

Design Paradigms and Tools for Modern Fourier Transform Hardware

Submitted in partial fulfillment of the requirements for
the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Lawrence Tang

B.S., Electrical and Computer Engineering, Cornell University
M.S., Electrical and Computer Engineering, Carnegie Mellon University

Carnegie Mellon University
Pittsburgh, PA

April 2026

Acknowledgments

The work throughout my PhD studies would not have been possible without the support of so many people around me. I would like to express my sincere gratitude to those who have helped me reach this point. First, I thank my advisors, Prof. Ken Mai and Prof. Franz Franchetti, for their continuous mentorship and guidance throughout my PhD. They gave me the freedom to explore a variety of different topics, helped me shape my own research approach and ideas, and believed and trusted in me throughout the entire journey. It has been an incredibly rewarding experience to gain so much knowledge across circuit design, computer architecture, and software systems, and many of our discussions have been great sources of inspiration. I would also like to thank my thesis committee members, Prof. James Hoe and John Shalf for their valuable feedback, insights, and questions which refined the quality and contributions of this thesis work.

My research group members, both on second floor and A-level, have been amazing collaborators and friends. Siyuan, Siddharth, Keshav, Chris, Kent, Oguz, Cagri, and Prashanth—I am grateful for all of the support and practical knowledge I’ve gained from our tapeouts and conversations; many of the test chips presented in this thesis would not have been possible otherwise. From the SPIRAL group, Naifeng, Sanil, Eric, Het, Tianyun, Emma, Quentin, and Yunhao—thank you for

all of the research collaborations and discussions, whether it was on code generation or some random new ideas, and for always taking the time to help me polish my own work. Thank you to my collaborators from SEI, John Wohlbier, Cathy Bernaciak, and David Graham for their guidance and help on the modulation recognition project. I would also like to thank the Apple NSI Initiative for their support of numerous academic tapeouts and opportunities for industry experience. Finally, I appreciate all of the talented graduate and undergraduate students that I have worked with over the years, for which there are probably too many to thank here.

I feel very fortunate to have made many friends, Siyuan, Naifeng, Shashank, Jason, Het, Upasana, Chris, Jen, Icey, and Elliot, here in Pittsburgh. We've shared so many fun and unforgettable experiences together, from casual hangouts to road trips and spontaneous hikes. To my friends outside of Pittsburgh from college and my hometown, all of our calls, group trips, and meetups have helped me relax through some of the stressful times, reminded me to keep things in perspective, and always left me with a breath of fresh air.

Finally, I thank my parents and brother for their unwavering support and encouragement. You have taught me to be patient, to be resilient, to be open-minded and curious, inspiring me to pursue my goals and challenge myself. For that, I am eternally grateful.

LAWRENCE TANG

Carnegie Mellon University

April 2026

Abstract

Many high performance FFT applications rely on software libraries for development and target a variety of computing architectures based on performance needs. However the performance scaling of modern computing platforms is now driven by improvements in energy efficiency, where the shift towards domain-specific hardware and advanced packaging gives rise to an opportunity to rethink the traditional FFT hardware design approach. The conventional approach of designing FFT hardware as a standalone IP block is relatively well understood for high throughput streaming signal processing applications. But this design methodology is fundamentally at odds with the requirements of high performance (embedded) computing, where software-like flexibility and programmability are crucial for application development.

This thesis proposes a new paradigm for designing specialized hardware targeting high performance, software-driven DFT-based applications that are executing on current and emerging computing systems. The design approach is based on the idea of hardware accelerating only the base cases of the FFTW software library, known as codelets, and retaining flexibility by allowing software to still specify their composition. A hardware architecture supporting a collection of different codelets can then be targeted through standard FFT library calls, making it transparent

to the application developer. These design principles are integrated from software stack down to custom hardware and realized through the development of design automation tools.

To showcase these design paradigms, we first present a flexible accelerator architecture that can be reconfigured in the style of FFTW codelet computations. Hardware codelet design for small size powers-of-2 and primes is driven by the SPIRAL code generation system. The accelerator orchestrates their computation and particular hardware instances can be automatically generated using a parametrized design automation framework. We then demonstrate how the generated accelerator can be integrated into a system-on-chip and coupled to a control processor. New software and code generation tools enable accelerator offload while the specialized hardware remains hidden behind domain-specific library calls. Finally we present silicon prototypes that showcase how the end-to-end design approach is applied and extended to applications in scientific computing and signal intelligence.

Contents

Acknowledgments	iii
Abstract	v
List of Figures	xii
List of Tables	xx
Chapter 1 Introduction	1
1.1 Motivation	2
1.2 Background	5
1.2.1 FFT Software Frameworks	6
1.2.2 FFT Hardware	7
1.2.3 Domain-specific Hardware Design	11
1.3 Proposed Approach	14
1.4 Contributions	14
1.5 Thesis Outline	15
Chapter 2 Accelerating FFTW Codelets	17
2.1 The Discrete Fourier Transform	17
2.1.1 Multi-dimensional DFT	18

2.1.2	Convolution	19
2.1.3	Forward and Inverse Transforms	22
2.1.4	Other properties of the DFT	23
2.1.5	Kronecker Product Notation	24
2.1.6	Tensor Product Identities	25
2.1.7	Short Vector Algorithms	26
2.2	“Hardening” FFTW-style Codelets	27
2.3	A First Radix-8 Codelet Test Chip	31
2.3.1	Radix-8 FFT Architecture	34
2.3.2	Diag Unit	34
2.3.3	Memory	35
2.3.4	Control Unit	36
2.3.5	Dataflow	36
2.3.6	Measured Results	37
2.4	The Hardware Codelet Microarchitecture	40
2.4.1	FFT Datapaths	41
2.4.2	Diagonal	44
2.4.3	Transposer	46
2.4.4	Supported Datapath Configurations	47
Chapter 3 A Design Automation Framework		50
3.1	An Architectural Template for Hardened Codelets	51
3.1.1	Architecture Overview	53
3.1.2	Descriptors	53
3.1.3	Frontend	54
3.1.4	IO Engine	55

3.1.5	Hardware Codelet	56
3.2	Hardware Design Flow	57
3.2.1	Generating Hardened Codelets	57
3.2.2	Generating the Codelet Accelerators	61
3.2.3	Evaluation of Overheads	63
3.3	Software Framework	65
3.3.1	Low-level Software Library	65
3.3.2	Simulation Models	66
3.3.3	LibraryX-ASIC	66
3.3.4	End-to-end Example	67
3.4	Generator Verification via Silicon Testchip	70
3.4.1	A Radix-4 Codelet Accelerator in 28nm	70
3.5	System Integration	72
3.5.1	RISC-V System-on-Chips	72
Chapter 4	Extending Hardware Capabilities	75
4.1	An Algorithm-based Approach to Soft Error Tolerance	75
4.1.1	Algorithm-Based Fault Tolerance	77
4.1.2	Interval Arithmetic	78
4.2	Interval Arithmetic-based Soft Error Tolerance	79
4.2.1	Hardware Redundancy via Interval Arithmetic	79
4.2.2	Error Detection	81
4.2.3	Forward Error Analysis	82
4.3	A Soft Error Tolerant FFT Accelerator in 28nm	85
4.3.1	Fast Fourier Transform	86
4.3.2	Interval Generation	87

4.3.3	Test Infrastructure	88
4.3.4	Silicon Prototype	88
4.3.5	Comparison against TMR	89
4.3.6	Fault Injection Simulation	90
4.3.7	Summary	91
4.4	A 16nm Heterogeneous SoC for Signal Modulation Recognition . .	91
4.4.1	Cyclostationary Processing	92
4.4.2	The Strip Spectral Correlation Algorithm (SSCA)	94
4.4.3	Accelerator Implementation for SSCA	96
4.4.4	SSCA Program Execution	98
4.4.5	RISC-V SoC Architecture	99
4.4.6	Post-PnR Layout	102
4.4.7	Results and Comparison	103
4.4.8	Discussion	105
4.5	DFTs of Real Data	106
4.5.1	Real FFT via Half-size Complex FFT	107
4.5.2	Real Codelet Architecture	110
4.5.3	Integration with Design Framework	112
Chapter 5	Performance Modeling and Discussion	113
5.1	Execution of LibraryX-ASIC Example	113
5.2	LibraryX-ASIC Results	116
5.3	Discussion	119
Chapter 6	Concluding Remarks	122
6.1	Lessons Learned	124

6.2 Future Work	126
Bibliography	128

List of Figures

1.1	Applications of $N \log N$ algorithms using the fast Fourier Transform.	3
1.2	Serial Delay Feedback FFT Architecture	8
1.3	Memory-based FFT Architectures	9
2.1	Overview of the FFTW library. Codelets are automatically generated pieces of C code that perform steps of the FFT computation at various small sizes. The library infrastructure implements data structures, recursive functions, and vectorization features that enable the flexible and adaptive nature of FFTW.	28
2.2	Example FFTW program. The function <code>fftw_plan_dft_1d</code> constructs and stores the plan as a composition of codelets and the actual execution takes place at the <code>fftw_execute</code> function call. . .	28
2.3	Example of a 4-point FFT invoking hardware codelets.	30
2.4	Example 4-point FFT codelet plan and execution call.	30
2.5	Comparison of the FFTW approach to the proposed hardware codelet approach.	31

2.6	Overview of the test chip design flow from SPIRAL script to generated RTL code to physical implementation. The SPIRAL script defines the parameters of the generated RTL-level description of the FFT hardware. Other functional blocks are hand designed and integrated with the generated code which then goes through the ASIC design flow.	32
2.7	System block diagram of test chip. The input FIFOs feed 16 single-precision numbers to the radix-8 FFT every cycle and output FIFOs capture 16 single-precision numbers from Diag. The chip can be run in a looping mode for power measurements.	33
2.8	Internal architecture of the Spiral-generated radix-8 FFT and Diag complex multiplier block. The FFT unit uses single-precision adders and constant multipliers. The Diag unit has 8 complex multipliers, where each complex multiplier has four multipliers and two adders.	35
2.9	Timing diagram of the dataflow through the FFT and Diag compute cores.	37
2.10	Die micrograph of 1.5mm $\times$ 1mm Spiral-FFT test chip in TSMC 28nm. The chip has 60 I/O pads and a core area of 1.2mm $\times$ 0.7mm. FIFO memory holds a total of 4.5 kB of data.	38
2.11	Breakdown of power consumption at 0.9V and area by block . . .	39
2.12	Shmoo plot of SPIRAL-FFT accelerator. Green areas represent the points at which the chip is functional. At nominal 0.9V the chip runs at 260 MHz and at a supply range of 0.65-1.15V the chip operates at 155-270MHz.	40

2.13	Configurable datapath implementing the equivalent of an FFTW codelet. The datapath is reconfigured during runtime based on the plan specification, which sets the values of the configuration registers.	41
2.14	FFT datapaths.	42
2.15	Diagonal Block	45
2.16	Systolic Array Transposer	47
2.17	A visual representation of the arrangements of the reconfigurable codelet. The green FFT datapaths can be reconfigured for different sizes or forward/inverse transforms. The Diag is shown in blue and transpose units are shown in purple.	49
3.1	High-level machine model. The accelerator interfaces to its own local memory and is coupled a controlling CPU. The system has a mechanism for data transfer to and from off-chip main memory.	51
3.2	FFT accelerator architectural template. The decoder interprets descriptors specified by the accelerator program to set the codelet configuration and fetch data according to the IO base address/stride patterns. The codelet datapath moves data to/from banked scratch-pad memory.	52
3.3	Automated Design Flow. The SPIRAL code generation system is used to co-design parts of the accelerator core and for code generation.	58
3.4	Generated Chisel code for radix-3 single-precision FFT hardware. This example shows how adders, multipliers, and pipeline registers are instantiated in the generated hardware.	60

3.5	Example radix-2, 3, 4, 8 generated FFT hardware. Each arithmetic unit performs real-valued operations and the inserted pipeline registers are also shown. All multipliers are constant multipliers in the FFT datapaths.	61
3.6	Overview of the accelerator generation process. User defined parameters including codelet sizes, data type, and local memory capacity are passed to the Chisel generator. The hardware codelets are pregenerated and contained in a separate Chisel package. . . .	62
3.7	Parametrized abstract class <code>DftIO</code> which all generated FFT modules must extend. The selected codelet sizes are then mapped to the corresponding FFT hardware module.	63
3.8	Percent overhead of descriptor fetch, decode, and control units out of total accelerator logic area.	64
3.9	Overview of the FFT accelerator microarchitecture.	67
3.10	The FFT application in written against FFTW will be executed on an FFT acclerator without user modification using LibraryX-ASIC. The output buffer will be populated as if nothing changed.	68
3.11	SPIRAL generated accelerator code. The generated code uses information from SPIRAL's FFT description including input ranges and input and output strides.	69
3.12	Summary of $1\text{mm} \times 1\text{mm}$ test chip implementation in a TSMC 28nm process and micrograph of generated accelerator	71
3.13	Voltage-frequency operating points and total power consumption measured from 0.65V - 1.15V. Operates from 350-1025 MHz and consumes 142.95 mW at 0.9V, 750MHz.	72

3.14	ESP Tile-based RISC-V SoC with generated FFT accelerator tile.	73
3.15	Accelerator program execution on the ESP RISC-V platform with custom FFT accelerator.	74
4.1	TMR technique using three copies of the module and a majority voter circuit.	76
4.2	Interval arithmetic based approach for soft error tolerance.	80
4.3	Interval arithmetic hardware block diagram for (a) addition, (b) subtraction, and (c) multiplication.	81
4.4	The four potential cases for an output interval: (i) no error, middle value accepted (ii) interval grows beyond bound, (iii) interval endpoints flipped, and (iv) high precision middle value outside of interval.	82
4.5	High level block diagram of the proposed method for error detection.	83
4.6	Example of interval growth after floating point operations. Each interval begins at 1 ulp distance but the resulting interval grows to 2 ulp due to rounding.	83
4.7	Analysis of an example 2-point FFT butterfly datapath to bound the output interval growth.	84
4.8	Hardware block diagram for interval checking using forward error analysis.	85
4.9	Architecture of the radix-8 FFT datapath and interval error detection.	86

4.10	System diagram of the CosmIC test chip. The datapaths include a complex radix-8 FFT and a systolic array, each replicated in interval arithmetic. Input data can be generated through the LFSRs or specific tests can be loaded into the on-chip memory. Additional test structures enable fault injection tests and error tracking. . .	87
4.11	1mm x 1mm die micrograph. There are 44 IO pads and three power domains. The protected accelerators are on the right side of the die. Additional test infrastructure including LFSRs, memories, interval generation, and clock generator are on the left side of the die. . .	89
4.12	Comparison of total area between TMR and interval arithmetic protected implementations.	90
4.13	Modulation Recognition composed of scan, detect, and characterize phases. We focus primarily on the computationally intensive spectral analysis algorithms in the characterize phase for symbol rate estimation.	92
4.14	Whiteboard view of the SSCA Algorithm.	95
4.15	Plots of input signal and steps of the SSCA algorithm for symbol rate estimation. The bottom right plot shows the strong cycle frequency features present in the spectral coherence at $\alpha = 0.25$. . .	96
4.16	SSCA Accelerator Architecture. The radix-8 hardware codelet now interfaces to banked scratchpad memories to store intermediate data and specialized buffers for loading input data in a strided fashion.	98

4.17	VexRiscv SoC consisting of a 32-bit RISC-V CPU tightly coupled to the SSCA accelerator and DMA engine for data transfers. There is a total of 2.27MB of on-chip scratchpad memory	100
4.18	Shared scratchpad to accelerator interface. Access to the memories is controlled such that requests can only come from the accelerator or AXI port.	101
4.19	Layout capture of the fabricated 2mm x 2mm test chip in a 16nm FinFET process. The SSCA accelerator and RISC-V core are in the middle of the die, with banked scratchpad memories on the top and bottom of the accelerator.	103
4.20	Summary of 2mm × 2mm test chip implementation in a TSMC 16nm FinFET process. The area breakdown considers the annotated SoC components shown in Fig. 4.19.	104
4.21	Real Codelet Datapath	111
5.1	The FFT application in (a) written against FFTW will be executed on an FFT accelerator without user modification using LibraryX-ASIC. The output buffer will be populated as if nothing changed. The generated code in (b) uses information from SPIRAL’s FFT description include input ranges and input and output strides. . .	114
5.2	High level abstract system model of the CPU-accelerator system. The model consists of a CPU controller coupled to the accelerator, fast on-chip local memory, and main memory. The micrograph shows a silicon-verified FFT accelerator implementing a part of the architecture in Fig. 3.9.	115

5.3	Flow chart demonstrating the execution of the generated accelerator program.	116
5.4	Test setup of custom FFT ASIC and integration of measured results into LibraryX-ASIC experimental setup.	117
5.5	FFTW on CPU vs. LibraryX-ASIC on accelerator system.	118
5.6	cuFFT on GPU vs. LibraryX-ASIC on accelerator system.	119
5.7	Breakdown of 4096 complex FFT executing on the accelerator. . .	120

List of Tables

2.1	Table of DFT properties between time and frequency domain. . .	23
2.2	Table of tensor product identities to manipulate DFT formulas. .	26
2.3	Implementation Summary	39
2.4	Supported mappings from configuration registers to datapath computation.	48
4.1	ASIC implementation summary and comparisons	105

Chapter 1

Introduction

The Discrete Fourier Transform (DFT), and in particular its $\mathcal{O}(N \log N)$ implementation using the fast Fourier Transform (FFT) algorithms, is ubiquitous in scientific and engineering applications. Many high performance FFT applications rely on software libraries for development and execute on systems such as supercomputers to embedded devices depending on performance needs and constraints. Yet the landscape of modern computing platforms is continuously changing—the driving factor for scaling performance now lies in improving energy efficiency, with a shift towards domain-specific hardware and advances in 2.5/3D packaging and integration. These technological shifts in system design give rise to a critical opportunity to rethink the traditional FFT hardware design approach for high performance, software-driven FFT applications.

The thesis of this dissertation is that there is a systematic approach for designing specialized FFT hardware for the flexibility required by high performance FFT software, addressing the need for a new generation of FFT hardware that fits the paradigm shifts in modern computing systems design. Traditionally, cus-

tom FFT hardware is treated as a standalone IP block where the design space is relatively well understood for high throughput streaming signal processing or communications applications. However this design methodology is fundamentally at odds with the need for software-like flexibility and programmability in high performance (embedded) computing. This insight suggests that a new hardware design approach should instead start from the perspective of the FFT software which drives these applications. This enables a powerful new generation of FFT-based hardware that is versatile enough for a domain of FFT-based applications while maintaining software programmability and benefiting from the energy efficiency of custom hardware.

1.1 Motivation

The implementation of the DFT using the fast Fourier Transform (FFT) algorithms plays a fundamental role in a number of modern electronic systems and computing workloads spanning digital signal processing (DSP) and communications systems [1–3], medical imaging [4, 5], scientific computing [6–10], and machine learning [11, 12]. In scientific computing, spectral methods for partial differential equation solvers including WarpX [13], NWChemEx [14], P³M [15], rely on large-scale, multi-dimensional FFTs to perform convolution operations. Signal processing or communication systems use one-dimensional transforms of hundreds or thousands of points for spectral analysis [16] or wireless protocol support such as WiFi or 5G/6G cellular standards [17–19]. Deep learning neural networks have explored the use of FFT-based methods for fast convolutions [20, 21] and as layers in transformer models and neural operators [12, 22, 23]. There are thus a plethora of different software and hardware-based approaches to address the needs of specific

applications.

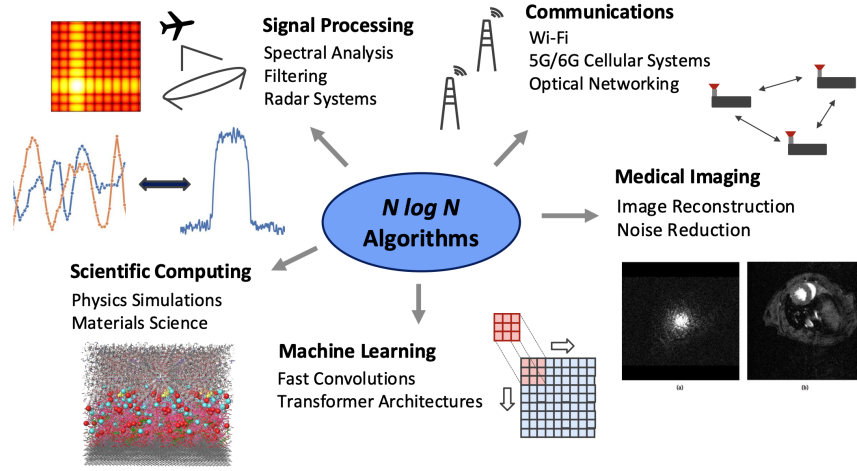


Figure 1.1: Applications of $N \log N$ algorithms using the fast Fourier Transform.

While software-based FFT implementations provide a higher degree of flexibility for different problems and computing architectures, dedicated hardware is often the solution of choice when tailoring to strict power, latency, or throughput constraints. The traditional approach of designing FFT hardware as a standalone IP block has been well studied [24], especially in the context of streaming DSP and communications systems. The design methodology is typically as follows: hardware designers consider application requirements and tradeoffs between the various known architectures, select and optimize a design point for the application, and finally integrate the fixed function IP block into the broader system. To ease the design process, there exists a number of FFT hardware generators which automatically generate hardware instances across a wide design space by providing a selection of different architectural parameters. There also exists a variety of vendor IPs that can be used for specific FPGA or ASIC targets. Fundamentally,

the design philosophy is that the hardware design choices are made strictly with an application requirement (e.g. throughput, latency, power) in mind.

In the past decade, the performance scaling of modern computing platforms has been driven by a shift to (i) heterogeneous architectures including CPUs, GPUs, and domain-specific architectures, and (ii) chiplet-based approaches leveraging advanced 2.5/3D packaging and integration. The key technology advances have been towards improving energy efficiency of systems as a means of continued performance scaling in the post-Dennard scaling era. So for domains such as scientific computing, autonomous systems, or radar processing—where the application software spends a significant fraction of time in FFT routines to process large amounts of data—there is an opportunity to realize a new generation of specialized FFT hardware that provides energy efficiency gains for the next high performance computing system.

However, the conventional wisdom in designing FFT hardware for signal processing systems starts to break down in application domains where requirements are fundamentally very different. In high performance computing (HPC) and high performance embedded computing (HPEC), FFT-based applications are driven by software programs that run on processors or accelerator devices, and instead interface to stored data through the system’s memory subsystem rather than continuously streaming data. For example in scientific computing, PDE solvers spend time in FFT-based spectral methods where software-like flexibility is crucial for application developers to realize DFTs of different sizes, dimensions, data patterns, and even algorithms. The existing design paradigm leads to inflexible FFT hardware implementations due to fixed problem types and algorithm support. Once implemented, designs can only target a set of fixed sized lengths and are often fur-

ther specialized for narrow application domains such as a specific communication protocol. Thus the standard FFT IP blocks or generators cannot be used in the context of such applications.

Another fundamental mismatch stems from the lack of a standard programming interface for users to easily target specialized FFT hardware. For IP blocks that are eventually integrated with software, the programmer must write against a custom API that requires application code modification along with some understanding of the hardware to take advantage of design-specific features.

Thus there are three key challenges which must be addressed to realize a new generation of FFT-based accelerators for the modern HPC/embedded system. First, a novel architecture which allows for software-like flexibility to support general FFT problem specifications in the context of broader applications. Second, a software stack must be co-designed with the hardware such that it can be targeted through standard software libraries and remains transparent to the user. Lastly, design automation tools are necessary to actually build the accelerators and automatically integrate the software stack.

1.2 Background

The diverse set of FFT-based applications has inevitably led to a large number of software and hardware implementation approaches that are used and fine tuned based on specific application requirements. In this section we discuss some of the prior state-of-the-art software and hardware approaches for implementing FFTs.

1.2.1 FFT Software Frameworks

There are several software FFT libraries targeting different computing architectures. In the following section we present software frameworks that are used for realizing efficient FFT code. These frameworks are critical for application developers who express their computations using domain-specific libraries.

FFTW. FFTW [25] is a general-purpose, adaptive software library for computing the FFT and has become widely used in FFT-based scientific applications. The system combines a domain specific language, code generator, and autotuning infrastructure to produce a portable and high performance FFT library. The code is automatically generated and tuned to the specific CPU architecture, and has been extended for SIMD vectorization and multi-threading support. FFTW provides a number of different routines including arbitrary size, forward or inverse, multi-dimensional, real and complex FFTs. The FFTW API has since become the de facto standard interface for computing FFTs in HPC.

