paradigm that will later be applied.

ParStreamCore. The first method results in each PE operating independently and asynchronously with respect to other PEs during each streaming stage: each PE is assigned a set of data points of the vector resident in main memory, and directly reads these data points from main memory, computes on them, and writes them back to main memory without interacting in any manner with the other PEs. This is visualized in Figure 4.6. In the resulting code, the outer loop is the parallel loop, while the inner loop is the streaming loop. Accordingly, we construct algorithms that use this method by first removing the parallel tag, and then removing the streaming tag, leading to the desired loop nesting:

$$\underbrace{I_n \otimes A_m}_{\text{parstreamcore()}} \rightarrow \underbrace{\underbrace{I_n \otimes A_m}_{\text{stream}(\psi)}}_{\text{par}(p,\mu)} \rightarrow I_p \otimes_{\parallel} I_b \otimes_{\equiv} (I_{n/bp} \otimes A_m), \quad \frac{nm}{bp} \geq \psi \tag{4.26}$$

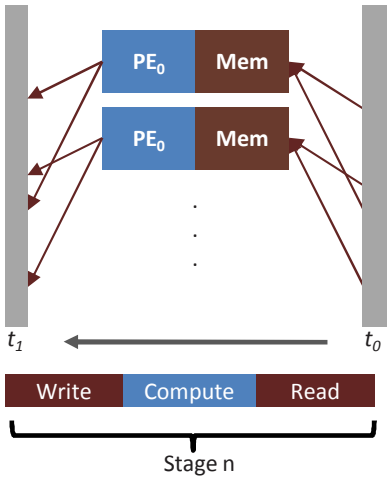


FIGURE 4.6: ParStreamCore algorithm visualization. Each PE works on independent chunks from main memory, thus requiring low synchronization overhead.

$$\underbrace{A_m \otimes I_n}_{\text{parstreamcore()}} \rightarrow \underbrace{\underbrace{A_m \otimes I_n}_{\text{stream}(\psi)}}_{\text{par}(p,\mu)} \rightarrow (L^{mpb}_m \otimes I_{n/pb})\big(I_p \otimes_{\parallel} I_b \otimes_{\equiv} (A_m \otimes I_{n/pb})\big)(L^{mpb}_{pb} \otimes I_{n/pb}),$$

$$\frac{n}{bp} \geq \psi \quad (4.27)$$

Note that there are no constraints on μ, because this method does not involve data exchanges directly between the PEs.

ParStreamCore: Constraints. Assuming C is the maximum kernel size that will fit on a single core, our constraints are:

$$mn/pb \leq C \quad \text{(kernel size constraint)}$$
$$mn/pb \geq \psi \quad \text{(memory packet size restriction, non-strided)}$$
$$n/pb \geq \psi \quad \text{(memory packet size restriction, strided)}$$

StreamParChip. In the second type, the PEs collectively bring in data that fits on the combined local memories. The kernel is then computed in parallel across the PEs. Data must be redistributed between the PEs after the load from memory, before the store to memory, or both. This is visualized in Figure 4.7. In the resulting code, the outer loop is the streaming loop while the inner loop is the parallel loop. Accordingly, we construct algorithms that use this method by

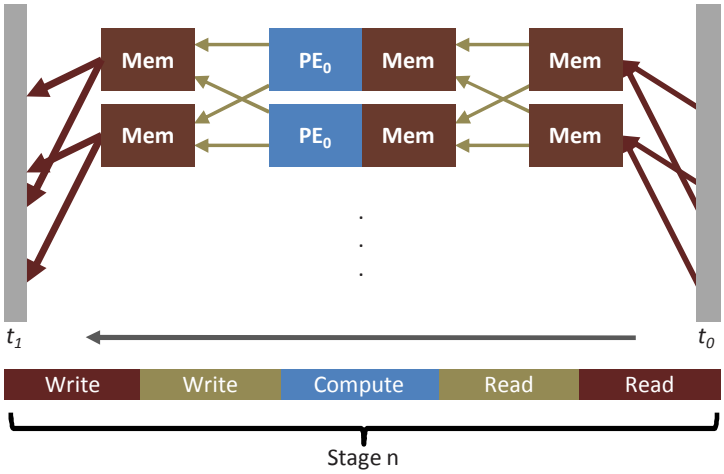


FIGURE 4.7: StreamParChip algorithm visualization. The main idea is to add an "on-chip stage," resulting in the benefit of an additional stage without the full additional cost of one. Larger packet sizes to main memory (and smaller on-chip packets) can be used with this algorithm. However, the PEs must be synchronized several times within each stage.

first removing the streaming tag, and then removing the parallel tag, leading to the desired loop nesting:

$$\underbrace{I_n \otimes A_m}_{\text{streamparchip()}} \rightarrow \underbrace{\underbrace{I_n \otimes A_m}_{\text{par}(p,\mu)}}_{\text{stream}(\psi)} \rightarrow I_b \otimes_{\equiv} I_p \otimes_{\parallel} (I_{n/bp} \otimes A_m) \tag{4.28}$$

$$\begin{aligned}
&\underbrace{A_m \otimes I_n}_{\text{streamparchip()}} \rightarrow \underbrace{\underbrace{A_m \otimes I_n}_{\text{par}(p,\mu)}}_{\text{stream}(\psi)} \rightarrow (L_m^{mb} \otimes I_{n/b})\big(I_b \otimes_{\equiv} \underbrace{(A_m \otimes I_{n/b})}_{\text{par}(p,\mu)}\big)(L_b^{mb} \otimes I_{n/b}) \\
&\rightarrow (L_m^{mb} \otimes I_{n/b})\big(I_b \otimes_{\equiv} ((L_m^{mn/b} \otimes I_{n/bp}) I_p \otimes_{\parallel} (A_m \otimes I_{n/bp})(L_{n/b}^{mn/b} \otimes I_{n/bp}))\big)(L_b^{mb} \otimes I_{n/b})
\end{aligned} \tag{4.29}$$

StreamParChip: Constraints. Assuming C is the maximum kernel size that will fit on a single core, our constraints are:

$$mn/bp \leq C \qquad \text{(kernel size constraint)}$$
$$n/b \geq n/b \qquad \text{(memory packet size restriction)}$$
$$n/bp \geq n/bp \qquad \text{(parallel packet size restriction)}$$

The main advantage of the StreamParChip method is that because it takes advantage of the *combined* local memories of all the PEs, the size of the kernels chunk used for computation scales with the number of PEs. Consequently, the problem size of the DFTs that can be computed using a given number of stages (before an additional stage is needed) also scales with the number of PEs. The main downside is the requirement of inter-PE data exchanges, which leads to one or two synchronization barriers per iteration.

Another way to look at the StreamParChip method is the following: we introduce an additional read/write stage, except, this stage is performed between the PEs' local memories (as opposed to main memory), and individually on each computed chunk. Therefore, it provides us with some of the benefits of an additional stage (including increased packet sizes to main memory), while not incurring the full cost of an additional stage.

In general, the two methods (ParStreamCore, StreamParChip) trade main memory data transfer packet size for the cost of inter-PE data exchange and synchronization. This works particularly well on platforms like chip-based distributed memory systems where the bandwidth between PEs far exceeds the main memory bandwidth.

4.3.2.1 Streaming, Parallelized Algorithms for Large-Sized 1D and 2D DFTs

We can now build algorithms for streamed, parallel large-sized 1D and 2D DFTs using the following two steps:

1. We apply a breakdown rule that breaks down the 1D or 2D DFT into two or three stages, using our algorithms presented earlier. Degree of freedom: the number of stages to pick.

2. We apply our parallelization and streaming technique to the factors. Degree of freedom: the technique to use (ParStream or StreamPar) for each factor.

Some choices within the degrees of freedom are resolved by the constraints imposed by each step. For example, a choice cannot made in the first step that will result in a dead end in the second step due to packet size or kernel size constraints being violated. These choices can be

eliminated by representing the algorithm, packet size, and kernel size constraints as a set of linear equations, and solving them to determine the minimum number of stages required for a particular problem size on a given platfom. The remaining choices in the degrees of freedom form a small search space, though which we search empirically by generating all variants.

4.3.3 Memory Bandwidth Bound on Streamed DFTs

When computing large sized DFTs on systems with a memory node, the maximum possible performance is bound not only by the peak compute performance, but also by the bandwidth between the PEs and main memory. In this section, we develop a model to determine upper bounds on DFT performance for a platform based on its architectural parameters, to help us predict and evaluate the performance of our generated DFTs on that platform.

For the FFT, as the problem size grows larger, the ratio of the number of compute operations to the number of loads and stores grows asymptotically by a factor of $O(\log n)$. For larger problem sizes that need to be streamed in from main memory, this implies that as the problem size increases, so must the performance of the FFT, as measured in flop/s, up to the architecture's peak performance limit. However, an important factor prevents this. As mentioned earlier, the effective memory bandwidth achieved typically decreases with a decrease in packet size. Packet sizes decrease with an increase in DFT problem size for a given number of stages. At the point where low packet sizes severely constrain performance, an extra stage can be added (at a cost) to the DFT to increase packet size.

Cost of adding a stage. Each stage involves reading an entire vector the problem size, computing, and writing it back. If performance is already bound by memory bandwidth, we cannot hide the costs of the additional loads and stores caused by the added stage. The point at which it becomes better to use an additional stage can be determined using our model below.

Handling twiddle factors. Since precomputed twiddle factors for a large problem size can themselves be too large to be distributed and stored on the local memories, we have two alternatives to handling twiddle factors: (a) store them in main memory and stream them in along with the data, and (b) if we are operating well below the compute peak, we can compute them on the fly without affecting overall performance. Although we discuss these in more detail in Chapter 5, we make a note of twiddle factors here because our model must take into account streamed twiddles.

Memory bandwidth bound on streamed DFTs. We will now see how to obtain a performance bound for DFTs based on memory bandwidth.

- B = Memory bandwidth for both reads and writes [in Bytes per second (B/s)]. For now,

we assume a simple memory bandwidth model, where we can achieve full memory bandwidth above a certain packet size threshold, and are unable to access memory below the threshold

- C = Bytes per complex element [in Bytes per complex element]
- s = Number of stages
- t = Number of twiddle stages
- N = DFT size
- n = DFT log size ($n = \log(n)$)

Total number of stages: $(s + 0.5t)$ (each twiddle stage involves only a read, and no write, and hence accounts for 0.5 times the cost of a regular stage).

Each stage in S reads 2^n complex elements, and writes 2^n complex elements.

We assume our performance is bound by the lower of the memory bandwidth and the peak compute performance.

Total data transferred per stage: 2^{n+1} complex elements, or $C \cdot 2^{n+1}$ bytes.

Time taken per stage: $\frac{(2^{n+1} * C)}{B}$ seconds

Since we assume we are bound by memory bandwidth, a DFT of size $\log(n)$ is completed in the amount of time above. So its performance is:

$$= \frac{5 * n * 2^n}{\frac{((s+0.5t) * 2^{n+1} * C)}{B}} \text{flop/s}$$

$$= \frac{5nB}{2C(s+0.5t)} \text{flop/s}. \tag{4.30}$$

To summarize, computing memory bandwidth-bound FFT performance on a given architecture consists of two parts:

1. Determine the number of stages involved, and identify stages that require streamed twiddles.
2. Use (4.30) to compute bandwidth-bound.

4.3.4 Streaming on Cache-based Architectures

The streaming techniques we presented can be used on cache-based architectures. Although we do not focus on or use cache-bases architectures in this thesis, we briefly discuss the applicability of our techniques to cache-based architectures. The difference is, we do not have fine control over what is on-chip and what is not at any given point. To get around this, we could use a combination of several techniques. First, we could use standard buffering techniques to avoid accessing memory in large strides (which can map to the same lines in cache and cause conflict misses). To buffer, we copy our data elements which we would have accessed in strides into a smaller array, and then perform operations on the array, and finally copy the array back into the original array. In addition, if the architecture supports it, we could use streaming reads and writes on the main array to ensure that they do not pollute the cache, but just get copied into our smaller array upon access.

4.3.5 Summary

In this section, we presented algorithms for latency optimized DFTs where the input and output data are resident in main memory. We summarize the algorithms, paradigms, and tags presented in this section:

- **Algorithms used:** Cooley-Tukey FFT (for 1D DFTs), row-column FFT (for 2D DFTs), vector recursion for breaking down the Cooley-Tukey algorithm into three stages.
- **Algorithms built:** 1–3 stage 1D and 2D FFTs, and 4-stage 2D FFTs for streaming using a single PE. In addition, we presented two techniques (ParStreamCore and StreamParChip) to combine streaming and parallelism, leading to several algorithms that use one or both of these techniques.
- **Paradigms used:** Parallel paradigm, streaming paradigm.
- **Tags used:** $\underbrace{A}_{\text{stream}(\psi)}$ tag to stream. $\underbrace{A}_{\text{parstreamcore}()}$ and $\underbrace{A}_{\text{streamparchip}()}$ to parallelize and stream an SPL construct.

