Automatic Library Generation and Performance Tuning for Modular Polynomial Multiplication

A Thesis
Submitted to the Faculty
of
Drexel University
by
Lingchuan Meng
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer Science
June 2015
Dedications

To my wife, Xiaoyu, for being the love of my life.

To my parents, Xianghua and Shumei, for all their love, support and patience over the years.
Acknowledgments

I would like to begin by thanking my advisor, Dr. Jeremy Johnson for providing me with the endless source of inspiration and numerous days proofreading the drafts of the papers and this thesis.

My special thanks extend to the members of my thesis examining committee: Dr. Bruce Char, Dr. Franz Franchetti, Dr. David Saunders and Dr. Ali Shokoufandeh. They have all provided great feedback on this document, as well as some lively discussion and valuable insights.

This work would not be possible without the contributions made by the SPIRAL team members. Yevgen Voronenko, Volodymyr Arbatov, and Richard Veras have patiently helped me with understanding, extending and enhancing the amazing SPIRAL project.

I have been fortunate to have discussed the topics in this thesis with some top researchers: Marc Moreno Maza and his students provided me with the modpn library and optimization ideas for polynomial arithmetic. Daniel Roche and Andrew Arnold offered in-depth discussions on the truncated Fourier transform. Pascal Giorgi shared some great ideas with me on the low-level optimization for various hardware features.

I have always enjoyed the enlightening conversation with my fellow students in the research lab and the TA office at Drexel University. A particular thanks goes to my lab mates over the years, Mark Boady, Gavin Harrison, Douglas Jones, Danh Nguyen and Tim Chagnon.

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

**LIST OF FIGURES** ........................................................................................................ vi
**LIST OF TABLES** ........................................................................................................... viii
**ABSTRACT** ......................................................................................................................... ix

1. Introduction ......................................................................................................................... 1
   1.1 Motivation ....................................................................................................................... 1
       1.1.1 Importance of Polynomial Arithmetic .................................................................... 2
       1.1.2 Difficulty of High Performance Implementations ................................................. 3
       1.1.3 Limitations of Hand-tuned Libraries .................................................................... 5
   1.2 Problem Statement ......................................................................................................... 6
   1.3 Related Work and Limitations of Exact Polynomial Libraries .................................... 9
       1.3.1 Automatic Performance Tuning .............................................................................. 9
       1.3.2 Polynomial Arithmetic .......................................................................................... 10
       1.3.3 SPIRAL ................................................................................................................. 11
   1.4 Contribution of the Thesis ............................................................................................ 12
   1.5 Organization of the Thesis ............................................................................................ 14

2. Background ........................................................................................................................ 18
   2.1 Transforms and Polynomials ....................................................................................... 18
   2.2 Fast algorithms: SPL, $\Sigma$-SPL, and OL ................................................................. 25
       2.2.1 SPL ....................................................................................................................... 26
       2.2.2 $\Sigma$-SPL ............................................................................................................. 31
       2.2.3 OL ......................................................................................................................... 34
   2.3 SPIRAL and Library Generation ..................................................................................... 36
   2.4 Autotuned Parallelism ................................................................................................. 40
       2.4.1 Vectorization .......................................................................................................... 41
       2.4.2 Parallelization ........................................................................................................ 42
   2.5 Conclusion ..................................................................................................................... 44

3. Modular Arithmetic ............................................................................................................. 46
   3.1 Hardware Support for Integer Arithmetic ...................................................................... 46
   3.2 Fast Algorithms ............................................................................................................. 51
       3.2.1 Fast Modular Multiplication Algorithms ............................................................. 52
       3.2.2 Vectorized Montgomery Algorithm ........................................................................ 55
   3.3 SPIRAL Extensions for Modular Arithmetic ............................................................... 58
   3.4 Conclusion ..................................................................................................................... 61

4. Modular FFT ....................................................................................................................... 63
   4.1 Modular DFT and Fast Algorithms ................................................................................. 63
   4.2 Library Generation ......................................................................................................... 67
   4.3 Performance Evaluation ............................................................................................... 70
   4.4 Conclusion ..................................................................................................................... 73

5. TFT and ITFT ..................................................................................................................... 75
   5.1 Background ................................................................................................................... 75
   5.2 TFT .............................................................................................................................. 78
5.2.1 Strict General-radix TFT Algorithm ........................................ 78
5.2.2 Parallel TFT Algorithm .......................................................... 83
5.2.3 Performance Evaluation ......................................................... 87
5.3 ITFT ...................................................................................... 90
5.3.1 Strict General-radix ITFT Algorithm ..................................... 90
5.3.2 Parallel ITFT Algorithm ....................................................... 102
5.3.3 Performance Evaluation ......................................................... 107
5.4 Conclusion ........................................................................... 108
6. Modular Polynomial Multiplication .............................................. 110
  6.1 Background ........................................................................ 110
     6.1.1 Convolution Theorem .................................................... 113
  6.2 Fast Algorithms .................................................................. 115
  6.3 SPIRAL Enhancements ......................................................... 117
  6.4 Performance Evaluation ........................................................ 119
  6.5 Conclusion .......................................................................... 120
7. Conclusion ............................................................................. 122
  7.1 Summary ............................................................................ 122
  7.2 Future Work ....................................................................... 123
Bibliography ............................................................................. 125
List of Figures

1.1 Input and output to the library generator. The high-performance generated libraries consist of three main components: recursive functions, base cases and target infrastructure. ................................................. 7

1.2 Performance comparison between the SPIRAL-generated TFT-based convolution library and modular DFT-based convolution library. ................................................. 13

1.3 A decomposition example of ITFT_{32,22,19}, where ○ represents the input values and • represents the untransformed values to be computed. ............................ 13

1.4 Performance comparison between the SPIRAL-generated parallel ITFT library and the SPIRAL-generated parallel inverse modular DFT library that pads to powers of two. ...................................................... 14

1.5 The vectorized Montgomery algorithm ......................................... 15

2.1 Classic program generation framework in SPIRAL .......................... 37

2.2 Library generation in SPIRAL ................................................ 39

3.1 A typical SIMD operation on \( v \)-way vector operands ......................... 47

3.2 2-way vectorized multiplication with floating point shuffle and cast 49

3.3 2-way vectorized multiplication with integer shuffle and unpack .......... 50

3.4 2-way vectorized multiplication with integer blend ........................... 51

3.5 The vectorized Montgomery algorithm ......................................... 57

3.6 An example of vector code generation for modular arithmetic in SPIRAL 61

4.1 Representation of the matrix factorization based on the Cooley-Tukey algorithm. Shades of gray represent values that belong to the same tensor substructure. ................................................................. 66

4.2 Performance comparison among the SPIRAL-generated modular FFT library and fixed-size code and the modular FFT implemented in modpn .... 71
5.1 Schematic representation of a radix-2 TFT whose transform size is 16 and input and output sizes are both 11 .................................................. 76

5.2 A decomposition example of TFT_{32,17,30}, where ○ represents the input values and ● represents the output values ............................................. 80

5.3 Possible factorization path of the relaxed general-radix algorithm ........ 85

5.4 An example of factoring TFT_{32,17,17} with the relaxed general radix TFT algorithm ................................................................. 87

5.5 Performance comparison between the SPIRAL-generated parallel TFT library and the SPIRAL-generated parallel modular DFT library that pads to powers of two .................................................. 89

5.6 A decomposition example of ITFT_{32,22,19}, where ○ represents the input values and ● represents the untransformed values to be computed ........ 97

5.7 An instance of the partial data coverage in each step of computing ITFT_{8,7,5,0}. The lines from northwest to southeast represent the input domain, and the lines from northeast to southwest represent the output range ............................................................................. 101

5.8 An example of factoring ITFT_{32,17,17} with relaxation .......................... 104

5.9 Co-generation/co-planning between TFT and ITFT ............................... 106

5.10 Performance comparison between the SPIRAL-generated parallel ITFT library and the SPIRAL-generated parallel inverse modular DFT library that pads to powers of two .............................................. 109

6.1 Extension in the base case generator to support multiple base case sets in one library ................................................................. 118

6.2 Performance comparison between the SPIRAL-generated TFT-based convolution library and modular DFT-based convolution library .... 121
List of Tables

1.1 A typical input to the extended code generator ........................................... 6

2.1 Translating SPL structures to code......................................................... 30

2.2 Translating $\sum$-SPL structures to code............................................. 33

3.1 A snapshot of a sample ISA ................................................................. 60
Abstract
Automatic Library Generation and Performance Tuning for Modular Polynomial Multiplication
Lingchuan Meng
Advisor: Jeremy Johnson, Ph.D.

Polynomial multiplication is a key algorithm underlying computer algebra systems (CAS) and its efficient implementation is crucial for the performance of CAS. In this context coefficients of polynomials come from domains such as the integers, rationals and finite fields where arithmetic is performed exactly without rounding error.

Obtaining peak performance on modern computer architectures is difficult due to the complexity of the memory hierarchy, and the need for efficient parallelism on multiple levels ranging from instruction level parallelism (ILP) and vector parallelism to multi-core parallelism. Tuning the performance of code to fully utilize all of these components requires modifying the implementation to the particular architecture and extensive experimentation to determine the best choices of algorithm and implementation strategies. Moreover, this time consuming process must be repeated whenever the architecture changes, as the particular choices for one machine are not optimal for another, even when relatively minor changes are made in the architecture.

In numeric (floating point based) scientific and mathematical libraries where performance is crucial, considerable effort has been made to optimize key routines across multiple architectures. This performance tuning has been aided by the use of automation where many code choices are generated and intelligent search is utilized to find the “best” implementation on a given architecture. The performance of autotuned implementations is comparable to, and in some cases better than, the best hand-tuned code.

In this thesis we design and implement algorithms for polynomial multiplication
using a fast Fourier transform (FFT) based approach. We improve on the state-of-the-art in both theoretical and practical performance. The SPIRAL library generation system is used to automatically generate and tune the performance of a polynomial multiplication library that is optimized for memory hierarchy, vectorization and multi-threading, using new and existing algorithms.

Many implementations of FFT-based polynomial multiplication use a power of two FFT. Inputs are zero-padded to the smallest power of two greater than the output size, and the resulting performance exhibits a “staircase” phenomenon where the computing time for problem sizes between powers of two is essentially equal to the time for using the larger power-of-two transform.

New algorithms for the truncated Fourier transform (TFT) and the inverse (ITFT) are presented that smooth the staircase phenomenon by exploiting the fact that some of the inputs are zero and not all of the outputs are needed. The new general-radix algorithms presented in thesis have the same asymptotic complexity of the fastest existing approaches, but extend the search space so that implementations can best adapt to the memory hierarchy. The new parallel algorithms introduce a small relaxation for larger problem sizes which trades off slightly higher arithmetic cost for improved data flow which allows full vectorization and parallelization.

The SPIRAL library generator is extended and enhanced to completely automate library implementation and optimization for polynomial multiplication, including the modular discrete Fourier transform, truncated Fourier transform, convolution and polynomial multiplication. The input to the generator is a specification of recursive algorithms for polynomial multiplication expressed in a high-level domain-specific language; the output is a vectorized and multi-threaded C++ library that supports general input size. The resulting libraries smooth out the staircase phenomenon while retaining the performance associated with state-of-the-art power of two libraries.
1. Introduction

This thesis addresses the automatic generation and performance tuning of a library for modular univariate polynomial multiplication. We use program generation and performance tuning techniques to obtain high performance implementations of new and existing algorithms that give both practical and theoretical performance improvements. Our library utilizes asymptotically fast algorithms based on the fast Fourier transform (FFT). The library is constructed and tuned in a bottom-up fashion. First, we provide efficient implementation of modular arithmetic. Second, fast libraries are built for the modular DFT and TFT using new general-radix and parallel algorithms. Finally, the DFT and TFT libraries are used for FFT-based convolution algorithms for polynomial multiplication.

In the introduction, we first discuss the motivation for autotuned high performance polynomial multiplication library. Second, we provide a formal problem statement of the thesis from both the algorithmic and the program generation perspectives. Third, we review the state-of-the-art of related work as well as their limitations. Fourth, we summarize the contribution of the thesis, including new algorithms and optimization techniques, and extensions and enhancements for a library generation system. Finally, we briefly introduce the organization of this thesis.

1.1 Motivation

The motivation for polynomial arithmetic library development is three-fold:

- Efficient polynomial arithmetic is a key component of CAS.
- High performance implementations become increasingly difficult to achieve with the recent hardware accelerations requiring extensive parallelism.
• Most high performance implementations in CAS are obtained by repetitive hand-tuning.

1.1.1 Importance of Polynomial Arithmetic

Polynomial arithmetic is a key component of symbolic computation, scientific computing and cryptography. CAS and certain cryptographic algorithms perform computations using exact arithmetic with integers, rational numbers, polynomials, rational functions, algebraic numbers and functions, and finite fields. Many algorithms for computing in those domains use modular techniques, where computations are performed modulo several integers or polynomials and combined using the Chinese Remainder Theorem or lifted for a single modular integer to obtain the solution in the original domain. Hence there is a need for efficient computation with modular polynomials. Moreover, multiplication with integer can be based on fast modular polynomial multiplication using the three primes algorithm or other related approaches. Furthermore, multivariate polynomial multiplication can be reduced to univariate polynomial multiplication using evaluation and interpolation. This justifies our focus on fast modular polynomial multiplication.

The performance of polynomial arithmetic has an essential impact on the overall performance of higher-level algorithms and systems that rely on it. High performance implementations of polynomial arithmetic have become increasingly difficult to achieve on modern processors, despite the abundance of fast algorithms and the peak performance they provide.

Many algorithms and techniques developed for numeric libraries can be used for exact polynomial arithmetic library generation and performance tuning. However, the nature of exact polynomial arithmetic, e.g., use of integer hardware and the need for division, along with the need for fast computation for small and large sizes and of
all degrees, will require altering and extending the existing algorithms and techniques, and more intriguingly, inventing new ones.

The importance of polynomial arithmetic, together with the difficulty of obtaining fast implementations on modern processors and the need to reoptimize for each processor all make the automatic library generation and performance tuning for polynomial arithmetic necessary, akin to the BLAS in numerical linear algebra.

1.1.2 Difficulty of High Performance Implementations

Complex and diverse designs in modern computer architectures, including instruction level parallelism, deep memory hierarchy, vector instructions and parallelization, require programmers with extensive domain knowledge and programming skills to repeat hand optimizations whenever a new platform becomes available, albeit the difference may only be an addition of a few new instructions. To complicate matters further, hardware vendors are typically biased against integer-related optimizations when facing design and cost constraints, which leads to delayed or missing optimizations compared to the floating-point optimizations. Although general purpose compilers with powerful optimization techniques provide very efficient code, carefully hand optimized implementations can typically outperform straightforward implementations by several orders of magnitude. Thus for key kernel routines, such as polynomial multiplication, we can not rely on the compiler to provide the desired performance.

Modern general-purpose processors have three types of on-chip parallelism: instruction-level parallelism (ILP) supported by the instruction pipelining and superscalar techniques, vector parallelism using the ISA (instruction-set architecture) vector extensions, and multithreaded parallelism supported by the multi-core architectures.

Instruction-level parallelism (ILP) was first realized in mid 1980s by using the
instruction pipelining, in which the processor works on multiple instructions in different stages of completion. A few years later, a similar idea called superscalar design emerged where multiple instructions are executed in parallel on separate arithmetic logic units (ALUs). ILP enables the parallelism by means out-of-order execution where instructions execute in any order that does not violate data dependencies. ILP is the only type of on-chip parallelism that requires no software development effort, although it may benefit from unrolling and instruction scheduling.

Vector parallelism, mostly SIMD (single instruction, multiple data) on commodity CPUs, is realized by adding vector extensions to the ISA to enable processing multiple data elements (vectors) in parallel. The SIMD parallelism can be exploited by directly using assembly instructions, by using the C/C++ intrinsics, or by relying on automatic vectorization techniques in general-purpose compilers. The latter tends to be suboptimal for most polynomial arithmetic problems due to lack of domain knowledge and high-level optimization. Obtaining full vectorization typically requires modifying the underlying algorithm and its implementations.

Thread-level parallelism (TLP) is the coarse-grain parallelism supported by incorporating multiple processing cores into a single CPU. TLP becomes increasingly important as the hardware vendors hit the clock frequency ceiling due to physical limitations, and shift to adding more cores onto the same die. Exploiting TLP requires programs with multiple threads of control.

Further complicating matters, the exponential growth of CPU speed has not been matched by the increase in memory bandwidth, resulting in the so-called “memory wall”, an overwhelming bottleneck in computer performance. Multi-level cache hierarchy, prefetching, etc. are designed to address the problem, which consequently require a high performance library to be optimized and tuned to the complex and deep memory hierarchy.
Finally, additional difficulty exists due to the requirement of integer and modular arithmetic in our implementations because of limited or delayed hardware support. Hardware vendors often have to prioritize hardware optimizations when facing design constraints. Due to the popularity of floating point numeric computation and the complexity of some integer algorithms, hardware optimizations are often biased against the integer-related operations. For example, integer division and mod are notoriously slow compared to floating-point division on the scalar level. The disparity is further widened as the latter can be performed at a higher throughput (4, 8 or 16) with the SIMD extensions. Even when equivalent integer and floating-point instructions are both supported by the SIMD extension, e.g. SSE (streaming SIMD extensions) and AVX (advanced vector extensions), the implementation of the vectorized integer instructions is often delayed. As we will discuss in Chapter 3, algorithms and optimization techniques must be developed to overcome the aforementioned limitations.

1.1.3 Limitations of Hand-tuned Libraries

Traditionally, polynomial arithmetic libraries are manually developed and tuned with limited vectorization and multi-threading. This requires the programmers to have extensive domain knowledge and programming skills. In addition, there are a large number of algorithms and implementation choices, along with too many tunable parameters for the programmer to explore by hand. Furthermore, a library tuned to a specific platform is in general not portable, which means that performance-critical libraries or subroutines must be reoptimized or reimplemented for new platforms.
1.2 Problem Statement

Motivated by the importance of polynomial arithmetic, the difficulty of obtaining high performance implementations, and the limitations of the existing polynomial libraries, the development of a high performance polynomial arithmetic library calls for automatic program generation and performance tuning, and new algorithms and optimization techniques.

To develop and implement a portable high performance library for modular polynomial multiplication that fully take advantage of all levels of parallelism available on modern processors.

The proposed library will be automatically generated and optimized for a high level specification of the algorithms that are used. The high level specification allows multiple algorithm choices to be explored without requiring extensive programming. Moreover, code generation will be utilized to provide vectorization and parallelism, thus allowing the library to adapt to different processors with different vector instructions and varying amounts of parallelism with different requirement for their full utilization. Automatic tuning is used to provide platform adaptability.

For example, a typical input to the extended code generator is

<table>
<thead>
<tr>
<th>Transform: ( \text{ModDFT}_{n,p,w} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{ModDFT}<em>{rs,p,w} \rightarrow (\text{ModDFT}</em>{r,p,w} \otimes I_s) T_p^{r,s}(I_r \otimes \text{ModDFT}_{s,p,w}) L_r^{s} )</td>
</tr>
<tr>
<td>( \text{ModDFT}_2 \rightarrow \begin{bmatrix} 1 &amp; 1 \ 1 &amp; -1 \end{bmatrix} )</td>
</tr>
<tr>
<td>Modular arithmetic: yes</td>
</tr>
<tr>
<td>Vectorization: 4-way SSE</td>
</tr>
<tr>
<td>Multithreading: yes</td>
</tr>
<tr>
<td>Multiple base case: no</td>
</tr>
<tr>
<td>Guided generation: no</td>
</tr>
</tbody>
</table>

Table 1.1: A typical input to the extended code generator
Figure 1.1: Input and output to the library generator. The high-performance generated libraries consist of three main components: recursive functions, base cases and target infrastructure.

The output is an autotuned library that is:

- optimized for modular arithmetic on the scalar and vector levels;
- vectorized using the available SIMD instructions;
- multi-threaded with a variable number of threads;
- optionally constructed with multiple base case patterns;
- optionally guided in the library generation or planning phase by a related operation;
- competitive in performance compared with the best available hand-tuned libraries.

Figure 1.1 illustrate the input and output of the library generator using the modular FFT as an example. The generator takes a high-level abstraction of the re-
cursive algorithm expression in the domain specific language, and generates a high-performance library automatically. The three main components forming the generated libraries are:

- A set of recursive functions derived based on the high-level recursive algorithms and needed to compute general-size inputs;

- A set of optimized base case implementations for small problem sizes which may also be looped and vectorized, serving as the termination of the recursion;

- An infrastructure responsible for function initialization, precomputation and adaptive search.

Our library uses FFT-based algorithms and classical approaches, as the asymptotically faster FFT-based approaches outperform classical methods for relatively small size polynomials. Since FFT libraries usually favor power of two sizes, this can lead to a “staircase” performance where multiplication of polynomials takes time approximately equal to the time for the larger powers of two. Our library explores various strategies, based on the TFT, to avoid this. This is crucial as users require inputs of all degrees.

Our library can serve as the foundation of a more general library for polynomial and integer arithmetic. Many basic polynomial arithmetic operations such as division can be efficiently reduced to multiplication. Moreover, calculations with univariate and multivariate polynomials can be reduced to computing with polynomials over finite fields, such as the prime field \( \mathbb{Z}_p \) for a prime number \( p \), via the so-called modular techniques. Additionally, multivariate polynomial arithmetic can be reduced to univariate polynomial arithmetic via evaluation and interpolation. Furthermore, most calculations tend to densify intermediate expressions even when the input and output polynomials are sparse. Therefore, the dense polynomial multiplication addressed in
this thesis can be beneficial to the sparse problems. This is not to say that their operations could not be more efficiently implemented directly; however, they would certainly benefit from our highly tuned library. This justifies our choice of modular polynomial multiplication and allows us to focus on the autotuning, which is the main objective of the thesis.

1.3 Related Work and Limitations of Exact Polynomial Libraries

This section reviews related work on automatic performance tuning, efficient polynomial arithmetic and the SPIRAL system. SPIRAL (http://www.spiral.net) is an extensible system for the automatic generation, optimization and platform adaptation of digital signal processing (DSP) algorithms. It is highlighted as our work will use and build on the SPIRAL system.

1.3.1 Automatic Performance Tuning

Automatic code generation and optimization systems have been proved to be able to automatically produce implementations comparable to, and in some cases better than, the best hand-tuned codes. Examples include ATLAS [7, 50] and FLAME [19] for dense linear algebra, SPARSITY [24] for sparse linear algebra, FFLAS/FFPACK [8, 9] for finite field linear algebra, and FFTW [13, 14] for the fast Fourier transform (FFT) and SPIRAL [25, 42, 49, 41] for more general DSP algorithms and library generation. Autotuning-based systems employ various techniques to produce fast code for a platform-specific architecture and instruction set with minimal human intervention. These techniques incorporate domain-specific knowledge and architecture features to provide large space of implementation candidates, and utilize empirical benchmarking, search and learning to achieve optimal performance.

To date, most work on autotuning has been done for numeric kernels used in sci-
entific computing and DSP. However, similar computations are needed for polynomial arithmetic where exact arithmetic is required, and yet only limited autotuning has been exploited. FFPACK [9] and FFLAS [8] are built on top of matrix multiplication and rely on highly tuned implementations of numerical BLAS library. Their symbolic routine uses ATLAS with some conversions in order to perform an exact matrix multiplication. Arithmetic in exact coefficient domains, such as arbitrary precision integers, rational numbers, and finite fields, in contrast to floating point arithmetic, are performed exactly without numeric error. Although some autotuning techniques can carry over to exact arithmetic, others need to be modified, extended or invented. Other applications requiring exact arithmetic include cryptography, computer algebra and theorem proving.

1.3.2 Polynomial Arithmetic

Despite the lack of autotuning for exact polynomial arithmetic, there has been extensive work on fast algorithms and hand-tuned high performance implementations. Algorithms and implementations for polynomial arithmetic have been developed with reduced arithmetic complexity, better adaptation to parallel architectures and improved space efficiency [15, 32]. In [46, 47], the author introduced a new variation of Fourier transform to be used by convolution, aiming to reduce arithmetic cost and avoid zero-padding. Then [20] improved the original radix-2 TFT/ITFT algorithm by optimizing for cache locality. [5] compared Chinese Remainder Theorem-based convolution algorithms with FFT-based algorithms, and presented a framework for convolution algorithm derivation. On top of transforms and convolutions, [21, 48] developed fast algorithm for univariate and multivariate polynomial multiplication. [26] is among the first publications to investigate algorithms for sparse polynomial arithmetic. Recently, [37] introduced parallel algorithms for sparse polynomial mul-
tiplication using heaps, and \cite{17, 23} focused on sparse interpolation for multivariate polynomial and for over finite fields, respectively.