Vendor Libraries. Vendor software libraries such as Intel MKL [26], Nvidia cuFFT [27], AMD rocFFT [28], IBM ESSL [29], and Apple vDSP [30] offer FFT routines optimized for the respective vendor computing devices. These routines use hand-tuned, specialized algorithms to tailor towards the vendor-specific computing architecture, thus provide some of the best performance for its targeted hardware platform. Due to the hand-tuned nature of vendor provided libraries, performance portability is often limited especially when compared to the highly portable FFTW library. As FFTW has become the de facto standard FFT interface, almost all of the above FFT libraries also implement a part of the FFTW interface.

FFTX and SPIRAL. The FFTX [31] project looks to build a more modern framework for FFT based applications while still maintaining exposure to the

familiar FFTW interface. The first component is a front end library interface that maintains backwards compatibility with FFTW but also provides a means to target higher level operations and new hardware platforms. For example, FFTX targets higher level FFT-based applications such as 1D pruned convolutions which require point-wise operations in addition to FFT calls. At the back end, automatic code generation for the application uses SPIRAL [32]. FFTX will make it easier to target new hardware platforms, including accelerators, for high performance FFT applications and exascale computing efforts.

SPIRAL [32] is a code generation system that targets a wide range of numerical and digital signal processing kernels, originally focusing on FFTs. It begins with a problem specified as a mathematical expression, described in an internal language called the Signal Processing Language (SPL). The SPL representation is implemented using one of the algorithmic breakdowns of the expression in the system. It then goes through a series of transformation stages which introduces loops and indexing patterns, compiler optimizations, and a final abstract code representation. The SPIRAL framework has been extended to numerical kernels beyond just FFTs, including graph analytics [33], cryptography [34, 35], and stencils [36] in scientific computing. The system has also been extended for hardware generation capabilities, including DFT IPs, sorting networks, and other linear signal transforms [37–39].

1.2.2 FFT Hardware

Dedicated FFT hardware, in particular for streaming DSP and communications systems, has been studied since the 1970s, leading to new architectures, design tool advances, and a plethora of optimizations to tailor towards strict power, la-

tency, or throughput constraints. For a given problem specification, the area and performance tradeoffs to meet such requirements are generally well understood.

FFT Hardware Architectures. The conventional FFT architectures for signal processing systems can be broken down into three different categories: pipelined, memory-based, or fully-unrolled [24].

Serial and Parallel Pipelined: Both serial and parallel pipelined FFT architectures process input data in a continuous flow consisting of $\log N$ compute stages, where each stage corresponds to an FFT computation stage [40]. Serial-pipelined architectures process data at a fixed throughput of one sample per cycle whereas the parallel-pipelined can be scaled to throughputs greater than a single element per cycle. The most common single-path delay feedback (SDF) architecture consists of $\log_2 N$ radix-2 butterflies that feed results back to a buffer such that the streaming inputs are aligned in the butterfly pattern for that stage. Other se-

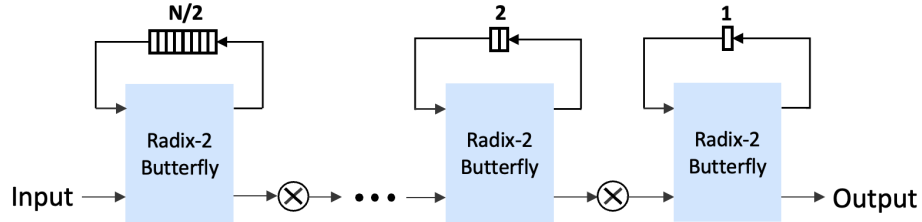


Figure 1.2: Serial Delay Feedback FFT Architecture

rial variants rearrange the buffers and commutators to create the serial feedforward (SFF) and serial delay commutator (SDC) versions, but are always fixed a throughput of one sample per cycle. The parallel-pipelined flavors follow similar principles but instead expose multiple processing paths for multiple samples to be processed

in parallel per cycle. Generally the pipelined FFT architectures are inflexible due to fixed length buffers that become wasteful when attempting to generalize across a wider range of sizes.

Memory-Based: A memory-based architecture, depicted in Fig. 1.3 consists of a processing element (PE) that reads and writes data to banks of memory to iteratively compute an FFT. The butterfly PE typically implements a small-sized radix- r FFT as the base case computation. The butterfly PE reads data from memory, computes on that data, and then stores the results back to memory, with the computation continuing iteratively until all stages of the FFT are complete. Multiple butterflies can be instantiated to scale the performance of the hardware, but to sustain pipeline saturation requires as many memory banks as butterfly inputs. There is flexibility in how the butterfly PE is actually implemented, where

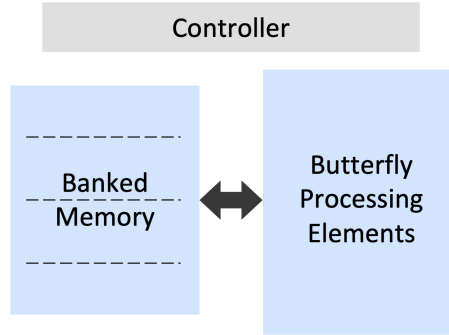


Figure 1.3: Memory-based FFT Architectures

higher-radix PEs offer higher throughput at the cost of increased memory bandwidth requirements and control logic complexity to maintain parallelism. Butterflies of small prime sizes such as 3 or 5 are also commonly used when required as

a base case in the FFT decomposition. Previous works have investigated a unified butterfly architecture that supports multiple small prime radices [41–43] for improved hardware reuse. Overall the complexity of the memory-based architectures comes in designing control logic to avoid bank conflicts for each butterfly, but can be simpler to generalize once implemented.

Fully Unrolled: Fully unrolled FFT architectures offer the highest throughput performance by processing a full sequence of inputs every cycle. The architecture is a direct mapping of the FFT signal flow diagram to arithmetic units, where each multiply/add required is implemented with its own multiplier/adder in hardware. It typically finds use in applications requiring the highest possible throughput of batched smaller sized FFTs [44] or as the PE in a memory-based architecture.

The tradeoff between throughput, area, and energy efficiency can be calibrated by carefully considering architectural choices along with mathematical optimizations such as radix choice, mixed-radix algorithms, rotations by roots of unity (e.g. CORDICs), or bit-width manipulations. There is extensive literature investigating each of these architectures to optimize for specific constraints, with some examples presented in the following section.

FFT Hardware IP and Generators. There are a number of academic and vendor design tools for automatically generating FFT hardware. Many FFT hardware generators [37, 45–49] generate designs which support only a single transform size, although they are highly parametrized for FFT length (mostly powers of 2), dimension, data type, and even performance constraints. In [43], Wang et. al build the first generator for runtime-reconfigurable, mixed radix FFT engines to support $2^n 3^m 5^k 7^l$ FFTs primarily targeting digital signal processing systems. The core is a reconfigurable butterfly derived from Winograd’s short Fourier Transform

algorithm [50].

Vendor FFT IP blocks from Xilinx [51], Intel [52], and Mentor Graphics [53] offer both variable and fixed size FFT IP cores for 1D powers-of-2 transform sizes, different arithmetic types, forward/inverse, and optimizations for DSP applications. The generated IP cores primarily target FPGAs, so they provide an interface to the vendor standard protocol.

Single instances of reconfigurable FFT designs have primarily been investigated in the domain of communication systems which must support different standards. The overarching approach across these designs and generators is the same—pipelined or memory-based FFT architectures are adapted/optimized such that they can target specific constraints or generalize to a fixed set of parameters.

1.2.3 Domain-specific Hardware Design

Designing domain-specific hardware is by nature time-consuming and challenging due to numerous factors that must be considered during design and deployment. Some of the main challenges lie in application understanding, novel architecture design, tools to guide high-level architectural decisions or design space exploration, and software programmability [54]. This section discusses recent state-of-the-art tools and approaches to promote faster design and verification time, hardware reusability, automatic optimizations, and improved software development toolchains. We focus on some of the recent approaches for automating hardware design and also take note of specific approaches for the development of FFT hardware.

High-Level Synthesis (HLS). High-level synthesis (HLS) raises the level of abstraction for hardware design by converting high-level C/C++/SystemC code to register-transfer level (RTL) designs, enabling faster hardware design time. Writ-

ing HLS code to achieve performance goals typically requires some code restructuring and understanding of the hardware architecture to guide the tool, but abstracts away low-level implementation details and automates RTL generation. The designer no longer needs to worry about cycle-level timings or detailed logic design. There are a few vendor HLS tools including AMD Vitis HLS [55], Intel oneAPI DPC++/C++ [56] to target FPGAs and Siemens (Mentor Graphics) Catapult HLS [53] for ASIC design. HLS-based methods are a popular solution for developing deep learning accelerators from high-level DNN model specifications [57–60] and have also been investigated for generating FFT hardware IP [61].

Hardware Generation Tools. Another approach to building domain-specific hardware is to design tools for hardware generation instead of single hardware instances. In contrast to HLS, developing hardware generators still requires an understanding of microarchitectural-level details to construct the design. One of the popular approaches used in both industry and academia is to build generators using high-level programming languages embedded in Scala or Python, such as Chisel [62], SpinalHDL [63], and Magma [64]. These hardware description languages (HDLs) provide hardware primitives which the designer uses to describe the hardware but enables one to use features of a high-level software programming language such as object-oriented programming, functional programming, and polymorphism to design highly parametrized, reusable generators. Users pass a set of compile-time parameters to the generator which then goes through its own compiler passes and optimizations before producing RTL-level descriptions. Some examples include generators for coarse-grained reconfigurable arrays (CGRAs) [65], CPU cores and accelerators [66–70], and embedded FPGA (eFPGA) fabrics [71].

The SPIRAL [32] system for hardware generation takes a code generation

approach starting from a high-level problem specification, then uses algorithmic breakdowns and multi-level DSLs to construct the hardware. It has previously been developed for generating streaming FFT hardware [37] and approaches for more flexible hardware generation [39] of different algorithms such as bitonic sorters and Walsh-Hadamard transforms. The approach for automating hardware design in this thesis builds upon the strengths of the SPIRAL approach and high-level hardware description languages. Complex datapaths that are challenging to manually design without deep domain knowledge are almost directly extracted from SPIRAL internal representations. A highly parametrized, architectural template of the accelerator developed in Chisel automatically instantiates these datapaths, enabling fast design times for realizing different hardware design points.

Accelerator Programming. Programming specialized hardware is a notorious pain point in adopting new domain-specific architectures. The goal of recent work in this area is to achieve the energy efficiency and performance of using heterogeneous hardware while abstracting away underlying hardware details from the application developer. Domain-specific languages and compilers [32, 72–76] improve programmer productivity by providing abstractions and constructs that enhance the expressability of computations, then automatically optimize those expressions and generate code to target different hardware platforms. New tools have been developed for deep learning, image processing, graph computations, and numerical kernels, where targeting various specialized architectures for those domains. There are also compiler approaches that specifically target FFT accelerators from legacy FFT programs [77]. The Fourier ACcelerator Compiler (FACC) adapter approach uses a generate and test system to match user code to accelerator APIs.

1.3 Proposed Approach

We propose a new paradigm for designing the next generation of FFT hardware that addresses the need for flexibility, programmability, and energy efficiency in accelerating high performance software-driven FFT applications. We observe that in FFTW [25], the de facto standard software library for FFTs, flexibility arises from recursion and that the FFT computation can be pushed to the leaves of the recursion. Thus to retain this flexibility we propose the idea of hardware accelerating only the base cases of the recursion, known as *codelets*, while the composition is still defined in software. This further allows the hardware to be exposed to the user through the standard FFTW API and therefore remain transparent to the application developer. We demonstrate the approach through a flexible architectural template, software stack, and a proof-of-concept automated design framework to build the accelerator. We then show how we can apply the approach to different applications by extending the hardware generator to support new architectural features. The design tools are used to build silicon testchips that validate the methodology and demonstrate gains in energy efficiency and performance.

1.4 Contributions

The focus of this thesis is on the design paradigms that enable new domain-specific FFT hardware for software-driven high performance computing or high performance embedded computing, and the tools necessary to realize the accelerators. Following these principles, it presents an end-to-end design methodology and new design tools to program, build, and integrate the accelerators into real systems. This thesis makes the following contributions:

- Demonstrates a flexible FFT hardware architecture based on accelerating FFTW-style codelets and shows how the flexibility can also be leveraged for realizing common DFT properties.
- Provides a design framework for automatically building and integrating the FFT-based hardware accelerators with application software. A hardware generator automatically produces accelerator instances that are targeted through a software stack co-designed as a drop-in replacement.
- Demonstrates the end-to-end approach for a few example FFT software programs through performance modeling.
- Presents extensions that expand the scope of the design framework and capabilities of the generated hardware accelerators.
- Builds silicon testchips that are showcase the design paradigms. Design tools are used for hardware development, verification, and testing while the software tools enable programmability of the flexible accelerators in full system-on-chip designs.

1.5 Thesis Outline

The thesis is organized as follows.

Chapter 2 first presents the relevant mathematical background on the Discrete Fourier Transform and properties that can be exploited in FFT hardware design. It then introduces the idea of accelerating FFTW-style codelets as a new design paradigm for FFT hardware and shows how the approach retains software-like flexibility and programmability of the hardware. The microarchitectural details

of a “hardened” codelet datapath highlight the necessary features to support the various codelet types of FFTW. Finally, it presents the first silicon test chip of a SPIRAL-generated hardware instance implemented in a 28nm process, showing a first look at how SPIRAL can be used in the ASIC design process.

Chapter 3 presents the end-to-end design automation framework, from software to hardware, to build and program the accelerators. It first details the hardware components: a hardware codelet generator and an architectural template of the accelerator implemented using the Chisel HDL. Next, it discusses the software stack that bridges application software to custom hardware, where accelerator programs can be automatically generated. A second test chip implemented in a 28nm process is designed, programmed, and characterized using the new design tools.

Chapter 4 discusses how the hardware framework is extended to applications in fault tolerant computing, digital signal processing, and high performance computing. While the underlying design principles remain the same, additional hardware features such as non-standard ALUs and custom hardware codelets become beneficial in these domains. This chapter highlights the different ways in which the design methodology can be tailored towards new applications.

Chapter 5 discusses the LibraryX-ASIC framework for hiding accelerator offload behind FFTW library calls and shows how the program executes on the accelerator. It then presents some code examples targeting the accelerator and an evaluation through performance modeling.

Chapter 6 concludes with final remarks and future directions.

Chapter 2

Accelerating FFTW Codelets

In this chapter we begin with a discussion on the Discrete Fourier Transform and its implementation using the fast Fourier Transform algorithms. We also present its extension to multi-dimensional DFTs and some of the key properties that can be exploited when designing FFT hardware. We then introduce the idea of “hardening” FFTW-style codelets and discuss the design of an initial silicon prototype based on this idea. Finally, we conclude with a detailed look at our proposed microarchitecture of a hardware codelet.

2.1 The Discrete Fourier Transform

The Discrete Fourier Transform (DFT) is the critical mathematical transformation between time and frequency domain representations of finite-length, discrete data that is used in many applications. The DFT is defined by the following formula [8]:

$$X[k] = \sum_{n=0}^{N-1} x[n]e^{-j2\pi kn/N}, \quad (2.1)$$

where $x[n]$ is a sequence of N complex inputs that is transformed to a complex, length- N sequence, $X[k]$. The complex roots of unity, $e^{-j2\pi kn/N}$, are commonly referred to as “twiddle factors” in the context of the DFT.

Fast Fourier Transform Algorithms. Computing a length- N DFT using Equation (2.1) results in a computational complexity of $\mathcal{O}(N^2)$. In practice, computing the DFT is often implemented using the fast Fourier Transform (FFT) algorithms to reduce the complexity to $\mathcal{O}(N \log N)$. The most well known and commonly used FFT algorithm is the Cooley-Tukey algorithm [78], which recursively decomposes the computation to smaller sized FFTs when N is not prime. FFT algorithms have since been extensively studied, resulting in many variants that optimize for arbitrary input size or different compute platforms and application requirements.

2.1.1 Multi-dimensional DFT

The 1D DFT expressed in Eq. 2.1 can be extended to the multi-dimensional DFT. Given a multi-dimensional sequence $x[n_0, \dots, n_{d-1}]$ of size $N_0 \times N_1 \times \dots \times N_{d-1}$, its d -dimensional transform is expressed as

$$X[k_0, \dots, k_{d-1}] = \sum_{n_0=0}^{N_0-1} \cdots \sum_{n_{d-1}=0}^{N_{d-1}-1} x[n_0, \dots, n_{d-1}] e^{-j2\pi \left(\frac{k_0 n_0}{N_0} + \dots + \frac{k_{d-1} n_{d-1}}{N_{d-1}} \right)} \quad (2.2)$$

The multidimensional DFT is separable, meaning that the higher dimensional transform can be broken down into N^{d-1} 1D DFTs over every dimension where the order of the dimensions does not matter. This is more easily seen once the

complex exponential of the sum is broken up:

$$e^{-j2\pi\left(\frac{k_0 n_0}{N_0} + \dots + \frac{k_{d-1} n_{d-1}}{N_{d-1}}\right)} = e^{-j2\pi\left(\frac{k_0 n_0}{N_0}\right)} \dots e^{-j2\pi\left(\frac{k_{d-1} n_{d-1}}{N_{d-1}}\right)} \quad (2.3)$$

Now the complex exponential and summations in Eq. 2.2 can be grouped or interchanged in different ways. This leads to a variety of algorithms and performance implications when implementing higher dimensional DFTs.

2.1.2 Convolution

A well known property of the DFT is that the circular convolution in one domain becomes multiplication in the other. This property is used in a number of different applications to reduce the computational complexity of the convolution, such as spectral methods for solving PDEs, large integer multiplication [79], and wireless communications. The circular convolution between two length N sequences, $\mathbf{x} = x[n]$ and $\mathbf{h} = h[n]$, can be written in matrix form as follows

$$\mathbf{y} = C_h \mathbf{x}, \quad (2.4)$$

where C_h is a circulant matrix defined as:

$$C_h = \begin{bmatrix} h[0] & h[N-1] & h[N-2] & \dots & h[1] \\ h[1] & h[0] & h[N-1] & \dots & h[2] \\ h[2] & h[1] & h[0] & \dots & h[3] \\ \vdots & & & \ddots & \vdots \\ h[N-1] & h[N-2] & \dots & & h[0] \end{bmatrix}. \quad (2.5)$$

Each column of the circulant matrix, C_h , is a circular shift of the previous column and the matrix vector multiplication expresses the circular convolution. Now we show that the DFT matrix diagonalizes a circulant matrix, where the eigenvectors are the basis vectors of the DFT matrix formed by complex roots of unity. Taking the k -th DFT basis vector, $\mathbf{f}_k$,

$$(\mathbf{f}_k)_n = W_N^{-nk},$$

where $k \in [0, N-1]$ and $W_N = e^{-j2\pi/N}$ and applying the circular convolution, we have

$$(C_h \mathbf{f}_k)_n = \sum_{m=0}^{N-1} h[(n-m) \bmod N] W_N^{-km}. \quad (2.6)$$

Recognizing the circular shift nature of the indices, we now substitute $\ell = (n-m) \bmod N$ and $m = (n-\ell) \bmod N$,

$$\begin{aligned} (C_h \mathbf{f}_k)_n &= \sum_{\ell=0}^{N-1} h[\ell] W_N^{-k(n-\ell) \bmod N} \\ &= \sum_{\ell=0}^{N-1} h[\ell] W_N^{-k(n-\ell)} \\ &= \sum_{\ell=0}^{N-1} h[\ell] W_N^{k\ell} W_N^{-kn}. \end{aligned} \quad (2.7)$$

From here we can see that the term $\lambda_k = \sum_{\ell=0}^{N-1} h[\ell] W_N^{k\ell}$ is the k -th element of the DFT of h , which is an eigenvalue of the circulant matrix since $W_N^{-kn} = (\mathbf{f}_k)_n$:

$$\begin{aligned} (C_h \mathbf{f}_k)_n &= \left(\sum_{\ell=0}^{N-1} h[\ell] W_N^{k\ell} \right) \cdot W_N^{-kn} \\ &= \lambda_k (\mathbf{f}_k)_n. \end{aligned} \quad (2.8)$$

The eigenvectors of a circulant matrix are therefore the columns of the DFT matrix, so the DFT matrix diagonalizes a circulant matrix and can be expressed as

$$C_h = F^{-1} \Lambda_h F \quad (2.9)$$

$$FC_h = \Lambda_h F \quad (2.10)$$

where $\Lambda_h = \text{diag}(H[0], H[1], \dots, H[N-1])$ is a diagonal matrix containing the DFT of h . The circular convolution of a vector x with h is then

$$F(C_h x) = \Lambda_h Fx = \Lambda_h X, \quad (2.11)$$

which is the property that the convolution of h and x , $C_h x$, in the time domain becomes an element-wise product in the frequency domain, $\Lambda_h X$.

This property applies directly to circular convolutions. For a linear convolution between a length N sequence $x[n]$ and a length M sequence $h[n]$ there is no circular wrap around in the computation:

$$y[k] = \sum_{n=0}^{N+M-2} h[n] x[k-n], \quad (2.12)$$

where $y[m]$ is of length $N + M - 1$. To realize the linear convolution with the

DFT, both sequences must be padded with zeroes to length $N + M - 1$, and then the circular convolution between the zero-padded sequences becomes equivalent to performing the linear convolution between $x[n]$ and $y[n]$.

2.1.3 Forward and Inverse Transforms

The forward and inverse DFT have the property that a forward DFT can actually be computed using an inverse DFT, and vice versa. This can be derived from the matrix vector form of the transforms. The DFT matrix, F , is a Vandermonde matrix where the elements are given by $F_{k,n} = e^{-j2\pi kn/N}$ for $k, n \in [0, N - 1]$. If F is the forward DFT matrix, let $\bar{F}$ be its conjugate. Then,

$$y = \bar{F}x = \overline{(Fx)} \quad (2.13)$$

The DFT matrix is unitary and symmetric, meaning that $F^{-1} = \bar{F}$. So the result above tells us that an inverse DFT can be calculated by simply conjugating the input, taking the forward transform, and conjugating the output.

Alternatively, we also show that an equivalent method is to swap the real and imaginary parts of the complex number before and after the forward transform. Suppose $z = a + bj$ is a complex number. Then the swap of the real and imaginary parts can be expressed mathematically as

$$j\bar{z} = j(a - bj) = b + aj. \quad (2.14)$$

Letting z_s denote the complex number with swapped real and imaginary parts, Eq. 2.14 can be rearranged to get $\bar{z} = -jz_s$. We can substitute this result back

into Eq. 2.13 and use linearity of the DFT:

$$y = \overline{(F(-jx_s))} = \overline{(-jFx_s)} = j\overline{(Fx_s)} \quad (2.15)$$

Since the operation j multiplied by the complex conjugate is equivalent to the swap, we have shown the result that the real and imaginary parts can be swapped before and after forward transform to compute the inverse.

2.1.4 Other properties of the DFT

Table 2.1 contains some of the other well known properties of the DFT.

Property	Time domain	Frequency domain
Linearity	$\alpha x[n] + \beta y[n]$	$\alpha X[k] + \beta Y[k]$
Time shift	$x[n - n_0]$	$e^{-j2\pi kn_0/N} X[k]$
Frequency shift	$e^{j2\pi nk_0/N} x[n]$	$X[k - k_0]$
Circular convolution	$(x \circledast y)[n]$	$X[k] \cdot Y[k]$
Multiplication	$x[n] \cdot y[n]$	$\frac{1}{N}(X \circledast Y)[k]$
Correlation	$(x \star y)[n]$	$X^*[k] \cdot Y[k]$
Conjugate symmetry	$x[n] \in \mathbb{R}$	$X[k] = X^*[-k]_N$

Table 2.1: Table of DFT properties between time and frequency domain.

Properties of the DFT present opportunities to design hardware that reuses much of the same building blocks, but can result in very different mathematical computations. For example, one common pattern is that element-wise multiplication in one domain, with complex exponentials or another sequence, maps to the convolution, correlation, or shift properties in the opposite domain. The following chapters discuss how we capture this insight and consider the tradeoffs between approaches when making architectural design decisions.