4.4 Throughput Optimized DFTs

So far, our FFTs have been optimized for latency. That is, by default, we optimize the cost of reading, computing, and writing out a single transform. We can also optimize our libraries for throughput, which is useful for applications that must compute a large number of DFTs one

after the other, with no intervening computation. In this thesis, we only focus on throughput optimization when a) the DFT input size is small enough to fit among the combined memories of the PEs, and b) when we have a memory node, and are optimizing for throughput for a stream of DFTs of the same problem size that are stored in the memory node.

Optimizing for throughput primarily involves minimizing the costs of data transfer from main memory to the PEs by overlapping computation with communication. Significant gains in performance cannot be achieved by optimizing large (larger than cache or local storage) sizes for throughput, since large sized DFTs are computed in parts from main memory, where we already overlap computation with communication as described in Section 4.3.

The Cell BE is the only target platform in this thesis that has a memory node, and thus, we discuss throughput optimization primarily in the context of the Cell BE.

Performing n multiple independent DFTs (which allows for throughput optimization) on contiguously stored input vectors is expressed in SPL by:

$$I_n \otimes \mathrm{DFT}_m .$$

In the remaining part of this section, the distinction between 1D and 2D DFTs does not matter since optimizing either for throughput involves the same techniques.

4.4.1 Streaming Small DFTs

In this case, we assume that we compute a large number of small DFTs on contiguous chunks of the input data, with data resident in main memory. *Small* DFTs are DFTs of problem sizes small enough to fit into the local memory of a single PE, or in the Cell, inside a single local store. Optimizing the streaming of small DFTs is easy, and we use two tools to do so: (a) we stream DFTs from an SPE to main memory, and (b) we use multiple SPEs in parallel, each working on a separate DFT problem.

In the simplest case, we express that we want to stream s DFTs from a single SPE to main memory in SPL as:

$$\underbrace{I_s \otimes_{\equiv} \mathrm{DFT}_m}_{\mathrm{stream}(\psi)}$$

We apply our rules from Table 3.2 to stream (4.4.1). Performance is bound by the main memory bandwidth of a single SPE in this case.

To improve performance, we can can trivially parallelize the above and have SPE working on a separate DFT. We express this in the following SPL expression, which computes $p \times s$ DFTs:

$$\underbrace{I_s \otimes_{\equiv} \underbrace{I_p \otimes_{\parallel} \mathrm{DFT}_m}_{\mathrm{par}(p,\mu)}}_{\mathrm{stream}(\psi)}$$

This leads to a loop nest with an outer streaming loop and an inner parallel loop. Below, we show how (4.4.1) translates to Σ-SPL. We note that since the DFTs work on completely independent problems, we push the outer streaming loop's gather and scatter into the inner parallel loop, using identities in [Franchetti et al., 2005]:

$$\begin{aligned}\sum_{j=0}^{s-1}{}_{\equiv} S_{\mathrm{DT}}(q)\left(\sum_{i=0}^{p-1}{}_{\parallel} \mathrm{S}(h)\,\mathrm{DFT}_m\,\mathrm{G}(h)\right) G_{\mathrm{DT}}(q) \rightarrow \\ \sum_{j=0}^{s-1}{}_{\equiv}\sum_{i=0}^{p-1}{}_{\parallel} S_{\mathrm{DT}}(q)\,\mathrm{S}(h)\,\mathrm{DFT}_m\,\mathrm{G}(h)\,G_{\mathrm{DT}}(q) \rightarrow \\ \sum_{j=0}^{s-1}{}_{\equiv}\sum_{i=0}^{p-1}{}_{\parallel} S_{\mathrm{DT}}(q)\,\mathrm{DFT}_m\,G_{\mathrm{DT}}(q) \qquad (4.31)\end{aligned}$$

Note that in the above, we eventually fused the outer sum's scatter and gather with the inner sum's scatter and gather, as described in [Franchetti et al., 2005].

Performance. Performance in our second case is bound by the total off-chip memory bandwidth. Notice that because of our loop ordering above (the parallel loop is the inner loop), our data access is performed in large chunks. Since main memory systems are typically optimized for this type of access, we can expect to achieve high memory performance, bound by main memory bandwidth specifications.

Note that here is a limit on the number of SPEs we can parallelize across before we saturate the main memory bandwidth. Also, we note that the main memory bandwidth bound increases as the problem size increases. This is because the computation ($5n\log(n)$) to load/store ratio ($2n$) for the DFT increases as the size increases. This is reflected in the bandwidth bound line on plot Figure 6.7(a) in Chapter 6.

4.4.2 Streaming Medium-Sized DFTs

Medium-sized DFTs are those that do not fit into the local store of a single SPE, but fit across the combined local stores of multiple SPEs. In this case, we parallelize the DFT across the SPEs, and stream such parallelized DFTs from main memory. In SPL, this is:

$$\underbrace{I_s \otimes_{\equiv} \underbrace{\mathrm{DFT}_{mn}}_{\mathrm{par}(p,\mu)}}_{\mathrm{stream}(\psi)}$$

Since we parallelize the DFT in this case, we end up with two parallel loops obtained by breaking down the DFT with our parallelization rules. At this stage, we can optionally apply the optimization we applied to small DFTs: we can push in the DMA scatter and gather into the parallel loops, and combine them. However, by doing so, while performance may improve in some cases, it might become worse in other cases. The reason for this is because when in the outermost sum, the DMA gather and scatter read and write in large packet size, while pushing them and fusing them with the parallel sum's gather and scatter cause them to now read and write memory packets in small sizes:

$$\sum_{j=0}^{s-1}{}_{\equiv} S_{\mathrm{DT}}(q) \left(\sum_{i=0}^{p-1}{}_{\shortparallel} \mathrm{S}(h)\,\mathrm{DFT}_m\,\mathrm{G}(h) \sum_{i=0}^{p-1}{}_{\shortparallel} \mathrm{S}(h)\,\mathrm{DFT}_n\,\mathrm{G}(h) \right) G_{\mathrm{DT}}(q) \rightarrow$$
$$\sum_{j=0}^{s-1}{}_{\equiv} \left(\sum_{i=0}^{p-1}{}_{\shortparallel} S_{\mathrm{DT}}(q)\,\mathrm{DFT}_m\,\mathrm{G}(h) \sum_{i=0}^{p-1}{}_{\shortparallel} \mathrm{S}(h)\,\mathrm{DFT}_m\,G_{\mathrm{DT}}(q) \right) \quad (4.32)$$

We perform this optional final step only when performance is increased as a result. This can either be determined experimentally, or be determined by a minimum packet size threshold which must be respected by the fusing rule.

4.5 Rewriting for Usage Scenarios

We have now discussed our parallel and streaming paradigms, and how to generate DFTs based on these paradigms. As illustrated by Figure 4.1, there are various usage scenarios we might need to generate code for. We have discussed most of these scenarios in previous sections. In this section, we summarize these scenarios and show how we use our tag guided rewriting process to generate algorithms for each scenario.

We can specify each of the usage scenarios depicted in Figure 4.1 by the appropriate use of tags, which then guide the rewriting system. Table 4.1 shows these tag based specifications corresponding to the scenarios in Figure 4.1. Table 4.1(a) shows latency-optimized scenarios (can be used to compute a single DFT), which were all discussed in Section 4.2 and Section 4.3. Table 4.1(b) shows throughput-optimized scenarios (available only when performing s DFTs), which were all discussed in Section 4.4.

Tagged Specification	Description
(a) Latency-optimized usage scenarios	
$\underbrace{\mathrm{DFT}_m}_{\mathrm{vec}(v)}$	Single DFT on a single PE, from local memory
$\underbrace{\mathrm{DFT}_m}_{\mathrm{stream}(\psi)}$	Large DFT streamed in parts to a single PE, from main memory
$\underbrace{\mathrm{DFT}_m}_{\mathrm{par}(p,\mu)}$	DFT parallelized across p PEs, from local memory
$\underbrace{\underbrace{\mathrm{DFT}_m}_{\mathrm{par}(p,\mu)}}_{\mathrm{stream}(\psi)}$	Large DFT parallelized across p PEs, from main memory, streamed in parts
(b) Throughput-optimized usage scenarios	
$\underbrace{I_s \otimes \mathrm{DFT}_m}_{\mathrm{stream}(\psi)}$	Streaming across s DFTs, using a single PE (small DFTs)
$\underbrace{I_p \otimes \underbrace{I_s \otimes \mathrm{DFT}_m}_{\mathrm{stream}(\psi)}}_{\mathrm{par}(p,\mu)}$	Streaming across s DFTs, parallelization across p PEs (many small DFTs)
$\underbrace{I_s \otimes \underbrace{\mathrm{DFT}_m}_{\mathrm{par}(p,\mu)}}_{\mathrm{stream}(\psi)}$	Streaming across s DFTs, each DFT parallelized to run on p PEs (mid-sized DFTs)

TABLE 4.1: Problem specification using tags for DFT usage scenarios. All DFTs assume a vectorization tag with innermost nesting.

4.6 Chapter Summary

In this chapter, we presented algorithms for generating code for latency optimized and throughput optimized DFTs. For latency optimized DFTs, we first presented our basic and large packet parallelization techniques. Next, we presented algorithms for latency optimized DFTs streamed from main memory, followed by algorithms and techniques for streamed, parallel DFTs. In all cases, we presented methods to generate code that explicitly managed data transfers (both from main memory to PEs and between PEs), and minimized their costs by overlapping them with computation. We then presented techniques to optimize small and medium sized DFTs for throughput. Finally, we presented problem specifications that can be used to generate code for a variety of usage scenarios.

In the next chapter, we will describe the rest of our system that generates code from the algorithms we presented here, and in addition, builds a tagging system to allow the user to create

high-level problem specifications.

CHAPTER 5

Program Generation

In the previous chapter, we explained how to adapt an FFT to the architectural paradigms of parallelization and streaming based on coarse hardware information in the form of a few parameters. This adaptation is implemented as a rewriting system and forms the core of our generator. In this chapter, we discuss the infrastructure of our generator built around this core. This includes both the generation of detailed input specifications from higher level user specifications, and the generation of an output program from an adapted algorithm represented in Σ-SPL. Our infrastructure uses and extends that of the existing SPIRAL. We explain both our infrastructure and the existing infrastructure from previous work to illustrate the inner workings of our system. We end the chapter with a brief example.

Figure 5.1 shows the overall program generation system. The input to our system is shown on the top, and the output is a C program or library, shown on the bottom. The labels on the left side show the knowledge our system includes to process a given stage.

System working. We give an overview of the system in Figure 5.1. A detailed explanation follows later.

- *Input:* The input to our system consists of DFT parameters (e.g., input size), problem specification (e.g., throughput or latency optimization, number of processors to use), and architectural specification (e.g., programming and memory paradigm, data transfer packet size to use).

- *System specification generation:* The first stage converts the input into a formal transform specification with appropriate tags. This is the form that can be processed by the next stage. We cover this in Section 5.1.

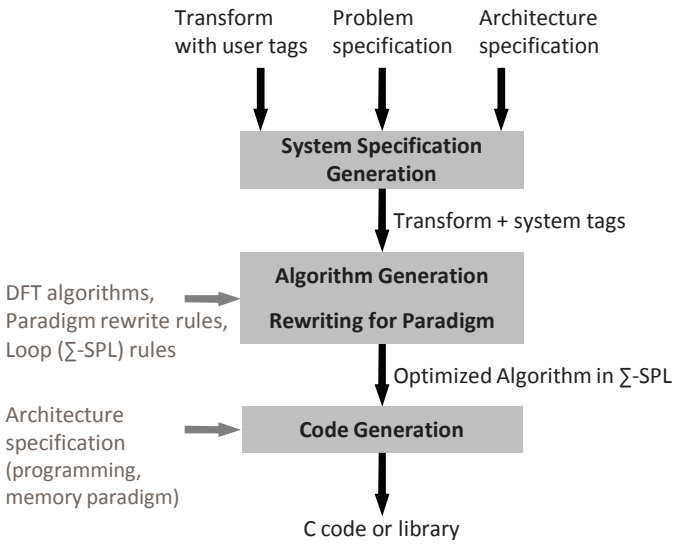


FIGURE 5.1: Program generation system overview.