However in aforementioned CAS, polynomial multiplication is typically implemented straightforwardly by definition, sometimes by the Karatsuba algorithm, and rarely by fast FFT-based algorithms. Recently, the \texttt{modpn} \cite{10, 30, 29, 39} library has been integrated into MAPLE \cite{31}, providing hand-optimized low-level routines implementing FFT-based fast algorithms for multivariate polynomial computations over finite fields, in support of higher-level code. The implementation techniques employed in \texttt{modpn} are often platform-dependent, since cache size, associativity properties and register sets have a significant impact and are platform specific.

1.3.3 SPIRAL

In this thesis, we use, extend and enhance the SPIRAL library generator for library generation and performance tuning. SPIRAL is a library generator that can generate stand-alone programs for DSP algorithms and numeric kernels. In contrast to other work in autotuning, SPIRAL uses internally the domain-specific language SPL to express divide-and-conquer algorithms for transforms as breakdown rules. For a user-specified transform and transform size, SPIRAL applies these rules to generate different algorithms, represented in SPL. Then \texttt{\sum}-SPL makes explicit the description of loops and index mappings. The \texttt{\sum}-SPL representation of algorithms are then processed by the built-in rewriting systems for performing difficult optimizations such as parallelization, vectorization, and loop merging automatically. The optimizations are performed at a high abstraction level in order to overcome known compiler limitations. The unparsers translate these algorithms into executable code. Based on the generated code, a search engine uses the dynamic programming technique to explore different choices of algorithms to find the best match to the computing platform. The
performance-based search is chosen over any high-level derivation as it can precisely predict the practical performance of the algorithms.

New transforms and kernels can be added by introducing new symbols and their definitions, and new algorithms can be generated by adding new rules. SPIRAL was developed for floating point and fixed point computation; however, since it uses a language to express problems instead of focusing on any specific problem, many of the transforms and algorithms can carry over to finite fields. For example, the DFT of size $n$ is defined when there is a primitive $n$th root of unity and many factorizations of the DFT matrix depend only on properties of primitive $n$th roots. In this case, the same machinery in SPIRAL can be used for generating and optimizing modular transforms and beyond.

1.4 Contribution of the Thesis

This thesis provides an automatically generated and optimized library for modular polynomial multiplication. We list the most important contributions developed in this thesis:

- We design and implement an efficient library for modular polynomial multiplication which is an order of magnitude faster than the state-of-the-art, as shown in Fig. 1.2. The library also provides efficient modular FFT and TFT.

- The vectorized and multi-threaded library is automatically generated and optimized from high level specification. The autotuning is done through extending and using SPIRAL.

- New algorithms are developed for the TFT and ITFT to better adapt to the memory hierarchy, and are more easily vectorized and parallelized, as shown in Fig. 1.3.
Figure 1.2: Performance comparison between the SPIRAL-generated TFT-based convolution library and modular DFT-based convolution library.

Figure 1.3: A decomposition example of ITFT_{32,22,19}, where ○ represents the input values and • represents the untransformed values to be computed.
Figure 1.4: Performance comparison between the SPIRAL-generated parallel ITFT library and the SPIRAL-generated parallel inverse modular DFT library that pads to powers of two

- The TFT and ITFT implementations smooth performance between powers of two compared to state-of-the-art power-of-two FFT performance, as shown in Fig. 1.4.

- We extend SPIRAL to fully automate the generation of high performance modular arithmetic which uses new and existing optimization techniques. An example is shown in Fig. 1.5.

1.5 Organization of the Thesis

In Chapter 2 we review the background of transforms and polynomials, their fast algorithms and corresponding expressions in the domain specific language and extensions, namely SPL (signal processing language), $\sum$-SPL and OL (operator language). Then we explain the general size library generation process in SPIRAL where the derivation of the library structure is performed. At the end of this chapter, we
Figure 1.5: The vectorized Montgomery algorithm
overview the parallelism framework that enables automatic vectorization and parallelization for the algorithms expressed in the domain specific language.

In Chapter 3, we focus on the automatic generation of the high performance modular arithmetic. We first review the hardware support for modular arithmetic. Then several fast algorithms for modular arithmetic are discussed and compared. Finally, we show how modular arithmetic is integrated into SPIRAL and automatically generated and optimized for both scalar and vector implementations.

In Chapter 4, we turn to the linear transforms. First, we investigate in the modular FFT. We show that some algorithms and techniques designed for the complex or real FFTs can be used for the modular FFT. The automatically generated modular FFT library is fully vectorized, multi-threaded and an order of magnitude faster than the baseline implementation representing the state-of-the-art of hand-optimized libraries. However, the modular FFT library and its fast algorithms exhibit the staircase phenomenon. This limitation calls for new algorithms that can utilize the known zero input values and truncated output values to smooth performance between powers of two with reduced the computational complexity.

In Chapter 5, we address the staircase phenomenon by introducing the TFT and ITFT. After reviewing the limitations of the existing algorithms and implementations, we follow by extending the fixed decomposition algorithms to general-radix algorithms that enable the search for optimal implementations that can better adapt to the memory hierarchy while maintaining the reduced complexity. Then we develop and implement parallel algorithms for TFT and ITFT that trade off a modest amount of extra arithmetic for improved vectorization and multi-thread parallelism. The TFT and ITFT do not explicitly recover the permuted data, as they are intended to be used together in convolution where the permutations can be canceled. Therefore, we extend SPIRAL to support the guided generation and planning between TFT
and ITFT. Finally, we show the performance gains of the generated TFT and ITFT libraries over the autotuned parallel modular FFT library.

In Chapter 6, we focus on the library generation and performance tuning for modular polynomial multiplication. The FFT-based fast algorithm is first reviewed. Then, the linear transforms from Chapter 4 and 5 are used as the building blocks for the polynomial multiplication library. We further introduce extensions to SPIRAL to support more complex library generation. Finally, we show the performance evaluation comparing the TFT-based library and power-of-two modular DFT-base library.
2. Background

In this chapter, we review the definitions of important linear transforms and polynomial arithmetic operations. We describe their fast algorithms in a domain specific language called SPL with extensions including the $\sum$-SPL and the OL. The $\sum$-SPL solves the formal loop merging problem in SPL and enables complex data manipulation. The OL extends the algorithm expressiveness and library generation from linear transforms to multi-linear kernels. The SPL and its extensions can be derived and translated into code by the SPIRAL library generator, which is also reviewed in this chapter: we begin with the automatic generation of general size library for arbitrary runtime sizes; then, we give an overview of the generation of vectorized and parallel libraries as we strive to obtain optimal performance on the diverse computing platforms.

2.1 Transforms and Polynomials

The DFT and its variations, including the modular DFT and TFT, serve as important building blocks in the fast algorithms for almost all non-trivial polynomial arithmetic operations. In this section, we formally define those transforms. Then, we review the definition of a polynomial and its most important operation, polynomial multiplication. The closely related convolution operations are also explained in this section.

Any linear transform can be defined as a sum of products or, more compactly, a matrix-vector product. The latter takes the form of

$$y = Mx,$$  \hfill (2.1)
where $M$ is the transform matrix, and $x$ and $y$ are the input and output vectors, respectively.

The DFT is a crucial operation in many areas of signal processing and computer science. FFTs exist that reduce the computational complexity from quadratic to $O(n \log n)$, by recursively factoring the DFT matrix $M$ into sparse matrices. In computer algebra, FFTs are used for fast polynomial and integer arithmetic and modular methods (i.e. computation by homomorphic images). Traditionally, the DFT is defined over the complex domain for real or complex inputs.

**Definition 1.** Given $n$ real or complex inputs $x_0, \ldots, x_{n-1}$, the DFT is defined as

$$y_k = \sum_{0 \leq l < n} \omega_n^{kl} x_l, \quad 0 \leq k < n,$$

(2.2)

where $\omega_n = \exp(-2\pi i/n)$ and $i$ is the imaginary unit.

The equivalent matrix view of $\text{DFT}_n$ is:

$$\text{DFT}_n = [\omega_n^{kl}]_{0 \leq k,l < n}.$$  

(2.3)

The straightforward implementation of the DFT requires $\Theta(n^2)$ many operations. It is well known that the FFTs can reduce the runtime to $O(n \log n)$. As we will show in the next section, essentially every FFT algorithm can be written in the form of

$$\text{DFT}_n = M_1M_2M_3M_4,$$

where $M_i$ is sparse and well structured. For example, $\text{DFT}_4$ can be factorized as

$$\begin{bmatrix}
1 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 \\
1 & 0 & -1 & 0 \\
0 & 1 & 0 & -1
\end{bmatrix} =
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\cdot
\begin{bmatrix}
1 & 1 & 0 & 0 \\
1 & -1 & 0 & 0 \\
0 & 0 & 1 & 1 \\
0 & 0 & 1 & -1
\end{bmatrix} =
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}.$$
After reviewing the domain specific language SPL later, we will be able to clearly show the structural properties of these sparse matrices and how we can describe the recursive algorithms in a declarative representation.

In computer algebra, calculations with univariate and multivariate polynomials can be reduced to computing with polynomials over finite fields, such as the prime field \( \mathbb{Z}_p \) for a prime number \( p \), via the so-called modular techniques. Therefore the modular DFT plays a vital role in fast algorithms for polynomial arithmetic, akin to the complex DFT in DSP applications.

**Definition 2.** Given \( n \) integer inputs \( x_0, \ldots, x_{n-1} \), the modular DFT is defined as

\[
y_k = \sum_{0 \leq l < n} \omega_n^{kl} x_l, \quad 0 \leq k < n.
\]

Therefore, the \( n \)-point modular DFT matrix is

\[
\text{ModDFT}_{n,p,\omega_n} = [\omega_n^{kl}]_{0 \leq k, l < n},
\]

where \( \omega_n \) is a primitive \( n \)th root of unity in \( \mathbb{Z}_p \).

An \( n \)-th root of unity of a ring \( \omega \in R \) is any element such that \( \omega^n = 1 \). If \( n \) is the smallest integer such that \( \omega^n = 1 \), then \( \omega \) is called a *primitive \( n \)-th root of unity*. The complex domain \( \mathbb{C} \) contains a primitive \( n \)-th root of unity for any \( n \), for instance \( \exp(-2\pi i/n) \) as seen in (2.2). The primitive roots of unity are more complicated in finite fields such as \( \mathbb{Z}_p \). Formally, a finite field \( \mathbb{F}_p \) with \( p \) elements contains a primitive \( n \)-th root of unity if and only if \( n \mid (p - 1) \). If the multiplicative group of \( \mathbb{F}_q \) has a generator \( \omega \), then \( \omega^{(p-1)/n} \) gives one \( n \)-th primitive root of unity.

Other than the difference in the primitive root of unity, the similarity between (2.2) and (2.4) and between (2.3) and (2.5) are obvious, respectively. As we will see in Chapter 4, the fast algorithms for the complex DFT can be migrated to the...
Due to the simplicity of implementation and the uniqueness of the primitive \( n \)-th root of unity in finite fields, many implementations of the modular FFT favor the inputs with sizes equal powers of two \([15, 32]\), as the underlying \( \mathbb{Z}_p \) is characterized by a \( p \) such that \( p - 1 \) is a multiple of a big power of two. Algorithms exist \([44, 4, 18]\) which do not have this constraint, however their implementation is more difficult and harder to optimize. Therefore, the inputs are typically zero-padded to the smallest power of two, and the resulting performance exhibits a “staircase” phenomenon where the computing time for sizes in between powers of two is essentially equal to the time for using the larger power of two FFTs.

The truncated Fourier transform (TFT) \([46]\) was developed to take advantage of the practical situations where some input values are known as zero and not all output values are needed. In \([46]\), van der Hoeven introduced a radix-2 algorithm for the TFT, and showed how to invert the TFT (ITFT) with “cross butterflies”, alternating the row and column transforms. Next we define the TFT and ITFT.

**Definition 3.** The TFT of size \( N = 2^n \) is defined as an operation that takes the inputs \( x_0, \ldots, x_{l-1} \) and produces \( y_0, \ldots, y_{m-1} \), with

\[
y_i = \sum_{j \in 0, \ldots, l-1} x_j \omega_N^{j[i]_n}, \tag{2.6}
\]

where \( i \in \{0, \ldots, m - 1\} \), \( 1 \leq l, m \leq N \), and \( [i]_n \) is the bit-reversal of \( i \) at length \( n \).

The only complication with this computation is the bit-reversal. For example, assuming \( N = 2^4 = 16 \), then the binary representation of 3 is 0011. Reversing the bits yields the bit-reversed \( [3]_4 = 1100 = 12 \). Compared to (2.2) and (2.4), we notice that the TFT generates a permuted output without explicit recovery of order. As the TFT and ITFT are designed to be used together in convolutions, the permutations in the
forward and inverse transforms can effectively cancel each other if the factorization
trees are mirrored, resulting in a correctly ordered output after the convolution. As
we will see in Chapter 5, the mirroring constraint requires new mechanism to be
developed in the library generator.

Inverting the TFT is more involved than inverting the complex or modular DFTs.
The inverse TFT (ITFT) uses the “cross butterflies”, and interleaves the row and
column transforms as it recovers the untransformed data and optionally one trans-
formed data element. The input to the ITFT is an initial segment of the transformed
data concatenated with the scaled final segment of the untransformed data, and the
output is the initial segment of the scaled untransformed data, and optionally the
next transformed data element.

**Definition 4.** The ITFT of size $N$ is defined as an operation that takes the inputs
$\bar{x}_0, \ldots, \bar{x}_{m-1}, Nx_m, \ldots, Nx_{l-1}$ and produces $y_0, \ldots, y_{m-1}$ and optionally $y_m$ if $f = 1$,
where

$$y_i = Nx_i, \text{ for } i \in \{0, \ldots, m-1\}, \text{ and } y_m = \bar{x}_m \text{ if } f = 1,$$

where $\bar{x}_i$ is the transformed value, $x_i$ is the untransformed value, $f \in \{0,1\}$, $1 \leq l, m + f \leq N$, and $l \geq m$.

When $m = l = N$, the TFT and ITFT reduce to the regular modular DFT and
inverse modular DFT. The input and output indices for the TFT and ITFT can be
further generalized to be from a wider class of subsets of $\{0, \ldots, N - 1\}$. In this
thesis, we restrict ourselves to contiguous input and output segments that both start
at index 0, which is sufficient for the intended application like modular polynomial
multiplication.

Next, we define univariate polynomial and polynomial multiplication. Then, we
review the convolutions that are crucial in the fast algorithms for polynomial arith-
metic.
Definition 5. Let $R$ be a commutative ring, such as $\mathbb{Z}$, a univariate polynomial $a \in R[x]$ in $x$ is a finite sequence $(a_0, \ldots, a_n)$ of elements of $R$ (the coefficients of $a$), for some $n \in \mathbb{N}$, and we write it as

$$a = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 = \sum_{0 \leq i \leq n} a_i x^i. \quad (2.8)$$

In practice, dense and sparse polynomials require different representations in data structure. For dense polynomials, we can represent $a$ by an array whose $i$th element is $a_i$. This assumes that we already have a way of representing coefficients from $R$. The length of this representation is $n + 1$.

Next, we define the univariate polynomial multiplication.

Definition 6. Let $a = \sum_{0 \leq i \leq n} a_i x^i$ and $b = \sum_{0 \leq i \leq m} b_i x^i$, the polynomial product $c = a \cdot b$ is defined as $\sum_{0 \leq k \leq m+n} c_k x^k$, where the coefficients are

$$c_k = \sum_{0 \leq i \leq n, 0 \leq j \leq m \atop i+j=k} a_i b_j, \text{ for } 0 \leq k \leq m + n. \quad (2.9)$$

The naive implementation of polynomial multiplication has an $\mathcal{O}(n^2)$ complexity. The Karatsuba algorithm reduces the cost to $\mathcal{O}(n^{1.59})$. The FFT introduced earlier can be used in convolutions to further enable fast algorithms with a complexity of $\mathcal{O}(n \log n)$.

The convolution is at the core of our polynomial multiplication library. Beyond polynomial arithmetic, convolution also has applications in signal processing and efficient computation of large integer multiplication and prime length Fourier transforms. Next, we present the definitions of linear and circular convolutions, and interpret them from different perspectives to show the connection between convolutions and the polynomial multiplication.
Both linear and circular convolutions can be viewed from three different perspectives: (1) as a sum, (2) as a polynomial product and, (3) a matrix operation. As a result, polynomial algebra can be used to derive algorithms and the corresponding matrix algebra can be used to manipulate and implement algorithms.

Definition 7. Let $u = (u_0, \ldots, u_{M-1})$ and $v = (v_0, \ldots, v_{N-1})$. The linear convolution $u * v$ is defined as

$$\begin{align*}
(u * v)_i &= \sum_{k=0}^{N-1} u_{i-k}v_k, \quad 0 \leq i < M + N \tag{2.10}
\end{align*}$$

If $u$ and $v$ are viewed as the coefficient vectors of polynomials, i.e.,

$$
\begin{align*}
  u(x) &= \sum_{i=0}^{M-1} u_i x^i, \\
  v(x) &= \sum_{j=0}^{N-1} v_j x^j,
\end{align*}
$$

then the linear convolution $u * v$ is equivalent to polynomial multiplication $u(x)v(x)$.

The sum form of linear convolution is also equivalent to the following matrix vector multiplication.

$$
\begin{pmatrix}
  u_0 \\
  u_1 & u_0 \\
  \vdots & u_1 & \ddots \\
  u_{M-1} & \vdots & \ddots & u_0 \\
  u_{M-1} & u_1 \\
  \ddots & \vdots \\
  \vdots & \ddots & \ddots & \ddots \\
  u_{M-1}
\end{pmatrix}
\cdot
\begin{pmatrix}
  v_0 \\
  v_1 \vdots v_{N-1}
\end{pmatrix}
\tag{2.11}
$$

The circular convolution of two vectors of size $N$ is obtained from linear convolution by reducing $i - k$ and $k$ in 2.10 modulo $N$.

Definition 8. Let $u = (u_0, \ldots, u_{N-1})$ and $v = (v_0, \ldots, v_{N-1})$. The circular convolu-
tion \( u \circledast v \) is defined as

\[
(u \circledast v)_i = \sum_{k=0}^{N-1} u_k v_{(i-k) \mod N}, \quad 0 \leq i < N.
\] (2.12)

Similar to the polynomial and matrix perspectives of linear convolution, the circular convolution can be obtained by multiplying polynomials \( u(x) \) and \( v(x) \) and taking the remainder modulo \( x^N - 1 \). In terms of matrix algebra, circular convolution can be interpreted as the product of a circulant matrix \( \text{Circ}_N(u) \) times \( v \), where all columns of the matrix are obtained by cyclically rotating the first column.

\[
\begin{bmatrix}
  u_0 & u_{N-1} & u_{N-2} & \cdots & u_1 \\
  u_1 & u_0 & u_{N-1} & \cdots & u_2 \\
  \vdots & \ddots & \ddots & \ddots & \vdots \\
  u_{N-2} & \cdots & u_1 & u_0 & u_{N-1} \\
  u_{N-1} & u_{N-2} & \cdots & u_1 & u_0
\end{bmatrix}
\cdot \begin{bmatrix}
  v_0 \\
  v_1 \\
  \vdots \\
  v_{N-2} \\
  v_{N-1}
\end{bmatrix} \quad (2.13)
\]

The convolution theorem introduced in Chapter 6 leads to fast convolution algorithms based on the aforementioned linear transforms, where two forward and one inverse transforms are required. The OL extension reviewed in the next section can express the fast algorithms in the declarative representation, similar to the fast algorithms for transform being expressed in the SPL.

2.2 Fast algorithms: SPL, \( \sum \)-SPL, and OL

We extend and use the SPIRAL library generator which uses its built-in domain specific language and extensions to describe and derive the fast algorithms as high-level abstractions. The foundational SPL expresses the fast transform algorithms as recursive factorization of a dense transform matrix into sparse matrices, which is ideal
for many classical algorithms. The $\sum$-SPL solves the formal loop merging problem and enables complex data manipulation, which is required by the TFT and ITFT algorithms in Chapter 5. The OL further extends the algorithm expressiveness and library generation from linear transforms to multi-linear kernels, which is critical to the convolutions in Chapter 6. As we review the language and its extensions, we show how the classical fast algorithms can be expressed with them.

2.2.1 SPL

SPL is a high-level domain-specific language that expresses recursive DSP algorithms at a high abstraction level as sparse matrix factorization. SPL expressions can be derived and optimized by the rewriting systems in SPIRAL to adapt to target computing platforms. The capability of SPL's expressiveness is bounded by the fast transform algorithms that can be interpreted as a factorization of the dense transform matrix into a product of structured sparse matrices. That is, if $M = \prod_{i=1, \ldots, k} M_i$, the original matrix-vector multiplication $y = Mx$ is equivalent to the sequential matrix-vector multiplications with the sparse matrices from $M_k$ to $M_1$ (right to left).

The main components in SPL are:

- Predefined sparse and dense matrices;
- Arbitrary matrices of predefined structure;
- Arbitrary unstructured matrices; and
- Matrix operators.

The predefined matrices included the transforms, either as the nonterminals (e.g., $ModDFT_{n,p,\omega_n}$) or the building blocks, such as the $n \times n$ identity matrix $I_n$ and the
$2 \times 2$ butterfly matrix $F_2$:

$$I_n = \begin{bmatrix} 1 & \cdots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \cdots & 1 \end{bmatrix}, \quad F_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \quad (2.14)$$

Arbitrary matrices of predefined structures include the permutation and diagonal matrices. The permutation matrix $L^n_{rs}$ where $n = rs$ permutes the input vector as

$$is + j \mapsto jr + i, \quad 0 \leq i < r, \quad 0 \leq j < s. \quad (2.15)$$

If the input vector is viewed as a row-major 2-D matrix of size $s \times r$, then $L^n_{rs}$ transposes the matrix. For example:

$$L^6_2 \cdot x = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} x_0 \\ x_2 \\ x_4 \\ x_3 \\ x_1 \\ x_5 \end{bmatrix} \quad (2.16)$$

The diagonal matrix $\text{diag}(a_0, a_1, \ldots, a_{n-1})$ is an $n \times n$ diagonal matrix filled with $a_i$:

$$\text{diag}(a_0, a_1, \ldots, a_{n-1}) = \begin{bmatrix} a_0 & 0 & \cdots & 0 \\ 0 & a_1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{n-1} \end{bmatrix} \quad (2.17)$$
Arbitrary unstructured matrices are all the other matrices that cannot be factorized into smalling building blocks using the matrix operator introduced below.

The three main matrix operators are:

- The matrix product: \( A \cdot B \);
- The direct sum: \( A \oplus B = \begin{bmatrix} A \\ B \end{bmatrix} \);
- The tensor product: \( A \otimes B = [a_{k,l}B], A = [a_{k,l}] \).

The tensor product serves as the key construct in SPL, in that it captures loops, data independence, and parallelism concisely. Two special cases of the tensor product worth noting arise when one of the matrix is the identity matrix. In particular,

\[
I_n \otimes A = \begin{bmatrix} A \\ \vdots \\ A \end{bmatrix} = A \oplus \ldots \oplus A,
\]

(2.18)

expresses the block diagonal stacking of \( n \) copies of \( A \), which means the input vector to \( I_n \otimes A \) is divided into \( n \) equal-size subvectors, each of which is independently multiplied with \( A \).

The second special case is

\[
A \otimes I_n = \begin{bmatrix} a_{0,0}I_n & \cdots & a_{0,m-1}I_n \\ \vdots & \ddots & \vdots \\ a_{m-1,0}I_n & \cdots & a_{m-1,m-1}I_n \end{bmatrix},
\]

(2.19)

assuming \( A \) is \( m \times m \). This can be viewed as \( A \) operating on vectors of size \( m \) instead of scalars or as a tight interleaving of \( n \) copies of \( A \). The input vector is divided into \( m \) equal subvectors of size \( n \), and \( A \) is applied to these \( n \)-subvectors as if they were
The SPL language can be used to express the recursive algorithms as breakdown rules for the nonterminals of transform. A breakdown rule typically decomposes a transform into several smaller transforms or converts it to another transform of lesser complexity. Next, we use \( \rightarrow \) to indicate that the rules are applied by replacing the left-hand side of the rule by the right-hand side.

For example, the general-radix Cooley-Tukey FFT, the most widely used FFT algorithm, can be expressed as:

\[
\text{DFT}_{rs} \rightarrow (\text{DFT}_r \otimes I_s) T^rs_s (I_r \otimes \text{DFT}_s) \text{L}^rs_r,
\]

(2.20)

where \( T^rs_s \) is the twiddle matrix.