2.1.5 Kronecker Product Notation

The DFT can be expressed in another way using the Kronecker product, or tensor product, notation. The Kronecker product between an $n \times n$ matrix A and an $m \times m$ matrix B is defined as

$$A \otimes B = \begin{bmatrix} a_{0,0}B & a_{0,1}B & \cdots & a_{0,n-1}B \\ a_{1,0}B & a_{1,1}B & \cdots & a_{1,n-1}B \\ \vdots & \vdots & \ddots & \vdots \\ a_{m-1,0}B & a_{m-1,1}B & \cdots & a_{m-1,n-1}B \end{bmatrix} \quad (2.16)$$

where $a_{i,j}$ are elements of the matrix A and $a_{i,j}B$ is the scalar multiplication of $a_{i,j}$ with the matrix B . The resulting matrix is of size $mn \times mn$. SPIRAL [32] uses the Kronecker product to express different algorithmic breakdowns of the DFT. For example, the Cooley-Tukey FFT algorithm for a DFT of size n , where $n = mk$, can be written as

$$DFT_n = (DFT_m \otimes I_k) T_k^n (I_m \otimes DFT_k) L_m^n \quad (2.17)$$

where L_m^n is a stride permutation, I_m is the $m \times m$ identity matrix, and T is the twiddle factor matrix. The first step of the transform is the stride permutation, L , which is defined as

$$L_m^{mn} : in + j \mapsto jm + i, \quad 0 \leq i < m, 0 \leq j < n. \quad (2.18)$$

Although it is applied to a 1D vector, when the vector is viewed as a 2D $m \times n$ matrix L_m^{mn} can be interpreted as a transpose of the $m \times n$ matrix. The first

stage of DFTs is m blocks of size- k DFTs on contiguous vectors of size k . The results from the first stage are then scaled by the diagonal twiddle matrix, T_k^n . The last stage of the transform is k DFTs of size m where the subvectors are now interleaved rather than contiguous. Here, we also see that the DFT stages are of size k and m , so can be broken down recursively into smaller size butterflies.

Higher-dimensional DFTs can also be expressed using the Kronecker product. For example, the 2D and 3D DFTs are given by:

$$DFT_{n \times m} = DFT_n \otimes DFT_m \quad (2.19)$$

$$DFT_{k \times n \times m} = DFT_k \otimes DFT_n \otimes DFT_m \quad (2.20)$$

The formulas above can be transformed using tensor product identities to manipulate the algorithm they express and therefore how parallelism is exploited. The variety of mathematical transformations and breakdowns yield implementations that can be optimized for specific hardware platforms. The next section introduces some of these identities and algorithm variants.

2.1.6 Tensor Product Identities

Transformations on the mathematical expressions are based on tensor product identities which result in different algorithms. The following table shows some of the identities used to manipulate the DFT formulas. The table first describes some of the basic distributive and commutative identities. The second half of the table shows various stride identities and factorizations. The DFT formulas in Eq. 2.17 and 2.20 are transformed using these identities.

$(BC)^\top = C^\top B^\top$
$(A \otimes B)^\top = A^\top \otimes B^\top$
$I_{mn} = I_m \otimes I_n$
$A \otimes B = (A \otimes I_m)(I_n \otimes B)$
$I_n \otimes (BC) = (I_n \otimes B)(I_n \otimes C)$
$(BC) \otimes I_n = (B \otimes I_n)(C \otimes I_n)$
$A \otimes B = L_n^{mn}(B \otimes A)L_m^{mn}$
$(L_m^{mn})^{-1} = L_n^{mn}$
$L_n^{kmn} = (L_n^{kn} \otimes I_m)(I_k \otimes L_n^{mn})$
$L_{km}^{kmn} = (I_k \otimes L_m^{mn})(L_k^{kn} \otimes I_m)$
$L_k^{kmn} = L_{km}^{kmn} L_{kn}^{kmn}$

Table 2.2: Table of tensor product identities to manipulate DFT formulas.

2.1.7 Short Vector Algorithms

A variant of the Cooley-Tukey algorithm in Eq. 2.17 was derived for short vector length machines such as CPUs with SIMD extensions [80] [81]. First, the leftmost expression $\text{DFT}_m \otimes I_k$ of the Cooley-Tukey breakdown is already vectorized. It can be viewed as a single DFT_m on vectors of length k . The expression $I_m \otimes \text{DFT}_k$ is manipulated using the identities in Table 2.2 such that almost all computations are vectorized. Its formula is:

$$\begin{aligned} \text{DFT}_n = & \left((\text{DFT}_{k/\nu} \otimes I_{m/\nu}) \otimes I_\nu \right) T_m^n \left(I_{k/\nu} \otimes (L_m^{m\nu} \otimes I_\nu) \right. \\ & \left. \times (I_{m/\nu} \otimes L_\nu^{2\nu}) (\text{DFT}_m \otimes I_\nu) \right) \left(L_{k/\nu}^{n/\nu} \otimes I_\nu \right) \end{aligned} \quad (2.21)$$

where ν is the vector length and $n = km$. Only the stride permutation $L_\nu^{2\nu}$ is not vectorized as it represents a permutation within the vector. The short vector variant of the Cooley-Tukey algorithm packs complex data elements into the ν -way vector and all computations are now ν -way vector parallel. In SIMD machines,

parallelism is realized across space where a machine vector register stores the entire vector, but the approach here can be used to target specialized FFT hardware which also parallelize across time.

2.2 “Hardening” FFTW-style Codelets

The fundamental kernels of the FFTW [25] approach to computing FFTs are fixed, small size transforms known as *codelets*. A decomposition of the transform is then specified by the *plan* data structure which defines the composition of codelets and their respective parameters. The *executor* manages execution of the transform for a given input data vector by interpreting the plan and invoking codelets with the specified parameters.

The standard FFTW library consists of a collection of codelet types and sizes including powers-of-2 up to 64 and small primes (e.g. 3, 5, 7). Concretely, a size- p FFTW twiddle codelet performs element-wise scaling of inputs by twiddle factors and then computes an in-place size- p DFT. The no-twiddle codelet computes only an out-of-place size- p DFT. Each type of codelet operates on data at different input/output strides specified by the plan.

The adaptivity of the library to specific hardware platforms is supported through plans and recursive functions. The planner has the flexibility of composing codelets in different ways to compute FFTs of real/complex data and of arbitrary size or dimension. A code generator, **genfft**, automatically generates the codelets as straight-line C code. An overview of the FFTW library is depicted in Fig. 2.1.

The structure of an example FFTW application is shown in Fig. 2.2. The function **fftw_plan_dft_1d** constructs and stores the plan as a composition of codelets and the actual execution takes place at the **fftw_execute** function call.

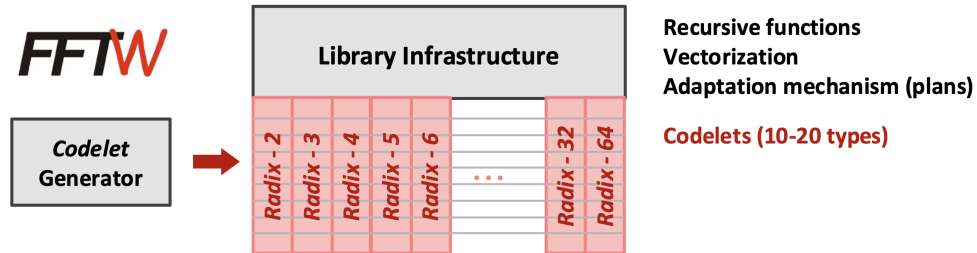


Figure 2.1: Overview of the FFTW library. Codelets are automatically generated pieces of C code that perform steps of the FFT computation at various small sizes. The library infrastructure implements data structures, recursive functions, and vectorization features that enable the flexible and adaptive nature of FFTW.

```

1  #include <fftw3.h>
2
3  int main() {
4      // Declare in/out arrays and plan
5      fftw_complex *in, *out;
6      fftw_plan p;
7
8      // Allocate/Init
9      in = (fftw_complex *) fftw_malloc(sizeof(fftw_complex) * N);
10     out = (fftw_complex *) fftw_malloc(sizeof(fftw_complex) * N);
11     ...
12
13     // Construct 1D complex FFT plan
14     p = fftw_plan_dft_1d(2*N, in, out, FFTW_ESTIMATE);
15
16     // Execute FFT plan
17     fftw_execute(p);
18
19     // Post-process and cleanup
20     ...
21     fftw_destroy_plan(p);
22     fftw_free(in), fftw_free(out);
23 }
```

Figure 2.2: Example FFTW program. The function `fftw_plan_dft_1d` constructs and stores the plan as a composition of codelets and the actual execution takes place at the `fftw_execute` function call.

We observe that once an FFTW codelet is invoked, there is no requirement that the computation takes place in software. The codelet can instead be accelerated in hardware with the actual composition of the codelets still defined in software. Thus accelerating only the kernel of FFTW enables a large degree of flexibility for the breakdown of the FFT computation.

A hardware accelerator for codelets can be exposed as a FFTW compatible API and can also be targeted by FFTX. While the FFTW API is still maintained to the user, a custom hardware accelerator carries out the actual execution of the codelets. The software infrastructure can be provided through a C library which looks like FFTW to effectively hide the dispatching of codelets in hardware. Then, the hardware is designed such that it is compatible with the software library and can interpret and execute codelet launches. Fig. 2.3 shows a software abstraction of how codelets are composed and then invoked in a 4-point FFT example. The first two launches of the hardware kernel computes two DFTs of size 2 with the respective base address and stride parameters. A memory fence is issued due to the data dependency and the final two kernel launches then complete the computation. The software attempts to batch up as many independent codelets as possible, but has the freedom to compose them in any way.

Fig. 2.4 then shows how that sequence of codelets would be stored in a specialized data structure and then executed by a separate function call.

Such an accelerator can be designed to concurrently process batched codelet invocations and integrated with a CPU that acts as a control processor. The proposed hardware design approach [82] can also be viewed analogously to the FFTW library structure shown in Fig. 2.1. A configurable codelet datapath should support the various codelet types and a hardware state machine manages

```

1  // Function executing 4-point DFT
2  void execute_dft4(_Complex double *X, _Complex double *Y) {
3
4      _Complex double U[4];
5      _Complex double T[1] = {__I__};
6
7      // Launch hardware kernels
8      dft2_kernel(U, X, NULL, 1, 2, 0);
9      dft2_kernel(U+2, X+1, T, 1, 2, 0);
10
11     // Memory Fence
12     #pragma pipeline fence
13
14     // Continue invoking kernels
15     dft2_kernel(Y, U, NULL, 1, 2, 0);
16     dft2_kernel(Y+1, U+1, NULL, 1, 2, 0);
17 }
18

```

Figure 2.3: Example of a 4-point FFT invoking hardware codelets.

```

1  // Data structure containing codelet parameters
2  descriptor_t dft4[] = {
3      {U, X, NULL, 1, 2, 0},
4      {U+2, X+1, T, 1, 2, 0},
5      FENCE,
6      {Y, U, NULL, 1, 2, 0},
7      {Y+1, U+1, NULL, 1, 2, 0},
8      DONE
9  };
10
11  // Execute plan
12  execute(dft4);

```

Figure 2.4: Example 4-point FFT codelet plan and execution call.

the execution of batched codelet invocations, depicted in Fig. 2.5.

We demonstrate a first look of this idea through a hardware accelerator test chip that executes the key computational kernel, a twiddle codelet. The accelerator processes independent fixed size FFT computations and twiddle factor multiplica-

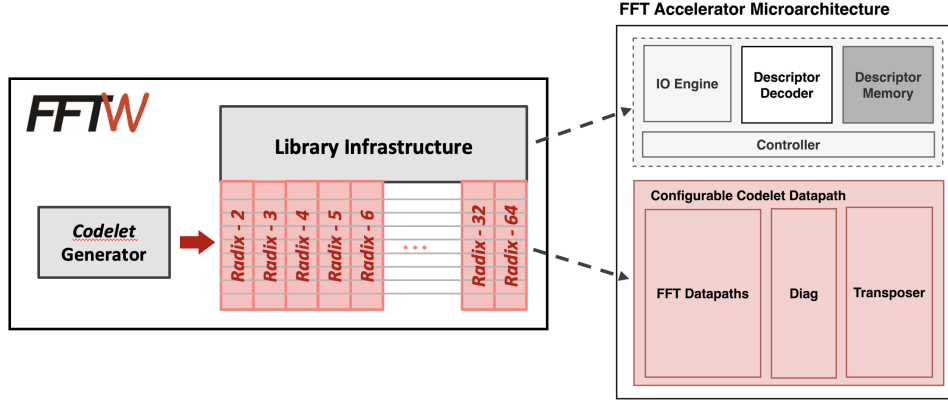


Figure 2.5: Comparison of the FFTW approach to the proposed hardware codelet approach.

tions in a pipelined fashion.

2.3 A First Radix-8 Codelet Test Chip

An initial test chip of an FFTW codelet accelerator is designed and taped out in a TSMC 28nm process. The core computation accelerates a radix-8 twiddle codelet, consisting of a fixed radix-8 FFT followed by eight general twiddle factor multiplications. In this section we first discuss how SPIRAL is integrated into the ASIC design flow to co-design the hardware. We then detail the function and microarchitecture of each of the main components of the test chip.

Hardware Generation. To facilitate design of the FFT hardware, we use SPIRAL to automatically generate the RTL description for a radix-8 complex FFT. The design flow proceeds as follows and is illustrated in Fig. 2.6.

SPIRAL uses a script that first defines the parameters for the generated FFT hardware. Within the SPIRAL system an algorithm is chosen and the problem is

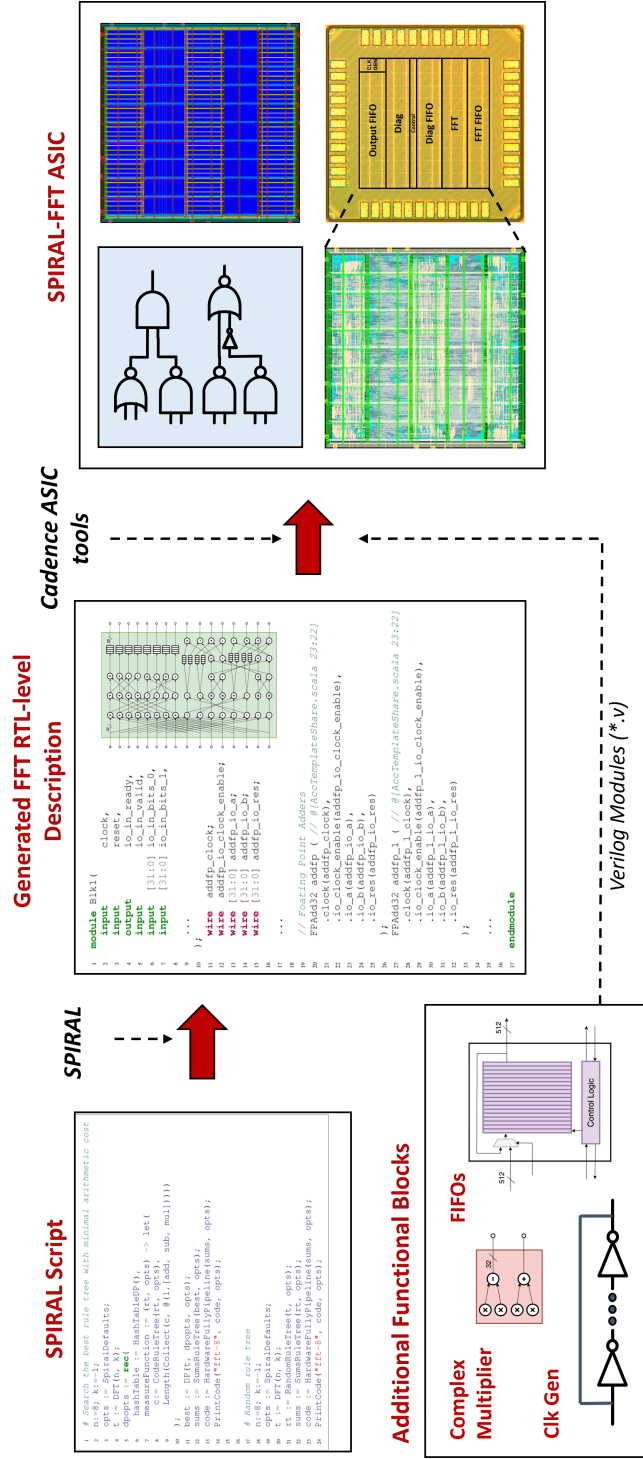


Figure 2.6: Overview of the test chip design flow from SPIRAL script to generated RTL code to physical implementation. The SPIRAL script defines the parameters of the generated RTL-level description of the FFT hardware. Other functional blocks are hand designed and integrated with the generated code which then goes through the ASIC design flow.

2.3.1 Radix-8 FFT Architecture

The SPIRAL-generated FFT hardware is a fully unrolled radix-8 architecture, giving the highest possible throughput at one full vector of outputs per cycle. Additionally, the design uses single-precision floating point arithmetic to support a wider variety of scientific applications which may have different dynamic ranges.

As shown in Fig. 2.8, the FFT hardware implements exactly the signal flow diagram of an 8-point FFT where every arithmetic operation is mapped to a dedicated adder/multiplier. The dataflow is also optimized such that the multipliers in the FFT unit are constant multipliers with one fixed input since the twiddle factors within the radix-8 FFT are fixed constants. Pipeline registers are automatically inserted to ensure that each set of FFT computations remains aligned in the datapath. Each floating point adder is further pipelined to three stages: one for the exponent comparison, another for the mantissa addition, and a final stage for normalization. The floating point multiplier is pipelined to two stages where one stage performs the mantissa multiplication and the other completes the normalization stage. The radix-8 FFT is aggressively pipelined to 13 stages to increase throughput of the accelerator.

2.3.2 Diag Unit

The eight complex outputs of the FFT are fed directly to the Diag unit, which consists of eight complex multipliers to perform all complex twiddle multiplications in parallel. General twiddle multiplication at the end of the fixed 8-point FFT forms the FFTW equivalent of a “twiddle codelet”. A complex multiplier is implemented using four floating point multipliers and two floating point adders. The design of each adder is exactly the same as those in the FFT core, however

the multiplier is instead a two-input single-precision multiplier since the twiddle factors are not fixed constants in the Diag unit. All complex multiplies occur in parallel to maintain steady-state pipeline, the Diag unit is pipelined to a total of 5 stages.

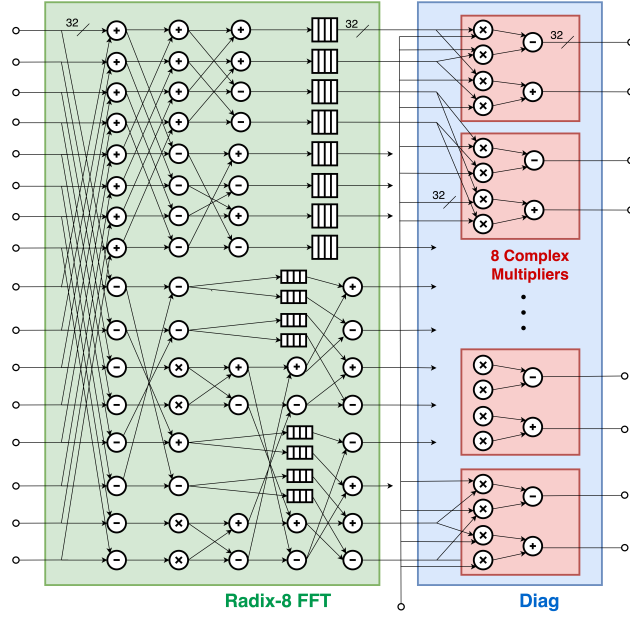


Figure 2.8: Internal architecture of the Spiral-generated radix-8 FFT and Diag complex multiplier block. The FFT unit uses single-precision adders and constant multipliers. The Diag unit has 8 complex multipliers, where each complex multiplier has four multipliers and two adders.

2.3.3 Memory

In order to sustain steady-state pipeline operation, we design 24-stage shift register FIFOs which directly input and capture complex single-precision numbers to and from the FFT/Diag units every cycle. One set of FIFOs holds the input FFT data, another holds complex twiddle factors, and the final set holds the output

data coming from the Diag block. The input FIFOs are designed so that their outputs can loop back to the input so that the system continuously run. The design contains a total of 4.5 kB of FIFO memory which is implemented as synthesized memories. The on-chip memory is designed in this way so that we can demonstrate the achievable throughput performance of the hardware kernel.

2.3.4 Control Unit

The control unit processes incoming data from off-chip through an asynchronous I/O protocol to load in test vectors to the FIFOs. We load 4-bit nibbles which the control unit then assembles together to form the 32-bit FP numbers which populate the FIFOs. The load-in data includes a set of bits allocated as metadata to inform the control unit which mode to operate the chip in. The FFT/Diag core can be configured to run in “compute” mode to run through one full set of test vectors in the input FIFOs and write to the output FIFO, or “looping” mode which continually runs the test inputs through the FFT/Diag units to obtain power measurements.

2.3.5 Dataflow

Fig. 2.9 shows the how data flows through the twiddle codelet accelerator and a detailed timing diagram. Independent batches of codelet computations flow through the FFT and Diag block every cycle. An output FIFO captures each batch of radix-8 twiddle codelet where the Diag input data is sequenced such that it aligns with the incoming FFT computation.

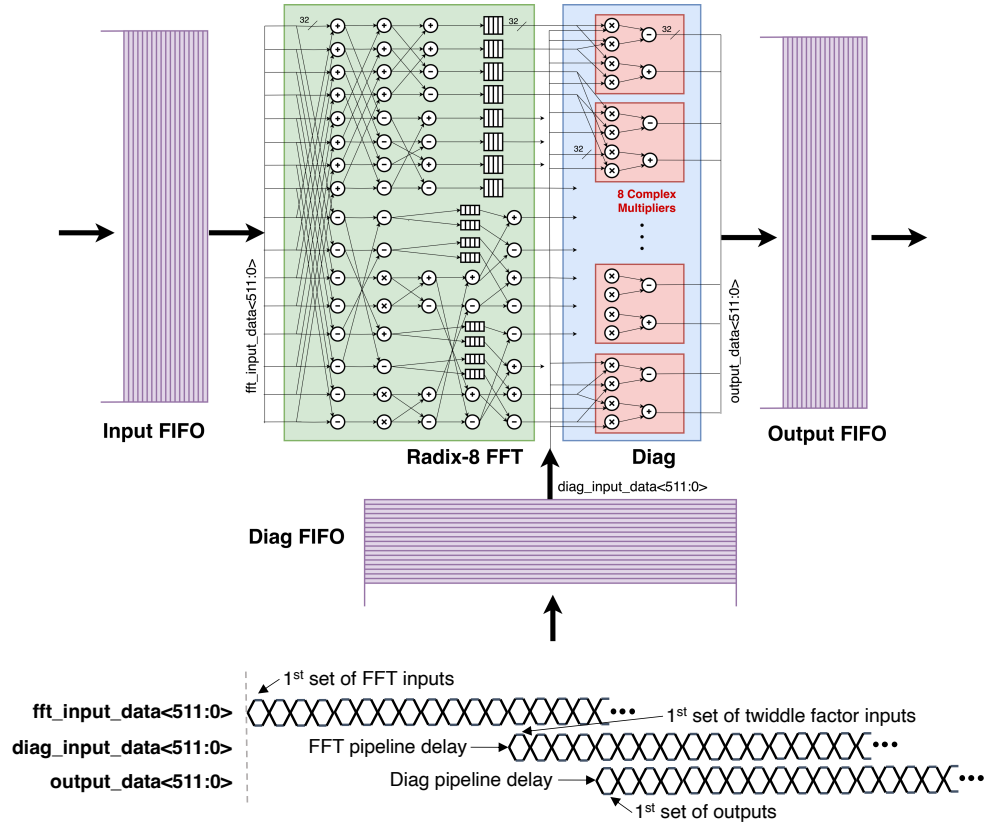


Figure 2.9: Timing diagram of the dataflow through the FFT and Diag compute cores.

2.3.6 Measured Results

The FFT accelerator test chip is fabricated in a TSMC 28nm process and has a core size of 1.2mm x 0.7mm. The radix-8 FFT core occupies 0.118mm² and the Diag unit occupies 0.151mm², with the remaining area occupied by FIFO memory and control/test infrastructure. Figure 2.11b shows a more detailed area breakdown by functional block. Since the design is a fully unrolled architecture, it achieves a throughput of $8 \times f_{clk}$ complex samples per second.

Functionality is verified with test vectors consisting of 24 sets of eight complex

single-precision floating point data and eight complex twiddle factors to completely fill up the input FIFOs. After running the computation we read out the contents of the output FIFO from the test chip and compare against the corresponding output obtained from RTL simulation. A micrograph of the die is shown in Fig. 2.10 and the chip implementation details are summarized in Table 2.3.