- *Algorithm generation and rewriting for paradigm:* This stage is responsible for generating algorithms for the specified problem which are adapted to the specified paradigm and parameters. This stage is implemented as a rewriting system and outputs the resulting algorithm in Σ-SPL. The core of these operations were covered in Chapter 4. We present implementation details in Section 5.2.

- *Code generation:* Finally, using the architecture specifications, the Σ-SPL algorithm is translated into C code, possibly containing MPI calls, DMA directives, and vector intrinsics. We cover this in Section 5.3, and address some lower level issues in Section 5.4.

- *Output:* The final is either an optimized C function (for a fixed input size DFT) or a C library (for the general input size DFT).

5.1 System Specification Generation

In this section, we describe the first stage or block in Figure 5.1. As shown, the input consists of three parts: the tagged transform, a problem specification, and an architecture specification. We discuss each of these parts in detail below, and then show how to use them to generate system specifications that can be used by the second stage of our system.

Tag	Description	Parameters	No-tag default assumption
$\overbrace{A}^{\text{parallelize}_\text{u}(p)}$	Parallelize	p (optional)	Run on single PE
$\overbrace{A}^{\text{memory}_\text{u}()}$	Vectors in main memory	*(none)*	Vectors in local memory

TABLE 5.1: User tags as indicated by the subscript u.

5.1.1 User Tags and System Tags

The scenarios shown in Section 4.5 are all relevant but also pose a problem for the use of our system. First, the user is required to have knowledge of the system: specifically, what problem sizes would not fit in the local memories, and what number of PEs would yield the fastest performance for a particular problem size. Second, the scenarios make implicit assumptions about the location of the input and output data. For example, they assume that transforms small enough to fit in local memory are stored in local memory, which may not be the case. Third, as we saw in Chapter 4, tags may interact during rewriting. In particular, this happens with the streaming and parallelization tags as well as with the large packet parallelization and the vectorization tag.

To resolve these issues, we group tags into two tag spaces: user tags and system tags. As their name implies, the former are specified by the user, while the latter are automatically introduced by the system as a part of the rewriting process. Note that users can also specify system tags if needed: this gives the user the flexibility to specify as little or as much as possible, and allows the system to automatically fill in (deterministically or through experimental evaluation) the rest of the required tags and parameters. Below, we describe these two tag spaces and how our tagging system works.

User tags. User tags are those specified by the user. The set of available user tags are shown in Table 5.1, along with the required parameters and defaults when the tag is not included in the specifications. User tags have the subscript "u." Available tags include the parallelization tag which specifies that the user wants parallelization (and optionally, for a specified number of processors), and the memory tag, which specifies that the input and output vectors are in main memory.

User tags can be thought of as "higher-level" tags, as they allow the user to minimally specify the requirement at a high level. They do not rewrite the transform itself, but rather get replaced first with system-level tags. User tags can be specified in any order.

System tags. System tags are automatically generated by our system using information from user tags and the architecture specification. The set of available system tags is shown in Table 5.2. All these were introduced in Chapter 3. Note that the user can also create a more detailed, lower

Tag	Description	Parameters
$\underbrace{A}_{\text{par}(p,\mu)}$	Use small packet parallelization	p, μ
$\underbrace{A}_{\text{parBig}(p)}$	Use large packet parallelization	p
$\underbrace{A}_{\text{stream}(\psi)}$	Use streaming (for throughput and large-sizes)	ψ
$\underbrace{A}_{\text{StreamAndPar()}}$	Use both streaming and parallelization (using ParStreamCore or StreamParChip)	*none*

TABLE 5.2: System tags.

level requirement specification by using any or all of the system tags.

Rewriting expressions to remove the individual parallelization and streaming tags were discussed in Chapter 3. The last tag in Table 5.2 is rewritten with the following two rules:

$$\underbrace{AB}_{\text{StreamAndPar()}} \rightarrow \underbrace{A}_{\text{StreamAndPar()}} \underbrace{B}_{\text{StreamAndPar()}}$$

$$\underbrace{A}_{\text{StreamAndPar()}} \xrightarrow{\text{search}} \underbrace{A}_{\text{parstreamcore()}}, \underbrace{A}_{\text{streamparchip()}}$$

A $\xrightarrow{\text{search}}$ implies that our system searches for the best among the choices on the right hand side. The second rule above has two choices. Note that large packet parallelization ($\underbrace{A}_{\text{parBig}(p)}$) does not work in conjunction with streaming.

Systems tag nesting. For the rewrite system to work correctly and find the set of applicable rules at any point during the rewriting, our system tags must be nested correctly. Tags can be classified based on the three paradigms we us:

1. Vectorization: $\underbrace{A}_{\text{vec}(v)}$
2. Parallelization: $\underbrace{A}_{\text{par}(p,\mu)}, \underbrace{A}_{\text{parBig}(p)}, \underbrace{A}_{\text{parBig}(p,v)}$
3. Streaming: $\underbrace{A}_{\text{stream}(\psi)}, \underbrace{A}_{\text{mem}(s)}$
4. Both parallelization and streaming: $\underbrace{A}_{\text{parstreamcore()}}, \underbrace{A}_{\text{streamparchip()}}, \underbrace{A}_{\text{StreamAndPar()}}$

System tags are nested thus:

1. The relative nesting of any of the streaming and parallelization tags is determined during rewriting based on whether the StreamAndPar() tag is rewritten into a StreamParChip() tag or a ParStreamCore() tag, as explained in Section 4.3.2.

2. The vectorization tag is always introduced at the innermost nesting level, as it must be the last one to be removed by the rewriting system.

5.1.2 Architecture specification

Our system requires the following knowledge about the target architecture:

- maximum number of of PEs available
- local memory size determines the largest transform C that will fit into local memory
- presence of a memory node (yes/no)
- minimum packet size to use for inter-PE communication (μ)
- minimum packet size to use for memory-PE communication (ψ)

In addition, code for the following are required for generating output code:

- non-blocking send or receive
- memory fence
- synchronization barrier

The SIMD instruction set is also required as explained in [Franchetti et al., 2006b].

5.1.3 Problem specification

In addition to the transform specification and the architecture specifications, the following are required as input to our system as a part of the problem specification:

- Type of code desired: fixed size or general size
- Data distribution (block distributed, block cyclic)

5.1.4 Generating System Specifications from User Specifications

We use a combination of the user-tagged transform, the architecture specification, and the problem specification to generate a system specification. A system specification consists of the transform with system tags. The following steps convert the user input to a system-tagged transform:

1. *Streaming:* if the architecture specification shows that the input transform is too large to be held in either one or multiple PEs (depending on the presence of the $\underbrace{A}_{\text{parallelize}_\text{u}(p)}$ tag), then a streaming or a streaming and parallel tag is added:

$$A \rightarrow \begin{cases} \underbrace{A}_{\text{stream}(\psi)}, & \text{if no parallelization was specified} \\ \underbrace{A}_{\text{StreamAndPar()}}, & \text{if parallelization was specified} \end{cases}$$

   In the above, ψ is obtained from the architecture specification.

2. *Parallelization:* If a $\underbrace{A}_{\text{StreamAndPar()}}$ tag was not added in the previous step, and the user has specified a $\underbrace{A}_{\text{parallelize}_\text{u}(p)}$ tag, then the following parallel tags are searched over:

$$\underbrace{A}_{\text{parallelize}_\text{u}(p)} \xrightarrow{\text{search}} \{ \underbrace{A}_{\text{par}(p,\mu)}, \underbrace{A}_{\text{parBig}(p)} \}$$

   In the above, μ is obtained from the architecture specification. Note that if p is left unspecified, then all valid values of p that will hold the transform (as determined from the architectural specifications) are searched over for the optimal one.

3. *Memory tag:* the memory tag is handled thus:

$$\underbrace{A}_{\text{memory}_\text{u}()} \rightarrow \begin{cases} A, & \text{if any of the stream tags exist} \\ \xrightarrow{\text{search}} & \{ \underbrace{A}_{\text{stream}(\psi)} \text{ (or } \underbrace{A}_{\text{StreamAndPar()}} \text{)}, S_{\text{DT}}(q) AG_{\text{DT}}(q) \} \end{cases}$$

   In other words, vectors are assumed to reside in main memory if any of the streaming tags exist. If not, vectors can either be streamed in from main memory, or can simply be read and written to main memory (without streaming) using explicit memory instructions, and our system searches over these options for the best solution.

Problem specifications. To handle various data distributions, our system simply adds appropriate permutations around the user specified transform based on the problem specification. Currently, block and block cyclic data distributions are supported.

5.2 Algorithm Generation and Rewriting for Paradigm

In the previous section, we discussed how to generate a system-tagged SPL specification from the user-tagged transform, problem specification, and architecture specifications. This section explains the second stage in Figure 5.1 which is the core of our system: it produces a platform-adapted algorithm specified in Σ-SPL. Although the main ideas in this process were covered in Chapter 3 and Chapter 4, this section serves as a wrapper to those chapters, and provides the necessary implementation details. Since the program generation is significantly different for fixed size and general size code, we discuss them separately below.

5.2.1 Fixed Size Program Generation

We describe the steps involved in generating fixed size FFT code. This step is the second stage shown in Figure 5.1. We note the degrees of freedom involved in each step, if any. We first describe our work, and then briefly discuss how it composes with previous work that targets SIMD vectorization and the memory hierarchy.

Step 1: Algorithm generation from system-tagged SPL. Algorithm breakdown rules are applied to convert the transform into tagged SPL constructs. Algorithm breakdown rules include the Cooley-Tukey FFT, the vector recursion rule, the row-column algorithm, and other 2D breakdowns covered in Chapter 4. In addition, the type of parallelization that will be used (small packet or large packet parallelization) is also decided in this step, since the two types of parallelization cannot be mixed with each other. Applying algorithm breakdown rules followed by applying the appropriate paradigm rules allows our system to automatically arrive at a variant FFT algorithm that is well suited to the target architecture. Degrees of freedom (all subject to rule constraints): the number of stages as decided by the application of the vector recursion rule, or any of the 2D multi-stage rules; choice of parallelization.

Step 2: Σ-SPL generation from algorithm. The tagged SPL constructs are converted into Σ-SPL using the rewrite rules presented in Table 3.1 and Table 3.2. The Σ-SPL versions are mappable to the target paradigms. At this stage, constructs that are tagged both stream and par may have a degree of freedom, and are re-tagged with either the ParStreamCore rule

or the StreamParChip rule. Degrees of freedom (subject to rule constraints): selection of ParStreamCore or StreamParChip. At the Σ-SPL level, the system performs formal loop merging based on previous work in Franchetti et al. [2005], and on rules from our work such as and .

Search. Steps 1 and 2 above contain degrees of freedom which exist because we do not know beforehand the options that will lead to the best performance. We handle such degrees of freedom simply by generating code for all the options available and searching for the fastest code. Since the degrees of freedom are limited, the search space is small and tractable.

Optimizing for other architectural paradigms. Our work composes with previous work on vectorization and blocking for the memory hierarchy. All our evaluation platforms incorporate SIMD vectorization, handled using the work in [Franchetti et al., 2006b]. In addition, PEs on MPI systems typically have a deep local memory hierarchy, and the Cell can be considered to have a single memory hierarchy layer consisting of register space. To optimize for the memory hierarchy, previous work performs a search over a larger search space at the lower recursion levels of the algorithm breakdown using dynamic programming, as discussed in [Püschel et al., 2005].

5.2.2 General Size Library Generation

We describe the steps involved in generating general size libraries based on our description in Section 2.4.4. Note that we only use the general size library generation system to produce libraries for our large packet parallelization algorithm. Streaming and small packet parallelization are currently not implemented for the general size case.

In the previous section, we showed how to rewrite the transform based on a set of rules to obtain a variant FFT that is suited to the target platform. The overall idea in generating general size code is analogous: we feed this set of rules into Autolib. However, unlike the program generation process for fixed size code, the result is a set of Σ-SPL expressions (as opposed to an algorithm in Σ-SPL) which represent the recursion step closure (as described in Chapter 2 and in [Voronenko, 2008]) for the input rules. When translated into mutually recursive functions, the recursion step closure forms the general size library.

Instead of allowing Autolib to generate the entire variant FFT, for the purpose of simplicity, we manually apply (4.12) to the DFT, and mark parts of it which translate directly into MPI code or local permutations, and only feed in the remaining computation stages into Autolib. The entire process is shown in Figure 5.2, and described below.