Other FFT algorithms, including the Prime Factor Algorithm and Rader’s Algorithm, can be similarly expressed in SPL as follows:

\[
\text{DFT}_{rs} \rightarrow V^{-1}_{r,s} (\text{DFT}_r \otimes I_s)(I_r \otimes \text{DFT}_s)V_{r,s},
\]

(2.21)

\[
\text{DFT}_{rs} \rightarrow W^{-1}_n (I_1 \oplus \text{DFT}_{n-1}) E_n (I_1 \oplus \text{DFT}_{n-1}) W_n.
\]

(2.22)

For a complete description of a transform algorithm, base case rules are also needed in addition to the breakdown rules. For example:

\[
\text{DFT}_2 \rightarrow \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}
\]

(2.23)

For a given transform with multiple breakdown rules, the recursive application of the rules at each level leads to a large space of formulae. As we can see in Table 2.1, the SPL formulae can be directly translated into code.

The tensor product also naturally arises in multidimensional transforms. For
<table>
<thead>
<tr>
<th>Formula $M$</th>
<th>Code for $y = Mx$</th>
</tr>
</thead>
</table>
| $F_2$      | $y[0] = x[0] + x[1]$;  
        | $y[1] = x[0] - x[1]$;  
        | for (i=0; i<k; i++) {  
        | for (j=0; j<n/k; j++) {  
        | $y[j+i*n/k] = x[j*k+i]$;  
        | }  
| $L^n_k$    |  
        | (A ⊕ B)  
        | <code for: y[0:1:k−1] = A*x[0:1:k−1]>
        | <code for: y[k:1:k+m−1] = B*x[k:1:k+m−1]>
        | for(j=0; j<n; j++) {  
        | <code for: y[jk:1:jk+m−1] = A*x[jk:1:jk+m−1]>
        | }  
| $(I_n ⊗ A)$ |  
        | (A ⊗ I_n)  
        | <code for: y[j:n:j+n(k−1)] = A*x[j:n:j+n(k−1)]>
        | for(j=0; j<n; j++) {  
        | <code for: y[j:n:j+n(k−1)] = A*x[j:n:j+n(k−1)]>
        | }  

Table 2.1: Translating SPL structures to code.

instance, 2D and 3D DFTs respectively can be written as

$$\text{DFT}_{r \times s} = \text{DFT}_r \otimes \text{DFT}_s,$$  \hspace{1cm} (2.24)

$$\text{DFT}_{r \times s \times t} = \text{DFT}_r \otimes \text{DFT}_s \otimes \text{DFT}_t,$$  \hspace{1cm} (2.25)

where the tensor product of multiple 1-D DFTs can be rewritten as

$$\text{DFT}_{r \times s} = (\text{DFT}_r \otimes I_s)(I_r \otimes \text{DFT}_s),$$  \hspace{1cm} (2.26)

$$\text{DFT}_{r \times s \times t} = (\text{DFT}_r \otimes I_s \otimes I_t)(I_r \otimes \text{DFT}_s \otimes I_t)(I_r \otimes I_s \otimes \text{DFT}_t),$$  \hspace{1cm} (2.27)

respectively, leading to the *row-column algorithm*.

The above rules are the standard way of implementing multidimensional transforms. Higher dimensional versions are derived similarly, with the associativity of $\otimes$
giving more variants. For example, if each 1-D transform $T$ can be factored into a product of sparse matrices, $T = M_1 \ldots M_i$, we can obtain vector-radix type multidimensional algorithms, by systematically interleaving the factors, as shown below with a 2-D example:

$$
(T \otimes T) = (M_1 \ldots M_i \otimes M_1 \ldots M_i) = (M_1 \otimes M_1) \ldots (M_i \otimes M_i). \tag{2.28}
$$

### 2.2.2 $\sum$-SPL

In this section, we overview the $\sum$-SPL\textsuperscript{[19]}, an extension of the SPL designed to solve the formal loop merging problem and to enable complex data manipulation. The transition of SPL $\rightarrow \sum$-SPL is usually automated, provided that the algorithms can be completely described by SPL. In this thesis, the general-radix TFT and ITFT algorithms have to be expressed in $\sum$-SPL due to the more involved data access.

$\sum$-SPL consists of four components:

- Index mapping functions;
- Scalar functions
- Parametrized matrices; and
- Iterative sum $\sum$.

An index mapping function is a function that maps an interval onto an interval. The index mapping functions are expressed in terms of primitive functions and function operators to capture their structures and enable many simplification. An integer interval is denoted by $\mathbb{I}_n = \{0, \ldots, n - 1\}$, and an index mapping function $f$ with domain $\mathbb{I}_n$ and range $\mathbb{I}_N$ is denoted as

$$
f : \mathbb{I}_n \rightarrow \mathbb{I}_N; i \mapsto f(i).
$$
We use the short-hand notation \( f^{n\rightarrow N} \) to refer to an index mapping function of the form \( f : \mathbb{I}_n \rightarrow \mathbb{I}_N \). An important primitive function is the *stride* function defined as

\[
h_{b,s} : \mathbb{I}_n \rightarrow \mathbb{I}_N; i \mapsto b + is.
\]

The parameterized matrices we will use in this paper are the gather and scatter matrices, described by their defining functions:

\[
G(r^{n\rightarrow N}) \quad \text{and} \quad S(w^{n\rightarrow N}).
\]

A gather matrix \( G_r \) denotes the reading or loading of input values according to the index mapping function \( r \), and a scatter matrix \( S_w \) denotes the writing or storing of input values according to the index mapping function \( w \).

Let \( e^n_k \in \mathbb{C}^{n\times1} \) be the column basis vector with the 1 in \( k \)-th position and 0 elsewhere. The gather matrix is defined by the index mapping \( r^{n\rightarrow N} \):

\[
G(r^{n\rightarrow N}) := [e^N_{r(0)} | e^N_{r(1)} | \cdots | e^N_{r(n-1)}]^T.
\]

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

\[
G(h_{0,1}) = \begin{bmatrix}
1 \\
\ddots \\
1
\end{bmatrix}, \quad S(h_{0,1}) = G(h_{0,1})^T,
\]

The iterative sum \( \sum_{i=0}^{n-1} A_i \) is used to represent loops. It encodes a loop in which each iteration produces a unique subvector of the final output vector.

As an example, we show how a tensor product \( \otimes \) is converted into a sum. \( A \) is
<table>
<thead>
<tr>
<th>Formula $M$</th>
<th>Code for $y = Mx$</th>
</tr>
</thead>
</table>
| $G(r^{n\rightarrow N})$ | \( \textbf{for} (i = 0; i < n; i++) \)  \[
y[i] = x[f(i)];
\]  
| $S(r^{n\rightarrow N})$ | \( \textbf{for} (i = 0; i < n; i++) \)  \[
y[f(i)] = x[i];
\]  
| $\text{perm}(p^n)$ | \( \textbf{for} (i = 0; i < n; i++) \)  \[
y[i] = x[p(i)];
\]  
| $\text{diag}(f^{n\rightarrow C})$ | \( \textbf{for} (i = 0; i < n; i++) \)  \[
y[i] = f(i) \ast x[i];
\]  
| $\left(\sum_{i=0}^{n-1} A_i\right)$ | \( \textbf{for} (i = 0; i < n; i++) \{ \)  \[
\text{\quad \quad code \quad for: \quad y = A_{i,j} \ast x>};
\}  

Table 2.2: Translating $\sum$-SPL structures to code.

assumed as $n \times n$ and the domain and range in the stride functions are omitted for simplicity. $I_m \otimes A$ that duplicates $m$ copies of $A$ along the diagonal can be described as:

\[
\begin{bmatrix}
A \\
\vdots \\
A
\end{bmatrix}
= \sum_{i=0}^{m-1} S_i \cdot A G_{i,n}
\]

Another important reason for the introduction of $\sum$-SPL is the formal loop merging \[11\]. As an example, consider the SPL formula

\[(I_m \otimes A_n) P, P \text{ a permutation matrix} \]  

(2.29)
which can be produced by applying (2.20). Σ-SPL further rewrites the formula to merge the two explicit data passes into one loop that implements the data reorganization specified by $P$ as readdressing the input of the computational kernel $A_n$. Therefore, (2.29) is rewritten as

$$
\sum_{j=0}^{m-1} (S_w_j A_n G_{p_{or_j}}),
$$

where the permutation matrix is now also expressed in terms of its defining permutation function $p$, and is directly incorporated into the nearby gather operation (“◦” is the composition of functions).

2.2.3 OL

The OL, a framework for automatic generation of fast numerical kernels, is recently developed as an extension to the SPL and integrated into SPIRAL. OL provides the structure to extend SPIRAL beyond the transform domain. The control flow of many kernels is data-independent, which allows OL to cast their algorithms as operator expressions. In this thesis, the multi-linear operations like the linear and circular convolutions require using OL.

The three main components of OL framework are:

• the operator language (OL) that describes the kernel algorithms;

• the hardware tags that describe architecture features; and

• the tagged OL, a common abstraction of architecture and algorithms.

In this section, we focus on OL, as the hardware tagging and optimization are discussed in details in Section 2.4. OL is similar to SPL, in that it describes structured divide-and-conquer algorithms for data-independent kernels. The algorithms
are encoded as breakdown rules, which can be recursively derived to generate a large implementation space. OL is a generalization of SPL which as designed only for linear transforms.

The basic building blocks of OL are operators that are \( n \)-ary functions on vectors. The arity of an operator is a 2-tuple \( (r, s) \), meaning that the operator consumes \( r \) vectors and produces \( s \) vectors. Note that the linear transforms reviewed earlier in this chapter all have the same arity \((1, 1)\). It is also worth noting that the matrices are viewed as vectors stored linearized in row major order. Therefore, the permutation operator \( L_{rs}^r \) seen in 2.20 that transposes an \( r \times s \) matrix is of arity \((1, 1)\).

The basic operators can be combined into OL formulae by the high-order operators, as the parameters provided to the functions on operators. For example, the composition, denoted by \( \circ \), is a generalization of the matrix multiplication in SPL that composes adjacent linear transforms. For instance,

\[
L_{rs}^r \circ P_{rs}
\]

is an arity \((2, 1)\) operator formula that first multiplies point-wise two matrices and then transposes the result.

The cross product of two operators, denoted by \( \times \), is used in the convolution formula where two forward transforms are applied to two input vectors. It applies the first operator to the first input set and the second operator to the second input set, and then combines the outputs. For example,

\[
L_{rs}^r \times P_{rs}
\]

is an arity \((3, 2)\) operator formula that transposes the its first input, and multiplies pointwise the second and third inputs, producing two output vectors.
The most important SPL operator, the tensor product $\otimes$, is also the most important higher order operator in OL, after it is formally extended. The generalization of the tensor product from SPL to OL is omitted here, as it is beyond the scope of this thesis.

Recursive algorithms can be expressed on OL breakdown rules. For example, the circular convolution $\text{Conv}_n$ is an arity $(2,1)$ operator as defined in Section 2.2. $\text{Conv}_n$ can be performed with the forward and inverse DFTs, which are then recursively broken down with the algorithms like $2.20$, $2.21$, $2.22$.

$$\text{Conv}_n \rightarrow \text{DFT}_{n,-1} \circ P_n \circ (\text{DFT}_{n,1} \times \text{DFT}_{n,1}),$$  \hspace{1cm} (2.30)

Where $\text{DFT}_{n,-1}$ and $\text{DFT}_{n,1}$ are the inverse and forward transforms, respectively.

The capability of expressiveness of OL also include the matrix multiplication, sorting network, Viterbi decoding and synthetic aperture radar (SAR). In addition to OL, for the high performance generation of kernels, hardware paradigms are also created to capture the essential properties of families of hardware. Paradigms tags are used to denote paradigm properties, such as cache line length, vector unit length and number of threads. The tags are replaced by the actually platform during the compilation of the OL formulae. Rewriting, compiling and unparsing rules have also been developed and integrated into SPIRAL for OL and the paradigm tags.

2.3 SPIRAL and Library Generation

SPIRAL started as a program generator that can generate stand-alone programs for DSP algorithms. In contrast to other work in autotuning, SPIRAL uses internally the domain-specific language SPL to express divide-and conquer algorithms for transforms as breakdown rules. For a user-specified transform and transform size,
SPIRAL applies these rules to generate different algorithms, represented in SPL. Then $\sum$-SPL makes explicit the description of loops and index mappings. The $\sum$-SPL representation of algorithms are then processed by the built-in rewriting systems for performing difficult optimizations such as parallelization, vectorization, and loop merging automatically. The optimizations are performed at a high abstraction level in order to overcome known compiler limitations. The unparsers translate these algorithms into executable code. Based on the generated code, a search engine uses the dynamic programming technique to explore different choices of algorithms to find the best match to the computing platform. Fig 2.1 shows the classic program generation framework in SPIRAL.

![Figure 2.1: Classic program generation framework in SPIRAL](image-url)
The classic program generation framework was thoroughly used and extended in our previous work to generate fixed size modular FFT algorithms. However, the framework is limited by the type of code it could produce, in that transform sizes and other parameters must be known at generation time.

SPIRAL has been further extended to support general size library generation\cite{49}. To generate a library for a given transform, the system has to be able to generate code for transforms of symbolic size, i.e., the size becomes an additional parameter. While the fixed-size transforms can be decomposed by the adaptive generation framework, the breakdown rules can only be applied at runtime for unknown/symbolic size, since the applicabilities of different rules are dependent on the transform size.

This change leads to a new framework fundamentally different from the fixed-size code generation, as seen in Fig.2.2

- The **Library Target** refers to desired library type, infrastructure and the target language.

- The **Library Structure** operates at the algorithm level, and is independent of the library target. The Library Structure module compiles the breakdown rules of the transform into a recursion step closure, which corresponds to a set of recursive functions.

To understand how a general size library is generated, recall the Cooley-Tukey factorization of \( \text{ModDFT}_{rs} \), where the modulus \( p \) and the root of unity \( \omega \) are omitted for simplicity.

\[
\text{ModDFT}_{rs} = (\text{ModDFT}_r \otimes I_s) T^r_s (I_r \otimes \text{ModDFT}_s) L^r_s.
\]

With \( \Sigma\text{-SPL} \), the twiddle factor \( T^r_s \) and the permutation matrix \( L^r_s \) can be fused into the nearby tensor products. As a result, the Cooley-Tukey FFT is computed in
two loops corresponding to the two tensor products decorated with the twiddles and the stride permutation.

\[
\text{ModDFT}_{rs} = (\text{ModDFT}_r \otimes I_s) T_r^* \underbrace{(I_r \otimes \text{ModDFT}_s)}_{\text{loop}} L_r^s.
\]

The fusion of the stride permutation requires a ModDFT function \(\text{moddf}_\text{str}\) with different input and output stride, and the fusion of the twiddles requires a ModDFT function \(\text{moddf}_\text{scaled}\) with an extra array parameter holding the scaling factors. These functions are implemented recursively based on the Cooley-Tukey algorithm, and do not require any new functions. Eventually, the modular DFT size becomes sufficiently small, and the recursion terminates by using codelet functions that compute small fixed-size transforms.
We say the functions \textit{moddft}, \textit{moddft\_str} and \textit{moddft\_scaled} form a so-called \textit{recursion step closure}, which is a minimal set of functions sufficient to compute the desired transform. The recursion step closure is the central concept in the library generation framework. The framework can compute the recursion step closure for a given transform and multiple algorithms via parametrization and descending, and generate corresponding base cases. Extensive details and examples can be found in [49].

In both frameworks, new transforms can be added by introducing new symbols and their definitions, and new algorithms can be generated by adding new rules. SPIRAL was developed for floating point and fixed point computation; however, many of the transforms and algorithms carry over to finite fields. For example, the DFT of size $n$ is defined when there is a primitive $n$th root of unity and many factorizations of the DFT matrix depend only on properties of primitive $n$th roots. In this case, the same machinery in SPIRAL can be used for generating and optimizing modular transforms.

2.4 Autotuned Parallelism

SPIRAL automatically derives the algorithms for parallel implementations. Rewriting systems have been developed to fully exploit two levels of parallelism: vector parallelism and thread parallelism. To generate vectorized and parallelized implementation, one first needs to obtain “fully optimized” SPL formulas, which means they can be mapped to efficient vector or parallel code. This mapping is possible as certain SPL constructs express parallelism. For example, $I \otimes A$ corresponds to a parallelizable loop with no loop-carried dependencies.
2.4.1 Vectorization

The rewriting system for vector code uses the following four components:

- **Vectorization tags** which mark the formula as "to be vectorized", and introduce the vector length.

- **Vector formula constructs** which express subformulae that can be fully mapped to SIMD code.

- **Rewriting rules** which transform SPL formulae into vector formulae.

- **Vector backend** that generates SIMD code from vector formulae.

During the rewriting, vectorization information is propagated through the formulae via a set of vectorization tags. For example, $\text{Vec}_\nu(A)$ means the formula construct $A$ is to be translated into vector code with vector length $\nu$.

The central formula construct in vector formula constructs is $A \otimes I_\nu$, where $A$ is an arbitrary matrix, and $I$ is an identity matrix of size $\nu$. Vector code is obtained on $\nu$-way short vector extensions by generating scalar code for $A$ and replacing scalar operations by respective $\nu$-way vector operations. $A \otimes I_\nu$ stipulates that the tensor product is to be mapped to vector code without any further manipulations. In addition, the complete set of efficient base cases for vectors of size $\nu$ are provided.

$$L_2^{2\nu}, L_\nu^{2\nu}, L_\nu^{\nu^2}, (I_{n/\nu} \otimes L_2^{2\nu}) D_n(I_{n/\nu} \otimes L_2^{2\nu})$$

where $D_n$ is any diagonal matrix. Both constructs marked with $\otimes$ and base cases are final, i.e., they will not be changed by rewriting rules.

Rewriting rules are used for non-final constructs in the framework to generate vectorized form, and are extensively covered in [12]. For example, $\text{Vec}_\nu(I_m \otimes A)$ can be rewritten into $I_{m/\nu} \otimes \text{Vec}_\nu(I_\nu \otimes A)$ based on the rewriting rules. SPIRAL derived
rules for permutations and tensor products. These rules are applied with pattern matching to rewrite an SPL formula into a final vectorized formula. Similar rewriting mechanism can also be applied to Σ-SPL vectorization.

For example, the tensor product $I_r \otimes \text{ModDFT}_s$ produced in the Cooley-Tukey factorization of ModDFT_{rs} can rewritten for vectorization as follows

\[
I_r \otimes \text{ModDFT}_s = I_{r/\nu} \otimes I_{\nu} \otimes \text{ModDFT}_s \\
= I_{r/\nu} \otimes (L_{\nu}^{s\nu}(\text{ModDFT}_s \otimes I_{\nu}) L_{\nu}^{s\nu})
\]

(2.31)

(2.32)

Note that in this formula ModDFT_{s} \otimes I_{\nu} is already vectorized. Then the stride permutation can also be rewritten into the fully vectorizable form as follows:

\[
I_{r/\nu} \otimes ((L_{\nu}^{s\nu} \otimes I_{\nu})(I_{s/\nu} \otimes L_{\nu}^{s\nu}))(\text{ModDFT}_s \otimes I_{\nu})(I_{s/\nu} \otimes L_{\nu}^{s\nu})(L_{s/\nu} \otimes I_{\nu})
\]

(2.33)

2.4.2 Parallelization

The same approach is followed by shared memory parallelization (smp) as with vectorization, starting with the formula tags and tagged SPL constructs, and finally obtaining parallel code. During parallelization, there are several issues one must address: 1) Load balancing: All processors should have an equal workload. 2) Synchronization overhead: Synchronization should involve as little overhead as possible and unnecessary wait should be eliminated at synchronization points. 3) Avoiding false sharing: Private data of different processors should not be in the same cache line as the same time, otherwise cache thrashing may occur, which leads to severe performance degradation.

To generate parallel code by rewriting SPL formulas, we first notice that SPL constructs have a direct interpretation in terms of parallel code. A SPL formula fully determines the memory access pattern of the generated program, and thus one can
st.tically schedule the loop iterations across $p$ processors through rewriting to ensure load balancing and eliminate false sharing. The components required in the parallelization extension to Spiral include: 1) *Parallelism tags* which introduces hardware parameters into the rewriting system. 2) *Parallel formula constructs* which denote the subformulas that can be perfectly mapped to *smp* platforms. 3) *Rewriting rules* which transform general formulas into parallel formulas. 4) *Parallel backend* than maps parallel formulas into parallel executable code.

*Parallelism tags* introduces the important parameters of *smp* machines: the number of processors $p$, and the cache line length $\mu$, e.g. $\text{Par}_{p,\mu}(A)$.

*Parallel formula constructs*. For arbitrary $A$ and $A_i$ in $\mathbb{C}^{m\mu \times m\mu}$, the expressions

$$y = (I_p \otimes A)x, \quad y = \bigoplus_{i=0}^{p-1} A_i x,$$

are embarrassingly parallel on $p$ processors as they express block diagonal matrices with $p$ blocks. If the matrix dimensions are a multiple of $\mu$, then each cache line is owned by exactly one processor, thus preventing false sharing. If all $A_i$ have the same computational cost, the resulting program is load balanced. For the tensor product and direct sum in Spiral, respective tagged versions have been introduced, namely $I_p \otimes_{||} A, \bigoplus_{i=0}^{p-1} || A_i, P \otimes \mu$. The tagged operators declare that a construct is fully optimized for *smp* machines and does not require further manipulation.

The general formulas are transformed into fully optimized formulas with *rewriting rules*. Example of rules include:

- $\text{Par}_{p,\mu}(AB) \rightarrow \text{Par}_{p,\mu}(A)\text{Par}_{p,\mu}(B)$, which expresses that in matrix multiplication each factor will be rewritten separately, and

- $\text{Par}_{p,\mu}(A_m \otimes I_n) \rightarrow \text{Par}_{p,\mu}((L_m^{mp} \otimes I_{n/p}) (I_p \otimes (A_m \otimes I_{n/p}))(L_p^{mp} \otimes I_{n/p}))$, which handles tensor product with identity matrices. It distributes the workload evenly
among the $p$ processors and execute as many consecutive iterations as possible on the same processor.

Other rules deal with breaking down stride permutations and product of a permutation and a large identity matrix.

The parallel backend currently uses OpenMP to generate parallel C code. The extension to support smp code is straightforward. The only thing needs to be added is the translation of the constructs

$$I_p \otimes_{||} A, \bigoplus_{i=0}^{p-1} A_i,$$

into parallel code for $p$ threads. The other parallel construct, $P \otimes I_\mu$ is merged with the adjacent loops by $\Sigma$-SPL optimization.

With the optimization techniques discussed above, SPIRAL works like a human expert in both DSP algorithms and code tuning. SPIRAL now autonomously explores algorithm and implementation space, optimizes at both the mathematical/algorithmic and at the code level, and exploits platform-specific features to create the best implementation for a given computer. Further, SPIRAL can be extended and adapted to generate code for new transforms, to exploit platform-specific special instructions, and to optimize for various performance metrics.

2.5 Conclusion

In this chapter, we reviewed the definitions and algorithms for the modular DFT, TFT, convolution and modular polynomial multiplication. We also introduced the SPL language and its extensions that were designed to express the recursive algorithms at a high abstraction level to enable the derivation and optimization based on domain-specific knowledge. We summarized the state-of-the-art of the related
work and their limitations. The SPIRAL system is highlighted here, as we extend and enhance the system for our library generation and optimization. The two most important library generation components in SPIRAL are overviewed with examples to show how the recursion step closure and the parallelism can be automatically obtained. In later chapters, we will provide details on the necessary extensions to SPIRAL to support high-performance library generation for modular polynomial multiplication.
Recall that our main focus is the high performance implementation of polynomial multiplication, to which many basic exact arithmetic operations such as division can be efficiently reduced. Furthermore, the modular techniques help reduce the calculations with univariate and multivariate polynomials to computing with polynomials over finite fields. Finally, as most calculations tend to densify intermediate expressions even when the input and output polynomials are sparse, our primary effort can therefore be focused on the high performance implementation of dense polynomial multiplication over finite fields.

The performance of the underlying modular arithmetic directly affects the performance of the modular polynomial multiplication. In this chapter, we compare the existing algorithms for modular arithmetic as well as some lesser known optimization techniques, and present detailed vectorized counterparts, which are integrated into the library generation of modular DFT, TFT and ITFT, and convolutions. The vectorization can be automated by SPIRAL as the algorithms can be expressed as rewriting rules, provided that the modular data types, related vector ISAs and auxiliary components are properly added or extended in the library generator.