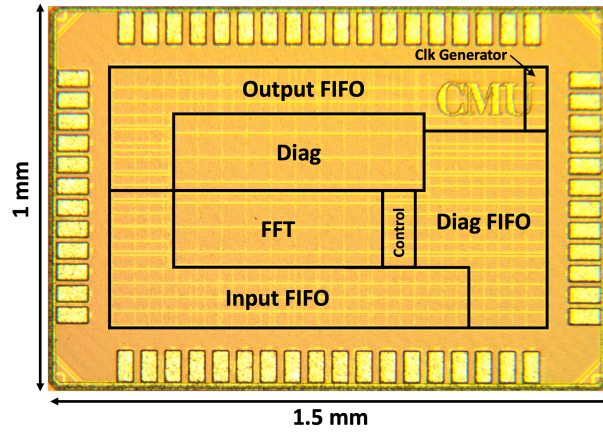


Figure 2.10: Die micrograph of 1.5mm $\times$ 1mm Spiral-FFT test chip in TSMC 28nm. The chip has 60 I/O pads and a core area of 1.2mm $\times$ 0.7mm. FIFO memory holds a total of 4.5 kB of data.

At nominal 0.9V the chip runs at 260 MHz and consumes a total of 126 mW when computing 8-point FFTs and twiddle factor multiplications at a throughput of 2.08 Gsamples/sec. The FFT and Diag units consume 75 mW and the FIFOs and clock generator consume 51 mW. A detailed breakdown of the measured dynamic power consumption is also shown in Fig. ??a.

Across a core voltage range of 0.65V-1.15V, the chip successfully operates at 155-270MHz. The Shmoo plot in Fig. 2.12 demonstrates the full voltage-frequency operating points at which the test chip is functional. This marks the first silicon

Technology	TSMC 28nm
Core Supply	0.9 V
Chip Area	1.5 mm ²
Core Area	0.84 mm ²
Frequency	455-1188 MHz
I/O	60
Memory	4.5 KB (FIFOs)
Power	472 mW (at 1V)
Throughput	3.8-9.52 GS/s
FFT8 Latency	15.1-39.5 ns

Table 2.3: Implementation Summary

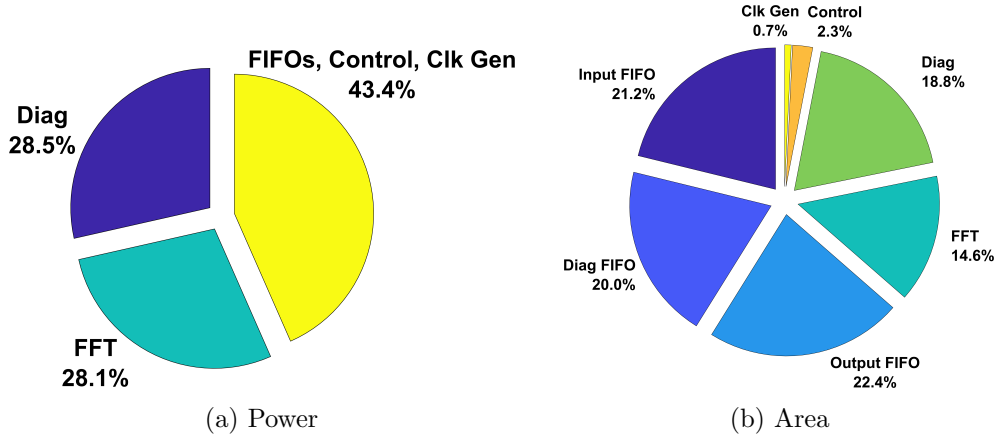


Figure 2.11: Breakdown of power consumption at 0.9V and area by block

tapeout in which SPIRAL is used as a part of the design flow. In the following section we discuss how the core of this accelerator is used as a key component in designing the complete configurable hardware codelet.

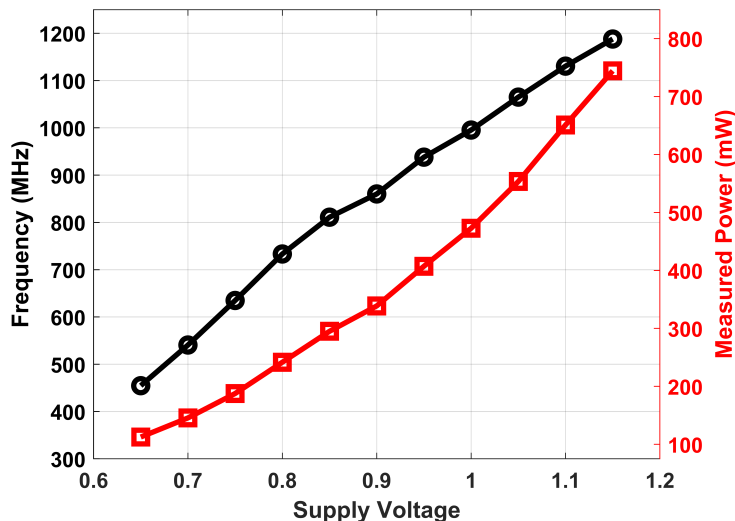


Figure 2.12: Shmoo plot of SPIRAL-FFT accelerator. Green areas represent the points at which the chip is functional. At nominal 0.9V the chip runs at 260 MHz and at a supply range of 0.65-1.15V the chip operates at 155-270MHz.

2.4 The Hardware Codelet Microarchitecture

The hardware-accelerated codelet must support the flexibility of executing different small size FFTs and the choice of computing twiddle or no-twiddle codelets. Thus we design a reconfigurable codelet datapath that supports a fixed set of small size FFTs and complex element-wise multiplications. Furthermore, a dedicated hardware transposer is required for permuting inputs/outputs within batches of codelet invocations. A special case transpose codelet was also developed in FFTW [25].

An overview of the the configurable hardware datapath is depicted in Fig. 2.13. Registers are placed at the beginning and end of every block to pipeline between memory accesses. In between each functional block is a set of muxes that allow the operations in the datapath to be reordered or skipped. The values set in the

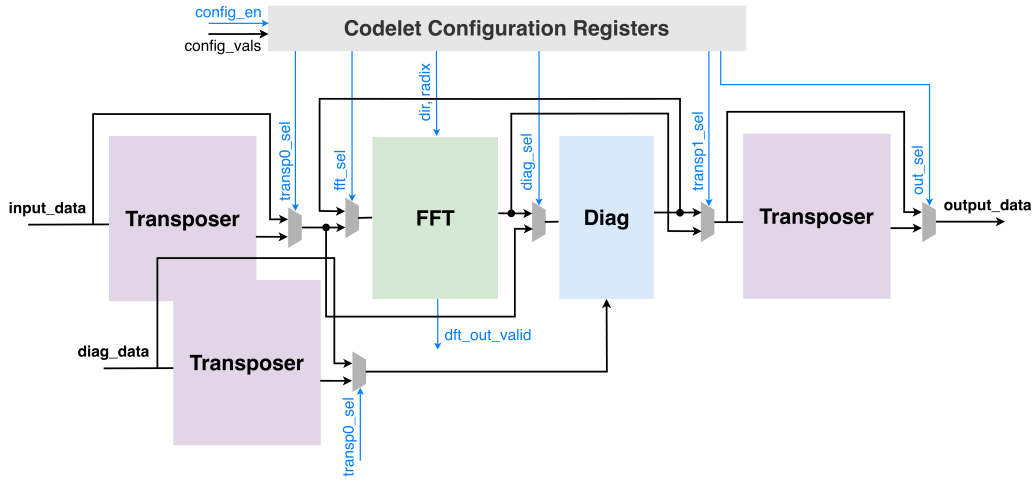


Figure 2.13: Configurable datapath implementing the equivalent of an FFTW codelet. The datapath is reconfigured during runtime based on the plan specification, which sets the values of the configuration registers.

configuration registers control the mux selects and are fixed for the entire duration of a batch codelet invocation. The entire accelerator design is written in the Chisel HDL [62] to generate Verilog RTL for various codelet sizes, data types, accelerator interfaces, or memory sizes. The following sections provide microarchitectural details of each of the main components and the supported reconfigurations of the datapath.

2.4.1 FFT Datapaths

The *FFT* block in Fig. 2.13 represents a collection of different complex FFT datapaths which must be decided at design time. Each small sized radix- r FFT is an independent hardware instance and the choice of which size is operating is based on the current radix value set in the configuration register. Once the user selected codelet sizes are instantiated, the accelerator will support only those sizes in hardware and the configuration register must be set accordingly. Similar to

FFTW, the collection of hardware codelets are automatically generated and the design flow will be discussed in more detail in Chapter 3.

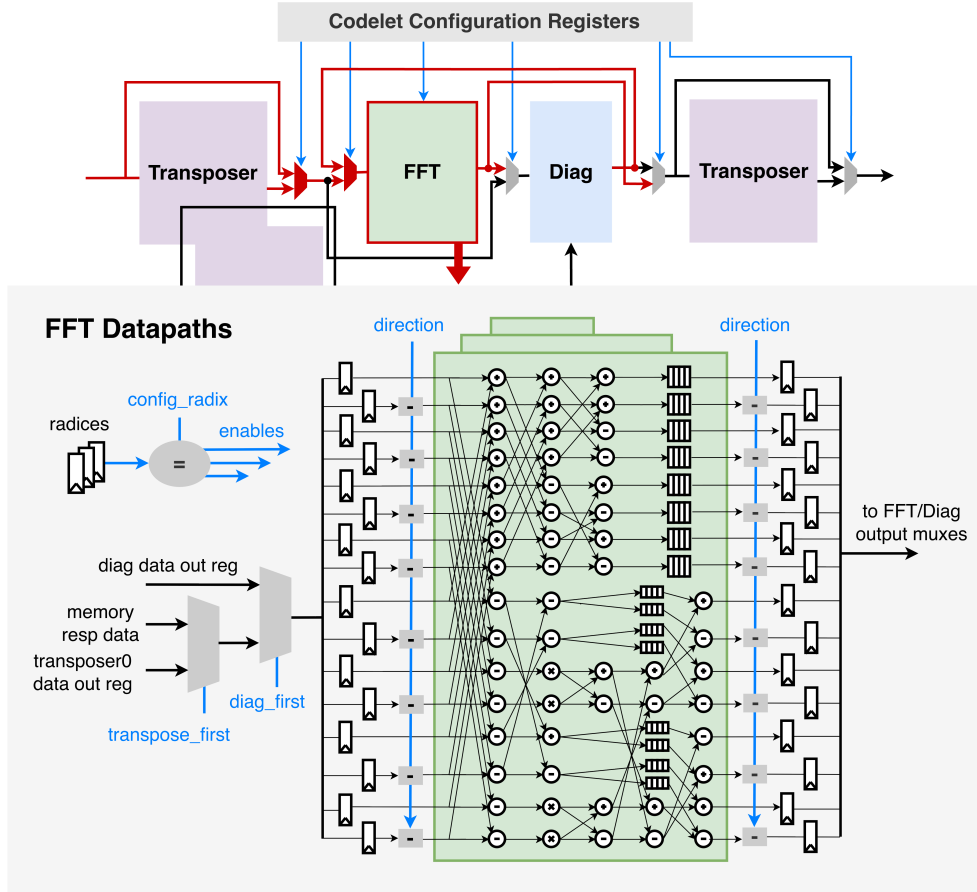


Figure 2.14: FFT datapaths.

Fig. 2.14 shows a detailed example of a radix-8 FFT where the other green blocks represent different FFT sizes. The values of the supported radices are contained in a bank of registers, and a comparison against the currently configured radix generates the control signals that enable only the desired radix datapath. The highlighted paths in red show how the FFT can take place after transposing,

without transposing, together with the Diag, or completely on its own as well. In the case the Diag is computed, the FFT can take place before or after it. A more complete discussion on the possible configurations of the hardened codelet is in Sec. 2.4.4.

Forward and Inverse Transforms. The *direction* signal controls whether a forward or inverse FFT is performed. We use the DFT property derived in Sec. 2.1.3 which says that an inverse transform can be performed via a forward transform by conjugating the inputs, taking the forward transform, and conjugating the outputs. This is relatively simple to realize in hardware by negating the complex input/output elements. For floating-point numbers it is just a mux that selects between the sign bit or its inversion. The other strategy is to swap real and complex elements on the way in and out of the kernel. This approach requires a mux for every input/output real and imaginary part, but could be more efficient for datatypes where negation is not as simple as flipping a sign bit.

Mixed Radix Support. Supporting multiple FFT radices instantiates separate hardware blocks for each required radix. In the case that a smaller size butterfly can evenly divide the larger radix, incoming data coming from all input lanes can be sequenced to the the smaller butterfly. Otherwise we assume that only the first r inputs to the codelet will be used. Other approaches [41, 43] explore how the existing butterfly structure of larger sizes can be reused to simultaneously support smaller sizes. For example, both radix-2 and radix-4 butterflies exist as part of the signal flow for a radix-8 butterfly. However previous work [43] has shown that going from constant multipliers in separate butterflies to general multipliers with mux logic incurs enough overhead that the two approaches become almost comparable. The reconfigurable butterfly requires multiplier optimizations [42] to

realize more improved area efficiency.

2.4.2 Diagonal

The *Diagonal* (*Diag*) block performs a diagonal matrix multiply using a set of element-wise complex multipliers as large as the maximum FFT radix size chosen. The element-wise multiplication can take place before or after the FFTs, or in the case of the no-twiddle codelet, omitted from the computation as well. We implement the complex multiplier using four multipliers and two adders, shown in Fig. 2.15. Registers are inserted before and after the multipliers to pipeline between memory accesses.

A common optimization technique for FFT hardware is to remove the first element-wise multiplier since for each butterfly computation, the first twiddle factor multiplication is with a value of 1. However, we leave the first multiplier here so that we can perform element-wise operations that occur between DFT stages, a common pattern seen in FFT-based algorithms such as convolutions or correlations. In these type of cross-stage FFT-based computations, it is not necessarily the case that the complex multiplication takes place with twiddle factors (roots of unity) since the second set of inputs need to be treated as another set of arbitrary complex values.

Alternatively, one could also implement the complex multiplier using three multipliers and five adders in the case that multipliers are costly to implement. Suppose we have two complex numbers, $x = a + bi$ and $y = c + di$. Then, the

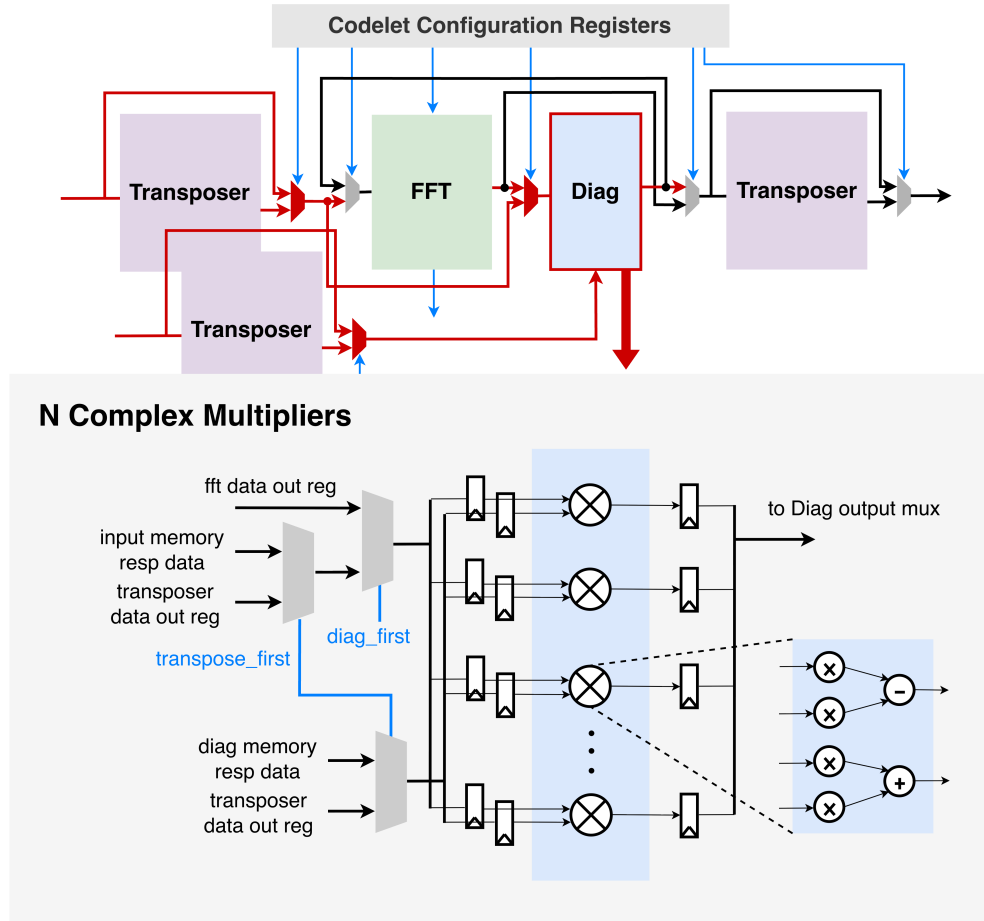


Figure 2.15: Diagonal Block

product, $z = xy$, is:

$$\text{Re}(z) = ac - bd \quad (2.22)$$

$$\text{Im}(z) = (a + b)(c + d) - ac - bd \quad (2.23)$$

and it becomes clear that the number of multipliers required is now reduced by one at the cost of three additional adders. The technique has been used in

area/resource-constrained ASIC and FPGA designs where multipliers are costly to realize.

2.4.3 Transposer

The *Transposer* is implemented in a systolic-array fashion in which “column” vectors of data are loaded in parallel over successive cycles, but then written back to memory in a “row”-wise fashion. Since codelet invocations are parallelized over time, this performs small block transposes within the batch. To maintain throughput over every cycle we duplicate the systolic arrays and ping-pong between loading data into one and loading data out of the other. The codelet pipeline provides the option of performing the transpose before the computation, after the computation, or both before and after the computation.

Fig. 2.16 shows the microarchitectural details of the transposer. The size of the array is again determined by the largest supported radix size in the FFT datapath. The transpose unit placed after the computation must first select between FFT or Diag outputs. As data flows out of the array, the codelet outputs are selected from one of the two ping-pong arrays. Finally a set of muxes determine whether to select transposed or non-transposed outputs. In the case that the codelet is configured for non-transposed outputs, the transposer is disabled to prevent unnecessary switching of the array registers.

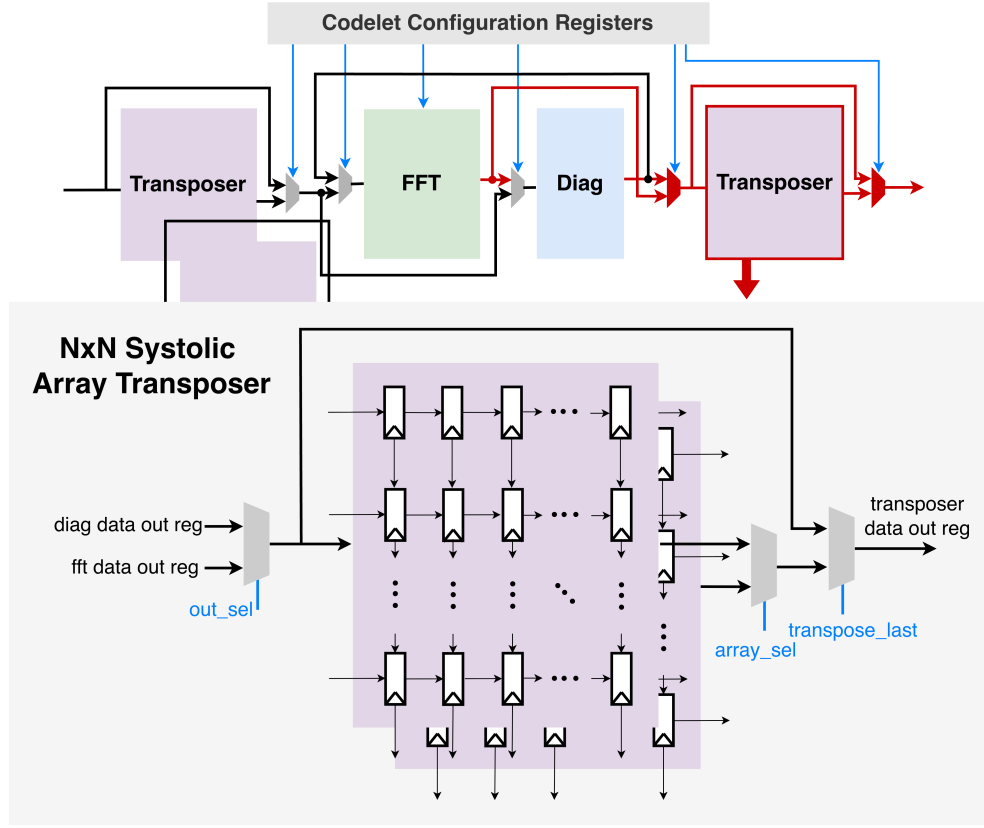


Figure 2.16: Systolic Array Transposer

2.4.4 Supported Datapath Configurations

The codelet configuration registers control the choice of FFT radix, forward or inverse transform, and the ordering of the FFT/Diag/Transposer blocks. The supported mappings from configuration values to hardware is detailed in Table 2.4 and also pictorially in Fig. 2.17.

At design time, the radices to support in the FFT datapath are first determined. For each radix, the datapaths are then fully unrolled and instantiated separately where the *radix* configuration register selects which butterfly is enabled for computation. Depending on the banking scheme of the memory subsystem,

Configuration Register				Datapath
Diag	Diag First	Transpose First	Transpose Last	
0	X	0	0	FFT
0	X	1	1	Transp, FFT
0	X	1	1	FFT, Transp
0	X	1	1	Transp, FFT, Transp
1	0	0	0	FFT, Diag
1	0	0	0	FFT, Diag, Transp
1	0	0	0	Transp, FFT, Diag
1	0	0	0	Transp, FFT, Diag, Transp
1	1	0	0	Diag, FFT
1	0	1	0	Diag, FFT, Transpose
1	1	1	0	Transp, Diag, FFT
1	0	1	1	Transp, Diag, FFT, Transp

Table 2.4: Supported mappings from configuration registers to datapath computation.

we may also choose to instantiate as many smaller radix butterflies to match the supported IO bandwidth.

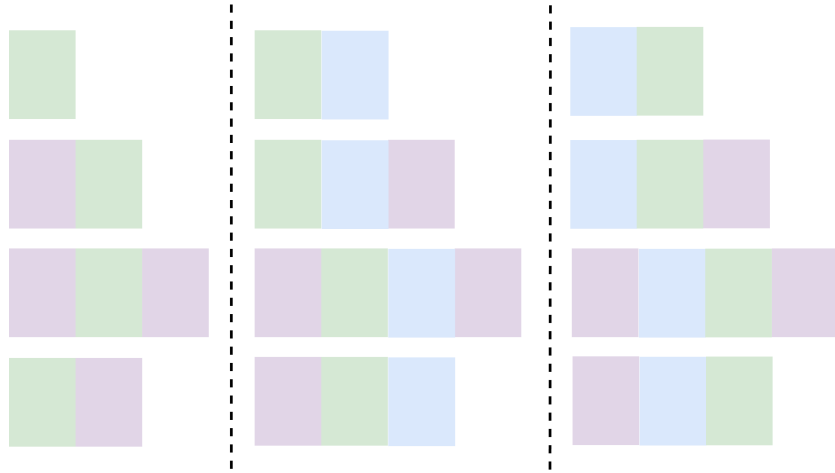


Figure 2.17: A visual representation of the arrangements of the reconfigurable codelet. The green FFT datapaths can be reconfigured for different sizes or forward/inverse transforms. The Diag is shown in blue and transpose units are shown in purple.

Chapter 3

A Design Automation Framework

Design tools are crucial to realize the new paradigms and build the accelerators. Similar to FFTW, where a code generation tool was needed to generate software codelets, the analogous hardware generation tools are needed for design automation. In this chapter we first discuss the design automation framework for generating the codelets and the accelerator instantiating the codelets. Then, we discuss the software tools developed to program it and walk through some examples that show how we can hide the accelerator behind an FFTW library call. To validate both the software and hardware design tools, we present a silicon testchip that implements our generated RTL design and executes custom accelerator programs. Finally, we conclude by showing how the accelerator can be integrated into a RISC-V system and coupled to a control processor.