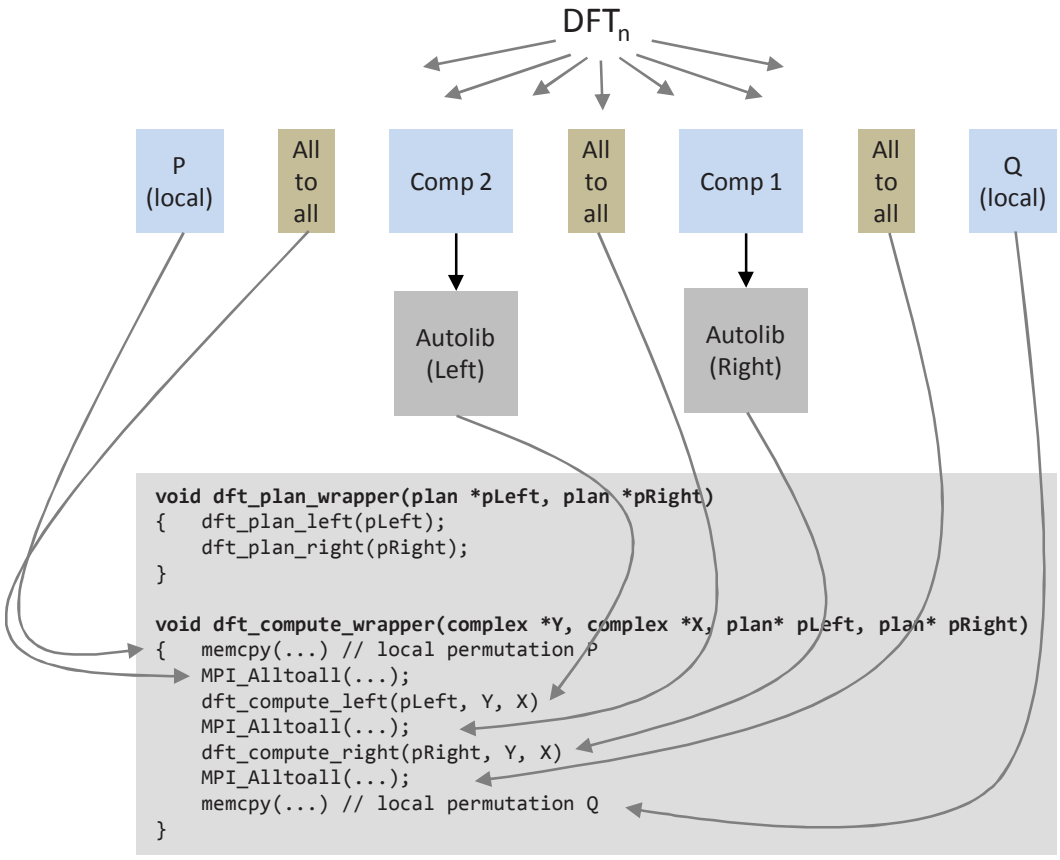


FIGURE 5.2: Generating DFTs in Autolib.

Step 1: Apply (4.12) to the DFT. As shown in (4.12), this produces seven stages including two computation stages, three global communication stages, and two local permutation stages.

Step 2: For each of the two computation stages, we input the computation (including the surrounding local permutations) into the Autolib system. A separate library is produced for each of these computation stages.

Step 3: Wrapper code implements the entire transform by including the local permutations (implemented using `memcpy()`s), global permutation (implemented using `MPI_Alltoall`), and calls to each of the two libraries that implement the computation.

The process works similarly for 2D DFTs and other variations based on data distributions. Note that the resulting code may have fewer communication or permutation stages.

Search. Search in general size library generation is conducted by the generated code, rather than by the program generation system as is the case for fixed size code. Autolib thus encodes

the set of algorithms required and the degrees of freedom into the library itself, as discussed in [Voronenko, 2008].

5.3 Code Generation

The third step of our system shown in Figure 5.1 is to convert Σ-SPL generated in the previous step to C code. In the fixed input size case, the Σ-SPL generated in the previous step is the actual algorithm. In the general input size case, the set of Σ-SPL expressions generated in the previous step correspond to the set of mutually recursive functions that are required for the implementation. In either case, the generated Σ-SPL expressions consist of regular, parallel, streaming, and vector loops which can be readily translated into C code. Program generation is performed based on Table 2.4. Explicit memory access instructions such as MPI and DMA are generated based on translation rules discussed in Chapter 3.

5.4 Low Level Concerns

In this section, we address several concerns associated with low level implementation details. We first present several architectural constraints on the Cell and our solutions, followed by issues with computing or loading twiddle factors, and conclude with a brief discussion of synchronization and memory barriers required for parallelization and streaming.

5.4.1 Cell Platform Constraints

The Cell BE has several platform imposed restrictions which our system considers. We discuss them briefly here.

Temporary space. Since the Cell has limited local memory (256kB local store per SPE), to minimize the temporary space required by our code, we use a simple version of the memory arena technique described in [Hanson, 1989]. Before generating C code, our system determines the peak memory storage required by the DFT function at any given point in the code, and allocates a memory arena of this size. Temporary space is then allocated and deallocated from this arena, thus minimizing the amount of total temporary space required.

DMA alignment constraints. The Cell has alignment constraints for DMA instructions. Effectively, we cannot perform DMAs under 16 bytes, and can perform DMAs only in multiples of 16 bytes. We can also perform DMA only on naturally aligned data. For two power sizes and vectorizable non-2-power sizes beyond a minimum size, our packet size restrictions guarantee that DMA instructions are aligned correctly. For other odd sizes which we do not address in this

thesis, alignment issues may be solved with techniques presented in [Franchetti and Püschel, 2007].

Memory banks and iteration skewing. The Cell's main memory is divided into 16 banks for performance reasons. Memory addresses are striped cyclically across these banks at a 128 byte block size. In other words, memory addresses corresponding to the bytes 0 to 127 are assigned to bank 0, addresses corresponding to bytes 128 to 255 are assigned to bank 1, and so on. Accessing memory addresses from only a subset of the banks results in a reduced effective bandwidth, since the architecture relies on a data access pattern that evenly uses all banks to achieve the full main memory bandwidth of 25.6 GB/s. However, when our parallelized streaming algorithms access main memory at large two-power strides, they may end up accessing only a subset of the banks.

To address this issue and increase performance in this situation, we use a simple iteration skewing technique. With the natural strides found in the FFT, as the SPEs progress through their loops, they all access the same bank at the same time. Instead, we skew the iterations with respect to the SPEs in such a way that while the first SPE is working on its first iteration of the loop, the second SPE is working on its second iteration, and so forth. This evens out the bank access pattern and increases the effective main memory bandwidth.

[Chow et al., 2005] addresses the same problem by offsetting the imaginary parts of the complex input and output arrays in such a way that the real and imaginary parts are always mapped to different memory banks. Although this allows the authors to achieve higher performance, the technique imposes a data layout on the application, in addition to being restricted to a split-complex input format. Our technique suffers from neither drawback.

5.4.2 Twiddle Factors

All DFT computations involve multiplications with twiddle factors. Since generating twiddle factors when computing a DFT is highly expensive (because of the sine and cosine computations involved), SPIRAL's previous approach involved pre-computing and storing twiddle factors at library initialization time. This means that each twiddle factor is simply loaded from memory when required.

When using recursive DFTs, several twiddle factor tables, varying from small to large (corresponding to the various problem sizes in the recursion tree) may be required. In a distributed memory environment, the smaller of these tables can easily be precomputed and replicated across each PE. However, this strategy does not work for larger twiddle tables that either do not fit in local memory, or are large enough to be different for each PE. We discuss our approach to this issue here.

Distributing twiddle factors. We handle the precomputed twiddles table for the topmost (largest) problem size in the recursion tree by distributing it across the PEs when the fit in local memory. Each PE computes only its portion of the twiddles table at library initialization time.

Streaming in twiddle factors. For architectures such as the Cell with limited local memory but larger main memory, we store the precomputed twiddle factors in main memory, and stream them in along with parts of the input vector when needed. We use a double buffering technique, similar to our streaming paradigm, to load twiddle factors in the background, one loop iteration ahead, to minimize the impact of such loading on run time. However, a part of the memory bandwidth is used up by the twiddle factors, which can impact run time on on platforms where memory bandwidth bounds performance.

Online computation of twiddle factors. When memory bandwidth bounds performance, this means that we potentially have unused compute cycles available to us, while bandwidth comes at a premium. In this case, one solution is to pre-compute small seed tables that fit in local memory space, but need additional computation (which must be relatively inexpensive) for conversion into actual twiddle factors. Although we do not currently implement this technique in our system, this may potentially increase performance by a factor that can be estimated using our formula in Section 4.3.3.

5.4.3 Synchronization Barriers

Our output code requires the use of two types of barriers: synchronization barriers and memory barriers (also known as memory fences).

Synchronization barriers. Synchronization barriers are a point in code at which all PEs must arrive before any can proceed past them. Barriers are required whenever PEs end up using data generated by other PEs. We identify and mark the location of barriers at a high level using rewrite rules shown in Table 3.1.

Ideally, our barriers must be fast, so as to not impact performance by adding to the run time. We need as many barriers as we have compute stages, which means we need three barriers to compute a single DFT for any of the two parallel DFT algorithms in this thesis. Fast barrier implementations may already exist on a platform. MPI defines a barrier call, `MPI_barrier()` which is typically already optimized for a given platform. On the Cell, we use mailbox communications to synchronize across the SPEs. Our synchronization barriers cost between 600 and 1200 cycles, depending on the number of SPEs being used.

Memory barriers. Memory barriers are points in code where a specified set of outstanding data transfer requests that were made at an earlier point using non-blocking calls, must be completed before execution can proceed. Architectures that support non-blocking data trans-

fer requests also have programming interfaces to issue memory barriers. This makes generating these simple, once the points at which they must be inserted have been identified. These points marked up during algorithm generation, based on the streaming rewrite rules, as shown in Table 3.2.

5.5 Example

In this section, we provide a small example to show how the program generation system works. We illustrate our example in Figure 5.3, and discuss its progress through the stages of our program generation system (shown in Figure 5.1).

Input. We specify the input to be a DFT transform of size 2^{18}, and request it to be parallelized across all available PEs. Our problem specification consists of requesting single precision fixed size code using a regular data distribution. Our architecture specification is that of a Cell BE with four available SPEs. Given the size of the local stores of the Cell BE, we compute $C = 2^{12}$. Based on empirical data about the interconnect performance, We also determine values for μ (minimum packet size for inter-SPE data transfers) and ψ (minimum packet size for chip to main memory data transfers) and include this in the architecture specification. We also specify the DMA instruction set.

System specification generation. As shown in Figure 5.3, based on our architecture specification, our system determines that we must parallelize across four SPEs, and that we also need to stream the data from main memory as it will not fit on-chip. Consequently, a StreamAndPar() tag is added with the appropriate parameters. A SIMD vector tag is also added by default. The end result is the transform with all required tags correctly nested.

Algorithm generation and rewriting for paradigm. Breakdown rules are applied to our transform. The system determines that we must use a three-stage streaming algorithm. One set of possible parameters for the 1D DFT three-stage algorithm is shown in Figure 5.3. The system searches for the fastest code over other parameters and alternatives including the ParStreamCore and StreamParChip versions of each of the factors.

Code generation. The output of the previous stage is an algorithm in Σ-SPL (not shown). This is converted into C code using the architecture specification to generate platform-specific calls.

Output. The final output is the fastest code found over searching alternatives, and tailored to the input specifications.