3.1 Hardware Support for Integer Arithmetic

Modular arithmetic occurs in many algorithms for symbolic computation. However, the cost-effective design philosophy derived by the mainstream CPU manufacturers tends to find the integer divide hardware costly from both physical size and performance perspectives, which makes adding multiple units to a core prohibitive. This compromise leads to the long latency and low throughput of integer divisions.
Considerable amount of effort has been made by the leading CPU manufacturers such as Intel® and AMD® to improve the integer division algorithms in their hardware designs during the recent decades, but the modular arithmetic operations are still comparably slower than their floating-point counterparts. As a result, dedicated implementations of the modular arithmetic are required, since the performance gains from low-level operations will produce improvements in a wide variety of problems.

A data level parallelism exploited by most of the modern CPUs is the SIMD (single instruction, multiple data) parallelism. The SIMD instructions operate on vectors of data, therefore converting scalar instructions to the equivalent SIMD instructions is referred as vectorization in this thesis. Figure 3.1 shows a typical SIMD operation, in this case a \( v \)-way vector multiplication, where the single instruction \( \text{VMUL} \) is applied to multiple data from the input vectors A and B.

\[
\begin{array}{cccccc}
A_{v-1} & \cdots & A_1 & A_0 \\
& \cdots & & \\
B_{v-1} & \cdots & B_1 & B_0 \\
& \cdots & & \\
R_{v-1} & \cdots & R_1 & R_0 \\
\end{array}
\]

Figure 3.1: A typical SIMD operation on \( v \)-way vector operands

The first widely-deployed desktop SIMD was Intel’s MMX extension to the x86 architecture, unofficially known as the “multimedia extension”. Since then, the majority of the development effort by the CPU manufacturers has been devoted to multimedia
processing, hence to the floating-point SIMD extensions. As a result, integer SIMD extensions are often delayed, altered or even missed when the manufactures face the design constraints.

As a warm-up, we show the complexity of multiplying two unsigned 32-bit integers and then gathering the high 32-bit and low 32-bit of the products using the existing SIMD operations. This operation is an important step in the vectorized Montgomery algorithm [38]. Unlike the vector multiply of floating-point and double precision numbers, the integer vector multiply produces a product that needs twice the width of the multiplier (32 → 64).

Figure 3.2, 3.3, and 3.4 show three strategies for the same operation, where the light-gray shaded area denotes the low 32-bit and gray shaded area denotes the high 32-bit. All three strategies use _mm_mul_epu32 to produce the 64-bit products $T_i, i \in \{0, 1, 2, 3\}$, where $T_2$ and $T_0$ are obtained by multiplying vectors $A$ and $B$, and $T_3$ and $T_1$ are obtained by multiplying the vectors $A$ and $B$ right-shifted by 32-bit.

Figure 3.2 uses the floating-point shuffle instruction _mm_shuffle_ps which, controlled by an 8-bit immediate data, shuffles two floating point vectors into one with a latency of 1 and a throughput of 1. Type casting is required between the single-precision floating point vector type (_m128) and integer vector type (_m128i) before and after the shuffle. However, the type casting is handled by the compilers, and does not incur any overhead. Therefore, with 2 shuffles, we can gather all the high 32-bits and low 32-bits into two vectors, respectively, albeit the elements in the two vectors are out of order. Then, we can apply the integer shuffle instruction _mm_shuffle_epi32 on each of the two vectors to recover the correct order. Note that _mm_shuffle_epi32 can only shuffle integer elements within one vector, with a latency of 1 and throughput of 0.5 (except for on Haswell architecture where the throughput is 1). The disparity between the seemingly similar _mm_shuffle_ps and _mm_shuffle_epi32 also indicates
that the floating-point vectorization strategies cannot always be naively migrated to integer vectorization.

Figure 3.2: 2-way vectorized multiplication with floating point shuffle and cast

Figure 3.3 uses only the integer vector instructions. First, the vectors of 64-bit products are each shuffled by _mm_shuffle_epi32 to gather the high 32-bits and low 32-bits within each vector. Then, _mm_unpacklo_epi32 and _mm_unpackhi_epi32 interleave the 32-bits within the low 64-bits and high 64-bits, respectively, of the two shuffled vectors to gather all the low 32-bits and high 32-bits in the correct order. Note that _mm_unpacklo_epi32 and _mm_unpackhi_epi32 both have a latency of 1 and throughput of 0.5 (except for on Haswell architecture where the throughput is 1).

Figure 3.4 uses the integer blend instruction available in AVX2. The vector containing 64-bit products $T_2$ and $T_0$ is shuffled with _mm_shuffle_epi32 to flip the high
2-way multiplication (integer shuffle/unpacks)

\[
\begin{array}{cccc}
A_3 & A_2 & A_1 & A_0 \\
B_3 & B_2 & B_1 & B_0 \\
T_{2h} & T_{2l} & T_{0h} & T_{0l} \\
T_{3h} & T_{3l} & T_{1h} & T_{1l} \\
T_{3h} & T_{2h} & T_{1h} & T_{0h} \\
T_{3l} & T_{2l} & T_{1l} & T_{0l}
\end{array}
\]

Figure 3.3: 2-way vectorized multiplication with integer shuffle and unpack

and low 32-bits within each product, resulting in the correct interleaved order of the 4 high 32-bits in the two product vectors. Next, \texttt{mm\_blend\_epi32} is applied to the product vectors twice with different 8-bit immediate data to gather the high 32-bits in-order and low 32-bits out-of-order. The vector containing the low 32-bits is shuffled again to recover the correct order. The \texttt{mm\_blend\_epi32} is only available in the Haswell architecture with a latency of 1 and a throughput of 0.33.

Figure 3.2, 3.3, and 3.4 show a typical scenario of integer vectorization. The integer SIMD instructions made available under the design constraints often overlap with each other in terms of functionality, causing confusions in deciding the best implementation strategy. They also often work differently than the seemingly similar floating point counterparts, preventing the floating point vectorization strategies from being directly migrated to integer vectorization. When the functionality is identical, the integer vector instructions often tend to have higher latency or lower throughput compared to their floating point counterparts. These limitations make vectorizing
the fast modular arithmetic algorithms challenging.

### 3.2 Fast Algorithms

In this section, we review the fast algorithms for modular arithmetic operations, with a particular focus on the pivotal modular multiplication, since the modular addition and subtraction are straightforward. The fast algorithms operate over the finite fields defined by word-size primes. A key assumption of the fast algorithms is that arithmetic with powers of two is highly efficient, which is true on modern hardware. The arithmetic operations modulo a power of two can be performed extremely quickly with bitwise shifts and logical instructions. Note that the algorithms, namely the Barrett algorithm and Montgomery algorithm, are well-known and implemented
as scalar code in some existing libraries, but the vectorized implementations were first devised in [36].

3.2.1 Fast Modular Multiplication Algorithms

Barrett algorithm [3] precomputes an approximation of the inverse of $P$, which enables a close approximation to the quotient of $ab$ divided by $P$. Suppose $\lceil \log_2(P) \rceil = k$, i.e., $P$ can be represented with $k$ valid bits. Let $P' = \lfloor 2^{2k}/P \rfloor$, a rescaled numerical inverse of $P$ stored as a precomputation. The Barret algorithm computes $ab \mod P$ as follows:

**Algorithm 1 Barrett Algorithm**

```plaintext
function Barrett(a, b)
    m ← ⌊ab/2^k⌋
    q ← ⌊mP'/2^k⌋
    t ← ab − qP
    while t ≥ P do
        t ← t − P
    end while
    return t
end function
```

The value of $t$ before the while loop is less than $4P$, therefore the while loop executes at most 3 times. Hence the total arithmetic operations consist of three multiplications, at most four subtractions, and two bitwise shifts that perform the integer division by a power of two.

Montgomery algorithm [38] is similar to Barrett algorithm, in that it also requires precomputation with $P$. But the algorithm gains further efficiency by changing the representation to the so-called residue class. The algorithm also replaces the expensive integer division by $P$ with bitwise operations such as shift and masking.
For $P > 1$, the algorithm defines an $P$-residue to be a residue class modulo $P$.

First, the algorithm selects a $R$ coprime to $P$ such that $R > P$ and computations modulo $R$ are inexpensive to process. Let $R^{-1}$ and $P'$ be integers satisfying $0 < R^{-1} < P$ and $0 < P' < R$ and $RP^{-1} - PP' = 1$, which can be computed using the extended Euclidean algorithm. For $i \in \mathbb{Z}_P$, the residue class $\bar{i}$ is defined as $iR \mod P$. With this residue system, we can quickly compute $\bar{a}\bar{b}R^{-1} \mod P$ from $\bar{a}$ and $\bar{b}$ if $0 \leq \bar{a}\bar{b} < RP$, as shown in the algorithm below:

**Algorithm 2 Montgomery Algorithm**

```plaintext
function REDC($\bar{a}, \bar{b}$)
    $T \leftarrow \bar{a}\bar{b}$
    $m \leftarrow (T \mod R)P' \mod R$ \quad \triangleright m \equiv \bar{a}\bar{b}P' \mod R
    $t \leftarrow (T + mP) / R$ \quad \triangleright mP \equiv \bar{a}\bar{b}P' \equiv -\bar{a}\bar{b} \mod R \Rightarrow t \equiv \bar{a}\bar{b}R^{-1} \mod P$
    if $t \geq P$ then
        return $t - P$
    else
        return $t$
    end if
end function
```

The correctness of the algorithm is shown in the comments lead by $\triangleright$. Given two numbers $x, y \in \mathbb{Z}_P$, let $z = \text{REDC}(\bar{x}, \bar{y})$. Then $z = \bar{x}\bar{y}R^{-1} \mod P \equiv (xR)(yR)R^{-1} \mod P \equiv xyR \mod p$. Also, $0 \leq z < P$, so $z$ is the product of $x$ and $y$ in the residue representation. Note that the change of representation does not affect the modular addition and subtraction algorithms.

The total arithmetic operations consist of three integer multiplications, but fewer addition/subtractions compared to Barrett algorithm. As we later vectorize the algorithm, we further see that the second multiplication is a “short product” where only the low-order bits are needed. This observation is a key to efficient adaptation
to the available integer vector instructions. The overhead of converting to and from
the residue representation is clearly expensive, but it becomes negligible in most ap-
plications where long sequences of modular arithmetic is performed. Therefore, we
will focus on the efficient implementation and integration of Montgomery algorithm
in SPIRAL.

Montgomery algorithm can be optimized further when the prime \( P = c2^n + 1 \),
also known as a Fourier prime when \( c \) is small and \( n \) is sufficiently big \([28]\). More
precisely, we require \( n \geq l/2 \) where \( l \) is the bit length of \( P \) and \( R = 2^l \). Machine
word-size multiplication modulo such Fourier primes can be done efficiently with the
algorithm below:

\[ \text{Algorithm 3 Montgomery Algorithm over Fourier Primes} \]

\[
\begin{align*}
\text{function } & \text{FOURIERREDC}(\bar{a}, \bar{b}) \\
q_1 & \leftarrow \bar{a}\bar{b}/R \\
r_1 & \leftarrow \bar{a}\bar{b} \mod R \\
q_2 & \leftarrow c2^n r_1/R \\
r_2 & \leftarrow c2^n r_1 \mod R \\
q_3 & \leftarrow c2^n r_2/R \\
t & \leftarrow q_1 - q_2 + q_3 \\
t & \leftarrow t + (t \gg 31) \& P \\
t & \leftarrow t - P \\
t & \leftarrow t + (t \gg 31) \& P \\
\text{return } t 
\end{align*}
\]

We first show that \( t \equiv \bar{a}\bar{b}R^{-1} \mod P \). By the definitions of \( q_1 \) and \( r_1 \), we have
\[ \tilde{a} \tilde{b} = q_1 R + r_1. \] Then we have

\[ \tilde{a} \tilde{b} R^{-1} \equiv q_1 + r_1 R^{-1} \pmod{P} \]

\[ \equiv q_1 - c 2^n r_1 R^{-1} \pmod{P} \] \(\because\) since \(c 2^n \equiv -1 \pmod{P}\)

\[ \equiv q_1 - q_2 - r_2 R^{-1} \pmod{P} \] \(\because\) since \(q_2 = c 2^n r_1 / R, r_2 = c 2^n r_1 \pmod{R}\)

\[ \equiv q_1 - q_2 + q_3 + r_3 R^{-1} \pmod{P} \] \(\because\) since \(q_3 = c 2^n r_2 / R, r_3 = c 2^n r_2 \pmod{R}\)

\[ \equiv q_1 - q_2 + q_3 \pmod{P} \] \(\because\) since \(r_3 = 0\)

To understand why \(r_3 = 0\), we have \(r_3 = c 2^n \pmod{R} \equiv c 2^n (c 2^n r_1 - q_2 R) \pmod{R} \equiv c^2 2^{2n} r_1 - c 2^n q_2 R \pmod{R}\), where both \(c 2^n q_2 R\) and \(c^2 2^{2n} r_1\) are multiples of \(R\), as \(R = 2^l\) and \(l \leq 2n\), resulting in \(r_3 = 0\). It can be further proved that \(t \in \{- (P - 1), 2(P - 1)\}\).

Due to the aforementioned advantage over the Barrett algorithm, the Montgomery algorithm is used throughout the library generation for modular polynomial arithmetic in SPIRAL. Next, we introduce the vectorized Montgomery algorithm that utilizes the available SIMD instructions for improved data-level parallelism.

### 3.2.2 Vectorized Montgomery Algorithm

Vectorization of the Montgomery algorithm is more complicated compared to the vectorization of modular addition and subtraction. Given the size \(l\) of the multipliers, e.g., a machine-word size of 32, the multiplication steps in the algorithm generate the intermediate products that require \(2l\)-bits to store. This observation, coupled with the availability of the integer SIMD extensions, requires the vectorized implementation to be well designed to fully utilize the available vector width for optimal performance.

Next, we show how a SIMD extension with \(sl\)-bit vector registers can be efficiently utilized in a mixture of \(s\)-way and \(s/2\)-way vectorization in order to vectorize the
Montgomery algorithm (Algorithm 2) with \( l \)-bit inputs. That is, we fully exploit the vector level parallelism and only reduce to \( s/2 \)-way when the intermediate products are needed in full. Next, we use 32-bit multipliers and SSE 4.2 with 128-bit vector registers as a running example.

Figure 3.5 illustrates the vectorized Montgomery algorithm which proceeds from top to bottom. The labels correspond to the variables and expressions in Algorithm 2. The solid lines denote operations on the vectors, and the dashed lines denote that the vectors are simply kept and used later. The prefix of the operators, such as 2 and 4, indicate whether the operations are 2-way or 4-way.

First, two 2-way integer multiplications are performed to produce four 64-bit \( T \)’s. Note that in Algorithm 2, \( T \) is then reduced to \( Z_R \) and multiplied with \( P' \), after which the product is again reduced to \( Z_R \) to produce \( m \). Here, the first reduction is applied to the lower 32-bits of \( T \)’s after they are properly gathered, and the multiplication only needs to generate the low 32-bit of the product (hence the 4-way \texttt{mm_mullo_epi32}), as the second reduction to \( Z_R \) only operate on the low 32-bit per the definition of \( R \). The reduction to \( Z_R \) can be efficiently performed as a bitwise and with \( R - 1 \) to produce \( m \).

Next, \( m \) is 2-way multiplied with \( P \), whose product is then 2-way added to \( T \). The 2-way addition is required as carries may happen across the low and high 32-bits. \( t_h \)’s and \( t_l \)’s are then gathered in-order before being divided by \( R \). The division is performed separately on the high and low 32-bits. The high 32-bits is a multiple of \( R \), therefore division by \( R \) is equivalent to a bitwise left shift; The low 32-bit divided by \( R \) is equivalent to a bitwise right shift. Both shifts are safe based on the definition of \( R \). Then the shifted results are 4-way added to produce \( t \)’s. Finally, \( t \)’s are reduced to \( Z_P \) by selectively subtracting \( P \) where \( t_i \geq P \).
Figure 3.5: The vectorized Montgomery algorithm
3.3 SPIRAL Extensions for Modular Arithmetic

SPIRAL used to concentrate on the numeric code generation and optimization, as the input to the DSP algorithms and numeric kernels are either complex or real. In our problem domain of modular polynomial arithmetic, the performance of the underlying modular arithmetic is critical. In this section, we introduce the extensions to SPIRAL’s limited integer arithmetic generation to support efficient modular arithmetic generation, including new data types, vector ISAs and library templates.

Data types play a central role at many levels of code generation and rewriting in SPIRAL. The built-in data types can be divided into two categories: 1. the primitive data types, such as TReal and TComplex, and 2. the composite data types, such as TVect. The primitive data types are associated with the operators in scalar arithmetic operations, which can be consumed by the type-based IR rewriting rules and by the corresponding unparsers. The composite data types, as the name indicates, are compositions of the primitive data types. For example, a 4-way floating point vector of is represented as TVect(TReal , 4), where 4 indicates the width of the vector.

Both the primitive and composite data types can be pairwise unified when two or more operators of different types are presented in the same arithmetic operation. Roughly speaking, the type unification rules are similar to the type conversions in languages like C when two arguments of the operation are of different types. For example:

\[ a \in TComplex, \ b \in \{ TInt, TUInt, TReal, TComplex \}, \text{Unify}(a, b) \rightarrow TComplex, \]
\[ a \in TVect(t_a, s_a), \ b \in TVect(t_b, s_b), \text{Unify}(a, b) \rightarrow TVect(\text{Unify}(t_a, t_b), \text{max}(s_a, s_b)). \]

(3.1)
(3.1) shows that if the data type of one operator is $TComplex$, then the unified data type is $TComplex$ if the other operator is any type in \{$TInt, TUInt, TReal, TComplex$\}; If the two operators are both $TVect$, then the Unify function unifies their primitive data types and takes the longer vector length in the resulting composite type.

New modular data types and corresponding unification rules are implemented to support the generation of modular arithmetic, namely $TModInt$, $TModInt64$ and $TModReal$. The integer modular types are directly associated with the 32-bit and 64-bit integers in `stdint.h`, respectively, as an effort to precisely control the integer size in code generation and optimization. On the other hand, $TModReal$ is used in the algorithms where conversions between integer and floating point are required. The new unification rules enforce the modular arithmetic when at least one operator is a modular type. For example:

$$a \in TModInt, b \in \{TBool, TUInt, TInt, TModInt\}, \text{Unify}(a, b) \rightarrow TModInt.$$  

The data types also guide the generation of compilable code when the (rewritten) IR is unparsed. Therefore the unparsers must be extended to properly generate modular code. Additionally, the library generation in SPIRAL parameterizes the library template files where many components are hard coded for real or complex arithmetic. The template files have since been extended for modular arithmetic, but the details are omitted in this thesis.

SPIRAL maintains internally a collection of ISAs (instruction set architecture) that capture the essential features of the potential target platforms. The target ISA is used during the library generation process in order to produce valid and efficient code for the given platform. Table 3.1 shows the snapshot of a newly added ISA called “SSE 4x32m” for the vectorized modular arithmetic. As we can see, the ISA encapsulates the platform-specific features, including vector width (in terms of number of
elements), the primitive data type, and short-vector memory operations, etc.

<table>
<thead>
<tr>
<th></th>
<th>SSE 4x32m</th>
</tr>
</thead>
<tbody>
<tr>
<td>info</td>
<td>“SSE 4-way 32-bit modular integer ISA”</td>
</tr>
<tr>
<td>v</td>
<td>4</td>
</tr>
<tr>
<td>t</td>
<td>TVect(TModInt, 4)</td>
</tr>
<tr>
<td>ctype</td>
<td>“int32_t”</td>
</tr>
<tr>
<td>includes</td>
<td>() \rightarrow\ Concat(“stdint.h”, ...)</td>
</tr>
<tr>
<td>svload_init</td>
<td>(vt) \rightarrow ...</td>
</tr>
<tr>
<td>svstore_init</td>
<td>(vt) \rightarrow ...</td>
</tr>
</tbody>
</table>

...  

Table 3.1: A snapshot of a sample ISA  

The scalar and vectorized Montgomery algorithms shown in Section 3.2 can be automatically generated by SPIRAL with the new extensions. The algorithms are encoded as rewriting rules at the IR (intermediate representation) level. Then, the rewritten IR implementing the algorithm is unparsed by a built-in unparsers to generate compilable code that is valid and efficient for the target platform.

Figure 3.6 shows the workflow of this process using the vectorized algorithm as an example. An IR expression assign(res, mul(a, b)) where res is of type TVect(TModInt, 4) can be captured by the pattern defined in the type-based IR rewriting rule. The rule encapsulates the vectorized algorithm and produces the algorithm expressed as a chain of IR expressions including the declaration of temporary variables. Then, the rewritten IR expressions are unparsed by the vector unparsers to produce efficient implementation of the vectorized Montgomery algorithm. Note that the scalar and vector unparsers in SPIRAL have also been expanded considerably to support the generation of integer and modular arithmetic, but the details are omitted in this thesis.
3.4 Conclusion

In this chapter, we focus on the high performance implementation of the low-level modular arithmetic. We reviewed the hardware support for integer and modular arithmetic, whose limitations requires dedicated implementations of modular arith-
metric using the fast algorithms. Then, we investigated in the well-known algorithm as well as a lesser known technique for the modular multiplication. We showed that the fast algorithms can be vectorized where the full vector width $w$ can be exploited except for a few inevitable $w/2$-way operations. Finally, we explained the extensions in SPIRAL for the automatic generation of high performance modular arithmetic, including the new data types and rules, ISAs, IR rewriting rules, and unparsing rules.
In computer algebra, all the asymptotically fast algorithms for polynomial and large integer arithmetic and modular methods rely on the FFT as a crucial subroutine. In particular, for polynomial and integer multiplication, the FFT is used via the convolution theorem where two forward and one inverse FFTs are performed. FFT algorithms reduce the computational complexity from quadratic to $O(N \log N)$ by recursively factoring a large transform into smaller transforms.

Calculations on univariate and multivariate polynomials can be reduced to computing with polynomials over finite fields, such as the prime field $\mathbb{Z}_p$ for a prime number $p$, via the modular techniques. Moreover, most calculations tend to densify intermediate expressions even when the input and output polynomials are sparse. Therefore, we focus on the automatic library generation and performance tuning for the modular DFT in this chapter.

We begin by reviewing the definition of the modular DFT and its fast algorithms. Next, we explain the automatic generation and optimization of a vectorized and parallel modular DFT library. Finally, we evaluate the performance of the autotuned library by comparing it to the performance of a hand-tuned high performance library used in MAPLE. The result shows that the autotuned parallelism improves the performance by an order of magnitude.

4.1 Modular DFT and Fast Algorithms

Recall the definition of modular DFT:
Definition 9. Given \( n \) integer inputs \( x_0, \ldots, x_{n-1} \), the modular DFT is defined as

\[
y_k = \sum_{0 \leq l < n} \omega_n^{kl} x_l, \quad 0 \leq k < n.
\]  

(4.1)

Therefore, the \( n \)-point modular DFT matrix is

\[
\text{ModDFT}_{n,p,\omega_n} = \left[ \omega_n^{kl} \right]_{0 \leq k,l < n},
\]  

(4.2)

where \( \omega_n \) is a primitive \( n \)th root of unity in \( \mathbb{Z}_p \).

An \( n \)-th root of unity \( \omega \in \mathbb{R} \) is any element such that \( \omega^n = 1 \). If \( n \) is the smallest integer such that \( \omega^n = 1 \), then \( \omega \) is called a primitive \( n \)-th root of unity. The complex numbers \( \mathbb{C} \) contain a primitive \( n \)-th root of unity for any \( n \), for instance \( \exp(-2\pi i/n) \) as seen in 2.2. The primitive roots of unity are more complicated in finite fields such as \( \mathbb{Z}_p \). Formally, a finite field \( \mathbb{F}_p \) with \( p \) elements contains a primitive \( n \)-th root of unity if and only if \( n \mid (p - 1) \). If the multiplicative group of \( \mathbb{F}_q \) has a generator \( \omega \), then \( \omega^{(p-1)/n} \) gives one \( n \)-th primitive root of unity.

Cooley-Tukey algorithm. Let \( n = rs \), then the divide-and-conquer step in the Cooley-Tukey algorithm [6], the most important FFT algorithm, can be represented as a sparse matrix factorization with the tensor product:

\[
\text{ModDFT}_{n,p,\omega_{rs}} = (\text{ModDFT}_{r,p,\omega_r} \otimes I_s) T^n_s (I_r \otimes \text{ModDFT}_{s,p,\omega_s}) L^n_r.
\]  

(4.3)

In (4.3), \( T^n_s \) is a diagonal matrix containing twiddle factors.