3.1 An Architectural Template for Hardened Codelets

We design the accelerator to fit in a high level machine model consisting of (1) a host CPU coupled to the accelerator, (2) the accelerator which interfaces to fast on-chip local memory, and (3) main memory with a mechanism for data transfer, as depicted in Fig. 5.2.

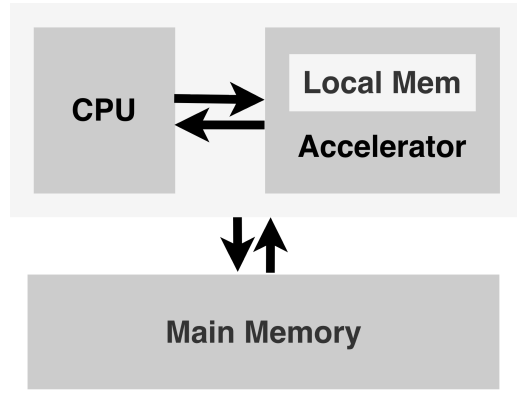


Figure 3.1: High-level machine model. The accelerator interfaces to its own local memory and is coupled a controlling CPU. The system has a mechanism for data transfer to and from off-chip main memory.

We also design the necessary architectural components such that the accelerator is compatible with the software library and can interpret and execute codelet launches. Finally, we present the design automation tools needed to automatically generate and instantiate hardened codelets in an accelerator through a hardware generator written in Chisel.

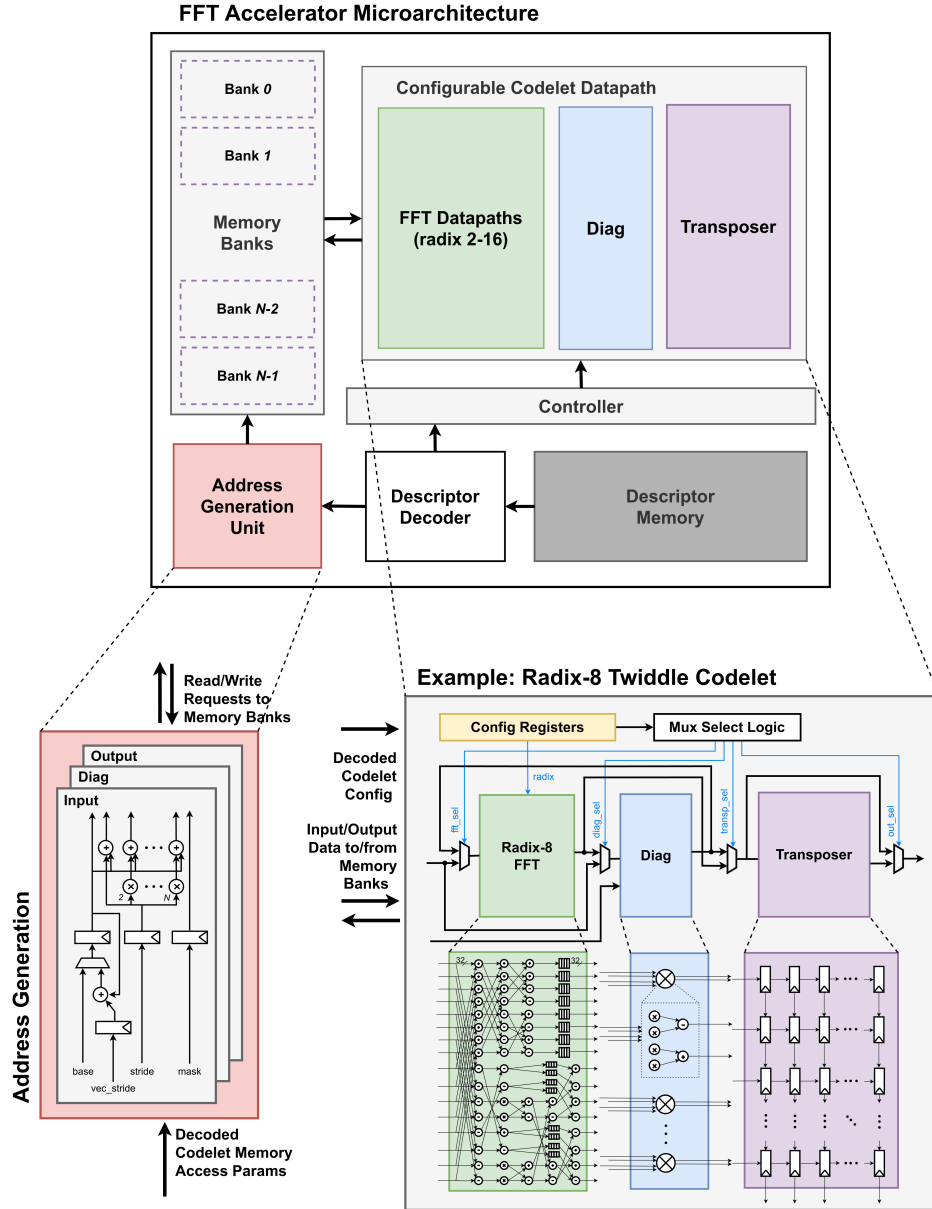


Figure 3.2: FFT accelerator architectural template. The decoder interprets descriptors specified by the accelerator program to set the codelet configuration and fetch data according to the IO base address/stride patterns. The codelet datapath moves data to/from banked scratchpad memory.

3.1.1 Architecture Overview

The overall accelerator architecture is illustrated in Fig. 3.2. At the core of the accelerator is the configurable hardware codelet, deeply pipelined to support concurrent execution of batched codelets. The accelerator operates in a manner similar to that of a long vector machine in which computation is parallelized over time, but instead small size FFTs are batched and concurrently executed in the codelet pipeline. Thus parallelism is actually exploited over space and time. The hardware codelet interfaces to fast on-chip memory which can be realized in different ways. In this work, we implement the on-chip memory as scratchpad SRAM banks that are managed explicitly in software. Input/output data is stored in a banked scratchpad with as many SRAM banks as the maximum FFT radix size. Data is transferred from local scratchpad memory to main memory through the host CPU’s data transfer mechanism and the accelerator’s interface to the system. The following sections now cover each of the major components of the accelerator.

3.1.2 Descriptors

The codelet configuration is determined at runtime based on the plan specification, encoded as a program consisting of *descriptors* that specify parameters for batched codelet execution. Descriptors contain two key components: the codelet configuration parameters discussed in Sec. 2.4.4 and the input/output/diag base address, strides, vector strides, and vector lengths for loading and storing data to memory. The descriptors are packed into 32-bit wide instructions and each descriptor is constructed of multiple instructions. The implementation of the descriptors is also flexible enough that additional parameters can be incorporated if needed.

Memory parameters specify the addressing pattern for retrieving and storing a batch of data. Successive inputs to the FFT are addressed according to the base address offset and stride, where the first complex input requests data at $\text{baddr} + \text{offset}$, where baddr is a pointer to scratchpad memory. The ensuing elements up to the size of the currently operating radix are addressed at stride increments. Then, the next sequence of data is retrieved starting from the base address offset incremented by a vector stride, $\text{baddr} + \text{offset} + \text{vstride}$ and the following elements are again accessed at the same stride value. Independent vectors of data are loaded up to the size of the batch defined by the descriptor. The same addressing mode is used for loading Diag data and for writing data back to memory, but each will have its own set of parameters for the batch.

In addition to the memory and codelet configuration descriptors, there are also some special case parameters for additional pipeline control. First, a flag to indicate a pipeline flush allows the hardware to continuously execute descriptors if there are no data dependency concerns across batches. Second, a special case descriptor for a memory fence is also required to synchronize codelet invocations. The controlling CPU can then resume the operation once all data transfers and codelet invocations reach the fence point.

3.1.3 Frontend

The accelerator programs are first stored to a local descriptor memory. The host CPU sends both a start command and a base pointer to the program to initialize execution. Then, the front-end begins to fetch descriptors and interpret the encoded codelet parameters. Once a descriptor has been completely decoded, if the codelet pipeline is not busy then the parameters are passed directly to the respec-

tive datapath configuration registers and the memory address generation units. Otherwise, the decoded parameters are sent to a descriptor queue. If the current in-flight descriptor does not need to be flushed then the parameters at the head of the queue are fetched immediately after the previous batch is completely issued. If a pipeline flush is required, the next descriptor and batch is issued after the in-flight descriptor indicates completion. The descriptor queue allows the frontend to run ahead of the compute pipeline so that the core does not stall on descriptor decoding. Because the depth of the codelet pipeline can be deep, the ability to continuously issue batches without pipeline flushes is important.

3.1.4 IO Engine

The IO engine manages data movement from the compute pipeline to local memory and the accelerator interface to the system's memory subsystem. The interface from local on-chip memory to the hardware codelet is controlled by address generation units that take decoded memory descriptor parameters and issue packets of memory requests using the addressing pattern described in Sec. 3.1.2. Each memory bank is connected to a single input/output element of the maximum sized radix and sequential addresses are interleaved across scratchpad banks. The address generators will receive the next descriptor's memory parameters once the current batch has completed its load/store requests and any pipeline flushes are handled.

The second aspect of the IO engine is to handle the accelerator's interface to the system in which it plugs into. Its actual implementation depends on the system architecture, but can be designed to interface to a system bus, memory controller, etc. The final section of this chapter will show specific examples of how this can

be designed for different types of systems.

3.1.5 Hardware Codelet

The datapath details of the hardware codelet were presented in Sec. 2.4.1. Here we discuss some of the additional considerations for interfacing with the accelerator frontend, scratchpad memories, and controlling CPU. The configuration of the compute pipeline will change the load/store sequencing of the data and depth of the pipeline. To handle this we use a latency-insensitive interface that sends a ‘valid’ bit with the input data which flows through the pipeline and is pulled out at certain points for controlling the issuing of load/store requests. If the Diag multiplications occur after the FFT, then the valid signal coming out of the FFT is used to initiate the load from scratchpad for Diag inputs. We use the final valid signal that exits the pipeline with the output data to initiate the write request back to scratchpad memory. We can also backpressure or stall the compute pipeline through a ‘ready’ signal sent on the output side of the pipeline.

The codelet controller is a state machine that tracks the batch as it flows through the pipeline. When the current in-flight batch does not specify a pipeline flush, the controller only updates the next batch memory parameters for the address generation units in the IO engine. The accelerator program guarantees that the codelet configuration does not change across batches with no pipeline flushes. For batches which must be flushed, the controller stalls the update of all codelet and memory parameters until all data is written back to the scratchpad. Memory fence descriptors send the controller into an idle state where it waits until the CPU indicates computation can resume. The accelerator will also raise a status flag to indicate to the CPU that the fence has been reached.

3.2 Hardware Design Flow

The end-to-end design flow from the software stack to the generated hardware is shown in Fig. 3.3. We leverage SPIRAL [32] in two ways, first as a design-time tool and second as a runtime tool for software and hardware generation. At design-time, the hardware needs to be instantiated based on the problem types that the applications span. During this phase we specify the data types, codelet sizes, memory capacities, and descriptor parameters for the hardware generator to produce an accelerator design. The FFT datapaths and element-wise multipliers are generated by SPIRAL using the codelet sizes passed to the generator. The Chisel hardware generator automatically instantiates generated codelets into the accelerator and emits SystemVerilog RTL which can be used in a standard ASIC toolflow.

3.2.1 Generating Hardened Codelets

Automatically generating the codelet datapaths is a key component for enabling rapid design cycles and prototyping using the design framework. The basic hardware codelet template is written in Chisel and provides a mechanism for automatically instantiating a collection of FFT datapaths and the Diag element-wise multipliers. The hardware codelets are completely unrolled such that the entire signal flow of the FFT is realized in space. To facilitate hardware generation of the unrolled FFT basic blocks, we use the SPIRAL [32] code generation system which contains internal representations of the FFT signal flows.

The generation process begins with a problem specification for the desired FFT size (e.g. DFT_8), and the transform is completely unrolled using the algorithmic

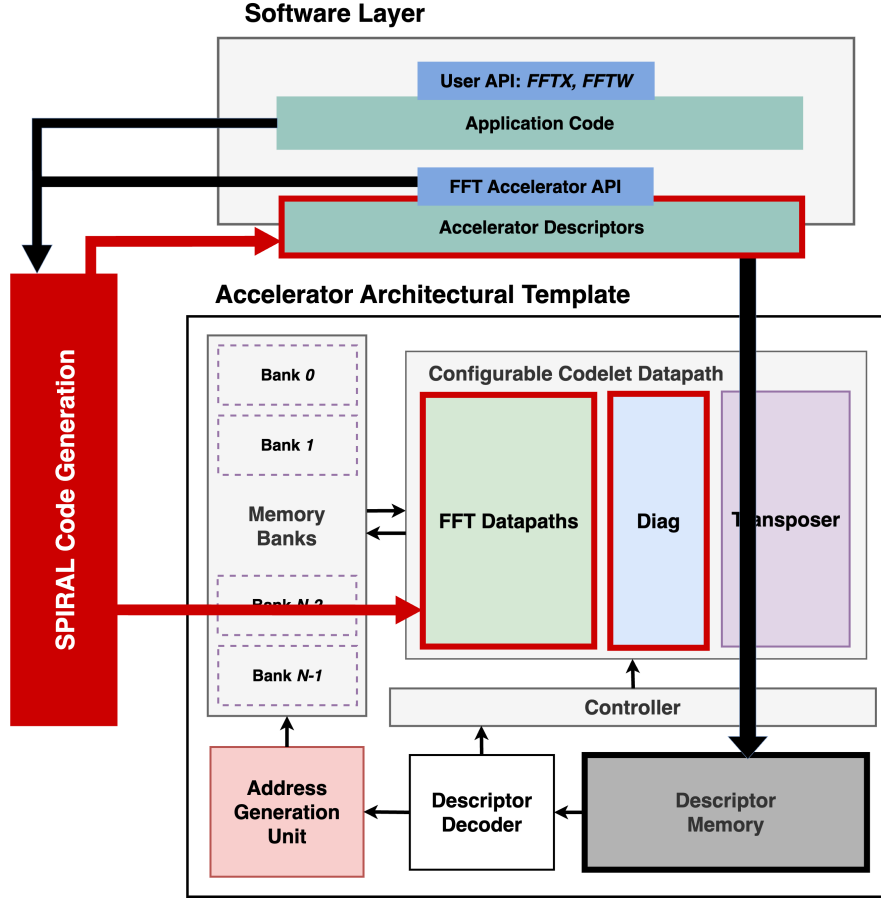


Figure 3.3: Automated Design Flow. The SPIRAL code generation system is used to co-design parts of the accelerator core and for code generation.

breakdowns in SPIRAL. At the lowest level DSL, the abstract code representation called icode contains the unrolled DAG for the entire transform. However the DAGs for generating software basic blocks cannot be directly unparsed for hardware generation since pipeline register insertion is required to ensure that data vectors remain aligned through the computation. We perform a timing analysis pass on the icode representation, where the latencies of arithmetic units can be passed in as parameters to the analysis.

Starting from the primary inputs to the module, we assign an abstract notion of time that starts at 0. We then trace the DAG and track the delays of the inputs at every node. If they match, then we increment the incoming time according to the latency of the node and assign the updated value to the resulting operand. If they do not match, then we insert a ‘delay’ node that is annotated with the required delay value to align the faster operand with the slower one. Then the delayed operand is assigned the new time which now matches with the other operand. After performing the timing pass, we unparse the updated DAG to the Chisel HDL [62].

Our implementations take advantage of the object-oriented, functional abstraction paradigms introduced by high-level hardware description languages such as Chisel. Unparsing the icode representations to Chisel is relatively simple because an adder, multiplier, or even shift register FIFO can be instantiated in a single line of code. Implementations of specific arithmetic units are provided in a separate Chisel package. The current codelet generator contains parametrized floating point units for half, single, and double-precision arithmetic, but we note that different data types can be supported in the future. Fig. 3.4 shows an example of the SPIRAL-generated Chisel code for a single-precision, complex radix-3 FFT which is the minimal example that shows the effect of the timing pass. The combination of SPIRAL’s mathematical framework and Chisel’s higher level programming abstractions creates a powerful generator for automatically realizing the hardware codelets.

Example codelet design. The following figures show a few small size powers-of-2 and prime size FFTs which are automatically generated using SPIRAL. The blocks depicted in Fig. 3.5 and Fig. 3.4 support single-precision floating point

```

1  package dft.accel
2
3  import chisel3._
4  import chisel3.util._
5  import _root_.circt.stage.ChiselStage
6
7  import dft._
8  import ChiselFloat._
9
10 class Dft3() extends DftIO(UInt(32.W), 3){
11
12     val beat = Wire(Bool())
13     val a38, a40, a41, a44, s7, s8, shift1, shift2,
14         shift3, shift4, shift5, shift6, t69, t70, t71, t72,
15         t73, t74 = Wire(UInt(32.W));
16     val sreg_sig_out = Wire(UInt(1.W));
17     t69 := FPAdd(io.X.bits(2), io.X.bits(4), beat)
18     t70 := FPAdd(io.X.bits(3), io.X.bits(5), beat)
19     a38 := FPMult("h3F000000".U, t69, beat)
20     shift1 := ShiftRegister(shift3, 1, 0.U, beat)
21     t71 := FPSub(shift1, a38, beat)
22     a40 := FPMult("h3F000000".U, t70, beat)
23     shift2 := ShiftRegister(shift4, 1, 0.U, beat)
24     t72 := FPSub(shift2, a40, beat)
25     a41 := FPSub(io.X.bits(3), io.X.bits(5), beat)
26     s7 := FPMult("h3F5DB3D7".U, a41, beat)
27     a44 := FPSub(io.X.bits(2), io.X.bits(4), beat)
28     s8 := FPMult("h3F5DB3D7".U, a44, beat)
29     shift3 := ShiftRegister(io.X.bits(0), 3, 0.U, beat)
30     t73 := FPAdd(shift3, t69, beat)
31     io.Y.bits(0) := ShiftRegister(t73, 4, 0.U, beat)
32     shift4 := ShiftRegister(io.X.bits(1), 3, 0.U, beat)
33     t74 := FPAdd(shift4, t70, beat)
34     io.Y.bits(1) := ShiftRegister(t74, 4, 0.U, beat)
35     shift5 := ShiftRegister(s7, 3, 0.U, beat)
36     io.Y.bits(2) := FPAdd(t71, shift5, beat)
37     shift6 := ShiftRegister(s8, 3, 0.U, beat)
38     io.Y.bits(3) := FPSub(t72, shift6, beat)
39     io.Y.bits(4) := FPSub(t71, shift5, beat)
40     io.Y.bits(5) := FPAdd(t72, shift6, beat)
41     sreg_sig_out := ShiftRegister(io.X.fire, 10, 0.U, beat)
42
43     beat := sreg_sig_out === 0.U | io.Y.ready
44     io.X.ready := beat
45     io.Y.valid := sreg_sig_out
46 }

```

Figure 3.4: Generated Chisel code for radix-3 single-precision FFT hardware. This example shows how adders, multipliers, and pipeline registers are instantiated in the generated hardware.

arithmetic where each adder is pipelined to four stages and each multiplier consists of two pipeline stages. The latencies of each arithmetic unit are passed to

the SPIRAL hardware generation script such that the timing pass can insert the appropriate shift register delays into the datapath.

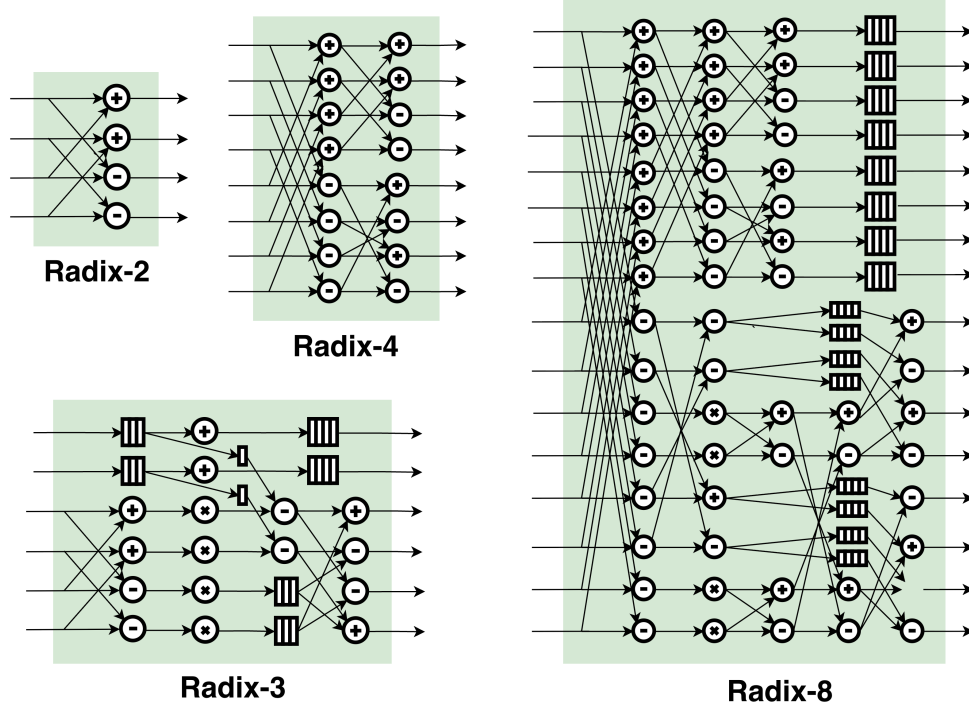


Figure 3.5: Example radix-2, 3, 4, 8 generated FFT hardware. Each arithmetic unit performs real-valued operations and the inserted pipeline registers are also shown. All multipliers are constant multipliers in the FFT datapaths.

3.2.2 Generating the Codelet Accelerators

The top-level accelerator generator exposes compile-time parameters which the user specifies in order to generate the full codelet accelerator from the architectural template. These include data type, memory capacity, and desired codelet sizes which are all passed down as parameters to subcomponents of the top-level Chisel accelerator. The system automatically instantiates as many scratchpad memory

banks as the maximum requested codelet size. The descriptor memory, decoder, controller, and IO engine are each implemented as parametrized Chisel modules which are connected together in a top-level accelerator module.

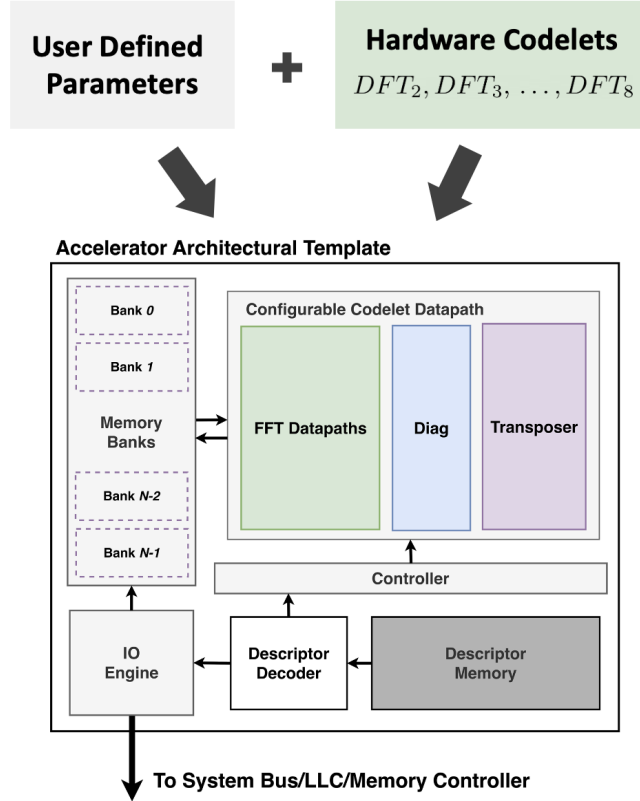


Figure 3.6: Overview of the accelerator generation process. User defined parameters including codelet sizes, data type, and local memory capacity are passed to the Chisel generator. The hardware codelets are pregenerated and contained in a separate Chisel package.