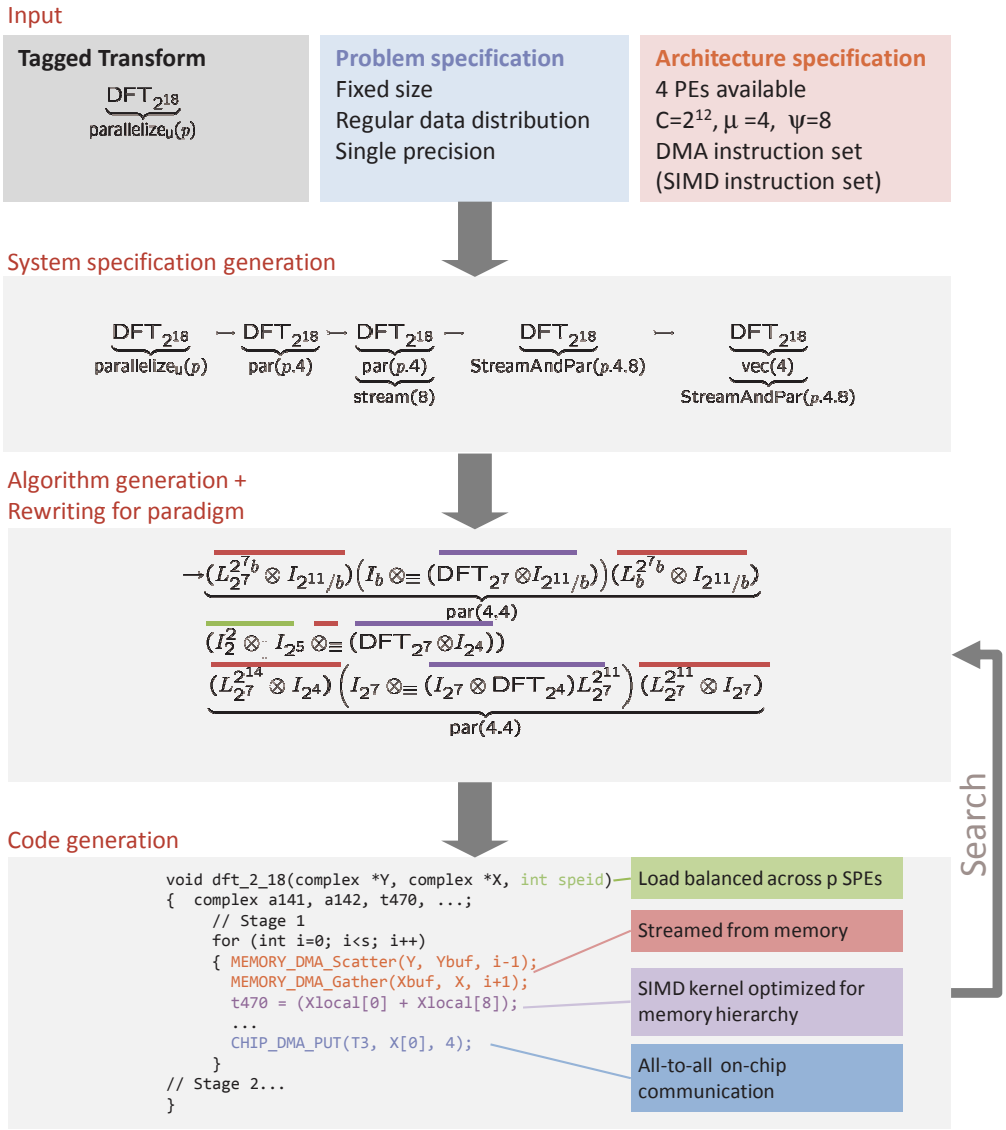


FIGURE 5.3: Program generation example.

CHAPTER 6

Experimental Results

In this chapter, we present the performance of the DFT code generated by our system. We first present our experimental setup. Then, we present the performance of our generated single-PE vectorized kernel, which serves as a baseline reference for the remaining results. For the remaining sections, results, we follow the general outline of Chapter 4: we first present performance results for latency optimized distributed DFTs, followed by latency optimized distributed DFTs streamed from main memory, and conclude with the performance of throughput optimized DFTs.

6.1 Experimental Setup

We evaluate our generated single precision 1D and 2D DFT implementations on the Cell BE processor, and on three MPI clusters: the Axon, the Cray XT4, and the Cray XT5. The architectural details of these platforms were discussed in Chapter 3.1. We present other platform details and software details below.

Cell processor. We used a single Cell processor of an IBM Cell Blade QS20 running at 3.2 GHz. Our code was compiled with the gcc compiler for the SPU, `spu-gcc`, from the Cell SDK version 3.0.

Axon. On the Axon, we use version 4.1.2 of the gcc compiler to compile our generated libraries with the following flags: `-msse3 -std=c99 -O3 -fstrict-aliasing`. The Axon uses OpenMPI version 1.4.1, which is an MPI-2 implementation.

Cray XT4 and XT5. On the Cray XT4 and XT5 machines, we use version 4.3.3 of the gcc compiler with the following flags: `-msse3 -std=c99 -O3 -fstrict-aliasing`. Both machines use

MPICH, which is an MPI-2 implementation.

Performance measure. All our plots report FFT performance in pseudo-giga-floating point operations per second (pseudo gflop/s), which is essentially inverse runtime, defined as:

$$5N\log_2(N)/(\text{runtime [s]} \cdot 10^9),$$

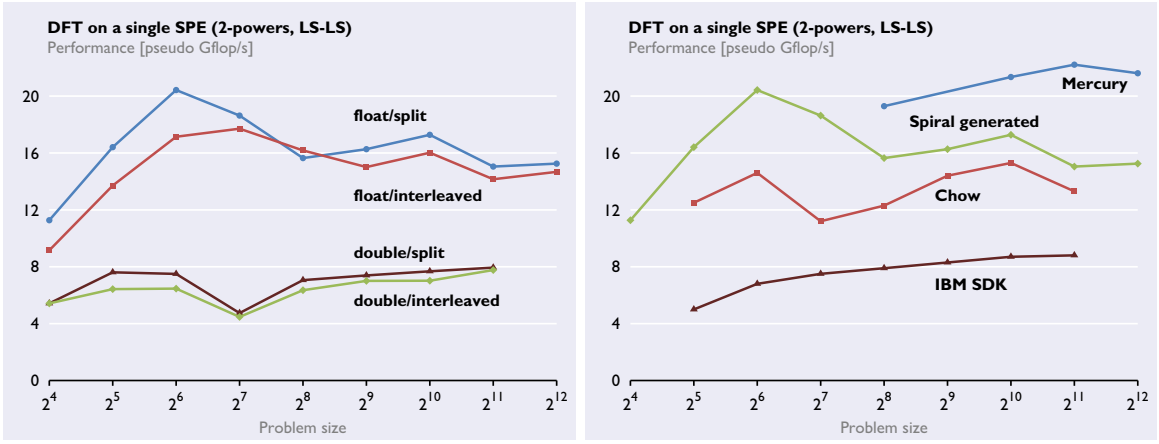
where N is the size of the 1D or 2D DFT kernel in complex samples (e.g., for a $\text{DFT}_{k\times k}$, $N = k^2$). This metric is based on the asymptotic operation count of the radix-2 Cooley-Tukey algorithm. As discussed in Section 4.2.5, this is an approximate and slight overestimate of the exact operations count which depends on the exact FFT used, as presented in [Frigo and Johnson, 2005]. However, this metric provides us with a convenient way to measure run time and compare it across platforms. Higher is better in all plots. We show only the forward transform's performance. The performance of the inverse transform is typically the same.

Timing methodology. For experiments that measured the performance of latency-optimized kernels, we measured the execution time of a single kernel. We timed an adequate number of iterations to ensure timing stability and precision. We measured the performance of throughput-optimized kernels by running several iterations and measuring the runtime of the steady state of the computation.

Timing mechanism. For obtaining timing information on the Cell BE, we use the decrementers available on the SPE. On all MPI platforms, we use the timing mechanism provided by MPI by calling `MPI_Wtime()`.

FFTW and flags used. We used FFTW version 3.3alpha1, which is a preview version of the new MPI interface. Previously, basic MPI support was available in FFTW 2.1.5, which had several limitations, and also provided poor performance in our tests as compared to the 3.3alpha1 version.

When we benchmark FFTW, we use its default planner flag, which is the FFTW_MEASURE flag (see [FFTW, 2008b] for details). With this option, FFTW measures the run time of FFTs on the target platform, and uses these measurements to guide its search. FFTW_PATIENT, which promises a larger search space was not used because of the time limitations on the supercomputing platforms. We note that we ran FFTW with FFTW_PATIENT on a random sample of problems on each of our target platforms, and found that the performance results obtained were never significantly better (within 4%) than those obtained when using FFTW_MEASURE.



(a) (b)

FIGURE 6.1: Baseline: single-SPE performance on the Cell BE.

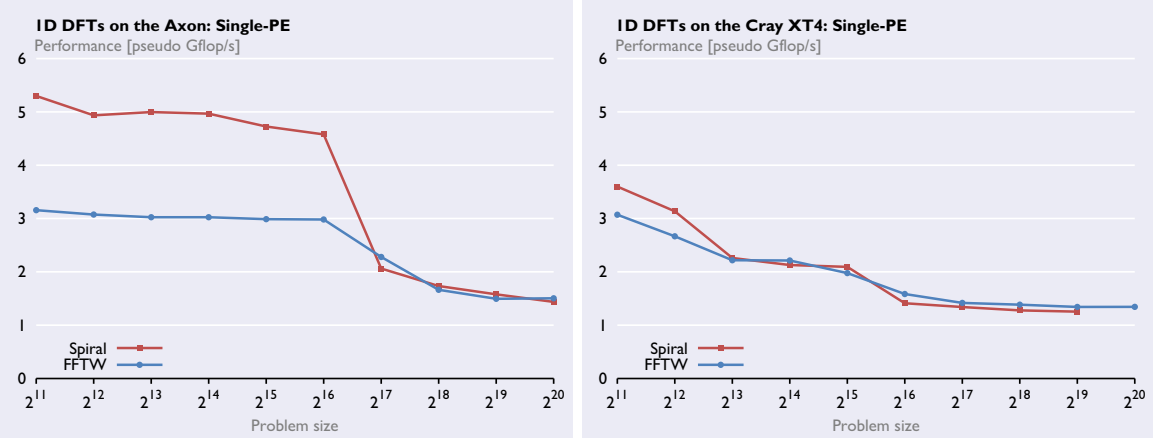
6.2 Baseline Single-PE Performance

We first show the performance of our libraries running on a single PE on each of the platforms. This serves as a baseline for comparison for the parallelized and streamed DFTs.

6.2.1 Cell

SIMD Vectorized DFT on a single SPE. Figure 6.1(a) shows the single-core performance of our generated DFT kernels for 2-power input sizes for single precision (labelled "float") and double precision (labelled "double") data resident in the local store. These vectorized DFT kernels are important building blocks in all our parallel and streaming implementations. We show performance for both the interleaved complex and the split complex data format to make a fair comparison to other libraries. Our generated code achieves 16–21 Gflop/s for single precision, and 8 Gflop/s for double precision.

In Figure 6.1(b), we compare the performance of our single-precision split-complex kernels to hand-tuned code from Mercury [Cico et al., 2006], Chow [Chow, 2008], and the IBM SDK FFT library [IBM, 2008] as measured in [Chow, 2008]. The maximum performance achieved by our single precision generated code is currently slower (0.73x–0.85x) than Mercury, although our code achieves its performance peak for smaller sizes. Our code is faster by a factor of 1.06x–1.6x when compared to Chow's library, and 1.83x–3x faster than IBM's SDK.



(a) (b)

FIGURE 6.2: Baseline: Single PE performance on MPI platforms.

6.2.2 Clusters

In Figure 6.2(a), we show the performance of SPIRAL generated code and that of FFTW running on a single PE on the Axon. Although SPIRAL generated code performs better than FFTW for in-cache sizes (less than size 2^{17}), these are largely irrelevant for comparisons with large parallelized FFTs, because large FFTs primarily use large, out-of-cache local FFTs.

In Figure 6.2(b), we show the performance of SPIRAL generated code and FFTW on a single PE of the Cray XT4. Performance for the Cray XT5 is similar.

6.3 Latency Optimized Distributed DFTs

We present the performance of our latency optimized distributed on each platform. In this section, we only consider problem sizes that fit within the combined local memories of the PEs.