An example of the Cooley-Tukey algorithm where \( r = s = 2 \) is

\[
\text{ModDFT}_{4,p,\omega_4} = (\text{ModDFT}_{2,p,\omega_2} \otimes I_2) \cdot T_2^4 \cdot (I_2 \otimes \text{ModDFT}_{2,p,\omega_2}) \cdot L_2^4,
\]  

(4.4)
which is equivalent to factorizing the ModDFT\(_4\) matrix

\[
\begin{bmatrix}
1 & 1 & 1 & 1 \\
1 & \omega_4 & \omega_4^2 & \omega_4^3 \\
1 & \omega_4^2 & \omega_4^4 & \omega_4^6 \\
1 & \omega_4^3 & \omega_4^6 & \omega_4^9
\end{bmatrix}
\] (4.5)

into the sparse matrices:

\[
\begin{bmatrix}
1 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 \\
1 & 0 & -1 & 0 \\
0 & 1 & 0 & -1
\end{bmatrix}
\cdot
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & \omega_4
\end{bmatrix}
\cdot
\begin{bmatrix}
1 & 1 & 0 & 0 \\
1 & -1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0
\end{bmatrix}
\cdot
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\] (4.6)

As previously mentioned, the substructures from applying recursive algorithms to transforms can be mapped to vectorized and multi-threaded codes, and formulae can be transformed to adapt to hardware parameters, such as vector length and number of computing cores. Fig. 4.1 shows such substructures where the four sparse matrices resulted from applying the Cooley-Tukey algorithm to a 16-point modular DFT are represented by four frames (only the non-zero values are plotted). In the first frame of ModDFT\(_4\) \(\otimes\) I\(_4\), for example, scalar operations can be replaced by 4-way vector operations, as each value in ModDFT\(_4\) is duplicated four times along the block diagonals. Further, in the third frame of I\(_4\) \(\otimes\) ModDFT\(_4\), it is obvious that the same computing kernel ModDFT\(_4\) is applied to four non-overlapping sub-vectors of the permuted input vector, which can be interpreted as a parallelizable loop with no loop-carried dependencies. In fact, the general forms of the first and the third frames, A \(\otimes\) I and I \(\otimes\) A where A is an arbitrary matrix, are the two typical formulae for automatic vectorization and parallelization, as we reviewed in Section 2.2.
Since the Cooley-Tukey factorization holds for any field with a primitive \( n \)th root of unity, the same machinery in SPIRAL can be used for generating and optimizing modular FFTs, provided the infrastructure is extended to support new data types and the code generation and compiler optimizations are similarly enhanced.

Other FFT algorithms, including the Prime Factor Algorithm \[18\] and Rader’s Algorithm \[44\], can be similarly expressed in SPL as follows:

\[
\text{ModDFT}_{rs} \rightarrow V_{r,s}^{-1} (\text{ModDFT}_{r} \otimes I_s)(I_r \otimes \text{ModDFT}_s)V_{r,s},
\]
(4.7)

\[
\text{ModDFT}_{rs} \rightarrow W_n^{-1} (I_1 \oplus \text{ModDFT}_{n-1}) E_n(I_1 \oplus \text{ModDFT}_{n-1}) W_n.
\]
(4.8)

The prime factor FFT in (4.7) requires that \( n = rs \) where \( r \) and \( s \) are relative primes \( \gcd(r,s) = 1 \), and computes the \( \text{ModDFT}_n \) using \( r \text{ ModDFT}_s \)'s and \( s \text{ ModDFT}_r \)'s as the Cooley-Tukey algorithm. The Rader FFT in (4.8) requires that \( n \) is prime and reduces \( \text{ModDFT}_p \) to 2 \( \text{ModDFT}_{p-1} \)'s.

The permutation matrices \( V \) and \( W \) correspond to the following permutations:

\[
v_{r,s}^r : i \mapsto (r \lfloor i/r \rfloor + s(i \mod r)) \mod rs, \gcd(r,s) = 1,
\]
(4.9)

\[
w_g^n : i \mapsto g^{i-1} \mod n \text{ if } i \neq 0, 0 \text{ otherwise.}
\]
(4.10)
4.2 Library Generation

In this section, we explain the generation of the modular DFT library using the Cooley-Tukey algorithm. We restate the breakdown rule in (4.3), replacing the twiddle matrix $T^n_s$ with $\text{diag}(d)$, where $d^{n\rightarrow n}$ is the scalar function providing the diagonal entries of $T^n_s$. The modulus $p$ and the root of unity $\omega$ are omitted for simplicity

$$\text{ModDFT}_n = (\text{ModDFT}_r \otimes I_{n/r}) \text{diag}(d) (I_r \otimes \text{ModDFT}_{n/r}) L^n_r.$$  \hspace{1cm} (4.11)

With $\Sigma$-SPL rewriting, the diagonal and the stride can be fused into the nearby tensor products. As a result, the Cooley-Tukey FFT is computed in two loops corresponding to the two tensor products decorated with the twiddles and the stride permutation.

$$\text{ModDFT}_n = \underbrace{(\text{ModDFT}_r \otimes I_{n/r}) \text{diag}(d)}_{\text{loop}} \underbrace{(I_r \otimes \text{ModDFT}_{n/r}) L^n_r}_{\text{loop}}.$$  \hspace{1cm} (4.12)

The fusion of the stride permutation $L^n_r$ requires a ModDFT function ($\text{moddft}_{\text{str}}$) with different input and output stride, and the fusion of the diagonal $\text{diag}(d)$ requires a ModDFT function ($\text{moddft}_{\text{scaled}}$) with an extra array parameter holding the scaling factors. These functions are implemented recursively based on the Cooley-Tukey algorithm, and do not require any new functions. Eventually, the modular DFT size becomes sufficiently small, and the recursion terminates by using codelet functions that compute small fixed-size transforms. The corresponding pseudo code looks like follows.

```c
void moddft(int n, int *y, int *x) {
    int r = choose_factor(n);
    // y= (I_r tensor ModDFT_{n/r}) L(n, r) * x
```
for(int i=0; i<r; ++i)
    moddft_str(n/r, r, 1, y+n/r*i, x+n/r*i);
/y=(ModDFT_r tensor I_n/r)diag(d)*y
for(int i=0; i<n/r; ++i)
    moddft_scaled(r, n/r, precomputed_d[i], y+i, y+i);
}
void moddft_str(int n, int in_str, int out_str, int *y, int *x) {
    //to be implemented
}
void moddft_scaled(int n, int str, int *d, int *y, int *x) {
    //to be implemented
}

We say the functions moddft, moddft_str and moddft_scaled form a recursion step closure, which is a minimal set of functions sufficient to compute the desired transform. The recursion step closure is the central concept in the library generation framework. The recursion step closure of ModDFT_n is systematically derived as follows:

**Step 1: Apply the breakdown rule.** The Cooley-tukey breakdown rule is applied to DFT. The non-terminals are tagged with the curly brackets.

\[
\{\text{ModDFT}_n\} = (\{\text{ModDFT}_r\} \otimes I_s) \text{diag}(d)(I_r \otimes \{\text{ModDFT}_s\}) L_r^n. \tag{4.13}
\]

**Step 2: Convert to \(\sum\)-SPL.** The SPL must be rewritten into \(\sum\)-SPL in order to merge the diagonal and the permutation with the adjacent tensor products. Note that the index mapping functions \(h\)'s are different in the decoration gathers and
scatters, but the details are omitted below.

\[
\left( \sum_{i=0}^{s-1} S(h)\{\text{ModDFT}_r\}G(h) \right) \text{diag}(d) \left( \sum_{j=0}^{r-1} S(h)\{\text{ModDFT}_s\}G(h) \right) \text{perm}(l_r^n).
\]

(4.14)

**Step 3: Apply loop merging rewrite rules.** The diagonal and the permutation can be merged with the adjacent loops, leading to

\[
\left( \sum_{i=0}^{s-1} S(h)\{\text{ModDFT}_r\} \text{diag}(d \circ h)G(h) \right) \left( \sum_{j=0}^{r-1} S(h)\{\text{ModDFT}_s\}G(l_r^n \circ h) \right).
\]

(4.15)

**Step 4: Apply index simplification rules.** The index mapping functions can be simplified as function composition.

\[
\left( \sum_{i=0}^{s-1} S(h)\{\text{ModDFT}_r\} \text{diag}(d \circ h)G(h) \right) \left( \sum_{j=0}^{r-1} S(h)\{\text{ModDFT}_s\}G(h) \right).
\]

(4.16)

**Step 5: Extract the required recursion steps.** So far we performed loop merging for one step of the Cooley-Tukey algorithm. To recursively merge loops in the sub-transforms, the decorations must be pushed to the inside of the current recursing steps, effectively expanding the scope of the recursion step tags.

\[
\left( \sum_{i=0}^{s-1} \{S(h)\text{ModDFT}_r, \text{diag}(d \circ h)G(h)\} \right) \left( \sum_{j=0}^{r-1} \{S(h)\text{ModDFT}_s, G(h)\} \right).
\]

(4.17)

In the two recursion steps above, two distinct recursion steps emerge in the dual
loop structure.

\[
\sum_{i=0}^{s-1} \{ S(h) \text{ModDFT}_r \text{diag}(d \circ h) G(h) \},
\]
\[
(4.18)
\]

\[
\sum_{j=0}^{r-1} \{ S(h) \text{ModDFT}_s G(h) \}.
\]
\[
(4.19)
\]

The first recursion step (4.18) contains the diagonal matrix which scales the input, corresponding to the \textit{moddft\_scaled}. The second recursion step (4.19) contains the input and output index strides, corresponding to the \textit{moddft\_strided}. Repeating steps 1-5 to the recursion steps (4.18) and (4.19) shows that no new recursion steps are needed, as the the decorations will be automatically merged. Therefore \{\text{ModDFT}_n\}, (4.18), and (4.19) form a recursion step closure, a sufficient set of recursion steps to recursively implement the modular DFT using the Cooley-Tukey algorithm.

4.3 Performance Evaluation

This section reports performance evaluation comparing the performance of the SPIRAL-generated fixed-size code and the autotuned parallel modular FFT library against the performance of the hand-optimized modular FFTs from the \texttt{modpn} library.

Test setup. All experiments were performed on an Intel Core i7 965 quad-core processor running at 3.2 GHz with 12 GB of RAM. Generated code was compiled with gcc version 4.3.4-1 with optimization set to O3. Vector code used SSE 4.2 with 4-way 32-bit integer vectors. The experiments were performed using 32-bit integers and 16-bit primes. The performance is measured by PAPI [40] and is averaged over 1000 runs with small observed performance variance. We report the performance in Gops (giga-ops) or billions of operations per second (higher is better), which is calculated assuming that modular DFT of size \(n\) takes a total of \((3/2)n \log(n)\) additions, subtractions and nontrivial multiplications.
Baseline implementation: modular FFT in modpn. The modpn library is a software library, mainly written in C, which supports general multivariate polynomial computations over finite fields based on fast algorithms, and targets efficient implementation of modular algorithms. modpn has shown performance improvement over Magma in [31], while Magma’s performance is comparably favored over NTL in [20] for polynomial multiplication. Therefore, modpn serves as an ideal baseline implementation to represent the state-of-the-art of polynomial arithmetic implementations. modpn takes advantage of certain hardware features such as SSE instructions and thus performs better on some platforms than others. Montgomery’s trick is used to avoid integer division, which can be costly on many machines. Various code optimization techniques are used (or have been attempted and abandoned later) such as explicit register allocation, thread-level parallelism, loop unrolling, and inclusion of
SPIRAL-generated code. SPIRAL generated algorithms use the Cooley-Tukey rule with a dynamic programming based search engine to select an “optimal” recursive breakdown strategy. Dynamic programming is only a heuristic since an optimal algorithm of a given size can depend on the context in which it is called; however, experience shows that it makes good choices.

As shown in Fig. 4.2, all SPIRAL generated codes are faster than the baseline modular FFT implementation. The fixed-size vector code provides speed-up of 26 times for medium sizes and 11 times for large sizes. The general size parallel library provides comparable speedups as the fixed-size code. The speedups are the result of the search engine selecting the optimal breakdown strategy for the given size that adapts to the memory hierarchy of the target platform. Loops are merged to reduce passes on the input data, and small transforms are fully unrolled to avoid loop overhead. Hardware tags and rewriting systems further optimize the generated code to fully exploit vector level parallelisms and thread level parallelism.

General size parallel library and fixed-size code. The performance of scalar and vector algorithms in the general size library are within 81% to 91% of the performance of corresponding fixed-size algorithms. For large sizes, thread level parallelism in the parallel library leads to roughly 1.5 time speedup over the fixed-size algorithm. The combined speedup for the general size parallel library over baseline implementation is roughly 15 times for large sizes, compared to 11 times of the fixed-size code.

Both fixed-size code generation and general size library use DP search engine to search for the optimal algorithm for a given size. For the fixed-size code, all parameters including the transform size are known at generation time, which enables computation of constants like the twiddle factors at generation time. The constants are then used in fully unrolled small size algorithms. For the general size library,
all or a partial set of the parameters are generated as symbolic variables that are instantiated at runtime. Therefore, a DP search engine is automatically integrated into the generated library to perform runtime planning.

4.4 Conclusion

In this chapter, we presented an automatically generated and optimized library for the modular DFT that is an order of magnitude faster than a hand-optimized library. We have shown how SPIRAL can be extended to generate high performance library for the modular FFT algorithms. The transform and algorithms are represented and optimized at a high abstraction level with little human effort. The generated library supports modular arithmetic, fully exploits vector and thread parallelisms of the target platform, and offers comparable performance improvement of the fixed-size implementations over the hand-optimized implementations. The general size library generation approach is advantageous over the fixed-size approach, as it compiles breakdown rules to recursive function closure and searches for optimal strategies with runtime parameters. This shows that the automated performance tuning can be incorporated into the general size library generation to take advantage of platform-dependent optimizations, in the context of quickly evolving hardware acceleration technologies.

The modular DFT will be used in the FFT-based convolution algorithm, where its performance gains will benefit the higher-level applications. However, the modular FFT library exhibits the staircase phenomenon. In Chapter 5, we will design and develop new algorithms for the TFT with lower arithmetic cost compared to the modular DFT, in order to smooth the performance between powers of two. The performance of the underlying transforms will be thoroughly analyzed to find the best transform candidate, as their performance directly affects the performance of
the FFT-based convolution and modular polynomial multiplication implementations.
5. TFT and ITFT

The truncated Fourier transform (TFT) and its inverse (ITFT) are invented to avoid the wasted computation in the FFT implementations where an arbitrary input size that is not exactly a power of two is zero-padded to the next power of two. However, two performance-critical properties of the FFT are lost in using the TFT: general-radix decomposition and parallelism, in that the algorithms are limited to fixed decomposition strategies and can not be expressed as sparse matrix factorization where substructures can be automatically tuned for vectorization and multi-threading.

In this chapter, we present new general-radix algorithms for the TFT and ITFT which expand the scalar implementation space where the optimal implementation adapted to memory hierarchy can be found and generated; Then, we present new parallel algorithms for the TFT and ITFT that trade off small arithmetic cost for full vectorization and improved multi-threaded parallelism.

The algorithms are automatically generated and tuned to produce arbitrary-size libraries. The new algorithms and the implementations smooth out the staircase performance associated with power-of-two modular FFT implementations, and provide significant performance improvement over zero-padding approaches even when high-performance FFT libraries are used.

5.1 Background

The TFT introduced in by Joris van der Hoeven follows the classic radix-2 divide-and-conquer FFT paradigm, which recursively splits the transform into two half-sized transforms where the input and output values non-contiguous and of arbi-
trary size. The author also shows how to invert the TFT with the “cross butterflies” and alternating row and column transforms in the ITFT. The algorithms accelerate the convolution-based univariate polynomial multiplication with reduced arithmetic cost. An example of the forward transform is shown in Fig. 5.1.

Recently, considerable effort has been made to improve the performance of TFT and ITFT implementations. In [20], Harvey reported improved performance of the TFT by decomposing a TFT of size $2^l$ into smaller transforms of sizes of $2^\lfloor l/2 \rfloor$ and $2^\lceil l/2 \rceil$, achieving improved cache performance. Harvey and Roche showed in [22] how the TFT can be done in-place. In [34, 1], Mateer and Arnold devised and improved the in-place cyclotomic TFT, respectively.

While the TFT reduces operation count and smooths out the staircase performance associated with padding, it is not clear that it is beneficial in practice, due to the limitations of fixed decomposition strategies and scalar implementation. To be comparable or beneficial to highly tuned FFT implementations, general-radix and parallel algorithms must be derived for optimal scalar code generation and efficient utilization of features like vector extension and multi-threading that are widely available on modern computing platforms.

Extensive examples [43, 49] have shown that a general Cooley-Tukey type algo-
Algorithm that allows arbitrary decomposition, usually accompanied with a search mecha-
nism, can generate optimal breakdown strategy for various computing platforms. Fur-
thermore, if the algorithms can be expressed as recursive structured sparse matrix
factorization, their implementations can be automatically derived and generated with
vectorization and parallelization. In the next two sections, we present general-radix
and parallel algorithms for both TFT and ITFT. The general-radix algorithms enforce
the strict truncation while generalizing the existing TFT algorithms. The parallel al-
gorithms introduce small relaxations for large transform sizes which trade off slightly
higher arithmetic cost for improved data flow which allows full vectorization and par-
allelization. The algorithms are automatically derived and tuned using the SPIRAL
system for code generation and adaptation. The new algorithms provide significant
performance improvement over approaches that pad to the next power of two even
when using high-performance FFT libraries.

As the general-radix algorithms may decompose TFT and ITFT along an arbitrary
decomposition path, the transformed result is permuted. Since the TFT and ITFT
are intended to be used together in the the convolution, their decomposition trees
must be mirrored. Note that the Cooley-Tukey algorithm can recursively decompose
a DFT of a composite size $mn = m \cdot n$ in two ways, as shown in (5.1) and (5.2).

$$\text{DFT}_{mn} = (\text{DFT}_m \otimes I_n) \cdot T^m_{mn} \cdot (I_m \otimes \text{DFT}_n) \cdot L^m_{mn} \quad (5.1)$$

$$= L^n_{mn} \cdot (I_m \otimes \text{DFT}_n) \cdot T^m_{mn} \cdot (\text{DFT}_m \otimes I_n). \quad (5.2)$$

The two variations represent two decomposition approaches which are differenti-
ated by when the permutation and stride DFT are applied. Since the Cooley-Tukey
algorithm applies to both forward and inverse DFT, the DFT-based convolution al-
gorithm can skip the permutations by applying (5.2) two forward DFT and (5.1) to
inverse DFT, since we have \( L_m^m \cdot L_m^n = I_{mn} \). Note that the mirrored breakdown trees must be imposed on the forward and inverse transforms, in order to cancel the recursively accumulated permutations. Moreover, this relationship can carry over to the TFT-based convolution, provided the TFT/ITFT algorithms are defined analogously to that of the DFT.

5.2 TFT

Recall the original definition of the TFT:

**Definition 10.** The TFT of size \( N = 2^n \) is defined as an operation that takes the inputs \( x_0, \ldots, x_{l-1} \) and produces \( y_0, \ldots, y_{m-1} \), with

\[
y_i = \sum_{j=0}^{l-1} x_j \omega_{N}^{[i]_n},
\]

where \( i \in \{0, \ldots, m-1\} \), \( 1 \leq l, m \leq N \), and \([i]_n\) is the bit-reversal of \( i \) at length \( n \).

The only complication with this computation is the bit-reversal. For example, assuming \( N = 2^4 = 16 \), then the binary representation of 3 is 0011. Reversing the bits yields the bit-reversed \([3]_4 = 1100 = 12\).

In this section we denote a truncated Fourier transform as \( \text{TFT}_{n,l,m} \), where \( n \) is the size of the transform, \( 1 \leq l \leq n \) is the size of the input (assuming \( x_{l} = \cdots = x_{n-1} = 0 \)), and \( 0 \leq m \leq n \) is the size of the truncated output.

5.2.1 Strict General-radix TFT Algorithm

A general-radix TFT algorithm is similar to that for DFT, except that the strict truncation yields non-uniform decomposition: the DFT recursively breaks down into two smaller transforms as in \([2.20]\), and the TFT usually breaks down into four smaller transforms.
Let \( n = rs, l_c = \lfloor l/r \rfloor, l_{c+1} = \lfloor l/r \rfloor, m_c = \lfloor m/r \rfloor, l_r = \min(l, r), m_r = m \mod r, \)
\( c_1 = \lfloor m/r \rfloor, c_2 = \lfloor m/r \rfloor - \lfloor m/r \rfloor, c_3 = l \mod r \) and \( c_4 = l_r - l \mod r, \) then the four smaller transforms (two row transforms and two column transforms respectively) generated by factoring \( TFT_n \) are

\[
TFT_{\bar{r}_1} = TFT_{r,l,r}, \quad TFT_{\bar{r}_2} = TFT_{r,l,m},
\]
\[
TFT_{\bar{c}_1} = TFT_{s,l,c+1,m}, \quad TFT_{\bar{c}_2} = TFT_{s,l,m}.
\]

and the algorithm can be expressed in \( \sum \)-SPL as:

\[
TFT_{n,l,m} = [(I_{\bar{c}_1} \otimes TFT_{\bar{r}_1}) \oplus (I_{\bar{c}_2} \otimes TFT_{\bar{r}_2})] \cdot T_{r,s}^n \cdot \left[ \sum_{c_1} (S_{\bar{c}_1} TFT_{\bar{c}_1} G_{\bar{c}_1}) + \sum_{c_2} (S_{\bar{c}_2} TFT_{\bar{c}_2} G_{\bar{c}_2}) \right],
\]

where “+” denotes non-overlapping matrix sum, \( \oplus \) denotes direct sum of matrices, and \( G_{\bar{c}_1}, G_{\bar{c}_2}, S_{\bar{c}_1} \) and \( S_{\bar{c}_2} \) are the decoration gather and scatter matrices, which together interleaver the computational kernels \( TFT_{\bar{c}_1} \) and \( TFT_{\bar{c}_2} \) onto the input vector. The detailed definitions of the decoration matrices are shown as below:

\[
S_{\bar{c}_1} = S(h_{l,r}^{n \rightarrow m}), \quad G_{\bar{c}_1} = G(h_{l,r}^{l \rightarrow l+1}),
\]
\[
S_{\bar{c}_2} = S(h_{c+1,i,r}^{n \rightarrow m}), \quad G_{\bar{c}_2} = G(h_{c+1,i,r}^{l \rightarrow l}).
\]

Note that the algorithm is applied to the input vector from right to left in the matrix factorization. Line 5.5 represents the two types of column transforms being applied to the proper columns based on \( l \) and \( m \), followed by the twiddle matrix. The direct sum of the two tensor products in line 5.4 represents the two types of row transforms being applied to contiguous subvectors of the input vector interpreted as
Figure 5.2: A decomposition example of $\text{TFT}_{32,17,30}$, where $\bigcirc$ represents the input values and $\bullet$ represents the output values.

Figure 5.2 shows the an instance of TFT where $l > r$ and $r \nmid l$. In this case, $l = 17$, $m = 30$, and $n = 32$ is factored as $8 \times 4$. frame (a) shows the input vector as a 2-D matrix based on the factorization, as well as one column transform $\text{TFT}_{c_1} (\text{TFT}_{8,5,8})$ being applied to left row(s) covered by lines going from bottom left to top right. In frame (b), three column transforms of $\text{TFT}_{c_2} (\text{TFT}_{8,4,8})$ are applied to the rest of columns. In frame (c), seven row transforms $\text{TFT}_{r_1} (\text{TFT}_{4,4,4})$ are applied to top rows, which are represented by the area covered by lines going from bottom right to top left. Finally, in frame (d), the last row transform $\text{TFT}_{r_2} (\text{TFT}_{4,4,2})$ is applied to the last row. Clearly, the general radix algorithm introduces non-uniform substructures in its recursive application. The need for explicit $\sum -\text{SPL}$ expression and the non-uniform substructures prevent it from being easily mapped to vectorization and parallelization.

Complexity Analysis. Let $\tilde{n} = \log_2(n)$, van der Hoeven showed that the computational complexity of the TFT and ITFT is bounded by $\tilde{m}/2 + \tilde{n}$ butterfly operations. Then Harvey proved that the cache-friendly TFT and ITFT follow the same estimates. The complexity analysis of the general-radix TFT and ITFT algorithms are similar to the analysis presented in [20]. The following theorem indicates that
the running time of the TFT is relatively smooth as a function of \( m \), the size of the truncated output.

**Theorem 1.** The number of executions of the TFT base case is bounded by \( \min((m-1)n/2 + n - 1, n\bar{n}/2) \).

**Proof.** For the base case \( n = 2 \), the number of base case execution is simply \( \min((m-1)/2 + 1, 2\cdot1/2) = 1 \), which is trivially true. For \( n \geq 4 \), we prove the two possible bounds in \( \min() \) individually by induction. Let \( \bar{r} = \log_2 r \) and \( \bar{s} = \log_2 s \).