The FFT hardware modules are pre-generated for sizes 2-8 and contained in a separate Chisel package. If other codelet sizes are desired, they can also be generated via SPIRAL and added to the package. To instantiate the desired set of FFT

modules, all SPIRAL-generated FFT Chisel code must extend a parametrized abstract class called `DftIO`. Then the sequence of codelet sizes, passed as a parameter to the top-level generator, can be mapped to a sequence of Chisel modules and their IO signals are connected to the top-level codelet datapath. The generated RTL is completely synthesizable and is compatible with a standard ASIC toolflow using timing SDC constraints. For ASIC designs, all memory modules need to be replaced with foundry-specific SRAM macros. Later in this chapter, we will present a testchip implementing a radix-4 codelet accelerator that validates the RTL produced by our generator.

```

1  package dft
2
3  import chisel3._
4  import chisel3.util._
5
6  abstract class DftIO[T <: Data](val data: T, val N: Int) extends Module {
7      val io = IO(new Bundle {
8          val X = Flipped(Decoupled(Input( Vec(2*N, data) )))
9          val Y = Decoupled(Output( Vec(2*N, data) ))
10     })
11 }

```

Figure 3.7: Parametrized abstract class `DftIO` which all generated FFT modules must extend. The selected codelet sizes are then mapped to the corresponding FFT hardware module.

3.2.3 Evaluation of Overheads

We quantitatively evaluate the area of the accelerator to show that the overhead of decoding and invoking batched codelets is low especially as the codelet size increases. The percent overhead is calculated by taking all logic area (in μm^2) excluding the hardware codelet and dividing by the total synthesized logic area.

The total accelerator area includes all logic required for the controller, IO engine, decoder, and configurable hardware codelet. All designs interface to 1KB of plan memory and 256KB of local scratchpad memory, but we exclude the memory macro area required for plan and data memories in the evaluation. All designs support single-precision floating point arithmetic and are synthesized at a 1GHz clock frequency using TSMC 16nm standard cell libraries.

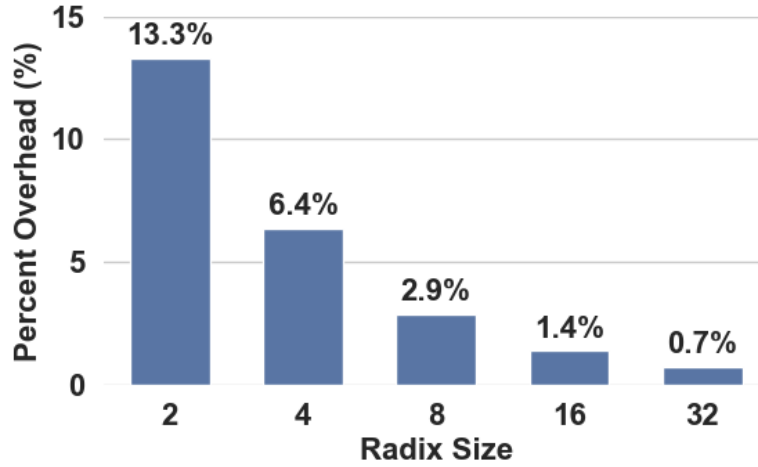


Figure 3.8: Percent overhead of descriptor fetch, decode, and control units out of total accelerator logic area.

The percent overhead of the frontend descriptor fetch, decode, and control logic area is about 13.3% for an accelerator with only a complex radix-2 codelet and gradually decreases to about 0.7% at a codelet size of 32. As the number of floating point adders/multipliers increases in the codelet, the overhead of the state machine and small ALUs for interpreting and executing programmable descriptors becomes extremely low. Furthermore, descriptors can specify different FFT algorithms to guarantee that the codelet pipeline is saturated. Thus if our design were compared

against a fully hardened ASIC at a fixed problem size, the additional overhead of programmability would not be significant.

3.3 Software Framework

Once the accelerator is instantiated, an FFT program for the accelerator can now be generated or handwritten. We provide a low-level C library which exposes an API to program and configure the accelerator for batched codelet invocations. When the code is compiled with the library, it generates the program binary which contains the descriptor parameters and can be interpreted by the accelerator. In this section we discuss components of the software tools that enable high-level application software to program the hardware. We also discuss the higher-level architectural model that was developed for validating the accelerator programs.

3.3.1 Low-level Software Library

A low-level software library translates batched codelets and their respective parameters to descriptors and program binaries which the accelerator can interpret and execute. This allows a user to compose an FFT computation on the accelerator by writing a standard C program against the library, which then builds a data structure containing the descriptors. The library functions then map directly to the low-level descriptor instructions supported by the accelerator. The flexibility of the hardware is exposed through this software layer and enables one to specify computations in a variety of different ways, similar to FFTW's plans.

The library provides the necessary assembler to translate the descriptor data structures to the accelerator specific binary that executes on the hardware. In Fig. [5.1b](#) we show an example C program with the descriptor data structure

that a user can construct, in this case the descriptors express a 64-point complex FFT using radix-8 FFTs. The configuration descriptors set the parameters for the codelet datapath while the following memory descriptors specify the IO base address, stride, vector stride, and vector length.

3.3.2 Simulation Models

The framework also provides a software architectural model that models transaction-level behavior of the hardware. It takes accelerator program binaries, decodes the descriptors and collects the parameters, then sequentially executes the batch codelet invocations using the decoded parameters. The models enable faster software development and verification cycles without needing to go to RTL simulation, making them particularly useful during hardware design cycles. In the next section, we will discuss how this layer of the accelerator API is exposed to higher level application code.

3.3.3 LibraryX-ASIC

While accelerators have many benefits, they also have a few challenges specifically in programmability and adoption. Accelerator designers define their unique end-to-end software stack for developer use, but this is challenging because it requires developers to potentially learn a new programming model to access the accelerator. This becomes increasingly more complicated for each new accelerator introduced, limiting program portability. To address the issues of programmability and portability of accelerator devices, we propose LibraryX-ASIC [83]. LibraryX-ASIC is an automated framework designed to hide the complexity of accelerator offload behind domain-specific software libraries like FFTW [25] for FFTs, or C++ Standard

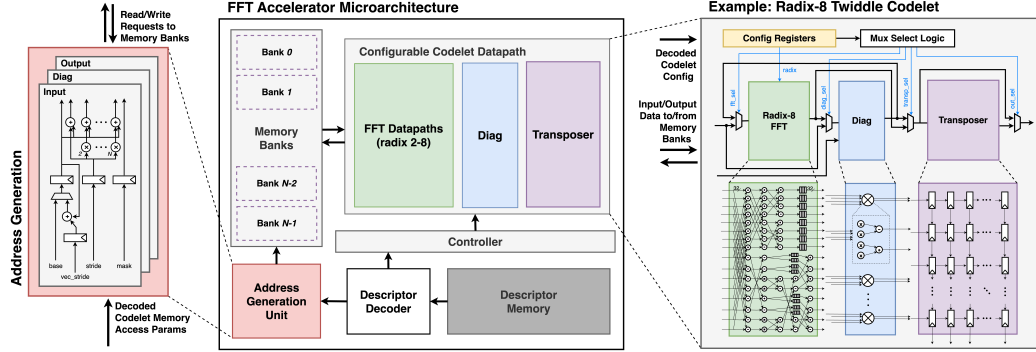


Figure 3.9: Overview of the FFT accelerator microarchitecture.

Template Libraries (STL). Using the library’s standard interface, LibraryX-ASIC recognizes the operation, generates an optimized accelerator implementation, and offloads it to the accelerator automatically, populating the output buffer upon completion. This relieves the programmer of having to worry about various accelerator offload paradigms.

3.3.4 End-to-end Example

We describe the LibraryX-ASIC system design using a simple FFT program shown in Fig. 3.10. We discuss how FFT library calls are recognized and captured. We then describe the high-level process to generate equivalent accelerator code using SPIRAL.

LibraryX-ASIC has been integrated with the proposed FFT-based accelerator to demonstrate how an application developer can target the accelerator through a standard software interface. Fig. 3.10 shows a simple FFT program in which the FFT library call is recognized and captured. At runtime the generated code, split into host code and accelerator code, shown in Fig. 5.1b is executed. The host code sets up the data transfer between the accelerator and main memory and invokes

```

1  #include <iostream>
2  #include <complex>
3  #include <vector>
4  #include "fftw3.h"
5  #include "LibraryX_ASIC.hpp"
6  using namespace std;
7  int main() {
8      int N = 64;
9      int sign = -1;
10     u_int f = FFTW_ESTIMATE;
11     vector<complex<float>> input(N);
12     vector<complex<float>> output(N,0.0);
13
14     buildInput(input);
15
16     //call is replaced with accelerator offload
17     //and executed
18     fftwf_plan p = fftwf_plan_dft_1d(N,
19         (fftwf_complex*)input.data(),
20         (fftwf_complex*)output.data(), sign, f);
21     fftwf_execute(p);
22
23     //output buffer contains accelerator result
24     checkOutput(output);
25
26     fftwf_destroy_plan(p);
27     return 0;
28 }

```

Figure 3.10: The FFT application in written against FFTW will be executed on an FFT accelerator without user modification using LibraryX-ASIC. The output buffer will be populated as if nothing changed.

the accelerator with the base pointer to the accelerator program. The accelerator program is contained in the `dft_desc` data structure as an array of descriptors and gets interpreted by the accelerator's descriptor decoder.

Capturing Library Calls LibraryX-ASIC uses a function call capturing technique called delayed execution to intercept library calls at runtime. This transforms library calls from operations performed on inputs and outputs to specifi-

```

1  #include "model.h"
2  #include "utils.h"
3  #include "accel.h"
4
5  Program dft_desc[7] = {
6      {CONFIGI,8,0,0,1,0,1,1,0,8},
7      {MEMI,MEM_IN,1,8,0xFF,LOCAL_MEM,0},
8      {MEMI,MEM_DIAG,1,8,0xFF,LOCAL_MEM,
9          LOCAL_MEM_REGION_SIZE};
10     {MEMI,MEM_OUT, 1, 8, 0xFF, LOCAL_MEM,0};
11     {CONFIGI,8,0,0,0,0,1,1,1,8};
12     {MEMI,MEM_IN,1,8,0xFF,LOCAL_MEM,0};
13     {MEMI,MEM_OUT,1,8,0xFF,LOCAL_MEM,0}
14 };
15
16 void dft64(float *Y, float *X) {
17     enter();
18     float *T23;
19     T23 = initTwiddles64();
20     dmaLoad(LOCAL_MEM, 0, 0, 8, 1, X, 8, 8, 8);
21     dmaLoad(LOCAL_MEM, 1, 0, 8, 1, T23, 8, 8, 8);
22     // Invoke Accelerator
23     executePlan(0, dft_desc);
24     dmaStore(LOCAL_MEM, 0, 0, 1, 8, Y, 8, 8, 8);
25     exit();
26 }

```

Figure 3.11: SPIRAL generated accelerator code. The generated code uses information from SPIRAL’s FFT description including input ranges and input and output strides.

cations describing the computation to be performed. In the case of FFTs this includes the type of transform, its dimensionality, and the types for the call’s input and output. At runtime the LibraryX-ASIC captured call is invoked. Here, LibraryX-ASIC extracts the library call information, building an SPL expression that describes the library call’s semantics. This SPL expression is then passed to the SPIRAL system for code generation.

SPIRAL Code Generation. The SPIRAL code generation system uses

the LibraryX-ASIC SPL expression to generate the ASIC implementation. This SPL expression goes through a series of transformation stages within SPIRAL that implement and optimize the FFT calculation. In the first stage, a specific algorithm is selected to instantiate the FFT. As the example FFT accelerator supports radix-8 hardware codelets, SPIRAL specializes its algorithmic breakdown for radix-8.

The FFT accelerator exposes an intrinsic C library for computation offload, and SPIRAL produces the intrinsic code for the FFT computation. Along with the actual operation, SPIRAL also produces the setup code to move the pointers from the host device to the accelerator, and performs memory cleanup once the operation is complete. An example generated FFT implementation is shown in Fig. 5.1b.

3.4 Generator Verification via Silicon Testchip

The previous silicon prototype showed a first look at how SPIRAL [32] is integrated as a part of the ASIC design flow for accelerating only the computational kernel.

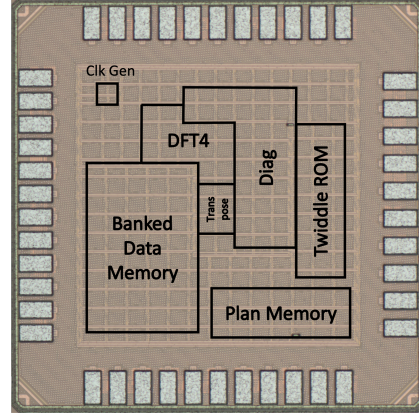
3.4.1 A Radix-4 Codelet Accelerator in 28nm

A generated radix-4 FFT accelerator design is taped out in a TSMC 28 nm process. The codelet datapath consists of an unrolled radix-4 complex FFT, four element-wise complex multipliers, and a 4×4 systolic array transposer. There is a total of 32 KB on-chip memory for data and descriptors implemented using SRAM macros. Twiddle factors up to FFT length 64 are programmed into a ROM. The data memory consists of four SRAM banks such that four complex elements can be fed into the radix-4 datapath per cycle. The accelerator RTL is entirely generated

using our design framework and verification is also facilitated with the software design tools to generate test vectors and golden models.

Technology	TSMC 28nm
Chip Area	1.0 mm ²
Core Area	0.49 mm ²
Core Voltage	0.65 - 1.15 V
Frequency	350-1025 MHz
IO Pads	44
Power Domains	3

(a) Implementation Summary



(b) Die micrograph

Figure 3.12: Summary of 1mm $\times$ 1mm test chip implementation in a TSMC 28nm process and micrograph of generated accelerator

The 1mm $\times$ 1mm FFT accelerator test chip is fabricated in a TSMC 28nm process and has a core size of 0.7mm x 0.7mm. The radix-4 datapath occupies about 0.024mm² and the Diag block occupies 0.045mm². Fig. 4.20 shows a summary of the die measurements and the accelerator layout. At nominal 0.9V the chip runs at 750 MHz and consumes a total of 142.95 mW to execute a 64-point complex FFT. The chip successfully operates in a range of 350-1025MHz at a supply voltage of 0.65V-1.15V. The black line in Fig. 3.13 demonstrates the voltage-frequency operating points at which the test chip is functional. For the same workload, the accelerator core consumes 93.12 mW and the memories consume 49.83 mW operating at 0.9V, 750MHz. The total measured power at the maximum operating frequency for a fixed voltage is shown in black in Fig. 3.13.

The 28nm programmable FFT accelerator supports both iterative and recur-

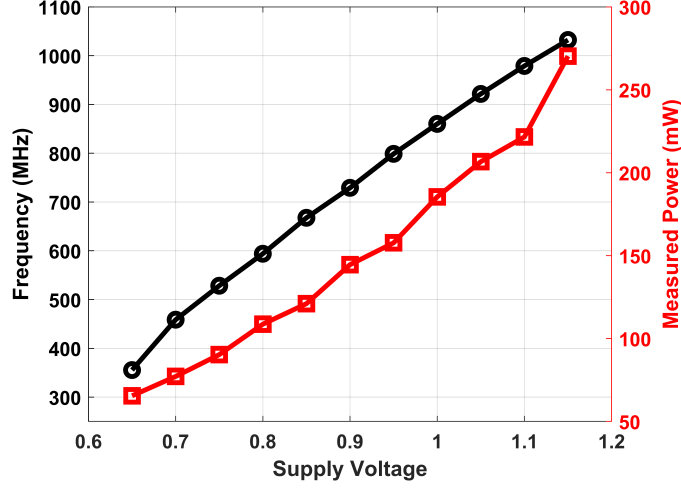


Figure 3.13: Voltage-frequency operating points and total power consumption measured from 0.65V - 1.15V. Operates from 350-1025 MHz and consumes 142.95 mW at 0.9V, 750MHz.

sive FFT algorithms for a few 1D and 2D powers-of-4 FFTs, demonstrating the highly flexible nature of our design approach. Furthermore, the silicon prototype verifies the functionality of our end-to-end design framework where generated software and hardware components ease much of the manual design effort in the ASIC design flow process.

3.5 System Integration

3.5.1 RISC-V System-on-Chips

The accelerator has been integrated into a few heterogenous SoC systems with RISC-V CPUs acting as the controlling processor. Here we show an example of how the accelerator fits into the ESP [84] SoC architecture and is coupled to the control processor.

Tile-based SoC Architecture The ESP platform for flexible SoC develop-

ment provides a tile-based architecture where tiles can be configured as processor, memory, or accelerator tiles. Any domain-specific accelerators inserted into the SoC can receive configuration parameters but are responsible for managing their own DMA request and responses. We have integrated the accelerator into a configuration of the ESP SoC shown in Fig 3.14. The accelerator plugs into the system NoC through the defined ESP interface and then manages its own packets through the NoC.

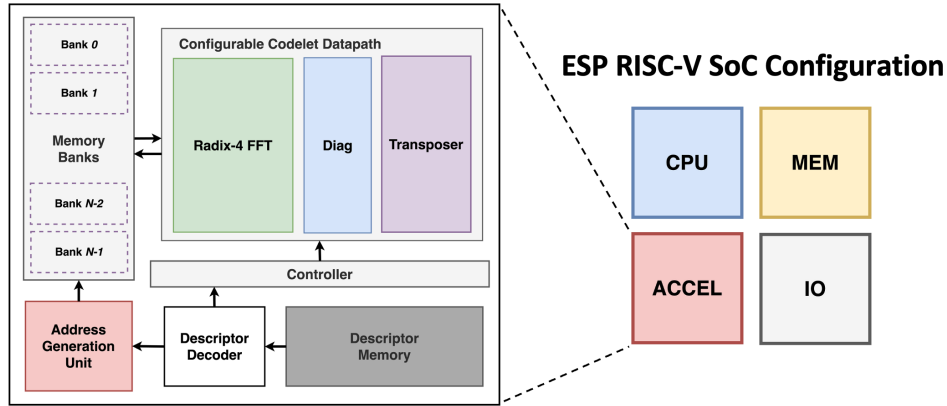


Figure 3.14: ESP Tile-based RISC-V SoC with generated FFT accelerator tile.

The extensible nature of our design allows us to integrate the ESP interface into the accelerator relatively easily. The IO engine of the accelerator architecture incorporates an additional state machine which handles the response and request streams to and from the NoC. The descriptors are also extended to support DMA requests which the accelerator must initiate.

Program execution. Below we show the execution flow of the program. The accelerator byte code is contained in a data structure whose base pointer is

passed to the accelerator upon invocation. The accelerator then initiates the DMA transfer to load the program from the pointer. Once the data transfer completes, the program execution begins and upon completion will initiate store requests back to the SoC memory.

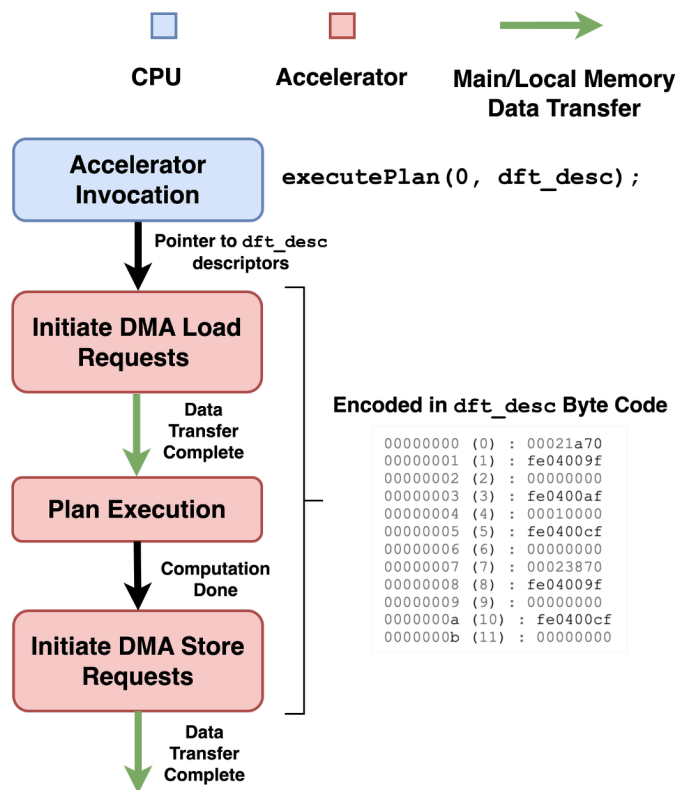


Figure 3.15: Accelerator program execution on the ESP RISC-V platform with custom FFT accelerator.

Chapter 4

Extending Hardware Capabilities

In this chapter we discuss several ways in which the design approach and hardware framework has been extended towards other applications in fault tolerant computing [85], digital signal processing, and high performance computing.

4.1 An Algorithm-based Approach to Soft Error Tolerance

Modern semiconductor technology scaling has led to an increased vulnerability in electronic systems to radiation-induced soft errors [86,87]. High energy particles, such as heavy ions, protons, or neutrons, can cause a temporary change of state known as a soft error upset (SEU) upon striking and interacting with the integrated circuit. Protecting against soft errors is crucial for circuits operating in safety-critical space/upper atmosphere applications where radiation environments

are harsh. Even large scale terrestrial systems, such as hyperscale data centers deploying integrated circuits (ICs) at high volume, are facing increasing reliability concerns due to effects from terrestrial radiation [88–90].

One of the most common design techniques to mitigate the effects of soft errors is to incorporate system-level hardware redundancy for error detection and correction. Using triple modular redundancy (TMR) or dual modular redundancy (DMR) schemes, the designer triples/doubles modules within the design and includes additional logic for error correction or detection. Both of these approaches can be leveraged at varying levels of granularity (e.g. at the register or functional unit-level) at around 2-4X area and power overheads. However the cost of TMR/DMR-based approaches is significant and becomes overwhelming, especially as power constraints and energy efficiency drive the performance scaling of new computing systems. In designing large modern SoCs, it has become critical to analyze tradeoffs between reliability, power, performance, and area so that only critical components are actually replicated [91,92]. Hence strategies to reduce the overheads while still providing tolerance against soft errors is increasingly important.

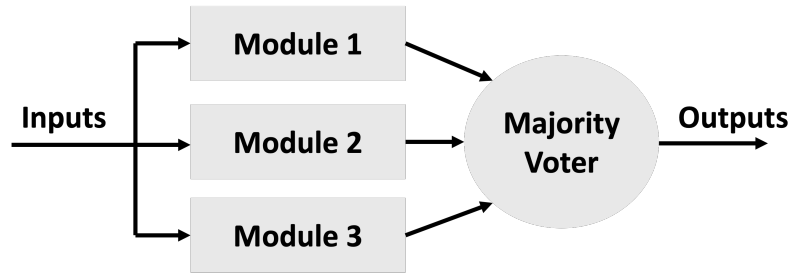


Figure 4.1: TMR technique using three copies of the module and a majority voter circuit.

Another method to protect against soft errors is to refactor computations such that they take advantage of properties of the algorithm itself for error detection or correction, an approach typically referred to as algorithm-based fault tolerance (ABFT) [93]. Exploiting algorithmic properties, such as transform invariants or matrix checksums, provides opportunities to detect and correct errors at low overhead without necessarily relying on system-level redundancy techniques. Inspired by the principles of ABFT, in this work we propose a new hardware redundancy approach for soft error tolerance in datapath modules. Our insight is that redundancy can be realized through floating-point interval arithmetic where lower precision interval arithmetic hardware reduces overheads compared to fully replicating modules. Error detection then relies on the guarantees provided by interval arithmetic and forward error analysis of the specific computation.

The following sections detail the approach and demonstrate how our design tools even enable fast hardware prototyping for non-standard arithmetic. In particular, it showcases how higher levels of abstraction representing the FFT datapaths leverage a Chisel library of arithmetic building blocks to generate unconventional FFT hardware instances.

4.1.1 Algorithm-Based Fault Tolerance

The idea of exploiting algorithm specific properties for fault tolerance was first demonstrated for matrix operations [93]. The key idea behind ABFT is to encode the data such that the encoding relation still holds after the algorithm is applied, enabling error detection or correction. The algorithm can then be modified to operate on the encoded data and produce encoded outputs, and finally the computation is distributed across processing units such that faults affect only certain parts of

the data. Error detection and correction can be performed based on the outputs of the original computation and the encoded outputs. The approach has been extended to a number of other domains such as digital signal processing [94–97], machine learning [98,99], and HPC scientific computing applications [100–103].