6.3.1 Cell

Parallel DFT across multiple SPEs, data in local stores. Figure 6.3(a) and (b) display the performance of our generated parallel, multi-SPE DFT kernels, with data resident in the local stores. These kernels use single-SPE vectorized kernels as building blocks. We generate and evaluate code for two standard data distributions: Figure 6.3(a) shows results for the standard block distribution (i.e., data is cut into p contiguous blocks for p SPEs). Figure 6.3(b) shows results for data in the *block cyclic* format, i.e., data is distributed across the SPEs' local stores in a round-

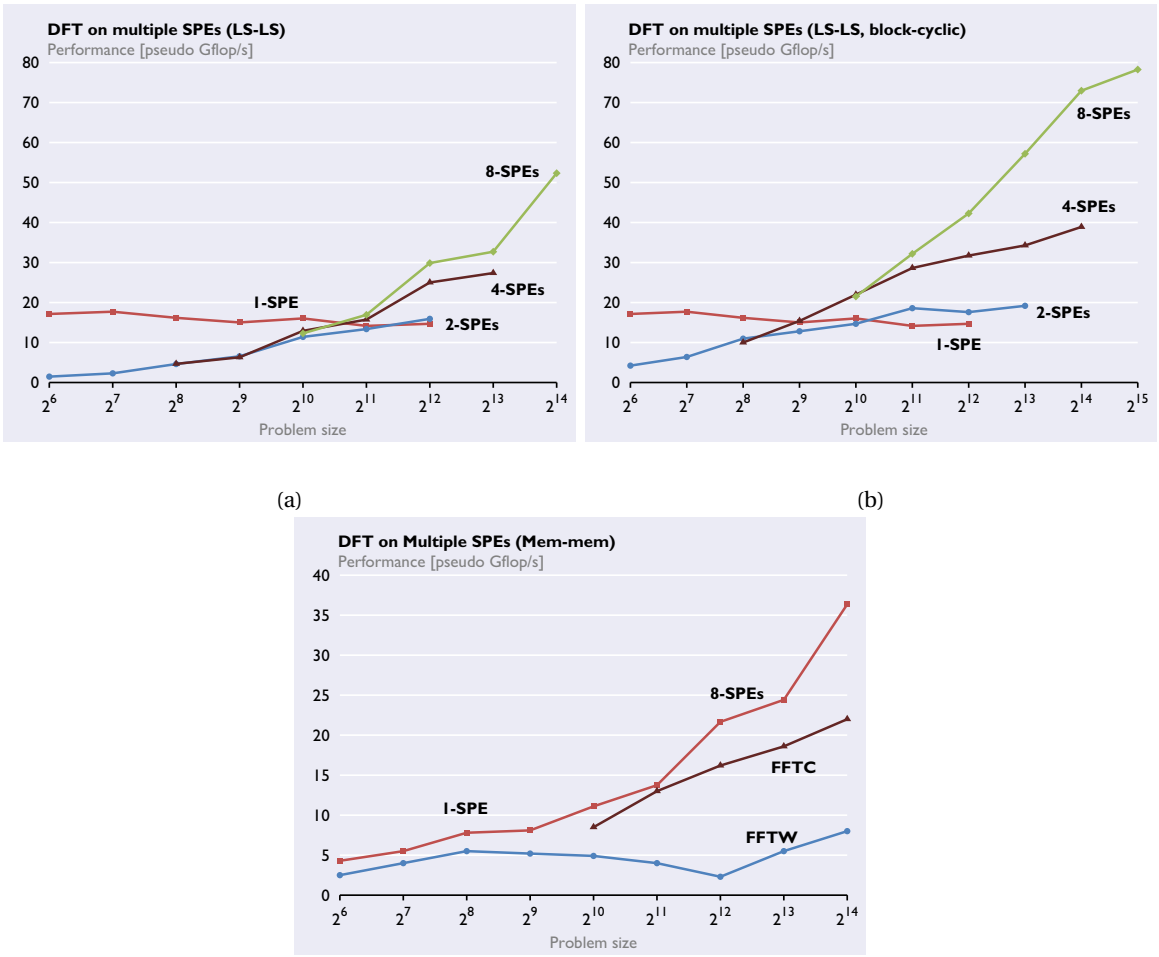


FIGURE 6.3: Latency optimized parallel DFTs on the Cell BE.

robin fashion using a suitable block size. We achieve close to 80 Gflop/s for a 2^{15}-point DFT across 8 SPEs using a block cyclic format, and about 53 Gflop/s for a 2^{14}-point DFT with a standard distributed format. The block cyclic data distribution requires less communication, and thus achieves better performance compared to the standard distribution.

DFT on multiple SPEs, data in main memory. Figure 6.3(c) displays the performance of parallelized DFT kernels with data beginning and ending in main memory. We compare our generated code to FFTW [Frigo and Johnson, 2005] and FFTC [Bader and Agarwal, 2007], with performance data taken from [FFTW, 2008a] and [Bader and Agarwal, 2007], respectively. Note that both implementations only provide *latency-optimized* DFT functions. To make a fair comparison with FFTW and FFTC, we specified a mem() tag, which simply adds DMA load and store

operations around our parallel multi-SPE kernels (shown in Figure 6.3(d)). This approach does not result in highly optimized code as communication and computation are not performed in parallel. Despite this limitation, as shown in Figure 6.3(f), our generated code already outperforms FFTW by a factor of 4.5x, and FFTC by a factor of 1.63x (for size 2^{14}).

FFTC is limited in its effectiveness because it uses a single algorithm, hard-coded for the 5 problem sizes shown. It is also hard-coded to use 8 SPEs. FFTW has been ported to the Cell processor [FFTW, 2008a], and is able to offload computation to the SPEs. However, FFTW is unable to achieve more than 22 (pseudo) Gflop/s of performance for any transform size on the Cell processor. Also, we speculate that it is unable to gain an advantage by offloading computation to the SPEs for transform sizes that are less than 2^{12}, limiting performance to under 5 (pseudo) Gflop/s for smaller sizes.

6.3.2 Clusters

In Figure 6.4(a), (c), and (e), we show the performance of 1D DFTs on the Cray XT4, Cray XT4, and the Axon, respectively. In Figure 6.4(b), (d), and (f), we show the corresponding performance of FFTW. Performance scales well on all these platforms. Our code achieves up to 12.5 Gflop/s on 32 PEs of the Cray XT4 (FFTW achieves up to 13.8 Gflop/s), 28.4 Gflop/s on 128 PEs of the Cray XT5 (FFTW achieves 26.9 Gflop/), and over 14 Gflop/s on 64 PEs of the Axon (FFTW achieves 16.2 Gflop/s).

For a given number of PEs on the Cray XT4 and XT5, we time our DFTs using two configurations with respect to the number of nodes versus the number of cores: one maximizes the number of nodes, while the other maximizes the number of cores. Applications may demand a particular configuration. On the XT4 and the XT5, surprisingly, maximizing the number of nodes consistently yields better performance than maximizing the number of cores for a given number of PEs.

In Figure 6.5(a) and (c), we show the performance of our generated 2D DFTs on the Cray XT4 and Cray XT5. We show the corresponding performance of FFTW in Figure 6.5(b) and (d). As of date, we were unable to obtain performance results for the 2D DFTs on the Axon due to unavailability of the machine. Again, the performance of our 2D FFTs scales well, though they are slightly slower than FFTW. Notice that 2D FFTs are faster than the 1D FFTs. This because they require only two communication stages as opposed to the three stages required by the 1D DFTs. Note that in some cases, we were unable to obtain FFTW performance for the larger number of PEs: this was because the FFTW planner took longer to complete its planning than the system allowed. The SPIRAL planner executes on a single node (as opposed to FFTW's planner which uses the entire system), which allows it to execute without demanding the use of all the PEs.

6.4 Latency Optimized DFTs Streamed from Memory

We evaluate latency optimized DFTs streamed from main memory on the Cell. In this section, we only consider sizes that are too large to fit on the combined local stores of the SPEs.

Streamed 1D DFTs, single SPE. In Figure 6.6(a), we show the performance of our 1D DFTs streamed using our two stage algorithm and the three stage algorithm. The three stage algorithm is faster for most cases. Given that we were unable to accurately predict the optimal number of stages for a given size, we search over both our algorithms to find the best code. Note that the performance of both these algorithms drops with an increase in problem size. This is because of the decrease in data transfer packet size as the problem size increases.

Streamed, parallelized 1D DFTs. In Figure 6.6(b), we show the performance of our large 1D DFTs streamed from main memory and parallelized across 4 SPEs. We compare our performance with [FFTW, 2008a] and [Sacco, 2008]. Although the performance of our generated code is comparable with both, we are unable to generate code for sizes larger than 2^{20} with our three stage algorithm, while FFTW can compute sizes up to 2^{22}.

Streamed 2D DFTs, single SPE. In Figure 6.6(c), we show the performance of small-to-medium-sized 2D DFTs on a single SPE. The data begins and ends in main memory and includes problem sizes which cannot be held completely in the local store. We use our two stage streaming algorithm. Data thus has to make two round trips from memory to the local stores. We observe performance of up to 11 Gflop/s for sizes that allow for reasonably large DMA packet sizes. Sizes that use smaller packet sizes see lower performance.

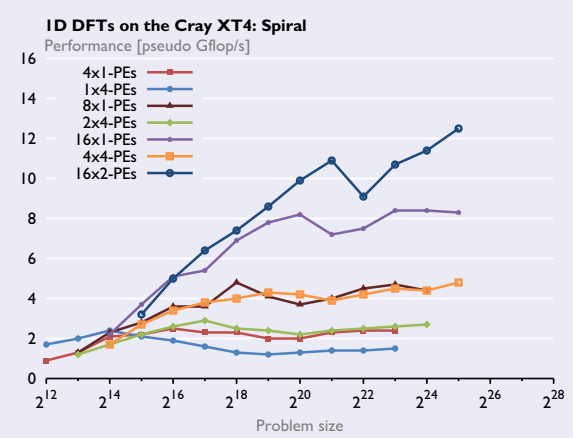
Streamed, parallelized 2D DFTs. Next, in Figure 6.6(d), we evaluate the performance of streamed, parallelized 2D DFTs for a range of sizes. These are also streamed from main memory using our streaming algorithms, but in addition, are parallelized across 4 SPEs. The performance of our code is again comparable to FFTW.

6.5 Throughput Optimized DFTs

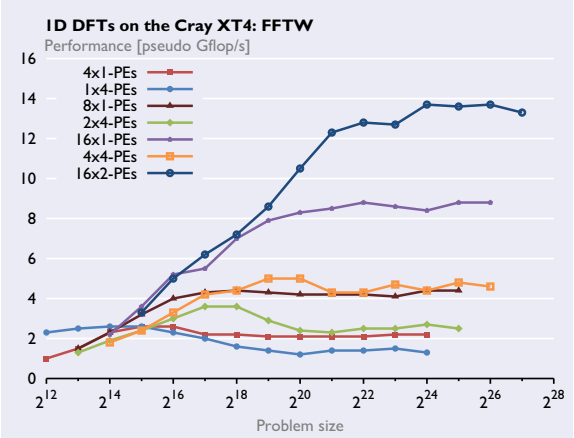
In Figure 6.7(a), we measure the performance of small DFT kernels (data resides in main memory but each DFT can fit in the local store of one SPE) in throughput mode. An independent copy of the single-SPE kernel runs on each SPE. Memory bandwidth is the limiting factor once more than 4 SPEs are being used, and we see sustained performance of up to 50 Gflop/s.

Figure 6.7(b) is a throughput-optimized version, and achieves about 30 Gflop/s on a DFT of size 2^{13} on 4 SPEs. It streams data from main memory to and the local stores, and converts data into a block cyclic data format on the fly, enabling the application of our fastest parallel block-

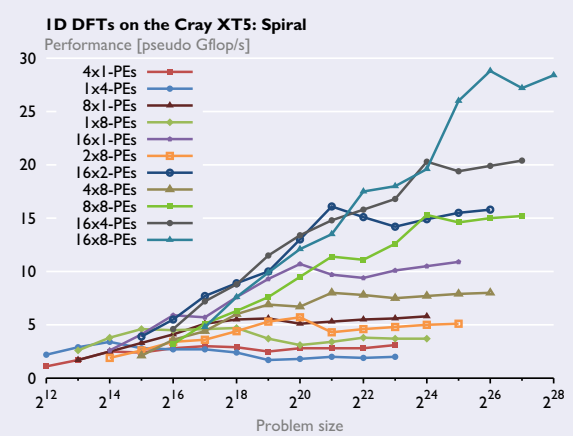
cyclic multi-SPE DFT kernels. This technique does not scale well to 8 SPEs due to the decrease in the size of the DMA packet size as the number of PEs increases.