We first prove the bound \( n\bar{n}/2 \).

- **Column transforms:** The size \( TFT_{\bar{c}_1} \) and \( TFT_{\bar{c}_2} \) is \( s \), therefore by induction the number of calls to base case in each column transform is bounded by \( ss\bar{s}/2 \). Hence the total number of calls by the column transform is bounded by \( rs\bar{s}/2 \), since there is at most \( r \) column transforms.

- **Row transforms:** The number of calls to base case by the row transforms \( TFT_{\bar{r}_1} \) and \( TFT_{\bar{r}_2} \) is bounded by \( c_2 r\bar{r}/2 = \lceil m/r \rceil r\bar{r}/2 \leq sr\bar{r}/2 \).

Therefore the sum is bounded by \( rs\bar{s}/2 + rs\bar{r}/2 = rs(\bar{s} + \bar{r})/2 = n\bar{n}/2 \).

Next we prove the more interesting bound \( (m-1)\bar{n}/2 + n - 1 \) which is associated with the size of the truncated output.

- **Column transforms:** By induction, the total number of calls by the column transform is bounded by \( r((m_c - 1)\bar{s}/2 + s - 1) \).

- **Top row transforms:** \( I_{c_1} \otimes TFT_{\bar{r}_1} \) calls the base case at most \( c_1 r\bar{r}/2 \) times, as the first bound is proved.

- **Bottom row transform:** \( I_{c_2} \otimes TFT_{\bar{r}_2} \) by induction calls the base case at most \( c_2 ((m_r - 1)\bar{r}/2 + r - 1) = (m_r - c_1)((m_r - 1)\bar{r}/2 + r - 1) \) times.
Therefore the sum of calls to base case is bounded by

\[
r((m_c - 1)\tilde{s}/2 + s - 1) + c_1(r\tilde{r}/2) + (m_c - c_1)((m_r - 1)\tilde{r}/2 + r - 1)
\]

\[
= \frac{1}{2}(r(m_c - 1)\tilde{s} + c_1r\tilde{r} + (m_c - c_1)(m_r - 1)\tilde{r}) + (r(s - 1) + (m_c - c_1)(r - 1))
\]

\[
= \frac{1}{2}(r\tilde{s}(m_c - c_1) + r\tilde{s}(c_1 - 1) + c_1r\tilde{r} + (m_c - c_1)(m_r - 1)\tilde{r})
\]

\[
+ (rs - r + (m_c - c_1)(r - 1))
\]

\[
= \frac{1}{2}(c_1r\tilde{n} + r\tilde{s}(m_c - c_1) - r\tilde{s} + (m_c - c_1)(m_r - 1)\tilde{r})
\]

\[+ (n - 1 + (m_c - c_1 - 1)(r - 1))
\]

\[
= \frac{1}{2}((m - m_r)\tilde{n} - r\tilde{s} + (m_c - c_1)((r\tilde{s} + (m_r - 1)\tilde{r}) + (n - 1 + (m_c - c_1 - 1)(r - 1)))
\]

\[
= \frac{1}{2}((m - 1)\tilde{n} + (1 - m_r)\tilde{n} - r\tilde{s} + (m_c - c_1)((r\tilde{s} + (m_r - 1)\tilde{r})
\]

\[+ (n - 1 + (m_c - c_1 - 1)(r - 1))
\]

\[
= \frac{1}{2}((m - 1)\tilde{n} + (m_r - 1)(\tilde{r}(m_c - c_1) - \tilde{n}) + r\tilde{s}(m_c - c_1 - 1))
\]

\[+ (n - 1 + (m_c - c_1 - 1)(r - 1))
\]

\[
= \frac{1}{2}(m - 1)\tilde{n} + n - 1 + \frac{1}{2}((m_r - 1)(\tilde{r}(m_c - c_1) - \tilde{n}) + r\tilde{s}(m_c - c_1 - 1))
\]

\[+ (m_c - c_1 - 1)(r - 1).
\]

Notice that the underlined terms are in the form of the desired bound. Let \( \varepsilon \) be the rest of the terms in the sum. Recall that \( m_r = m \mod r \), \( m_c = \lceil m/r \rceil \) and \( c_1 = \lfloor m/r \rfloor \). There are two cases based on \( r|m \). If \( r \nmid m \), then \( m_c - c_1 = 1 \) and \( m_r \geq 1 \). As a result, \( \varepsilon = (m_r - 1)(\tilde{r} - \tilde{n})/2 = -(m_r - 1)\tilde{s}/2 \leq 0 \). If \( r|m \), then \( m_c - c_1 = 0 \) and \( m_r = 0 \). Therefore, \( \varepsilon = (\tilde{n} - r\tilde{s})/2 - (r - 1) = (\tilde{r} + \tilde{s} - 2\tilde{r}\tilde{s})/2 - (r - 1) \leq -(r - 1) < 0 \), since \( 2\tilde{r} \geq \tilde{r} + 1 \). In conclusion, the truncated output-based bound holds in both cases.

Combining the two proved bounds gives the desired bound of the number of exe-
cutions of the TFT base cases: \( \min((m - 1)\tilde{n}/2 + n - 1, n\tilde{n}/2) \).

Theorem 1 shows that the running time of the general-radix TFT algorithm is relatively smooth with regard to the size of the truncated output. Solving \((m - 1)\tilde{n}/2 + n - 1 \leq n\tilde{n}/2\) yields \(m \leq n + 1 - 2n/\tilde{n} + 2/\tilde{n}\).

### 5.2.2 Parallel TFT Algorithm

In the general-radix TFT algorithm, recall that the two column transforms \(\text{TFT}_{c_1}\) and \(\text{TFT}_{c_2}\) take \(l_{c+1} = \lceil l/r \rceil\) and \(l_c = \lfloor l/r \rfloor\) as their inputs, respectively. If we relax \(l_c\) to \(\lfloor l/r \rfloor\), then all the column transforms become uniform. Similarly, if we relax the input to the last row transform \(\text{TFT}_{r_2}\) to \(r\), then all the row transforms become uniform. With such relaxations, the decomposition can be expressed in SPL, which can be further automatically derived and generated for improved vector and multi-thread parallelism.

However, the relaxation also raises an important question: will the performance gain from the truncation be preserved if the relaxation is performed at each decomposition step? In Theorem 2, we prove that the truncation can be preserved as the relaxed decomposition reaches a base case via an arbitrary decomposition path.

**Theorem 2.** Let \(N\) be the transform size of a given TFT, \(L\) be the size of the truncated input, and \(B\) be the size of the base case chosen by the search engine, then the size of the truncated input to the base case column transforms in the relaxed general TFT algorithm is bounded by \(\lceil LB/N \rceil\).

**Proof.** Let \(T = N/B\), and \(T = t_1t_2\ldots t_k\) be an arbitrary factorization. Then the arbitrary factorization of \(N\) introduces a relaxed input \(l_B\) when the base case \(B\) is
reached as a column transform:

$$l_B = \left\lceil \left\lceil \left\lceil \frac{L}{t_1} \right\rceil /t_2 \right\rceil \ldots /t_k \right\rceil$$  (5.6)

$$= \left\lceil \frac{L}{t_1 t_2 \ldots t_k} \right\rceil$$  (5.7)

$$= \left\lceil \frac{LB}{N} \right\rceil$$  (5.8)

To prove (5.6) \(\rightarrow\) (5.7) is equivalent to proving

$$\left\lceil \left\lceil \frac{L}{t_i} \right\rceil \right\rceil /t_j \leq \frac{q}{t_j}$$

We will prove this by contradiction below. Let \(q = \left\lceil L/t_i \right\rceil\), by the definition of the ceiling operation, we have

$$q - 1 < \frac{L}{t_i} \leq q$$

$$\Rightarrow \frac{q - 1}{t_j} < \frac{L}{t_i t_j} \leq \frac{q}{t_j}$$

$$\Rightarrow \left\lceil \frac{L}{t_i t_j} \right\rceil \leq \left\lceil \frac{q}{t_j} \right\rceil$$  (5.9)

Assume in (5.9) that \(\left\lceil \frac{L}{t_i t_j} \right\rceil < \left\lceil \frac{q}{t_j} \right\rceil\), then \(\exists k \in \mathbb{Z}\) s.t.

$$\frac{L}{t_i t_j} \leq k < \frac{q}{t_j}$$

$$\Rightarrow \frac{L}{t_i} \leq t_j k < q$$

But \(q = \left\lceil \frac{L}{t_i} \right\rceil\), which leads to a contradiction of \(q \leq t_j k < q\). Therefore, \(\left\lceil \frac{L}{t_i t_j} \right\rceil\) must be equal to \(\left\lceil \frac{q}{t_j} \right\rceil\) in (5.9), i.e. \(\left\lceil \frac{L}{t_i t_j} \right\rceil = \left\lceil \frac{L}{t_i t_j} \right\rceil\), and therefore, \(\left\lceil \left\lceil \frac{L}{t_i} \right\rceil /t_2 \right\rceil \ldots /t_k \right\rceil = \left\lceil \frac{L}{t_1 t_2 \ldots t_k} \right\rceil\)

Similarly, let \(M\) be the size of the truncated output of the original TFT, then the bound of the output size of the base case transform can also be proved as \(\left\lceil MB/N \right\rceil\). While the worst case of the arithmetic overhead introduced by the relaxation is pessimistic, the theorem shows that optimal arithmetic cost is determined by the avail-


able fixed-size base cases. Therefore, the overhead can be reduced by generating bigger fully optimized base cases.

Examples of how the factorization and the base case affect the computation overhead can be seen in Figure 5.3. Frame (a) shows the relaxation if 32 is factored as $2 \times 16$, the relaxed algorithm will perform 16 small column transforms of $TFT(2, 2, 2)$. Therefore, no truncation and the associated benefit of reduced arithmetic cost remains. Frame (b) shows another factorization option where 32 is factored as $8 \times 4$, then column transform of size 8 is recursively factored as $4 \times 2$. Assume size-4 TFTs are available as base cases, we see that the final input to base cases is $\lceil 17/8 \rceil = 3$.

The search engine in SPIRAL is capable of choosing the optimal factorization with the strict general-radix algorithm in base case generation, thus fully preserving the reduced arithmetic cost. And the generated library is also capable of searching for the optimal factorization with the relaxed parallel algorithm at runtime.

Next, we introduce a parallel TFT algorithm that introduce a small relaxation for larger problem sizes which trades off slightly higher arithmetic cost for improved data
flow which allows full vectorization and parallelization. The relaxation is accumulated in order to simplify the inverse transform ITFT.

Let \( N = 2^n = rs \) be the size of the transform, \( B = 2^b \) be the size of the base case transform generated by the strict general-radix algorithm in Section 5.2.1. \( L \) and \( M \) be the input and output sizes, respectively. Let \( l_s = \lceil LB/N \rceil \cdot 2^{\max([n/2],b)-b} \), \( m_s = \lceil MB/N \rceil \cdot 2^{[n/2]-b} \), \( l_r = \min(L,r) \), \( m_r = \min(M,r) \), then the row and column transforms are

\[
\begin{align*}
\text{TFT}_{\tilde{r}} &= \text{TFT}_{r,l_r,m_r}, \\
\text{TFT}_{\tilde{c}} &= \text{TFT}_{s,l_s,m_s}.
\end{align*}
\]

and the relaxed parallel Cooley-Tukey algorithm for TFT is:

\[
\text{TFT}_{N,L,M} = (I_{m_s} \otimes \text{TFT}_{\tilde{r}}) \cdot T_{r,s}^{m} \cdot (\text{TFT}_{\tilde{c}} \otimes I_{l_r}).
\]

Note that \( l_s \) and \( m_s \) now compute the accumulated relaxation from the lower levels of recursion, as the the relaxed decomposition strategy is used. The terms \( \lceil LB/N \rceil \) and \( \lceil MB/N \rceil \) in \( l_s \) and \( m_s \) are proved in Theorem 2 as the bounds of input and output sizes to the base case column transform.

The relaxation happens when \( r \nmid l \). As a result, all column transforms and all row transforms become uniform, respectively, enabling the expression of the algorithm in SPL. More specifically, the SPL expressions such as \( I \otimes A \) and \( A \otimes I \) can be automatically derived and optimized for vectorization and parallelization as illustrated in Section 2.2.

Also note that, compared to 5.2, 5.12 does not perform explicit permutation \( L \) during the recursive factorization. As a result, the transformed values are permuted based on the specific factorization path. However, as we use the TFT in the convolu-
Figure 5.4: An example of factoring $TFT_{32,17,17}$ with the relaxed general radix TFT algorithm.

Figure 5.4 shows an example of the relaxation. Assuming the base case size is 4, in frame (a), the relaxation is applied by adding the circled cells to the dotted cells to be computed together, therefore obtaining uniform row and column transforms. In frame (b), the right tensor product in (5.12) containing $TFT_c$ and the twiddle matrix are applied to the relaxed columns; and in frame (c), the left tensor product containing $TFT_r$ is applied to the relaxed rows. The relaxation includes the accumulated lower-level relaxations at each recursive level.

5.2.3 Performance Evaluation

The general-radix and parallel TFT algorithms are combined in the TFT library generation and optimization. The general-radix is used to generate fixed-size base cases in the library, since its strict truncation retains the reduced computation count. The base cases are generated using a single basic block of straight-line code. Inside the basic block, SPIRAL can apply constant propagation and dead code elimination.
tion to obtain an optimized TFT implementation that does not perform unnecessary operations. The parallel algorithm is used to generate the arbitrary size library, as it improves the vectorization and multi-thread parallelization with small relaxations. The arithmetic is performed over finite field, and the autotuned codes employ the Montgomery reduction [38] to avoid expensive division operations in modular arithmetic. The performance is reported in cycles measured by PAPI [40] and is averaged over 1000 runs with small observed performance variance.

**Baseline implementation.** A SPIRAL-generated parallel modular DFT library is used as the baseline implementation. The reference library pads input to the next power of two, and fully utilizes vector registers and multi-core. It has been shown in [35] that its performance is comparable to optimized fixed-size codes [36], and gains an order-of-magnitude speed-up over hand-optimized library [31]. The ModDFT library is represented by the dashed line in Figure 5.5, which exhibits clear jumps in running time when the lengths cross a power-of-two boundary.

**Autotuned TFT library.** The TFT library employs the strict general-radix algorithm for base case generation to reduce arithmetic cost. The built-in search engine uses the dynamic programming technique and measures the actual running time of smaller transforms as the input to the feedback loop to guide the generation of the base case sizes up to 16. For the library-level recursive breakdown, it applies the parallel algorithm, which trades off a slightly higher arithmetic cost for vectorization and parallelization. A search engine is integrated into the library to perform the DP search at runtime.

**Speedup.** The TFT library delivers a speedup of 33% – 40% over the high performance ModDFT library with length that just crosses a power-of-two boundary. As the length increases, the gap between the truncated transform and the full transform decreases. Overall, the TFT library’s performance is smooth with respect to the
output size. Also note that the relaxation introduces slightly higher arithmetic cost which is bounded by the optimized base case sizes. As a result, mini jumps can be seen between power-of-two jumps, which do not affect the overall smooth performance.

This section introduced new TFT algorithms for high performance TFT library generation. The general-radix TFT algorithm is devised that extends the fixed decomposition strategies to allow searching for optimal implementation in the entire implementation space. The parallel TFT algorithm achieves improved vectorization and multi-threading by introducing a small arithmetic overhead. The algorithms are expressed in a domain specific language and its extension, which enables automatic generation and optimization by the SPIRAL system. The effort results in a high-performance implementation of the TFT. The performance evaluation demonstrates the practical performance gains of TFT even when compared to an autotuned parallel modular DFT library.
5.3 ITFT

The inverse truncated Fourier transform (ITFT) is a key component in the fast polynomial and large integer algorithms introduced by van der Hoeven. The ITFT, with additional and irregular data passes and more complicated “cross butterflies”, poses additional challenges compared to the TFT. For obtaining high performance implementations, the complexity of the ITFT has limited implementations to fixed decomposition strategies and scalar codes.

This section reports a high performance implementation of the ITFT which poses additional challenges compared to that of the forward transform. A general-radix variant of the ITFT algorithm is developed to allow the implementation to automatically adapt to the memory hierarchy. Then a parallel ITFT algorithm is developed that trades off small arithmetic cost for full vectorization and improved multi-threaded parallelism. The algorithms are automatically generated and tuned to produce an arbitrary-size ITFT library. The new algorithms and the implementation smooths out the staircase performance associated with power-of-two modular FFT implementations, and provide significant performance improvement over zero-padding approaches even when high-performance FFT libraries are used.

5.3.1 Strict General-radix ITFT Algorithm

We begin by investigating the base cases of the ITFT. Then we derive a general-radix ITFT algorithm that enforces strict truncation. A complexity analysis shows that performance of the ITFT algorithm is relatively smooth with respect to the effective output size. At the end, we present the solutions to some practical challenges encountered during the fixed-size code generation.
ITFT Base Cases

Despite the similarity in their structures, the base cases of the ITFT are more involved than that of the modular DFT. The number of valid combinations of the ITFT parameters makes the ITFT base cases nontrivial. Next, we present the base cases as small matrices using the residue representation, as the Montgomery reduction is used thoroughly in the code generation. Regular base cases can be derived similarly.

Let \( \bar{i} = iR \mod p \) denote the residue representation of \( i \in \mathbb{Z} \) used in the Montgomery reduction, where \( R \) and \( p \) are the power-of-two constant and the modulus, respectively. Recall the size-2 base case of modular DFT matrix:

\[
\begin{bmatrix}
X_0 \\
X_1
\end{bmatrix} = \begin{bmatrix}
\bar{1} & \bar{1} \\
\bar{1} & \bar{-1}
\end{bmatrix} \cdot \begin{bmatrix}
x_0 \\
x_1
\end{bmatrix},
\]

(5.13)

where \( \bar{x}_i \) and \( \bar{X}_i \) are the input and output in the residue representation.

Let \( m \) be the output size, \( l \) be the input size, and \( cf \) be a cross flag that indicates whether a transformed value is computed (\( cf = 1 \)) or not (\( cf = 0 \)), then an ITFT of size 2 has seven base cases. Every ITFT base case is a variation of solving the two equations (5.14), (5.15) from the size-2 TFT with some variables known and some other variables to be computed.

\[
\begin{align*}
\bar{X}_0 &= \bar{x}_0 + \bar{x}_1 = \bar{x}_0 + \bar{x}_1, \\
\bar{X}_1 &= \bar{x}_0 - \bar{x}_1 = \bar{x}_0 - \bar{x}_1
\end{align*}
\]

(5.14) (5.15)

- \( m = 2, l = 2, cf = 0 \). In this case, \( \bar{X}_0 \) and \( \bar{X}_1 \) are known, and \( \bar{2x}_0 \) and \( \bar{2x}_1 \) are to be computed. The modular ITFT matrix is trivially identical to that of the modular DFT of size 2, which scalarizes the input vector with the dimension of
the transform when multiplied with TFT matrix of size 2:

\[
\begin{bmatrix}
1 & 1 \\
1 & -1
\end{bmatrix} \cdot \begin{bmatrix}
1 & 1 \\
1 & -1
\end{bmatrix} \cdot \begin{bmatrix}
x_0 \\
x_1
\end{bmatrix} = \begin{bmatrix}
2 & 0 \\
0 & 2
\end{bmatrix} \cdot \begin{bmatrix}
x_0 \\
x_1
\end{bmatrix} = \begin{bmatrix}
\frac{2x_0}{2x_1}
\end{bmatrix},
\]

where $\frac{2x_0}{2x_1}$ will be recovered at the end of the computation by multiplying with $R^{-1}$ to get $2x_0$ and $2x_1$, therefore completing the convolution.

- $m = 1, l = 2, cf = 1$. In this case, $\overline{X_0}$ and $\overline{2x_1}$ are known, and $\overline{2x_0}$ and $\overline{X_1}$ are to be computed. The base case matrix constructed by solving (5.14), (5.15):

\[
\begin{bmatrix}
\overline{2x_0} \\
\overline{X_1}
\end{bmatrix} = \begin{bmatrix}
\frac{2}{2} & 1 \\
1 & -1
\end{bmatrix} \cdot \begin{bmatrix}
\overline{X_0} \\
\overline{2x_1}
\end{bmatrix}.
\] (5.16)

From (5.14), $\overline{2x_0} = \overline{2X_0} - \overline{2x_1} = [\overline{2}, -1] \cdot [\overline{X_0}, \overline{2x_1}]^T$. From (5.15) - (5.14), $\overline{X_1} = \overline{X_0} - \overline{2x_1} = [1, -1] \cdot [\overline{X_0}, \overline{2x_1}]^T$.

- $m = 1, l = 1, cf = 1$. In this case, $\overline{X_0}$ is known ($\overline{x_1}$ is known to be 0). $\overline{2x_0}$ and $\overline{X_1}$ are to be computed. The base case matrix is obtained by dropping the second column of the matrix in (5.16):

\[
\begin{bmatrix}
\overline{2x_0} \\
\overline{X_1}
\end{bmatrix} = \begin{bmatrix}
\frac{2}{2} \\
1
\end{bmatrix} \cdot \begin{bmatrix}
\overline{X_0}
\end{bmatrix}.
\] (5.17)

- $m = 1, l = 2, cf = 0$. In this case, $\overline{X_0}$ and $\overline{2x_1}$ are known. $\overline{2x_0}$ is to be computed. The base case matrix is obtained by dropping the second row of the
matrix in (5.16):

\[
\begin{bmatrix}
\frac{2x_0}{2x_1}
\end{bmatrix}
= \begin{bmatrix} 2 & -1 \end{bmatrix} \cdot \begin{bmatrix} X_0 \end{bmatrix}.
\]  
\tag{5.18}

• \(m = 1, l = 1, cf = 0\). In this case, \(X_0\) is known, and \(\frac{2x_0}{2x_1}\) is to be computed. The base case matrix is obtained by dropping the second column and the second row of the matrix in (5.16):

\[
\begin{bmatrix}
\frac{2x_0}{2x_1}
\end{bmatrix}
= \begin{bmatrix} 2 \end{bmatrix} \cdot \begin{bmatrix} X_0 \end{bmatrix}.
\]  
\tag{5.19}

• \(m = 0, l = 2, cf = 1\). In this case, \(\frac{2x_0}{2x_1}\) are known. \(X_0\) is to be computed. The base case matrix is constructed based on (5.14):

\[
\begin{bmatrix}
X_0
\end{bmatrix}
= \begin{bmatrix} 2^{-1} & 2^{-1} \end{bmatrix} \cdot \begin{bmatrix} \frac{2x_0}{2x_1} \end{bmatrix}.
\]  
\tag{5.20}

• \(m = 0, l = 1, cf = 1\). In this case, \(\frac{2x_0}{2x_1}\) is known, and \(X_0\) is to be computed. The base case matrix is obtained by dropping the second column of the matrix in (5.20):

\[
\begin{bmatrix}
X_0
\end{bmatrix}
= \begin{bmatrix} 2^{-1} \end{bmatrix} \cdot \begin{bmatrix} \frac{2x_0}{2x_1} \end{bmatrix}.
\]  
\tag{5.21}

The increased number of base cases leads to two changes compared to the modular FFT: 1. the size of the generated library increases, as more base cases as well as their strided and vectorized variations are included for the recursive steps, and 2. new optimization strategies must be developed to properly handle the new residue constants in the base cases, such as \(2\) and \(2^{-1}\), which can otherwise lead to stiff
computational penalties. The first change is discussed in Chapter 7 and the second change is addressed in later in this section.

A Strict General-radix Algorithm for ITFT

The ITFT introduced in [46] follows the classic radix-2 divide-and-conquer FFT paradigm, which recursively splits the transform into two half-sized transforms. [20] shows that such paradigm has suboptimal locality, and introduces a cache-friendly ITFT algorithm, which recursively breaks down the transform into row and column transforms in a balanced style, akin to Bailey’s algorithm [2].

Extensive examples [43, 14] have shown that a general Cooley-Tukey type algorithm that allows arbitrary decomposition, usually accompanied with a search mechanism, can generate optimal breakdown strategy for any given problem size on various computing platforms. We devise a general-radix ITFT algorithm with strict truncation to achieve the reduced operation count. The new algorithm extends the existing algorithms from fixed decomposition strategies to arbitrary decomposition.

As a result of the truncation, the TFT cannot be easily inverted. The last row of the ITFT usually lacks sufficient information to be performed together with the other row transforms uniformly, until several rightmost column transforms produce the missing values. Then the rest of the column transforms are performed after the last row transform. The alternation between the rows and columns of the ITFTs makes the algorithm more complicated.