4.1.2 Interval Arithmetic

Interval arithmetic [104] is an approach for performing sound floating-point arithmetic as it provides the guarantee that the resulting interval must contain the real number that is the result of the mathematical expression. Operations on real numbers represented in floating point instead take place on intervals with floating point boundaries containing the real number. Concretely, an interval $\hat{x} = [x_l, x_u]$ consists of floating point bounds that guarantee the exact number is contained within the interval $\hat{x}$. Interval arithmetic operations for addition, subtraction, and multiplication are then defined as follows:

$$\hat{x} + \hat{y} = [\text{RD}(x_l + y_l), \text{RU}(x_u + y_u)] \quad (4.1)$$

$$\hat{x} - \hat{y} = [\text{RD}(x_l - y_u), \text{RU}(x_u - y_l)] \quad (4.2)$$

$$\hat{x} \cdot \hat{y} = [\min(\text{RD}(x_l \cdot y_l), \text{RD}(x_l \cdot y_u), \quad (4.3)$$

$$\text{RD}(x_u \cdot y_l), \text{RD}(x_u \cdot y_u)),$$

$$\max(\text{RU}(x_l \cdot y_l), \text{RU}(x_l \cdot y_u),$$

$$\text{RU}(x_u \cdot y_l), \text{RU}(x_u \cdot y_u))]$$

where $\text{RD}(x)$ rounds x towards negative infinity and $\text{RU}(x)$ rounds x towards positive infinity.

The guarantee of sound floating point arithmetic in interval arithmetic is valu-

able for safety critical applications in which round-off errors can be catastrophic [105–107]. Recent work has thus studied methods for automatically producing efficient interval arithmetic programs with higher certified accuracy [108, 109]. In this work we use this guarantee for designing an algorithm-based approach to soft error tolerant hardware.

4.2 Interval Arithmetic-based Soft Error Tolerance

In this section we assume the single event upset fault model in which exactly one internal node in the datapath can have its logic state flipped at a given clock period. The focus of this work lies in the protection of logic or datapath modules which implement a specific computation. Memories are assumed to be protected through ECCs and scrubbing logic. The following section discusses how we adopt interval arithmetic for lower overhead hardware redundancy and perform algorithm specific forward error analysis for error detection.

4.2.1 Hardware Redundancy via Interval Arithmetic

Fig. 4.2 shows the block diagram of the proposed technique. Module 1 contains the original hardware which executes its computation on the unmodified inputs. A lower precision version of the module executes the same computation in interval arithmetic using floating-point, providing hardware redundancy for error detection.

Inputs to the interval arithmetic block are lower precision intervals constructed such that the original number is guaranteed to be within the interval. The input values are first cast to a lower precision floating point representation and then rounded towards $+\infty$ and towards $-\infty$ to get the upper and lower endpoint values. Thus the constructed input intervals are small—the error margin is at most one

unit in least precision (ulp), the smallest difference between two adjacent floating point numbers. Constants within the module are also cast to intervals using the same procedure and in the case that the number is exactly representable in floating point will have the same endpoints. The interval outputs and the high precision outputs from the original module all feed into the error detection block.

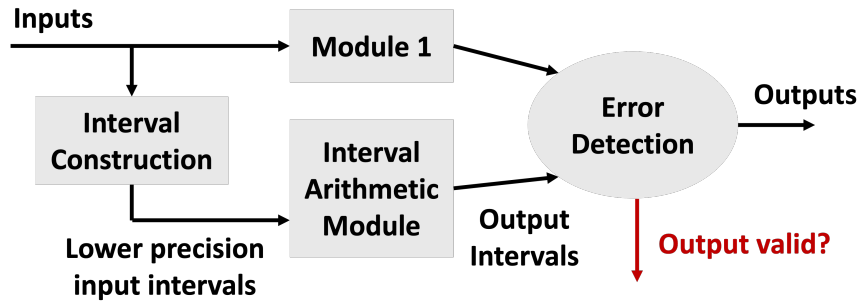


Figure 4.2: Interval arithmetic based approach for soft error tolerance.

Interval arithmetic addition and subtraction is implemented using two adders to compute the upper and lower intervals. For subtraction the second operands are swapped and the sign bit is flipped. In interval arithmetic multiplication, a naive implementation following Eq. 4.3 may use four multipliers for each of the operand combinations in computing one endpoint. Instead we perform sign bit logic to swap operands on the fly and require only two multipliers for interval multiplication. Constant multiplication is even simpler since the sign of one operand is known at design time. The arithmetic unit for the upper endpoint implements rounding towards $+\infty$ and for the lower endpoint rounding towards $-\infty$. Fig. 4.3 shows block diagrams of the interval arithmetic units.

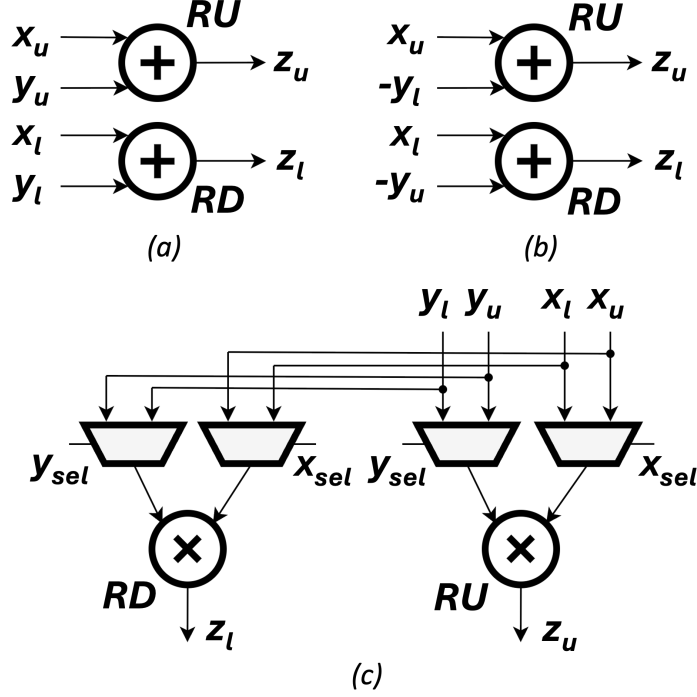


Figure 4.3: Interval arithmetic hardware block diagram for (a) addition, (b) subtraction, and (c) multiplication.

4.2.2 Error Detection

The guarantee of sound floating-point arithmetic and forward error analysis of the computation enable error detection. There are four general cases depicted in Fig. 4.4 which we must consider for the output interval and middle value. Fig. 4.5 shows a block diagram of the steps for error detection. For any interval $\hat{x} = [x_l, x_u]$, we first check that $x_u > x_l$ and $d(x_u, x_l) < T$ where $d(x, y)$ is the distance between two floating point numbers, x and y , in terms of ulp. The threshold value, T , is determined in terms of ulp through forward error analysis, which is discussed in the following section. If both conditions are satisfied then the interval is accepted

as valid. Otherwise we declare that there is an interval error.

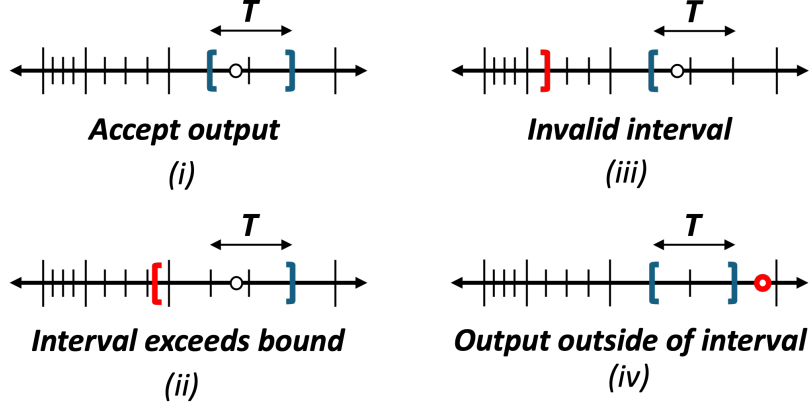


Figure 4.4: The four potential cases for an output interval: (i) no error, middle value accepted (ii) interval grows beyond bound, (iii) interval endpoints flipped, and (iv) high precision middle value outside of interval.

We then check whether the high precision middle value, x_m , is contained within the interval. If $x_l < x_m < x_u$, then we accept the high precision computation of the original module and can write the value to a protected memory. If the condition does not hold, then an error must have occurred in the high precision block and we mark the computation invalid. The output control signal can be used to indicate to a controller that the computation must be redone. We assume that the algorithms to be protected are numerically stable and bit flips that affect the lowest order mantissa bits while remaining within the bound are tolerable.

4.2.3 Forward Error Analysis

Using forward error analysis we can analyze the sequence of operations through the computation to provide a meaningful bound on the growth of the interval. We

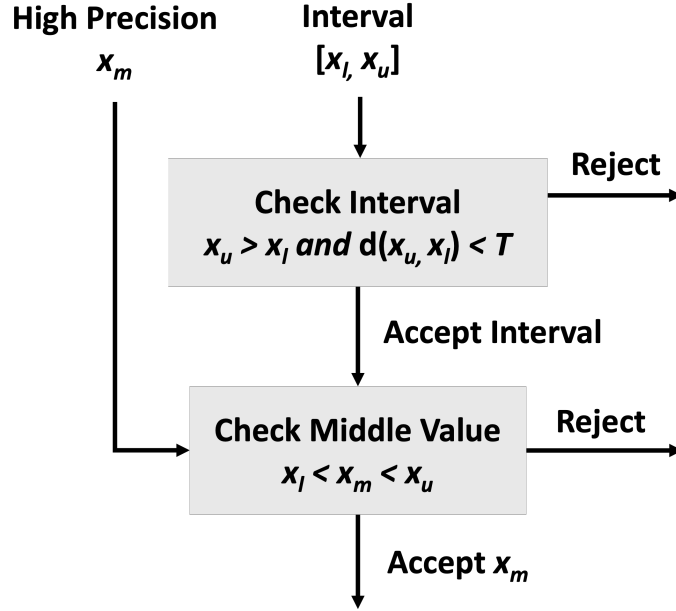


Figure 4.5: High level block diagram of the proposed method for error detection.

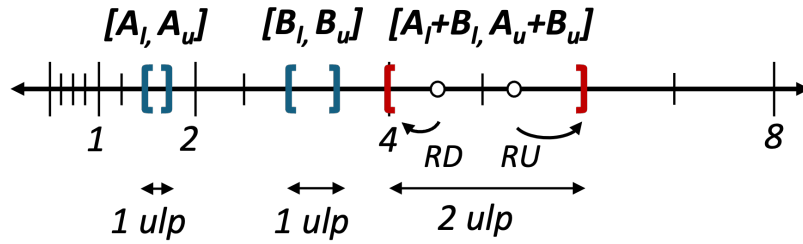


Figure 4.6: Example of interval growth after floating point operations. Each interval begins at 1 ulp distance but the resulting interval grows to 2 ulp due to rounding.

also assume that the dynamic range of the inputs and outputs are known either by properties of the application or through static analysis. All computations begin with intervals that are constructed such that the width is at most one ulp. In interval addition, rounding to $+\infty$ introduces an absolute roundoff error of at one

ulp. Rounding down incurs the same error, so the growth of the interval is bounded by at most two ulp for a single interval addition. Fig. 4.6 shows an example of how the resulting interval width can grow following interval addition. In multiplication, we can bound the growth of the interval by constraining multiplicand range. For example when computing FFTs, multiplication is always with twiddle factors so it is guaranteed that one multiplicand is in the range $[0, 1]$ and the growth is fixed to 2 ulp.

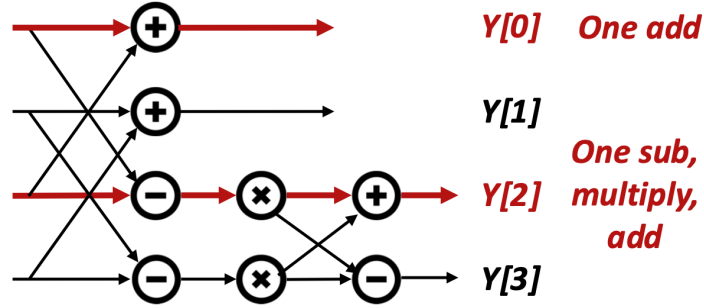


Figure 4.7: Analysis of an example 2-point FFT butterfly datapath to bound the output interval growth.

For each output of the module we analyze the critical path of additions, subtractions, and multiplications to compute the bound, T , on the growth of each interval. The bound is computed in terms of ulp whose absolute value depends on the range of the number. To avoid tracking dynamic range throughout the computation, we use a worst case bound by performing static analysis to determine the maximum ulp across the entire computation. The block diagram in Fig. 4.8 shows how we implement the interval check in hardware. The error detection module uses the exponent associated with the maximum ulp value to first normalize and

shift the interval mantissas. Then, the mantissas can be subtracted and compared against the bound, T . For every output interval we perform forward error analysis as shown in Fig. 4.7 for a 2-point butterfly and implement the corresponding error checking module in Fig. 4.8.

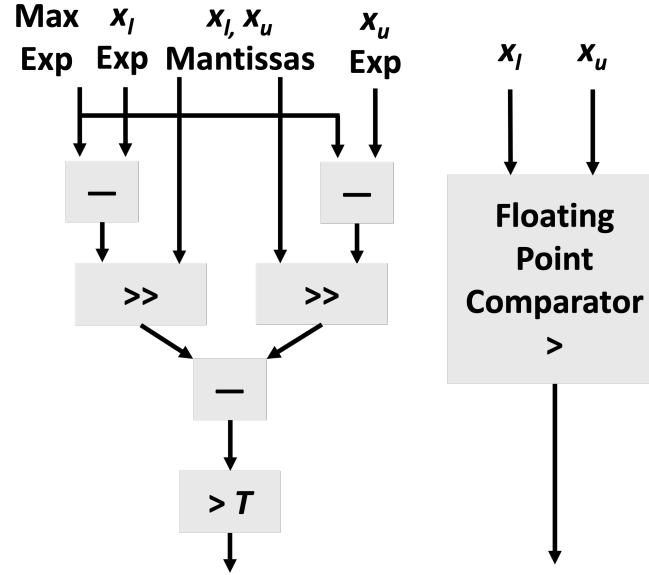


Figure 4.8: Hardware block diagram for interval checking using forward error analysis.

4.3 A Soft Error Tolerant FFT Accelerator in 28nm

We implement the algorithm-based approach using interval arithmetic for a radix-8 fast Fourier Transform datapath and a systolic array. We also design test structures to facilitate fault injections and error evaluation for each of the modules. Fig. 4.10 shows a block diagram of the test chip.

4.3.1 Fast Fourier Transform

The fast Fourier Transform (FFT) datapath shown in Fig. 4.9 implements a fully unrolled, complex radix-8 FFT supporting 32-bit single precision floating point arithmetic and interval arithmetic units using 16-bit half precision. Twiddle factor constants within the datapath are also converted to intervals. Both the original module and the interval arithmetic version are pipelined to 13 stages to ensure all outputs are aligned. Both single precision and half precision interval FFT hardware are automatically generated using the SPIRAL [32] code generation system.

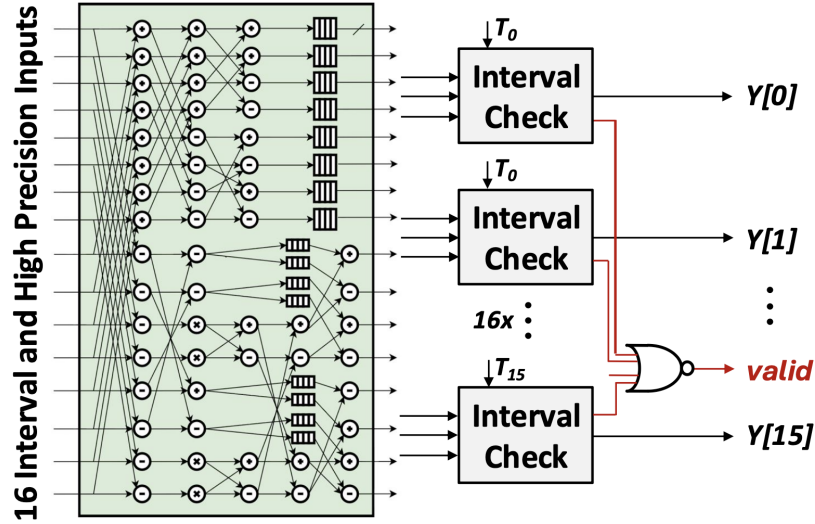


Figure 4.9: Architecture of the radix-8 FFT datapath and interval error detection.

Each 32-bit output and 16-bit interval feeds directly to the error detection block to check validity of the outputs. A final reduction step across all interval outputs indicates whether the computation can be accepted as valid. If invalid, batched FFT computations [82] can be restarted by a controller.

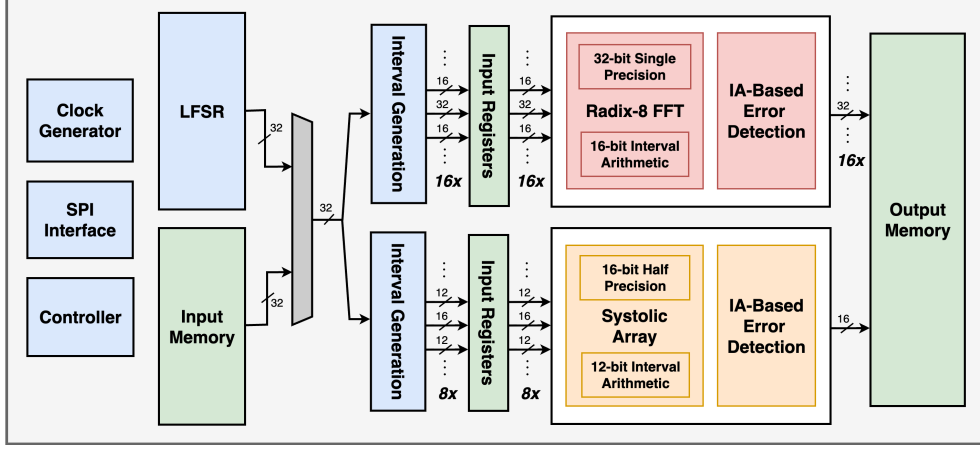


Figure 4.10: System diagram of the CosmIC test chip. The datapaths include a complex radix-8 FFT and a systolic array, each replicated in interval arithmetic. Input data can be generated through the LFSRs or specific tests can be loaded into the on-chip memory. Additional test structures enable fault injection tests and error tracking.

The bound on interval growth for the FFT is calculated by tracking the number of operations that occurs for each output. We also assume that the input dynamic range is bounded to values in the range of $[0, 1]$. For example, the first output in Fig. 4.9 incurs a maximum growth of 6 ulp from the three addition operations and we know that the maximum output dynamic range by properties of the FFT is $[0, 8]$. Thus the maximum exponent and the threshold bound is determined and can be used for error detection.

4.3.2 Interval Generation

We design a block for generating intervals on the fly as floating point numbers enter each of the computations. The numbers are rounded towards $+\infty$ and towards $-\infty$ for the desired lower precision representation: 16-bit half precision for the FFT and the custom 12-bit format for the systolic array. This creates intervals

which are exactly one ulp distance away from each other. The logic is entirely combinational for each input going into the respective block. This also enables the protected hardware to act as a drop-in replacement since the interface to the data does not change from the original hardware implementation.

4.3.3 Test Infrastructure

We implement linear feedback shift registers (LFSRs) to generate pseudo-random inputs for both the FFT and systolic array modules. For each of the modules we would also like to bound the input dynamic range, so we add additional post-processing logic to ensure the generated floating point inputs are within the fixed range. We also design synthesized memories so that specific test vectors can be loaded for test and evaluation post-silicon.

To facilitate fault injection tests, we insert a dedicated scan chain for all flops in the core datapaths. This allows us to scan out the state of the computation, flip any bit, and scan the modified state back in to resume computation. The error detection block also incorporates additional logic for tracking error counts and location across the output intervals. A ring oscillator-based on-die clock generator can provide frequencies up to the target 500 MHz core frequency.

4.3.4 Silicon Prototype

The 1mm x 1mm test chip is fabricated in a TSMC 28nm process and has a core area of 0.7mm x 0.7mm. A micrograph of the die and the layout of the core components is shown in Fig. 4.11.

The design targets a core frequency of 500 MHz. There are three power domains for the clock generator, FFT/systolic array, and other control/test logic.

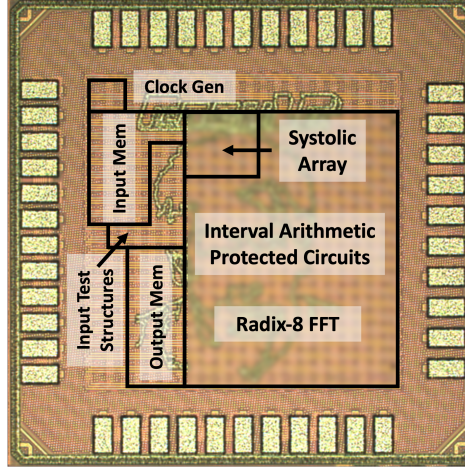


Figure 4.11: 1mm x 1mm die micrograph. There are 44 IO pads and three power domains. The protected accelerators are on the right side of the die. Additional test infrastructure including LFSRs, memories, interval generation, and clock generator are on the left side of the die.

The chip can be operated through the LFSRs which feed data directly to the datapaths or through specific test vectors that are loaded to the input memories. The input/output memories are large enough to saturate the pipeline of the datapaths.

4.3.5 Comparison against TMR

To evaluate our approach we implement a TMR version of the modules as a baseline comparison against the fabricated test chip. All results are extracted from post place-and-route designs and include the logic area required for error handling.

The proposed interval arithmetic approach achieves a 2 times reduction in area for the 8-point complex FFT and 3.3 times reduction for the systolic array compared to the traditional TMR approach. The greater area improvement in the systolic array design can be attributed to its higher fraction of multipliers which scale down faster with decreased bitwidth. We also note the potential for further

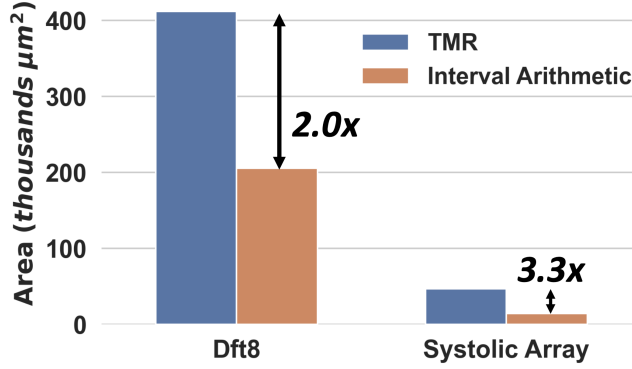


Figure 4.12: Comparison of total area between TMR and interval arithmetic protected implementations.

savings by lowering the interval precision.

4.3.6 Fault Injection Simulation

We discuss some results from fault injection simulation to demonstrate that our method effectively detects bit flips occurring in the datapath. The test cases presented below are extracting the 11th output of the FFT datapath and each one corresponds to a scenario presented in Fig. 4.4. For each result we present the lower, middle, and upper $\{x_l, x_m, x_u\}$ values with the affected value in bold.

1. Result: $\{\mathbf{0.6528}, 0.6505, 0.6509\}$. The interval check immediately catches that $0.6528 > 0.6509$ and signals that the result is invalid.
2. Result: $\{0.6504, \mathbf{0.6507}, 0.6509\}$. The center value's lower mantissa bit is affected, resulting in a value of 0.6507. However the interval result $[0.6504, 0.6509]$ is tight and still bounds the center value. Thus the error is tolerable and the result can be accepted.
3. Result: $\{0.6504, 0.6505, \mathbf{0.9004}\}$. There is an error in the MSBs of the

upper endpoint of the interval. The interval distance is much larger than the allowable bound and the result is declared as invalid.

4.3.7 Summary

This section introduced a new approach for building soft error tolerant hardware and showed how it can be applied to FFT accelerators. The design of this test chip showed that our hardware generation approach is extensible to new arithmetic units or techniques for FFT-based accelerators.

4.4 A 16nm Heterogeneous SoC for Signal Modulation Recognition

The deployment of modulation recognition (MR) systems on high performance, resource-constrained devices is critical for computationally intensive signal intelligence applications. Modern MR systems rely on a mix of machine learning (ML) approaches and traditional digital signal processing (DSP) techniques for blind signal classification, where signal characteristics are unknown a priori. The computational intensity of such algorithms has led to its development on complex, modern SoC systems consisting of CPUs, GPUs, FPGAs, and neural engines in order to reduce the time required for signal analysis, while also constrained to smaller form factors. Specialized accelerators thus offer the opportunity to realize increased performance and energy efficiency on such high performance embedded devices.