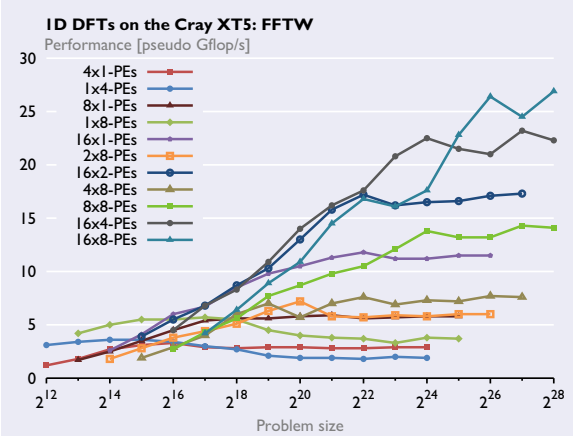
(a)



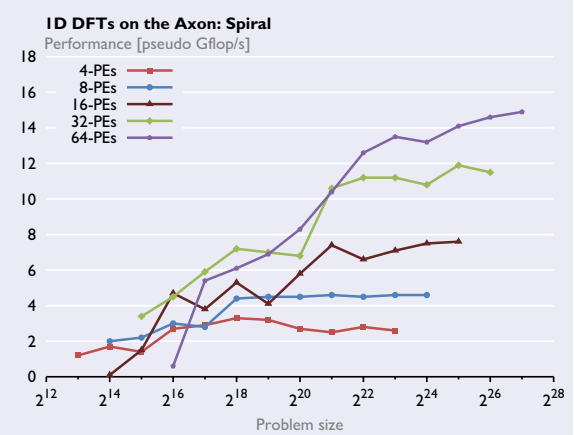
(b)



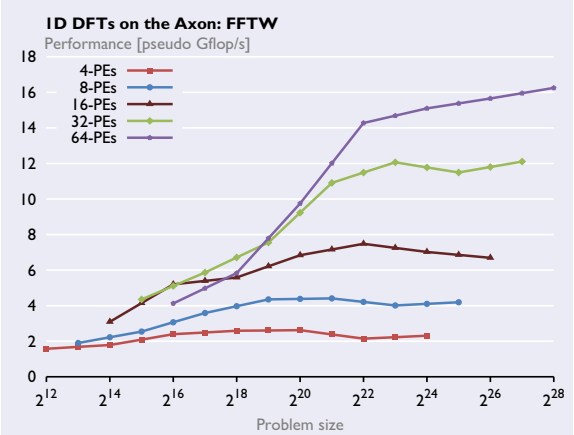
(c)



(d)



(e)



(f)

FIGURE 6.4: Latency optimized parallel DFTs on MPI platforms. A 16x2 in the legend means that 16 nodes with 2 cores per node (total of 32 PEs) were used.

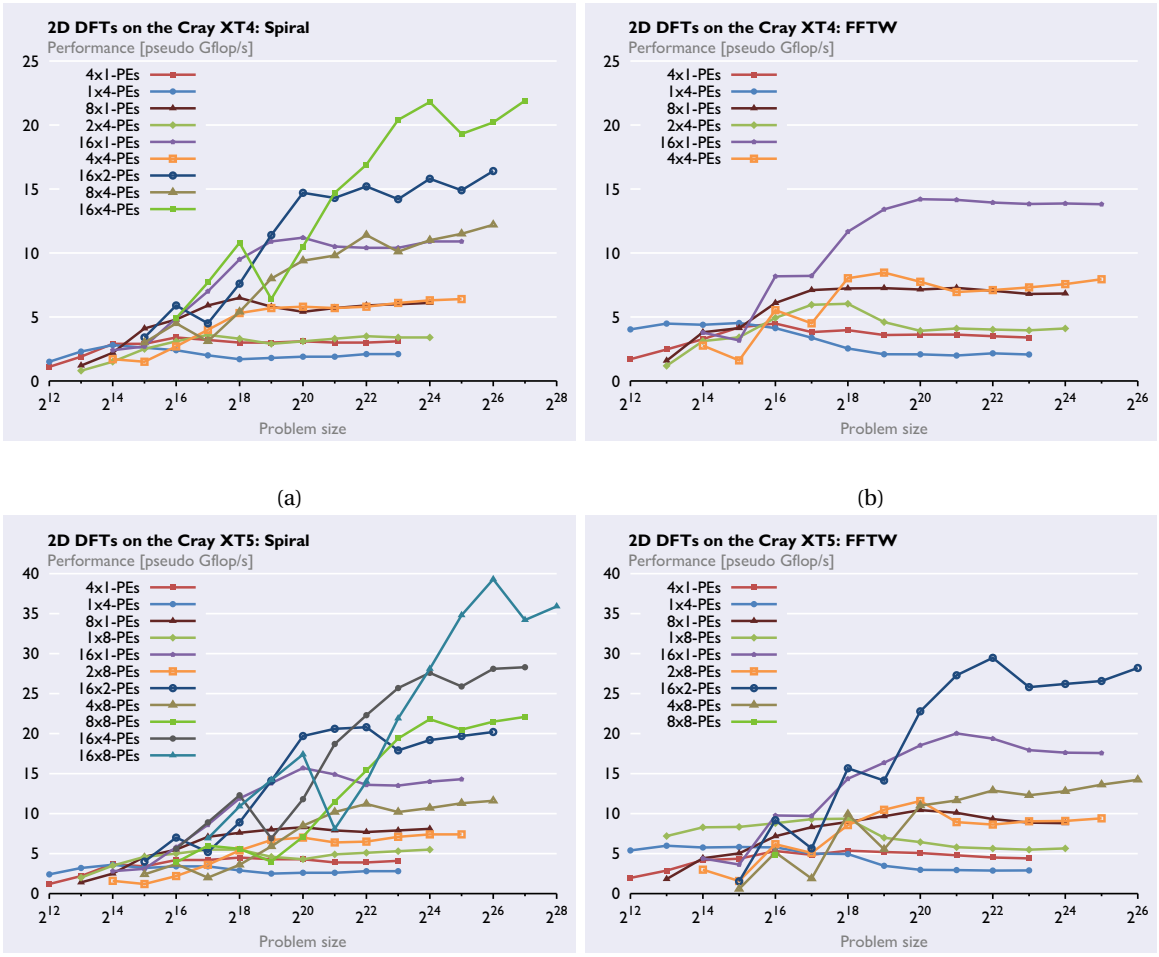


FIGURE 6.5: Latency optimized parallel 2D DFTs on MPI platforms. Problem sizes are shown with a single dimension on the x-axis for readability. For even powers of two, a size shown as 2^N corresponds to an input size of $2^n \times 2^n$ where $n = N/2$. For odd powers of two, 2^N corresponds to $2^n \times 2^{n+1}$. For example, a problem size shown as 2^{16} refers to a 2D DFT size of $2^8 \times 2^8$, and 2^{17}' refers to a size of $2^8 \times 2^9$. A 16x2 in the legend means that 16 nodes with with 2 cores per node (total of 32 PEs) were used.

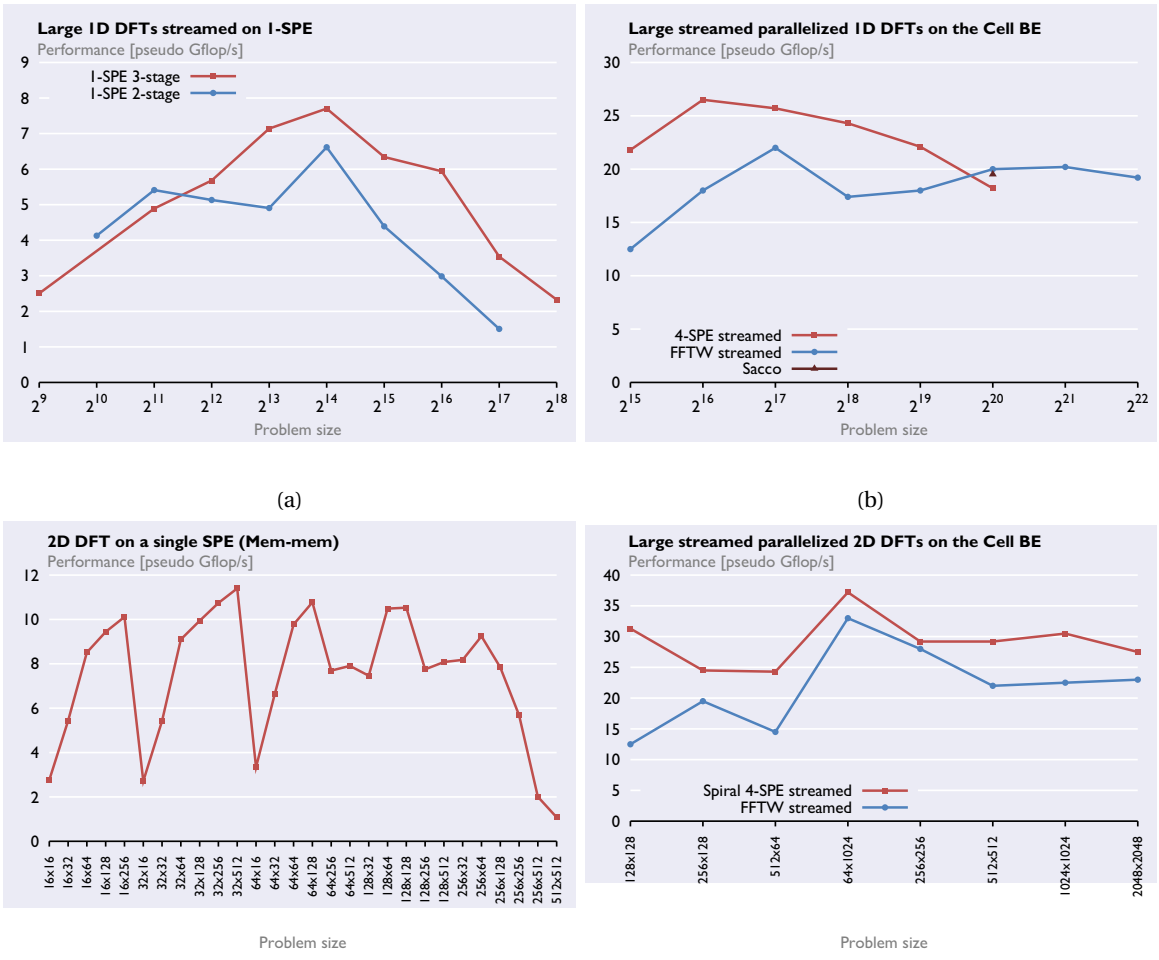
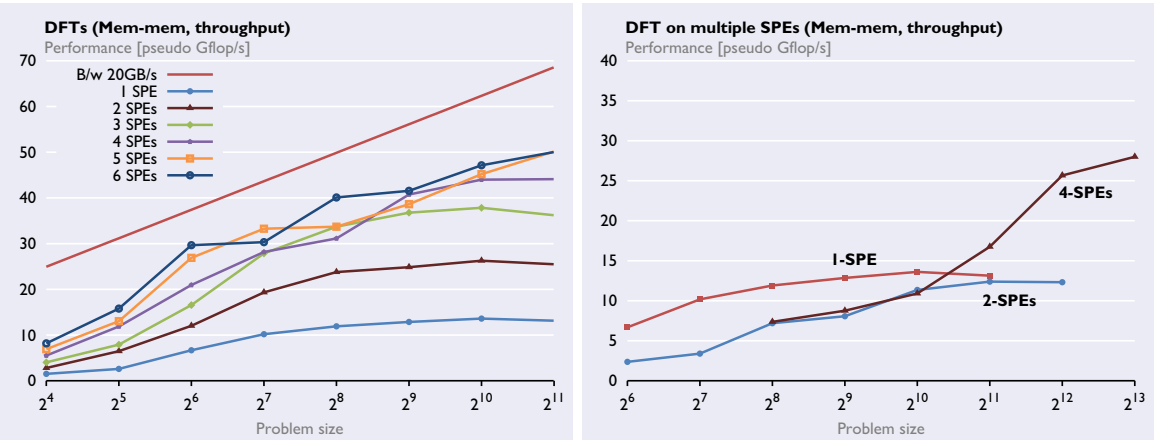


FIGURE 6.6: Latency optimized large DFTs streamed from main memory on the Cell BE.



(a) (b)

FIGURE 6.7: Throughput optimized DFTs on the Cell BE.

CHAPTER 7

Conclusion

We restate from our introduction:

> *The goal of this thesis is the computer generation of high-performance DFT libraries for distributed memory parallel processing systems, given only a high-level description of DFT algorithms and a set of platform and architecture parameters.*

In this thesis, we have presented a prototypical system that achieves this goal. Writing and tuning DFT libraries for distributed memory platforms is both important and difficult, and the main contribution of this thesis is the framework to formalize and automate the process and its actual implementation as a program generator.

As we discussed in Chapter 1, performance increases in computer platforms for the foreseeable future are expected to come primarily from the scaling of parallelization. The distributed memory paradigm is a vital component in scaling parallelization, which makes the paradigm a likely mainstay for the foreseeable future.