Denote an ITFT as $\text{ITFT}_{n,l,m,f}$, where $n$ is the size of the transform, $1 \leq m \leq n$ is the number of untransformed values to be computed, $0 \leq m \leq l \leq n$ is the size of the input values consisting of the transformed values and the complimentary scaled untransformed values, and $f \in \{0,1\}$ is a flag for whether the last untransformed value needs to be computed.
Let \( n = rs \), \( l_c = \lfloor l/r \rfloor \), \( l_{c+1} = \lfloor l/r \rfloor \), \( m_e = \lfloor m/r \rfloor \), \( m_{e+1} = \lfloor m/r \rfloor \), \( l_r = \min(l, r) \), \( \bar{m} = m \mod r \), \( \bar{l} = l \mod r \), \( c_1 = \max(\bar{m}, \bar{l}) - \bar{m} \), \( c_2 = l_r - \max(\bar{m}, \bar{l}) \), \( c_3 = \min(\bar{m}, \bar{l}) \), \( c_4 = \bar{m} - c_3 \), \( r_1 = \lfloor m/r \rfloor \), \( r_2 = f' \) and if \((\bar{m} + f) > 0 \) then \( f' = 1 \) else \( f' = 0 \), then the two row transforms and four column transforms generated by decomposing \( ITFT_n \) are:

\[
\begin{align*}
\text{ITFT}_{r_1} &= \text{ITFT}_{r,r,r,0}, \quad \text{ITFT}_{r_2} = \text{ITFT}_{r,l,r,f}, \\
\text{ITFT}_{c_1} &= \text{ITFT}_{s,l,c_{e+1},m,f'}, \quad \text{ITFT}_{c_2} = \text{ITFT}_{s,l,m_{e+1},f'}; \\
\text{ITFT}_{c_3} &= \text{ITFT}_{s,l,c_{e+1},m_{e+1},0}, \quad \text{ITFT}_{c_4} = \text{ITFT}_{s,l,m_{e+1},0}.
\end{align*}
\]

The algorithm expressed in \( \sum \)-SPL decomposes \( ITFT_{n,l,m,f} \) as:

\[
\begin{align*}
\left( \sum_{c_3} \left( S_{c_3} \text{ITFT}_{c_3} G_{c_3} \right) + \sum_{c_4} \left( S_{c_4} \text{ITFT}_{c_4} G_{c_4} \right) \right) \cdot (5.22) \\
\sum_{r_2} \left( S_{r_2} T_{r_2}^m \cdot \text{ITFT}_{r_2} G_{r_2} \right) \cdot (5.23) \\
\left( \sum_{c_1} \left( S_{c_1} \text{ITFT}_{c_1} G_{c_1} \right) + \sum_{c_2} \left( S_{c_2} \text{ITFT}_{c_2} G_{c_2} \right) \right) \cdot (5.24) \\
T_{r_1}^m \cdot \sum_{r_1} \left( S_{r_1} \text{ITFT}_{r_1} G_{r_1} \right), \quad (5.25)
\end{align*}
\]

where \( T_{r_1}^m \) and \( T_{r_1}^m \) represent lower and upper partial twiddle matrices, respectively, and \( G \) and \( S \) in each \( \sum \) are the gather and scatter decorations whose details are
shown as follows.

\[
S_{\tilde{c}_1} = S(h_{m+i,r}^{\tilde{c}_1m \rightarrow m+f'}) \\
S_{\tilde{c}_2} = S(h_{m+i,r}^{c_1m \rightarrow m+f'}), \\
S_{\tilde{c}_3} = S(h_{i,r}^{m+f \rightarrow m+c+1}), \\
S_{\tilde{c}_4} = S(h_{i,r}^{m+f \rightarrow m+c+1}), \\
S_{r_1} = S(h_{i,r,1}^{n-\min(r,l)}), \\
S_{r_2} = S(h_{i-m,c,1}^{r_2m \rightarrow \min(m+f,r)}), \\
G_{\tilde{c}_1} = G(h_{m+i,r}^{\tilde{c}_1m+i}), \\
G_{\tilde{c}_2} = G(h_{m+i,r}^{c_1m+i}), \\
G_{\tilde{c}_3} = G(h_{i,r}^{n-\min(c,2)}), \\
G_{\tilde{c}_4} = G(h_{i,r}^{n-\min(r,l)}), \\
G_{r_1} = G(h_{i,r,1}^{n-\min(r,l)}), \\
G_{r_2} = G(h_{i-r,1}^{r_2m \rightarrow \min(r,l)}),
\]

where \( i \) in each decoration matrix is the loop index from the enclosing iterative sum \( \sum \), respectively, \( \tilde{c}_1m = m + f \) if \( r_2 = 0 \), \( n \) otherwise, \( \tilde{c}_1l = l \) if \( r_1 = 0 \), \( n \) otherwise, \( \tilde{r}_2m = m + f \) if \( c_3 = 0 \) and \( c_4 = 0 \), \( n \) otherwise, \( \tilde{r}_2l = l \) if \( c_1 = 0 \), \( c_2 = 0 \) and \( r_1 = 0 \), \( n \) otherwise.

The algorithm is applied to the input vector from right to left. An example of decomposing ITFT_{32,22,19} is illustrated in Fig.5.6: in frame (a), the first 4 rows with complete information are performed followed by the partial twiddle matrix, as in line 5; in frame (b), the leftmost column transform of type ITFT_{\tilde{c}_2} is performed, as in line 4; in frame (c), the last row and the lower partial twiddle matrix are performed, as in line 3; and in frame (d), two types of column transforms ITFT_{\tilde{c}_3} and ITFT_{\tilde{c}_4} are performed as non-overlapping matrix sum defined in line 2.

**Complexity Analysis.** Let \( \tilde{n} = \log_2(n) \), van der Hoeven showed that the computational complexity of the TFT and ITFT is bounded by \( \tilde{n}m/2 + n \) butterfly operations. Then Harvey proved that the cache-friendly TFT and ITFT follow the same estimates. The complexity analysis of the general-radix ITFT algorithm is similar to the analysis presented in [20] and extends to general-radix decomposition. The following theorem indicates that the running time of the TFT, ITFT and TFT-based
multiplication is relatively smooth as a function of $m + f$, the size of the truncated output.

**Theorem 3.** The number of executions of the ITFT base case is bounded by $\min((m+f-1)\tilde{n}/2 + n - 1, n\tilde{n}/2)$.

Proof. For the base case $n = 2$, the number of calls to base case is trivially $\min((m+f-1)/2 + 1, 2 \cdot 1/2) = 1$, thus the bound holds. For the general case $n \geq 4$, let $\tilde{r} = \log_2 r$ and $\tilde{s} = \log_2 s$, the two bounds are proved by induction respectively.

We first prove the bound $n\tilde{n}/2$: (1) Column transforms: The size of the four column transforms $ITFT_{\tilde{r}_1} - ITFT_{\tilde{r}_4}$ is $s$, and by induction the number of calls to base case is bounded by $\sum_{i=1}^{4} c_i s\tilde{s}/2 \leq rs\tilde{s}/2 = n\tilde{s}/2$. (2) Row transforms: The number of calls invoked by the row transforms $ITFT_{\tilde{r}_1}$ and $ITFT_{\tilde{r}_2}$ is at most $sr\tilde{r}/2 = n\tilde{r}/2$. Therefore the sum of the calls by all transforms is bounded by $n\tilde{s}/2 + n\tilde{r}/2 = n\tilde{n}/2$. Thus the first bound is proved.

The second bound $(m+f-1)\tilde{n}/2 + n - 1$ can be proved by induction on the row and column transforms as well: (1) Top row transforms: The full transforms $ITFT_{\tilde{r}_1}$ by induction call the ITFT base case at most $c_5(r\tilde{r}/2) = m_c(r\tilde{r}/2)$ times. (2) Bottom row transforms: The calls made by $ITFT_{\tilde{r}_2}$ by induction is bounded
by \( c_0(\bar{m} + f - 1)\bar{r}/2 + r - 1 = f'(\bar{m} + f - 1)\bar{r}/2 + r - 1 \). (3) Right column transforms: The number of calls invoked by ITFT\( \hat{c}_1 \) and ITFT\( \hat{c}_2 \) is bounded by 
\[
(c_3 + c_4)(m_c + f' - 1)\bar{s}/2 + s - 1 = (l_r - \bar{m})(m_c + f' - 1)\bar{s}/2 + s - 1.
\]
(4) Left column transforms: ITFT\( \hat{c}_3 \) and ITFT\( \hat{c}_4 \) call the base case at most \((c_1 + c_2)(m_{c+1} - 1)\bar{s}/2 + s - 1 = \bar{m}(m_{c+1} - 1)\bar{s}/2 + s - 1\).

Before summing all the calls to the base case, we show that the computation of the calls from the column transforms can be simplified. If \( \bar{m} = m \mod r = 0 \), then 
\[
m_c(r\bar{r}/2) + f'(\bar{m} + f - 1)\bar{r}/2 + r - 1 + l_r((m_c + f' - 1)\bar{s}/2 + s - 1)
\]
\[
= \frac{1}{2}(m_c r\bar{r} + f'(\bar{m} + f - 1)\bar{r} + r\bar{s}m_c + r(f' - 1)\bar{s})
\]
\[
+ (f'(r - 1) + (l_r - \bar{m})(s - 1) + \bar{m}(s - 1))
\]
\[
= \frac{1}{2}(m_c r\bar{r} + f'(\bar{m} + f - 1) + r(f' - 1)\bar{s})) + (f'(r - 1) + l_r(s - 1))
\]
\[
= \frac{1}{2}((m - \bar{m})\bar{n} + f'(\bar{m} + f - 1) + r(f' - 1)\bar{s})) + (n - 1 + (f' - 1)(r - 1))
\]
\[
= \frac{1}{2}((m + f - 1)\bar{n} + (\bar{m} + f - 1)(f'\bar{r} - \bar{n}) + r\bar{s}(f' - 1)) + (n - 1 + (f' - 1)(r - 1))
\]
\[
= \frac{1}{2}(m + f - 1)\bar{n} + n - 1 + r\bar{s}(f' - 1)) + (f' - 1)(r - 1) + \frac{1}{2}(\bar{m} + f - 1)(f'\bar{r} - \bar{n}).
\]

Notice that the underlined terms are in the form of the desired bound. Let \( \varepsilon \) be the rest of the terms in the sum. If \( f' = 1 \), then \( \bar{m} + f > 0 \), hence \( \varepsilon = -\bar{s}(\bar{m} + f - 1)/2 \leq 0 \). If \( f' = 0 \), then \( \bar{m} + f = 0 \), hence \( \varepsilon = (\bar{n} - r\bar{s})/2 - (r - 1) \leq -(r - 1) < 0 \). In summary, the truncated output-based bound holds.
Theorem 3 indicates that the running time of the general-radix ITFT algorithm is also relatively smooth as a function of the effective output size $m + f$, despite the increased complexity.

The strict general-radix algorithm is used to generate fixed-size base cases in the resulting library, since its strict truncation maintains the optimal computation count. The base cases are generated using a single basic block of straight-line code. Inside the basic block, the SPIRAL system can apply constant propagation and dead code elimination to obtain an optimized TFT implementation that does not perform unnecessary operations. The arithmetic is performed over finite field using the Montgomery reduction.

**Fixed-size Code Generation**

The unique properties of the ITFT and its general-radix algorithm pose new challenges to the fixed-size code generation in the SPIRAL system, including mainly the data continuity issue and the efficient rewriting strategies. We describe our solutions to these practical questions, including breaking the existing assumptions and devising new optimizations.

**Shadow Structures.** (5.22)-(5.25) each requires a partial traversal on the input data, which poses two unique challenges: (1) the extra traversals compared to the regular DFT deteriorate cache locality and increase loop overhead, and (2) the partial traversals may drop some intermediate results that are needed in later steps but not the immediate next step(s), as the gathers and scatters in each step only concern the partial data segments to be operated by the kernel of the step. The first challenge is inherently inevitable when the strict truncations are enforced, and will be addressed in the next section with the introduction of the parallel TFT and ITFT algorithms. We will address the second challenge here as we introduce the shadow structures.
Each $\Sigma$-SPL structure composed together as a chain of computation assumes that the immediate previous structure provides a valid and dimension-agreeing domain of the intermediate result, and in turn produces a valid range for the immediate following structure as its own domain. This domain-range chaining mechanism works fine for regular transforms, as all $\Sigma$-SPL structures operate over the same domain and range, i.e., the entire input vector or the intermediate copies of it depending on whether the library performs inplace computation.

With the truncation on the input and output vectors, it becomes insufficient to only consider the computational structures, as they only operate on partial sections of the intermediate vectors. To circumvent this problem, we propose an expansion of the fixed-size truncated algorithms with shadow structures in order to preserve the completeness and continuity of the intermediate results.

Figure 5.7 shows an instance of the ITFT decomposition where the shadow structures must be added to avoid data loss. The algorithm execution starts from the leftmost column (0) as the input, and finishes at the rightmost column (5) as the output. The figure shows the domain and range of each data traversal in the columns (1)-(4), corresponding to the steps (5)-(2) in the ITFT algorithm, respectively. To compute $\text{ITFT}_{8,7,5,0}$, column (1) gathers $x[0], \ldots, x[3]$ to its computational kernel, and scatters the computed results to $b_1[0], \ldots, b_1[3]$, where $b_1$ is a temporary buffer. Then column (2) performs two regular size-2 column transforms at $(b_1[1], b_1[5])$ and $(b_1[2], b_1[6])$ and writes to $(b_2[1], b_2[5])$ and $(b_2[2], b_2[6])$, and a cross butterfly from $b_1[3]$ to $(b_2[3], b_2[7])$, where $b_2$ is another temporary buffer. When column (2) gathers $b_1[5]$ and $b_1[6]$, an invalid index error occurs, as column (1) only partially generated $b_1[0], \ldots, b_1[3]$. The same type of errors repeat as the algorithm execution proceeds.

The shadow structures are parameterized gather and scatter matrices that fill up the gaps resulted from the partial data traversals. Note that the shadow structures
Figure 5.7: An instance of the partial data coverage in each step of computing $ITFT_{8,7,5,0}$. The lines from northwest to southeast represent the input domain, and the lines from northeast to southwest represent the output range.

do not generate any arithmetic operations or increase the computational complexity, as they fix the missing links between the temporary variables from nonadjacent code areas by only inserting some new temporary variable assignments in the bridging code area. Then the new temporary assignments will be optimized out as the fixed-size codes are fully unrolled and optimization routines like copy propagation and dead code elimination are performed.

**New rewriting strategies.** The residue multiplicative identity $\bar{1}$ is analogous to the regular multiplicative identity, as $REDC(\bar{1} \cdot \bar{x}) = 1 \cdot R \cdot R \cdot x \cdot R^{-1} \mod p = x \cdot R \mod p = \bar{x}$. In SPIRAL, regular arithmetic operations in the intermediate representation (IR) are rewritten into corresponding modular arithmetic by the IR rewriting system. The modular multiplication rewriting strategy employs the Montgomery reduction to avoid the expensive integer division. The strategy identifies the residue
multiplicative identity and its negation, and avoids generating the whole instruction sequence of the Montgomery reduction.

Section 5.3.1 shows that $\overline{2}$ and $2^{-1}$ are among the residue constants in the ITFT base cases. They can also be optimized to avoid the full Montgomery reduction sequence. For multiplying with $\overline{2}$, $mul(\overline{2}, \overline{x})$ is rewritten as a modular addition $add(\overline{x}, \overline{x})$. For multiplying with $2^{-1}$, $mul(2^{-1}, \overline{x})$ returns $\overline{x}/2$, which can be optimized as at most two bit-wise operations, an addition and a regular multiplication. If $2|\overline{x}$, then $\overline{x}/2 \equiv \overline{x} \gg 1 \mod P$, where $\gg$ is the bitwise right shift; otherwise for $2 \nmid \overline{x}$, $\overline{x}/2 \equiv (\overline{x} + P) \gg 1$. A combined compact formula for the rewriting trick is $\overline{x}/2 = (\overline{x} + p \cdot (\overline{x} \& 1)) \gg 1$, where $\&$ is the bitwise and operation.

**TFT/ITFT co-generation.** The TFT and ITFT definitions exclude the permutations to recover the order of the output. Therefore, when general-radix decomposition is used, it is important to ensure that the TFT and ITFT of the same configuration use the symmetric decomposition trees, so that the correct order can be recovered after the convolution. This problem is addressed in Section 5.3.2 together with the co-planning at the library level.

### 5.3.2 Parallel ITFT Algorithm

So far the goal of the previous work has been focusing on minimizing the operation count to the extent possible, albeit at the cost of increasing the implementation complexity. Section 5.3.1 clearly shows that the additional and irregular data passes in the ITFT prohibit effective vectorization and parallelization.

To overcome the limit in ITFT parallelization, we propose a parallel algorithm that uses a small relaxation which trades off slightly higher arithmetic cost for simplified ITFT, improved data flow, which allows full vectorization and improved multi-threaded parallelism. The new algorithm is expressed in SPL, and can be derived and
tuned using the SPIRAL system for code optimization and generation.

Let $N = 2^n = rs$ be the size of the transform, $B = 2^b$ be the size of the base case transform generated by the strict general-radix algorithm in Section 5.3.1. Let $L$ and $M$ be the input and output sizes, respectively. If $N < B^2$, then $r = N/B$ and $s = B$. Let $l_s = \lceil LB/N \rceil \cdot 2^{\max(\lceil n/2 \rceil, b) - b}$, $m_s = \lceil MB/N \rceil \cdot 2^{\max(\lceil n/2 \rceil, b) - b}$, $l_r = \min(L, r)$, $m_r = \min(M, r)$, then the row and column transforms are

$$\text{ITFT}_r = \text{ITFT}_{r, l_r, m_r}, \quad (5.26)$$

$$\text{ITFT}_c = \text{ITFT}_{s, l_s, m_s}, \quad (5.27)$$

and the relaxed parallel Cooley-Tukey algorithm for ITFT is:

$$\text{ITFT}_{N, L, M} = (\text{ITFT}_c \otimes I_{l_r}) \cdot T_{r,s}^N \cdot (I_{m_s} \otimes \text{ITFT}_r). \quad (5.28)$$

The relaxation happens when $r \nmid l$. As a result, all column transforms and all row transforms become the uniform, respectively, enabling the expression of the algorithm in SPL. More importantly, the “cross butterfly” can be completely eliminated if the ITFT follows a mirrored decomposition tree of the forward transform, because of the accumulated relaxation in the TFT. Also note that, compared to 5.1, 5.28 does not perform explicit permutation $L$ during the recursive factorization. As a result, the untransformed values are permuted based on the specific factorization path at runtime. Therefore, a co-planning mechanism must be developed to ensure that the generated TFT and ITFT libraries follow the symmetric recursive step call graph when the configuration is the same, in order to effectively cancel the permutations in the convolution.

Figure 5.8 shows an example of the relaxation. Assuming the base case size is 4, frame (a) shows the relaxation as a that leads to the uniform row and column trans-
forms. Frame (b) shows the right tensor product in (5.12) containing the ITFT being applied to the relaxed rows; Frame (c) shows the left tensor product containing ITFT being applied to the relaxed columns. The twiddle factor $T_{r,s}^N$ is fused into adjacent loops as the SPL expression is rewritten and optimized as $\sum$-SPL expressions.

**Complexity analysis.** We show that the truncation can be preserved as the relaxed decomposition of ITFT reaches a base case via an arbitrary decomposition path, similar to Theorem 2 for TFT.

**Theorem 4.** Let $N$ be the transform size of a given ITFT, $L$ be the size of the truncated input, and $B$ be the size of the base case chosen by the search engine, then the truncated input to the base case column transforms in the relaxed general ITFT algorithm is bounded by $\lceil LB/N \rceil$.

*Proof.* Let $T = N/B$, and $T = t_1t_2\ldots t_k$ be an arbitrary factorization. Then the arbitrary factorization of $N$ introduces a relaxed input $l_B$ when the base case $B$ is
reached as column transform:

\[ l_B = \left\lceil \frac{L}{t_1 t_2 \ldots t_k} \right\rceil \]

\[ = \left\lceil \frac{L}{t_1 t_2 \ldots t_k} \right\rceil \quad (5.29) \]

\[ \quad = \left\lceil \frac{L}{t_1 t_2 \ldots t_k} \right\rceil \quad (5.30) \]

\[ = \left\lceil \frac{LB}{N} \right\rceil \quad (5.31) \]

To prove (5.29) → (5.30) is equivalent to proving \( \left\lceil \left\lceil \frac{L}{t_1} \right\rceil \right\rceil = \left\lceil \frac{L}{t_1 t_j} \right\rceil \). We will prove this by contradiction below. Let \( q = \left\lceil \frac{L}{t_i} \right\rceil \), by the definition of the ceiling operation, we have

\[ q - 1 < \frac{L}{t_i} \leq q \]

\[ \Rightarrow \frac{q - 1}{t_j} < \frac{L}{t_i t_j} < \frac{q}{t_j} \]

\[ \Rightarrow \left\lceil \frac{L}{t_i t_j} \right\rceil \leq \left\lceil \frac{q}{t_j} \right\rceil \quad (5.32) \]

Assume in (5.32) that \( \left\lceil \frac{L}{t_i t_j} \right\rceil < \left\lceil \frac{q}{t_j} \right\rceil \), then \( \exists k \in \mathbb{Z} \) s.t.

\[ \frac{L}{t_i t_j} \leq k < \frac{q}{t_j} \]

\[ \Rightarrow \frac{L}{t_i} \leq t_j k < q \]

But \( q = \left\lceil \frac{L}{t_i} \right\rceil \), which leads to a contradiction of \( q \leq t_j k < q \). Therefore, \( \left\lceil \frac{L}{t_i t_j} \right\rceil \) must be equal to \( \left\lceil \frac{q}{t_j} \right\rceil \) in (5.32), i.e. \( \left\lceil \frac{L}{t_j} \right\rceil = \left\lceil \frac{L}{t_i t_j} \right\rceil \), and therefore, \( \left\lceil \left\lceil \frac{L}{t_1} \right\rceil / t_2 \right\rceil / \ldots / t_k \right\rceil = \left\lceil \frac{L}{t_1 t_2 \ldots t_k} \right\rceil \)

Similarly, the bound of output size of the base case transform can be proved to be \( \left\lceil MB/N \right\rceil \). While the worst case of the arithmetic overhead introduced by the relaxation is pessimistic, the theorem shows that optimal arithmetic cost is determined by the available fixed-size base cases.
Parallel Library Generation

**TFT and ITFT co-generation/co-planning.** Let $S = \{TFT, ITFT\}$, both the fixed-size code generation and library planning of $t \in S$ must be *guided* by the decomposition paths of $S \setminus t$. The forward and inverse modular DFT are special cases of the TFT and ITFT whose symmetric decompositions are therefore also enforced.

The structural similarity between the algorithms of the TFT and ITFT makes the co-generation and co-planning feasible. Figure 5.9 illustrates the framework: if $t \in S$ is generated or planned, the breakdown trees or the recursive step call graphs are stored in the hash table or the DP knowledge files. The files can be used as vehicles to ensure the symmetry between the TFT and ITFT.

For the fixed-size code generation, a new parameter $<dtree>$is added to the non-terminals as a means of storing the decomposition trees and triggering the guided algorithms. When the co-generation is required, $<dtree>$reads the optimal decomposition tree from the guiding hash table, which is used to direct the generation of $S \setminus t$. The library-level co-planning uses pattern matching scripts to produce a symmetric DP knowledge file based on an existing one.

**Parameter computation optimization.** In the code generation, the *ceiling* operations $\lceil \rceil$ in $l_s$ and $m_s$ can be rewritten into bitwise operations to avoid the
external function calls to \( \text{ceil}() \) in the generated C/C++ libraries. Let \( l_1 = \lceil LB/N \rceil \), since \( L \) and \( N/B \) are both positive and their sum is free of integer overflow per the definitions, then \( l_1 = \lceil L/(N/B) \rceil = \lceil L/(1 \ll (n - b)) \rceil = (L - 1 + 1 \ll (n - b))/(1 \ll (n - b)) = (L - 1 + 1 \ll (n - b)) \gg (n - b) \), where \( \ll \) and \( \gg \) are the bitwise shift left and right, respectively. Then \( l_s = l_1 \cdot 2^{\max([n/2], b) - b} = l_1 \ll (\max([n/2], b) - b) = ((L - 1 + 1 \ll (n - b)) \gg (n - b)) \ll \max((n + 1) \gg 1 - b, 0) \).