The key observation that shaped our approach and solution is that communication costs can dominate run time and lead to poor performance. Thus, it is important to design algorithms that explicitly manage data transfer and minimize communication costs, which we accomplished by addressing all the main facets of the problem. First, we design parallel algorithms that accomplish data transfer using only packets large enough to minimize overhead costs and maximize achieved bandwidth. Second, we explicitly manage data transfers and perform them concurrently with computation (this requires architectural support), thus hiding data transfer costs. Third, when acceptable by the application, we present data distributions that can reduce communication costs. Finally, we show traversals of the DFT dataflow graph that can maximize the

performance of banked memory systems. In addition, we designed our techniques to be compatible with previous work that target optimizations for other architectural paradigms including SIMD vectorization.

In summary, our system enables the computer generation of high performance DFT libraries for current and future platforms that incorporate the distributed memory design at the chip level or the node level.

7.1 Current Limitations

As with most research work, our system has several limitations, which we discuss below.

Option space limitations.. The cross product of all options and scenarios that a system could generate code for is huge, as can be seen in Table 1.1. We do not support this entire space, in part due to the engineering effort involved.

Problem dimensions. We focus on 1D DFTs, and also produce code for 2D DFTs in this thesis, but do not consider 3D DFTs. Since these are easily expressed in SPL, an extension of our work to cover 3D DFTs should be straightforward.

Problem size limitations. In this thesis, we focus on two-power sizes. However, our techniques should be easily extensible to support non-two power composite sizes composed of small (≤ 7) primes that are are compatible with the underlying SIMD vectorization paradigm. Vectorization requires the sizes to be multiples of the square of the SIMD vector length. This would primarily be an engineering limitation. Extension to other sizes including larger primes may require the use of other FFT algorithms in addition to techniques similar to the ones described in [Franchetti and Püschel, 2007].

Other programming paradigms. Although we did not target programming paradigms such as UPC and Chapel, which are primarily useful for applications that use them, we do demonstrate that our system works with at least two different programming paradigms: a shared memory pthreads based DMA paradigm, and the distributed memory MPI paradigm. Since we perform most of our rewriting at a level where the programming paradigm itself is abstracted out, our work should be extensible to generate libraries for other programming paradigms with appropriate changes of the backend.

7.2 Future Directions

In addition to addressing the limitations discussed above, we discuss possibilities for extending this research work.

One area that warrants research is that of applying our parallelization and streaming techniques to compute DFTs on emerging hybrid architectures. In addition to a distributed memory design, such architectures may incorporate multicore parallelism, SIMD vectorization, GPUs, and FPGAs at each node. Further, distributed memory designs may exist at multiple levels of the parallelism hierarchy in such systems.

Another area of research is the extension of our work to transforms other than the DFT. Although we focus on DFT libraries, our approach is based on parallelizing at the SPL level, and hence should be applicable to other linear transforms that can be expressed in SPL.

Based on the recent generalization of SPL to the Operator Language (OL) [de Mesmay, 2010; Franchetti et al., 2009a], library generation for linear algebra functionality may be possible.

Bibliography

Jose Ignacio Aliaga, Francisco Almeida, Jose Manuel Badía, Sergio Barrachina, Vicente Blanco, Maria Castillo, Rafael Mayo, Enrique S. Quintana, Gregorio Quintana, Alfredo Remón, Casiano Rodríguez, Francisco Sande, and Adrian Santos. Toward the parallelization of gsl. *J. Supercomputing*, 48(1): 88–114, 2009. ISSN 0920-8542. 8

E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 3rd edition, 1999. ISBN 0-89871-447-8 (paperback). 12

David A. Bader and Virat Agarwal. FFTC: Fastest Fourier transform for the IBM Cell Broadband Engine. In *IEEE Intl. Conference on High Performance Computing*, pages 172–184, 2007. 2, 123

Gerald Baumgartner, Alexander Auer, David E. Bernholdt, Alina Bibireata, Venkatesh Choppella, Daniel Cociorva, Xiaoyang Gao, Robert J. Harrison, So Hirata, Sriram Krishanmoorthy, Sandhya Krishnan, Chi-Chung Lam, Qingda Lu, Marcel Nooijen, Russell M. Pitzer, J. Ramanujam, P. Sadayappan, and Alexander Sibiryakov. Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. *Proc. of the IEEE*, 93(2), 2005. special issue on "Program Generation, Optimization, and Adaptation". 12

L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997. ISBN ISBN 0-89871-397-8. 12

Andreas Bonelli, Franz Franchetti, Juergen Lorenz, Markus Püschel, and Christoph W. Ueberhuber. Au-

tomatic 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. 8, 27, 48, 55, 57, 71, 77

Kang Chen and Jeremy R. Johnson. A self-adapting distributed memory package for fast signal transforms. In *Proc. IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS)*, 2004. 8, 55, 56

Alex C. Chow, Gordon C. Fossum, and Daniel A. Brokenshire. A programming example: Large FFT on the Cell Broadband Engine. Technical report, May 2005. 115

Alex Chunghen Chow. Fast Fourier transform SIMD code generators for synergistic processor element of Cell processor. In *Workshop on Cell Systems and Applications*, 2008. 121

L. Cico, R. Cooper, and J. Greene. Performance and Programmability of the IBM/Sony/Toshiba Cell Broadband Engine Processor. In *Proc. of (EDGE) Workshop*, 2006. 121

J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. of Computation*, 19:297–301, 1965. 23, 27

Frédéric de Mesmay. *On the Computer Generation of Adaptive Numerical Libraries*. PhD thesis, Electrical and Computer Engineering, Carnegie Mellon University, 2010. 133

Frédéric de Mesmay, Srinivas Chellappa, Franz Franchetti, and Markus Püschel. Computer generation of efficient software Viterbi decoders. In *International Conference on High Performance Embedded Architectures and Compilers (HiPEAC)*, volume 5952 of *Lecture Notes in Computer Science*, pages 353–368. Springer, 2010. 28

FFTW. FFTW on the Cell processor, 2008a. `http://www.fftw.org/cell`. 123, 124, 125

FFTW. FFTW website, 2008b. `http://www.fftw.org/`. 120

Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computing*, C-21:948, 1972. 39

MPI Forum. Message passing interface, 1998a. http://www.mpi-forum.org/. 44, 45

MPI Forum. MPI 2.0 specifications, 1998b. http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html. 41

Franz Franchetti, Frédéric de Mesmay, Daniel McFarlin, and Markus Püschel. Operator language: A program generation framework for fast kernels. In *IFIP Working Conference on Domain Specific Languages (DSL WC)*, volume 5658 of *Lecture Notes in Computer Science*, pages 385–410. Springer, 2009a. 8, 133

Franz Franchetti and Markus Püschel. SIMD vectorization of non-two-power sized FFTs. In *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, volume 2, pages II–17, 2007. 115, 132

Franz Franchetti, Markus Püschel, Yevgen Voronenko, Srinivas Chellappa, and José M. F. Moura. Discrete Fourier transform on multicore. *IEEE Signal Processing Magazine, special issue on "Signal Processing on Platforms with Multiple Cores"*, 26(6):90–102, 2009b. 23

Franz Franchetti, Yevgen Voronenko, Peter A. Milder, Srinivas Chellappa, Marek Telgarsky, Hao Shen, Paolo D'Alberto, Frédéric de Mesmay, James C. Hoe, José M. F. Moura, and Markus Püschel. Domain-specific library generation for parallel software and hardware platforms. In *NSF Next Generation Software Program Workshop (NSFNGS) colocated with IPDPS*, 2008. 28

Franz Franchetti, Yevgen Voronenko, and Markus Püschel. Formal loop merging for signal transforms. In *Programming Languages Design and Implementation (PLDI)*, pages 315–326, 2005. 8, 33, 55, 74, 76, 78, 100, 112

Franz Franchetti, Yevgen Voronenko, and Markus Püschel. FFT program generation for shared memory: SMP and multicore. In *Supercomputing (SC)*, 2006a. 8, 28, 30, 32, 45, 48, 49

Franz Franchetti, Yevgen Voronenko, and Markus Püschel. A rewriting system for the vectorization of signal transforms. In *High Performance Computing for Computational Science (VECPAR)*, volume 4395 of *Lecture Notes in Computer Science*, pages 363–377. Springer, 2006b. 30, 31, 45, 50, 58, 69, 73, 80, 109, 112

Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proc. of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):216–231, 2005. 2, 7, 35, 56, 82, 87, 120, 123

FSF. GNU Scientific Library, 2010. http://www.gnu.org/software/gsl/. 7

John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. FLAME: Formal linear algebra methods environment. *TOMS*, 27(4):422–455, December 2001. 12

David R. Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Software: Practice and Experience*, 20(1):5–12, 1989. 114

Michael A. Harrison. *Introduction to Formal Language Theory*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1978. ISBN 0201029553. 22

M. Hegland. Block algorithms for FFTs on vector and parallel computer. In *Parallel Computing: Trends and Applications*, pages 129–136. North-Holland, 1994. 26

M. T. Heideman, D. H. Johnson, and C. S. Burrus. Gauss and the history of the fast Fourier transform. *Archive for History of Exact Sciences*, 34:265–277, 1985. 23

John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach, (Third Edition)*. Morgan Kaufmann, 2003. 41

Fast Fourier Transform Library Programmer's Guide and API Reference. IBM, 3.0 edition, 2008. 121

IBM. Parallel Engineering and Scientific Subroutine Library (ESSL), 2010. http://www-03.ibm.com/systems/software/essl/. 8

J. R. Johnson, R. W. Johnson, D. Rodriguez, and R. Tolimieri. A methodology for designing, modifying, and implementing FFT algorithms on various architectures. *Circuits Systems Signal Processing*, 9: 449–500, 1990. 7, 16

D. Mirković and S. L. Johnsson. Automatic performance tuning in the UHFFT library. In *Proc. Int'l Conf. Computational Science (ICCS)*, volume 2073 of *LNCS*, pages 71–80. Springer, 2001. 7

A. Norton and A. J. Silberger. Parallelization and performance analysis of the Cooley-Tukey FFT algorithm for shared-memory architectures. *IEEE Trans. Comput.*, 36(5):581–591, 1987. ISSN 0018-9340. 26

M. C. Pease. An adaptation of the fast Fourier transform for parallel processing. *Journal of the ACM*, 15: 252–264, 1968. 27

W. H. Press, B. P. Flannery, Teukolsky S. A., and Vetterling W. T. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 2nd edition, 1992. 2, 27

Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Pro-*

gram Generation, Optimization, and Adaptation", 93(2):232– 275, 2005. 7, 22, 28, 112

Sharon Sacco. Large multicore ffts: Approaches to optimization. In *Proc. High Performance Embedded Computing (HPEC)*, 2008. 125

Paul N. Schwarztrauber. Multiprocessor FFTs. *Parallel Computing*, 5:197–210, 1987. 27

Tim Stitt. An introduction to the partitioned global address space (pgas) programming model, 2010. Connexions http://cnx.org/content/m20649/1.7/. 41

Daisuke Takahashi. A blocking algorithm for parallel 1-D FFT on shared-memory parallel computers. *Lecture Notes in Computer Science*, 2367:380–389, 2002. 7, 8, 26

TOP500. Top 500 supercomputing sites, 2010. http://top500.org/. 4

Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990. ISSN 0001-0782. 72

Charles Van Loan. *Computational Framework of the Fast Fourier Transform*. SIAM, 1992. 16, 23, 24, 26, 27

Field G. Van Zee, Paolo Bientinesi, Tze Meng Low, and Robert A. van de Geijn. Scalable parallelization of FLAME code via the workqueuing model. *ACM Trans. Math. Softw.*, 34(2):1–29, 2008. ISSN 0098-3500. 12

Field G. Van Zee, Ernie Chan, Robert van de Geijn, Enrique S. Quintana-Ortí, and Gregorio Quintana-Ortí. Introducing: The libflame library for dense matrix computations. *IEEE Computation in Science & Engineering*, 11(6):56–62, 2009. 12

Yevgen Voronenko. *Library Generation for Linear Transforms*. PhD thesis, Electrical and Computer Engineering, Carnegie Mellon University, 2008. 8, 35, 112, 114

Yevgen Voronenko, Frédéric de Mesmay, and Markus Püschel. Computer generation of general size linear transform libraries. In *International Symposium on Code Generation and Optimization (CGO)*, pages 102–113, 2009. 35

R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001. 12

Jianxin Xiong, Jeremy Johnson, Robert W. Johnson, and David Padua. SPL: A language and compiler for

DSP algorithms. In *Programming Languages Design and Implementation (PLDI)*, pages 298–308, 2001. 15