Similarly, \( m_s = ((M - 1 + 1 \ll (n - b)) \gg (n - b)) \ll \max((n + 1) \gg 1 - b, 0) \). Note that the \( \ll \) and \( \gg \) in the final forms of \( l_s \) and \( m_s \) can not be merged into \( (n - b - \max([n/2] - b, 0)) = \min([n/2], n - b) \), since the ceiling relaxation of the numerator in \( l_1 \) from \( L \) to \( L - 1 + 1 \ll (n - b) \) obtains its correctness only with the denominator \( 1 \ll (n - b) \).

### 5.3.3 Performance Evaluation

This section reports the performance data comparing the ITFT library and inverse modular FFT library, both of which are automatically optimized and generated by SPIRAL. The performance is reported in cycles measured by PAPI \[40\] and is averaged over 1000 runs with small observed performance variance.

**Baseline implementation.** An SPIRAL-generated parallel inverse modular DFT (ModDFT) library is used as the baseline implementation. The reference library pads input to the next power of two, and fully utilizes vector registers and multi-core. It has been shown in \[35\] that its performance is comparable to optimized fixed-size codes \[36\], and gains an order-of-magnitude speed-up over hand-optimized library \[31\]. The ModDFT library is represented by the solid line in Figure 5.10 which exhibits clear jumps in running time when the lengths cross a power-of-two boundary.

**Autotuned ITFT library.** The TFT library employs the strict general-radix algorithm for base case generation to reduce arithmetic cost. The built-in search engine
uses the DP technique that measures the actual running time of smaller transforms as the input to the feedback loop to guide the generation of the base case sizes up to 8. For the library-level recursive breakdown, it applies the relaxed parallel algorithm, which trades off a slightly higher arithmetic cost for vectorization and parallelization.

**Speedup.** The 2-thread ITFT library delivers a speedup of 38% − 41% over the 2-thread high performance inverse ModDFT library with length that just crosses a power-of-two boundary. As the length increases, the gap between the truncated transform and the full transform decreases. The 4-thread ITFT library is faster than the 2-thread ITFT library by a factor of almost two, showing an near linear speedup as the available hardware threads are fully utilized.

Overall, the ITFT library’s performance is smooth with respect to the input size. Also note that, as we proved in Section 5.3.2, the relaxation introduces slightly higher arithmetic cost which is bounded by the optimized base case sizes. As a result, mini jumps can be seen between power-of-two jumps, which do not affect the overall smooth performance.

### 5.4 Conclusion

We presented new general-radix and parallel algorithms for the TFT and ITFT which improve on both the theoretical complexity and practical performance. The general-radix algorithms used in the base case generation expand the scalar implementation space in order to search for the optimal implementation that adapts to memory hierarchy. The parallel algorithms for the TFT and ITFT trade off small arithmetic cost for full vectorization and improved multi-threaded parallelism.

We used $\sum$-SPL and SPL to express the new algorithms. SPIRAL is extended and enhanced to support the automatic generation and optimization for the TFT libraries. The autotuned libraries smooth out the staircase phenomenon associated
Figure 5.10: Performance comparison between the SPIRAL-generated parallel ITFT library and the SPIRAL-generated parallel inverse modular DFT library that pads to powers of two with the power-of-two modular FFT implementations, and provide significant performance improvement over the zero-padding approach even when high-performance FFT libraries are used.
6. Modular Polynomial Multiplication

In this chapter, we build upon the performance gains from previous chapters, including the improved modular arithmetic and the parallel modular DFT and TFT, to obtain a highly optimized library for modular polynomial multiplication. The asymptotically fast multiplication algorithms utilize the autotuned linear transforms, applying the forward transform as an evaluation of the unknown product polynomial at a number of points and an inverse FFT as an interpolation to compute the coefficients of the product polynomial.

We first review the background, including the definitions and the Convolution Theorem. Then we introduce the fast algorithms based on different transforms for modular polynomial multiplication. Necessary extensions in SPIRAL are also explained as the library generation becomes more involved. Finally, we evaluate the performance of the modular DFT based library and the TFT-based library to show the practical performance gains from the new TFT algorithms over the power of two modular FFT algorithm.

6.1 Background

In this section, we review the polynomial-related definitions. Then we introduce the convolution theorem which states that the Fourier transforms of a convolution is the pointwise product of Fourier transform. The convolution theorem and the FFT or its variations can reduce the complexity of the convolution from quadratic to $O(n \log n)$, which serves as the foundation of the fast FFT-based multiplication algorithms.

Definition 11. Let $R$ be a commutative ring, such as $\mathbb{Z}_p$, a polynomial $a \in R[x]$ in
\(x\) is a finite sequence \((a_0, \ldots, a_n)\) of elements of \(R\) (the coefficients of \(a\)), for some \(n \in \mathbb{N}\), and we write it as

\[
a = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 = \sum_{0 \leq i \leq n} a_i x^i. \tag{6.1}
\]

For dense polynomials, we can represent \(a\) by an array where the \(i\)th element is \(a_i\). This assumes that we already have a way of representing coefficients from \(R\). The length of this representation is \(n + 1\). The representation of sparse polynomials is also well studied but beyond the scope of this thesis.

**Definition 12.** Let \(a = \sum_{0 \leq i \leq n} a_i x^i\) and \(b = \sum_{0 \leq i \leq m} b_i x^i\), the polynomial product \(c = a \cdot b\) is defined as \(\sum_{0 \leq k \leq m+n} c_k x^k\), where the coefficients are

\[
c_k = \sum_{0 \leq i \leq n, 0 \leq j \leq m \atop i + j = k} a_i b_j, \quad \text{for } 0 \leq k \leq m+n. \tag{6.2}
\]

The naive implementation of polynomial multiplication has an \(O(n^2)\) complexity. The Karatsuba algorithm [27] reduces the cost to \(O(n^{1.59})\). The FFT introduced earlier can be used in convolutions to further obtain fast algorithms with a complexity of \(O(n \log n)\).

The convolution is at the core of our modular polynomial multiplication library. Beyond polynomial arithmetic, the convolution also has applications in signal processing, efficient computation of large integer multiplication, and prime length Fourier transforms. Next, we present the definitions of linear and circular convolutions, and interpret them from three different perspectives to show the connection between convolutions and the polynomial multiplication.

Both linear and circular convolutions can be viewed as: (1) a sum, (2) a polynomial product, and (3) a matrix operation. As a result, polynomial algebra can be used to
derive algorithms and the corresponding matrix algebra can be used to manipulate and implement algorithms.

**Definition 13.** Let \( u = (u_0, \ldots, u_{M-1}) \) and \( v = (v_0, \ldots, v_{N-1}) \). The \( i^{th} \) component of the linear convolution \( u * v \) is defined as

\[
(u * v)_i = \sum_{k=0}^{N-1} u_{i-k}v_k, \quad 0 \leq i < M + N \tag{6.3}
\]

If \( u \) and \( v \) are viewed as the coefficient vectors of polynomials, i.e.,

\[
u(x) = \sum_{i=0}^{M-1} u_i x^i, \quad v(x) = \sum_{j=0}^{N-1} v_j x^j, \tag{6.4}\]

then the linear convolution \( u * v \) is equivalent to the polynomial multiplication of \( u(x)v(x) \) as defined in (6.2).

The sum form of linear convolution is also equivalent to the following matrix vector multiplication.

\[
u * v = \begin{bmatrix}
u_0 \\
\vdots \\
u_0 \end{bmatrix} \cdot \begin{bmatrix}
u_0 \\
\vdots \\
u_0 \end{bmatrix} = \begin{bmatrix}
u_1 & \cdots & \nu_0 \\
\vdots & \ddots & \vdots \\
\nu_{M-1} & \cdots & \nu_0 \\
\nu_{M-1} & \cdots & \nu_0 \\
\nu_{M-1} & \cdots & \nu_0 \\
\nu_{M-1} & \cdots & \nu_0 \\
\end{bmatrix} \cdot \begin{bmatrix}
u_0 \\
\vdots \\
u_0 \end{bmatrix} \tag{6.5}
\]

The circular convolution of two vectors of size \( N \) is obtained from linear convolution by reducing \( i - k \) and \( k \) in (6.3) modulo \( N \).

**Definition 14.** Let \( u = (u_0, \ldots, u_{N-1}) \) and \( v = (v_0, \ldots, v_{N-1}) \). The \( i^{th} \) component
of the circular convolution $u \circledast v$ is defined as

$$
(u \circledast v)_i = \sum_{k=0}^{N-1} u_k v_{(i-k) \mod N}, \quad 0 \leq i < N.
$$

(6.6)

Similar to the polynomial and matrix perspectives of linear convolution, the circular convolution can be obtained by multiplying polynomials $u(x)$ and $v(x)$ and taking the remainder modulo $x^N - 1$. In terms of matrix algebra, circular convolution can be interpreted as the product of a \textit{circulant matrix} $\text{Circ}_N(u)$ times $v$, where all columns of the matrix are obtained by cyclically rotating the first column.

$$
\begin{bmatrix}
    u_0 & u_{N-1} & u_{N-2} & \cdots & u_1 \\
    u_1 & u_0 & u_{N-1} & \cdots & u_2 \\
    \vdots & \ddots & \ddots & \ddots & \vdots \\
    u_{N-2} & \cdots & u_1 & u_0 & u_{N-1} \\
    u_{N-1} & u_{N-2} & \cdots & u_1 & u_0 \\
\end{bmatrix}
\cdot v
$$

(6.7)

The convolution theorem introduced next leads to fast convolution algorithms based on the aforementioned linear transforms, where two forward and one inverse transforms are required. The OL extension reviewed in Section 2.2.3 can express the fast algorithms in the declarative representation, similar to the fast algorithms for transform being expressed in the SPL.

\subsection*{6.1.1 Convolution Theorem}

We have learned the equivalence between the linear and circular convolutions and the polynomial multiplication. Next, we introduce the convolution theorem, which leads to fast FFT-based polynomial multiplication algorithms.

The convolution theorem states that the Fourier transform of a convolution is the
pointwise product of Fourier transforms. That is, convolution in one domain equals pointwise multiplication in the other domain.

**Theorem 5.** Let \( f \) and \( g \) be two polynomials \( \in \mathbb{R}[x] \) of degree less than \( n \), the convolution theorem states that

\[
\text{DFT}(f \odot g) = \text{DFT}(f) \cdot \text{DFT}(g) \tag{6.8}
\]

where \( \cdot \) denotes pointwise multiplication.

The forward transforms evaluate \( f \) and \( g \) at \( w^0, \ldots, w^{n-1} \). Its kernel is \( x^n - 1 \), and the theorem says that DFT mapping \( \mathbb{R}[x]/x^n - 1 \to \mathbb{R}^n \) is a homomorphism of \( R \)-algebras, where multiplication in \( \mathbb{R}^n \) is pointwise multiplication of vectors. The following commutative diagram clearly illustrates the mapping relationships:

\[
\begin{array}{ccc}
(R[x]/x^n - 1)^2 & \xrightarrow{\text{DFT} \times \text{DFT}} & R^n \times R^n \\
\downarrow \text{circular convolution} & & \downarrow \text{pointwise multiplication} \\
R[x]/x^n - 1 & \xrightarrow{\text{DFT}} & R^n
\end{array}
\tag{6.9}
\]

Then, by applying the inverse transform \( \text{DFT}^{-1} \), we can write:

\[
f \odot g = \text{DFT}^{-1}\{\text{DFT}(f) \cdot \text{DFT}(g)\} \tag{6.10}\]

By using the fast linear transform algorithms introduced in previous chapters, we obtain an efficient algorithm for computing the circular convolution, and thus for polynomial multiplication mod \( x^N - 1 \). In a ring \( R \) that has appropriate roots of unity to support FFT, the convolution in \( \mathbb{R}[x]/x^n - 1 \) and multiplication of polynomials \( f, g \in \mathbb{R}[x] \) with \( \text{deg}(fg) < n \) can now be performed in \( O(n \log n) \).
6.2 Fast Algorithms

We have learned that the circular convolution is equivalent to the polynomial multiplication in the ring \( R[x]/x^n - 1 \) and can be done efficiently via the convolution theorem. The linear convolution of size \( n \) can be obtained via circular convolution by zero-padding to size \( 2n \), leading to the polynomial multiplication in \( R[x] \). In this section, we present fast algorithms based on the modular FFT and TFT.

The fast circular convolution algorithm based on the modular DFT can be expressed in the OL as:

\[
\text{CirConv}_n \rightarrow \text{ModDFT}_n^{-1} \circ P_n \circ (\text{ModDFT}_n \times \text{ModDFT}_n), \quad (6.11)
\]

where the cross operator \( \times \) has an arity of \((2, 2)\) in this case and applies the left and right \( \text{ModDFT}_n \) to the two input vectors, respectively. Also recall that \( \circ \) represents the composition of operations, and \( P \) performs the pointwise multiplication with an arity of \((2, 1)\).

Similarly, the linear convolution with zero-padding, can be expressed as:

\[
\text{LinConv}_n \rightarrow \text{ModDFT}_{2n}^{-1} \circ P_{2n} \circ ((\text{ModDFT}_{2n} \circ ZP_{2n}) \times (\text{ModDFT}_{2n} \circ ZP_{2n})), \quad (6.12)
\]

where \( ZP \) is an operation that pads the input vectors with zeros at the end to the desired length. In practice, the modular DFT based fast linear convolution algorithm \((6.12)\) requires the input sizes to be powers of two and uses zero padding for arbitrary input sizes.

Other algorithm candidates include by-definition for small convolutions, and factorization of a circular convolution \((\mathbb{Z}[x]/(x^{2n}-1))\) to a circular convolution \((\mathbb{Z}[x]/(x^n-\)
and a nega-circular convolution \( (\mathbb{Z}[x]/(x^n + 1)) \) as in (6.13).

\[
\mathbb{Z}[x]/(x^{2n} - 1) \cong \mathbb{Z}[x]/(x^n - 1) \times \mathbb{Z}[x]/(x^n + 1) \tag{6.13}
\]

As we can see, the performance of convolutions essentially depends on the performance of the underlying transforms, which justifies our previous focus on the library generation and optimization for the modular FFT and TFT.

Next, we introduce the TFT-based convolution algorithm which achieves reduced arithmetic cost from the underlying TFT and ITFT. Let \( R \) be a commutative ring. We assume that \( R \) contains a principal \( L^{th} \) root of unity \( \omega \). Examples of \( R \) include \( R = \mathbb{Z}/(2^{L/2} + 1)\mathbb{Z} \) where \( L = 2^l \) and \( \omega = 2 \), which appears in the Schönhage-Strassen algorithm for multiplication in \( \mathbb{Z}[x] \) [16, 45].

The TFT and ITFT may be used to deduce a polynomial multiplication algorithm in \( R[x] \) as follows. Let \( g, h \in R[X] \), and \( u = gh \). Let \( z_1 = 1 + \deg(g) \), \( z_2 = 1 + \deg(h) \), \( n = z_1 + z_2 - 1 \), and assume that \( n \leq L \). Let \( g_0, \ldots, g_{z_1-1} \) be the coefficients of \( g \) and \( h_0, \ldots, h_{z_2-1} \) be the coefficients of \( h \). Compute

\[
\begin{align*}
(\hat{g}_0, \ldots, \hat{g}_{n-1}) &= \text{TFT}(L, z_1, n) \cdot (g_0, \ldots, g_{z_1-1}), \\
(\hat{h}_0, \ldots, \hat{h}_{n-1}) &= \text{TFT}(L, z_2, n) \cdot (g_0, \ldots, g_{z_2-1}),
\end{align*}
\]

and then compute \( \hat{u}_i = \hat{g}_i \hat{h}_i \) in \( R \) for \( 0 \leq i < n \). Then \( \hat{u}_0, \ldots, \hat{u}_{n-1} \) are the first \( n \) Fourier coefficients of \( u \), and \( u_j = 0 \) for all \( n \leq j \leq L - 1 \) since \( n = \deg(u) + 1 \). Therefore, we recover \( u \) via

\[
(Lu_0, \ldots, Lu_{n-1}) = \text{ITFT}(L, n, n) \cdot (\hat{u}_0, \ldots, \hat{u}_{n-1}).
\]
The TFT-based convolution algorithm can also be expressed in OL as follows:

\[ \text{Conv}_n \rightarrow \text{ITFT}_{L,n,n} \circ P_n \circ (\text{TFT}_{L,z_1,n} \times \text{TFT}_{L,z_2,n}). \] (6.14)

**Computational complexity.** The standard FFT algorithms compute the DFT or inverse DFT using \( \frac{L \log L}{2} \) 'butterfly operations'. In contrast, van der Hoeven showed that the TFT and ITFT may be computed using at most \( \frac{n \log L}{2} + L \) butterfly operations. Moreover, in the multiplication algorithm sketched above, only \( n \) pointwise multiplication are performed, compared to \( L \) multiplications incurred by the standard FFT method. Therefore, the ratio of the running time of the TFT/ITFT-based multiplication algorithm over that of the standard FFT-based algorithm is \( \frac{n}{L} + O((\log n)^{-1}) \), indicating that the performance is relatively smooth as a function of \( n \).

### 6.3 SPIRAL Enhancements

SPIRAL lacks the support for base case generation for multiple nonterminals within one library. In the modular polynomial multiplication library, the convolution and the linear transforms each requires a set of base cases for recursion termination and improved performance for the memory hierarchy. Therefore, the *base case generator* module has been extended to take as input and match multiple base case patterns and invoke the corresponding base case generation rules. Fig 6.1 illustrates the extension, where the dashed lines show the boundaries of the libraries to be generated.

The tag propagation is important in the OL formula derivation, especially when substructures are encapsulated in operators such as \( \times \), the cross operator. During the development, a small yet severe bug has been discovered which, under certain
For vectorized and multi-threaded library generation, two tags are propagated during the decomposition, namely $[\text{AParLib}, \text{AVecLib}]$. The $\text{AParLib}$ tag triggers the GT$_\text{HPar}$, and the $\text{AVecLib}$ tag triggers the GT$_\text{HVec}$, for parallelization and vectorization derivation, respectively. A small rewriting rule named $\text{TTag\_Cross}$ was designed to distribute the tags to the substructures under the cross operator. However, $\text{TTag\_Cross}$ used a left fold operation, which effectively reverses the order of the tags. When the $\text{AParLib}$ tag is last consumed, it prevents the inclusion of any base cases, resulting in “black holes” in the closure computation which eventually lead to library generation failure. The solution is simple: use the right fold in $\text{TTag\_Cross}$.

The planning framework has been extended to ensure the mirrored decomposition paths between the permuted TFT and ITFT within the circular convolution library. The DP knowledge entry is extended to include an execution signature generated each time when the top-level planning starts. The forward transform first tries to reuse
the entry that matches the function signature and the execution signature. If such entry does not exist, it tries to reuse an entry that matches the function signature, and constructs a new entry based on the matched entry and the current execution signature. If no entry is matched on the two said levels, the forward transform creates a DP knowledge entry via the planning framework with its current signature. Then the inverse transform can follow the mirrored decomposition path by reusing the DP knowledge entries for the forward transform with some extra string manipulation.

6.4 Performance Evaluation

This section reports the performance data comparing the TFT-based convolution library and the modular FFT based convolution library, both of which are automatically optimized and generated by SPIRAL. The performance is reported in cycles measured by PAPI [40] and is averaged over 1000 runs with small observed performance variance.

Baseline implementation. An SPIRAL-generated parallel ModDFT-based convolution library using Algorithm (6.12) is used as the baseline implementation. The reference library pads input to the next power of two, and fully utilizes vector registers and multi-core. Its underlying transform has been shown in [35] that its performance is comparable to optimized fixed-size codes [36], and gains an order-of-magnitude speed-up over hand-optimized library [31]. The ModDFT-based convolution library is represented by the solid black line in Figure 6.2 which exhibits clear jumps in running time when the lengths cross a power-of-two boundary.

Autotuned TFT-based convolution library. The TFT-base convolution library employs the strict general-radix algorithms for base case generation to reduce arithmetic cost. The built-in search engine uses the DP technique that measures the actual running time of smaller transforms as the input to the feedback loop to guide
the generation of the base case sizes up to 8. For the library-level recursive break-
down, it applies the relaxed parallel algorithms, which trades off a slightly higher
arithmetic cost for vectorization and parallelization. The TFT-based convolution
library is represented by the red line in Figure 6.2

**Speedup.** The 4-thread TFT-based convolution library delivers a speedup of
35% – 41% over the 4-thread high performance ModDFT-based convolution library
with length that just crosses a power-of-two boundary. As the length increases,
the gap between the TFT-based approach and the full transform based approach
decreases.

Overall, the TFT-based convolution library’s performance is smooth with respect
to the input size. Also note that, as we proved in Section 5.2.2 and 5.3.2, the relaxation
introduces slightly higher arithmetic cost which is bounded by the optimized base case
sizes. As a result, mini jumps can be seen between power-of-two jumps, which do not
affect the overall smooth performance.

6.5 Conclusion

In this chapter, we presented the automatic library generation and optimization
for modular polynomial multiplication. We reviewed the definitions of polynomial
multiplication and the closely-related convolutions. Then, we presented the convolu-
tion theorem which leads to the fast FFT-based convolution algorithms. With the
design and implementation of the TFT and ITFT, the FFT-based modular polyno-
mial algorithms can choose from the power of two modular DFT and the TFT. We
have shown that the performance gains from previous effort, including the improved
modular arithmetic and the parallel modular DFT and TFT, directly contribute to
the performance of the automatically generated and optimized library for modular
polynomial multiplication. Furthermore, the performance evaluation comparing the
Figure 6.2: Performance comparison between the SPIRAL-generated TFT-based convolution library and modular DFT-based convolution library

modular DFT based library and the TFT-based library shows the practical performance benefits from the new TFT algorithms.
7. Conclusion

7.1 Summary

In this thesis we presented an automatically generated and optimized library for modular polynomial multiplication based on the FFT. The autotuned libraries for polynomial multiplication and the underlying transforms improve on the state-of-the-art in practical performance. The SPIRAL library generation system has been extended and used to automatically generate and tune the libraries for memory hierarchy, vectorization and multi-threading, using new and existing algorithms.

The generation and optimization process took a bottom-up approach, starting from the optimized modular arithmetic. Then, the modular DFT and its fast algorithms were implemented and derived using the SPL language to generate a general-size parallel library that is faster than the state-of-the-art by an order of magnitude.

Next, aiming at smoothing the staircase phenomenon associated with the power of two FFTs and zero padding, new general-radix and parallel algorithms for the TFT and ITFT were presented with proved lower complexity. The algorithms are used together in the generation and optimization for the TFT and ITFT libraries which gain further speedup over the parallel modular DFT library, proving the practical performance benefit from using the TFT.

Finally, we constructed a parallel modular polynomial library based on the modular FFT and the TFT. The fast algorithms were expressed in the OL extension and derived automatically to utilize all levels of parallelism. We further extended SPIRAL to handle the evolved library generation requirements. The result shows that the TFT-based modular polynomial multiplication library smooths the staircase phenomenon for output sizes between powers of two associated with the modular DFT
based approach.

7.2 Future Work

Our library can serve as the foundation of a more general library for polynomial and integer arithmetic. Many basic polynomial arithmetic operations such as division can be efficiently reduced to multiplication. Moreover, calculations with univariate and multivariate polynomials can be reduced to computing with polynomials over finite fields, such as the prime field $\mathbb{Z}_p$ for a prime number $p$, via the so-called modular techniques. Additionally, multivariate polynomial arithmetic can be reduced to univariate polynomial arithmetic via evaluation and interpolation. Furthermore, most calculations tend to densify intermediate expressions even when the input and output polynomials are sparse. Therefore, the future work includes:

- **Exploration of non power-of-two FFTs and hybrid algorithms.** Algorithms exist that do not require the input sizes to be powers of two. Those algorithms are harder to generate and optimize, but can still be expressed in the domain specific language. It is also possible to achieve further performance improvement by combining more algorithms in a hybrid approach.

- **Multivariate polynomial multiplication** is naturally mapped to multidimensional convolution and to underlying multidimensional transforms. The multidimensional algorithms can be readily expressed by SPL.

- **Integer polynomial multiplication.** In many applications, polynomial multiplication in $\mathbb{Z}[x]$ is desired. It is possible to perform multiplication in $\mathbb{Z}[x]$ via multiplications in a collection of prime fields and reconstruct the integer result via the Chinese Remainder Theorem (CRT).
• **Large integer multiplication.** The Schönhage-Strassen algorithm uses the nega-circular convolution and weighted transform to multiply large integers. The underlying components in the algorithm are closed related to the components in our modular polynomial library. The automatic generation and optimization process developed in this thesis can be migrated to generate and optimize the large integer multiplication.

• **Cryptography.** Some cryptographic algorithms, such as the SWIFFT [33], are constructed using variations of the FFT over finite fields (e.g., the cyclotomic FFT). The optimized modular arithmetic and modular FFT presented in this thesis can be used or modified to improve their performance.
Bibliography