The MR system we study in this thesis consists of three phases: scan, detect, and characterize. Our focus lies in the ‘characterize’ phase, which has been shown

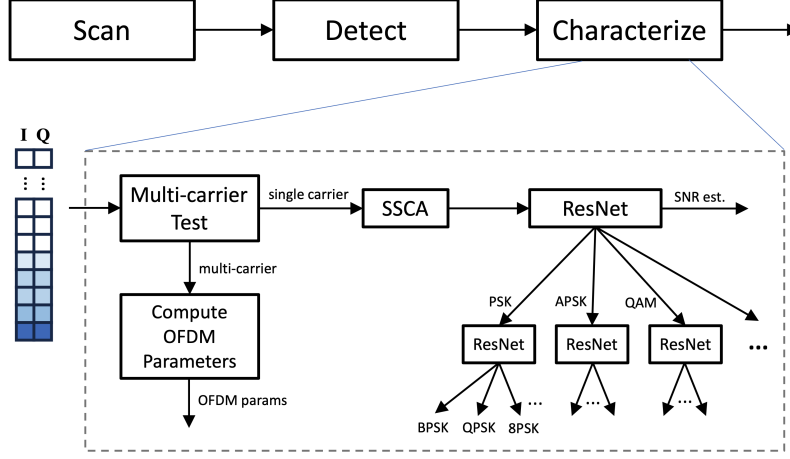


Figure 4.13: Modulation Recognition composed of scan, detect, and characterize phases. We focus primarily on the computationally intensive spectral analysis algorithms in the characterize phase for symbol rate estimation.

to take almost 50% of the entire execution time. During characterization, ML-based approaches using deep neural networks are developed for classification of signal modulation types. This is combined with traditional spectral correlation analysis techniques that analyze cyclostationary processes to produce estimates of the signal symbol rate. There are different computational methods for performing cyclic spectral analysis including frequency smoothing and time smoothing approaches that rely heavily on FFTs. We focus specifically on the Strip Spectral Correlation Algorithm (SSCA) used for analyzing cyclostationary signals.

4.4.1 Cyclostationary Processing

In this section we provide a brief background on cyclostationary processes and then outline the SSCA algorithm used to estimate the spectral correlation function (SCF). A *cyclostationary* process is a process that has statistical properties which vary periodically over time, in contrast to the widely used model of a *stationary*

process that has statistical properties which remain constant over time. Most real world modulated signals exhibit cyclostationary properties where their statistics, such as the mean, variance, or higher order moments, change at the fundamental symbol rate. Thus analyzing such properties can be used to extract signal features that aid modulation recognition systems.

The autocorrelation function of a signal is fundamental to cyclostationary processing. It is defined as

$$R_x(\tau) = \frac{1}{N} \sum_{n=-N/2}^{N/2} x \left[n + \frac{\tau}{2} \right] x^* \left[n - \frac{\tau}{2} \right] \quad (4.4)$$

where τ is the time delay. The autocorrelation function is a measure of the similarity of the signal against time-shifted versions of itself. So for example, if a signal transmits symbols at a rate of N symbols/second, then at multiples of the symbol period we would expect to see stronger autocorrelation. In this MR pipeline, the goal of cyclostationary processing is essentially to determine where the periodicity occurs in the autocorrelation function to estimate the signal symbol rate.

Taking the Fourier Transform of the time-varying autocorrelation function reveals information on the periodic patterns, known as cyclic features. The Cyclic Autocorrelation Function (CAF) is the Fourier series coefficients of the autocorrelation function:

$$R_x^\alpha[k] = \lim_{N \rightarrow \infty} \frac{1}{2N+1} \sum_{n=-N}^N x[n+k] e^{-j\pi\alpha(n+k)} (x[n] e^{j\pi\alpha n})^* \quad (4.5)$$

where α is the cycle frequency and k is the time delay. The cycle frequency, α , represents the frequency at which the signal's autocorrelation is changing. At $\alpha = 0$, the CAF becomes the standard autocorrelation function. Non-zero cycle

frequencies, $\alpha \neq 0$, are called cyclic features that indicate a stronger periodic correlation when non-zero.

The Spectral Coherence Function (SCF) now takes the Fourier Transform of the CAF with respect to the time delay of the autocorrelation, τ :

$$S_x^\alpha(f) = \sum_{k=-\infty}^{\infty} R_x^\alpha[k] e^{-i2\pi f k} \quad (4.6)$$

where α is again the cycle frequency and f is the spectral frequency. The intuition behind the SCF is that it reveals periodicities occurring in the power spectral density (PSD) of the signal. At $\alpha = 0$, the SCF is equivalent to the PSD. When the SCF is non-zero at certain cyclic frequencies, this means there is correlation at that spectral frequency and cyclic frequency pair. Estimating the SCF is the key computational task for symbol rate estimation in this MR system. There are both time-smoothed and frequency-smoothed methods which rely on high performance FFT implementations for real time operation.

4.4.2 The Strip Spectral Correlation Algorithm (SSCA)

We focus on one approach for SCF estimation, called the Strip Spectral Correlation Algorithm (SSCA) [110]. The SSCA is an FFT-based time smoothing approach that can be broken down into three parts for SCF estimation [111].

1. A series of N N_p -point sliding window Fourier transforms which channelize the input signal into N narrowband signals.
2. Multiplication of the N N_p -point signals with the input signal and complex exponentials.
3. N_p N -point Fourier transforms produces the final SCF estimates.

The SCF will have higher magnitude at cycle and spectral frequency pairs that are strongly correlated. These points need to be extracted in a final stage that computes a coherence value for every pair and thresholds them against a fixed value. The coherence values are extracted from an estimate of the power spectral density (PSD) which normalizes the SCF. A high-level view of the computational pattern is shown in Fig. 4.15.

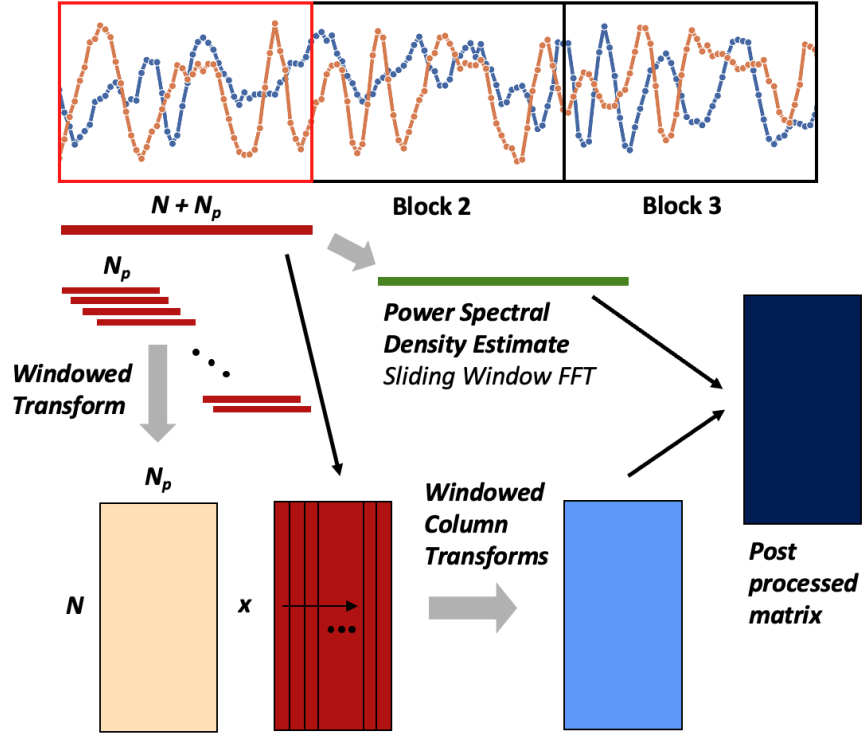


Figure 4.14: Whiteboard view of the SSCA Algorithm.

The following figures show an end-to-end example which applies the SSCA to an 8-PSK modulated signal with symbol rate of something. The complex-valued signal is shown in the top left diagram where the real and imaginary parts are

separated and shown individually. The top right plot is the estimate of the power spectral density of the signal.

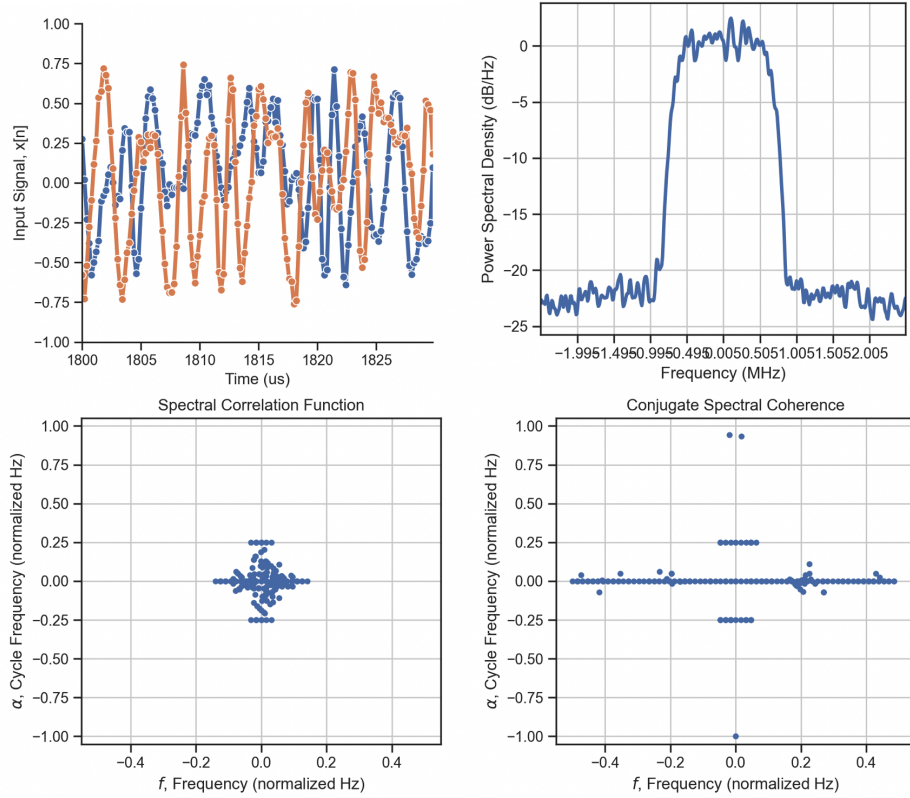


Figure 4.15: Plots of input signal and steps of the SSCA algorithm for symbol rate estimation. The bottom right plot shows the strong cycle frequency features present in the spectral coherence at $\alpha = 0.25$.

4.4.3 Accelerator Implementation for SSCA

In this section we demonstrate how our approach is applied to the design of specialized hardware for the SSCA and the minimal extensions incorporated into the architectural template. The computation breaks down to a pattern similar to a

two dimensional FFT but has a series of element-wise multiplications with the input signal that occurs between the transforms for each dimension. The transforms in the first stage of the algorithm are also sliding windows FFTs which occur over a fixed unit stride. The current accelerator architecture presented in Chapter 3 supports the multiplications occurring between the transforms but would benefit from additional components that support the sliding window nature of the initial transforms. Consecutive transforms in the first stage would waste computational cycles and power reloading the same $N_p - 1$ elements when only the next strided element is required for the following transform. To address this issue, we design a specialized buffer, similar to line or window buffers used for image processing, which first preloads the vector for the first transform. Then, the ensuing transforms only require the buffer to operate in a shift register mode that serially shifts out its first element and loads the next element from on-chip memory, effectively saving cycles and power spent on data movement. We integrate this structure into the accelerator template, enabling a new custom tailored accelerator that supports a common computational pattern seen in spectral analysis algorithms. The new accelerator architecture is depicted below in Fig. 4.16.

The two new components in the accelerator architecture are a dedicated input memory for buffering unprocessed signal data and the specialized buffer to support sliding window transforms. The input memory is double buffered such that the next block of data is fetched while we compute with the currently loaded block. The next iterations then ping pong between the two buffers. The accelerator descriptors are modified to support data movement to and from the sliding window buffers.

Modulated signal data is collected from a software-defined radio (SDR) to determine the required data precision and approximate FFT sizes the accelerator

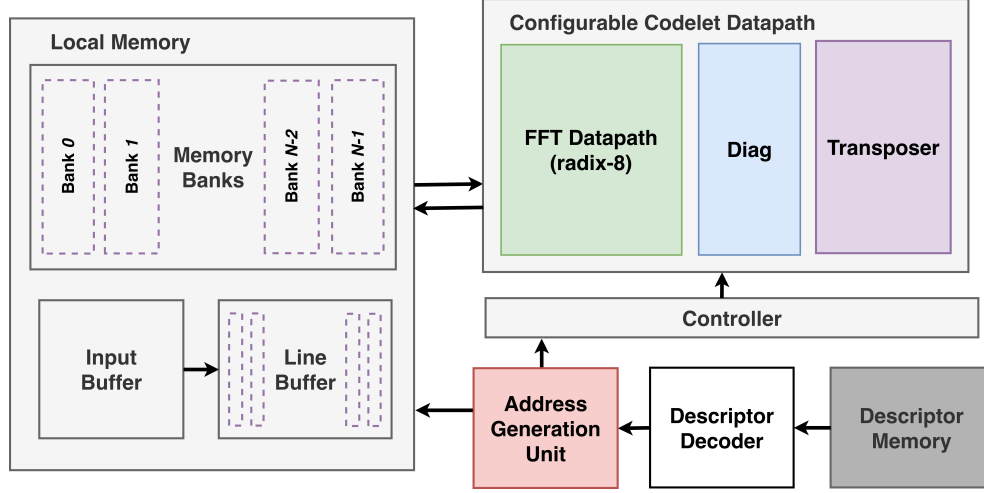


Figure 4.16: SSCA Accelerator Architecture. The radix-8 hardware codelet now interfaces to banked scratchpad memories to store intermediate data and specialized buffers for loading input data in a strided fashion.

will need to support. The transmitted data is mapped to 8-PSK symbols that are shaped with a square-root raised cosine pulse at a sampling rate of 5MHz. Python-based software implementations using the collected data show that 16-bit half-precision arithmetic units with parameters $N = 4096$ and $N_p = 64$ provide enough precision in the SCF estimate. We select a radix size of 8 and pass the necessary on-chip memory requirements and data precision to the generator to produce RTL-level descriptions of the accelerator.

4.4.4 SSCA Program Execution

The SSCA processes blocks of $N + N_p - 1$ samples at a time, where each block produces an estimate of the SCF. The steps of the program execution are described below:

1. The SSCA accelerator program targeting parameters of $N = 4096$ and $N_p = 64$ is moved to the accelerator’s dedicated plan memory through a CPU-initiated DMA transfer.
2. Once the program transfer has completed, the first block of $N + N_p$ samples is loaded into the input data memory which holds unprocessed signal data.
3. We use custom RISC-V instructions for accelerator invocation, which first passes a pointer to the SSCA accelerator program and then pulses the start signal to initiate accelerator execution.
4. The accelerator program completes and writes to a status register to indicate that is done. The CPU polls this status register using the CFU interface.

The first descriptor of the program instructs the accelerator to populate the line buffer and execute the first N_p -point transform. The next iteration of the N_p -point transforms retrieves data from the line buffer, which updates before the transform begins, and writes outputs to the banked shared scratchpad memory. The element-wise multiplications are then fused into the final stage and we continue with the set of N_p N -point transforms in the other dimension.

4.4.5 RISC-V SoC Architecture

The VexRiscv SoC [69] consists of a 32-bit RISC-V RV32IMF core tightly coupled to domain-specific accelerators targeting the MR pipeline in Fig. ?? and security schemes. The custom function unit (CFU) interface manages the interactions between CPU, accelerators, and DMA engine including accelerator invocation, data transfers, and other device-specific control/status signaling. There is 192KB of

on-chip memory and another 2MB of shared scratchpad memory which the accelerator and CPU can access. A dedicated on-die accelerator clock generator enables the SoC and accelerators to operate at independent frequencies for improved system energy efficiency. Because the accelerator operates at higher frequency than the CPU, we use asynchronous FIFOs to separate the clock domains at the CFU interface.

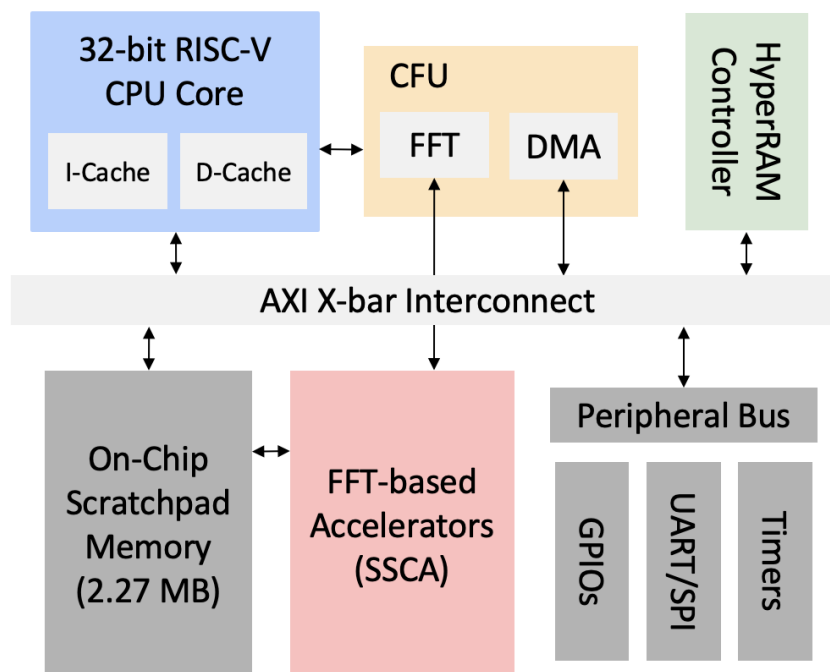


Figure 4.17: VexRiscv SoC consisting of a 32-bit RISC-V CPU tightly coupled to the SSCA accelerator and DMA engine for data transfers. There is a total of 2.27MB of on-chip scratchpad memory

The CPU to accelerator CFU interface supports custom RISC-V instruction commands that pass register file operands to the accelerator. The tightly-coupled

interface allows for fast configuration of the problem specific parameters including windowing values and the SSCA problem sizes. Accelerator memories including the plan, input, and data memories are all scratchpad-based SRAM memories that are shared with the RISC-V CPU. Access to the scratchpad is controlled through a multiplexer such that either the CPU or the accelerator accesses the memories, but both cannot interface simultaneously. When the scratchpad is configured for accelerator mode, the hardware codelet has access to the plan/input memories and the full bandwidth of the parallel data banks. In CPU mode, the shared scratchpad is accessed through requests sent to the AXI slave port.

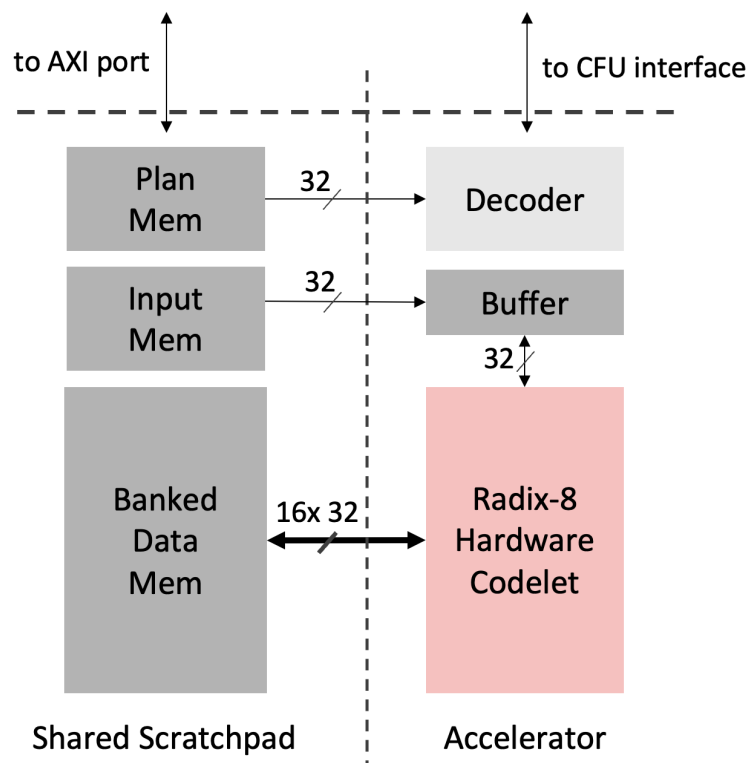


Figure 4.18: Shared scratchpad to accelerator interface. Access to the memories is controlled such that requests can only come from the accelerator or AXI port.

The single-port, 4KB plan memory holds the SSCA accelerator program specified by 32-bit wide descriptors. There is a total of 32.5KB of input memory to hold double-buffered blocks of $N + N_p$ samples of input signal data, interfacing directly to the end of the line buffer. The intermediate data memories provide 16 parallel banks of 1R1W memories to support the bandwidth required by the radix-8 FFT hardware codelet.

4.4.6 Post-PnR Layout

The SoC is implemented on a 2mm x 2mm die and fabricated in a 16nm FinFET process. Fig. 4.19 shows the post-PNR layout capture featuring the RISC-V CPU, SSCA Accelerator, caches, and shared scratchpad memories between the CPU and accelerator. Other circuits targeting deep learning and security schemes are not shown. The total core area is 3.38mm² and the annotated components occupy a total of 1.85mm².

The shared data scratchpads are organized into two groups of eight memory banks placed on either side of the SSCA accelerator. The RISC-V CPU, caches, and DMA engine are in the middle of the die. An additional 192KB of on-chip SRAM memory on the right side can be accessed directly from the CPU or through DMA requests. There are two clock generators, one SoC-level clock generator placed in the bottom right corner and another accelerator clock generator near the SSCA accelerator. We show an implementation summary and area breakdown of the SoC components below.

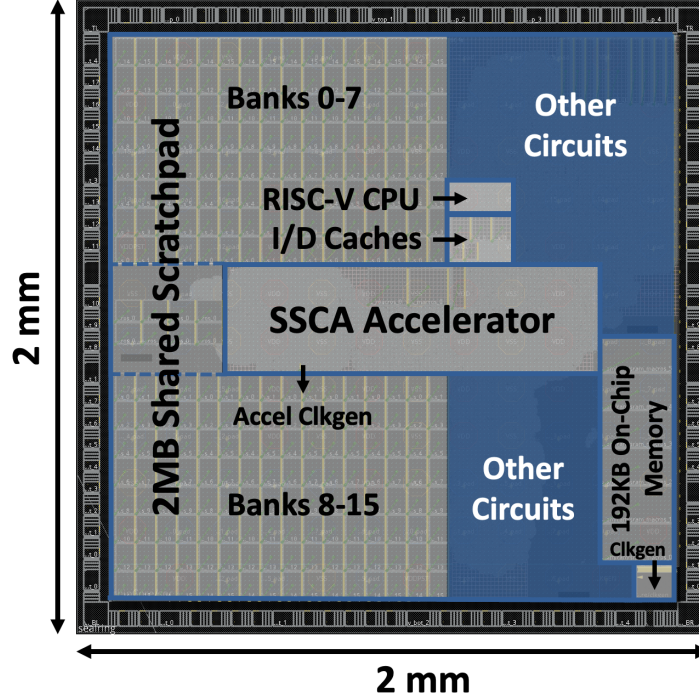


Figure 4.19: Layout capture of the fabricated 2mm x 2mm test chip in a 16nm FinFET process. The SSCA accelerator and RISC-V core are in the middle of the die, with banked scratchpad memories on the top and bottom of the accelerator.

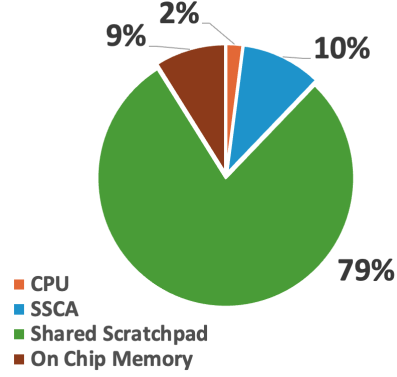
4.4.7 Results and Comparison

The SSCA accelerator completes the $N=4096$ and $N_p=64$ SCF estimate in 199K cycles at a 1 GHz accelerator clock frequency. When running the SSCA program on the accelerator, the entire SoC consumes 390mW of power based on Voltus power simulations using the post-PNR layout database.

We compare our custom accelerator targeting the SSCA against other CPU and GPU-based software implementations and previous FPGA designs. The software implementations of the SSCA target the 12-core Arm Cortex-A78AE 64-bit CPU and Ampere GPU on an NVIDIA Jetson AGX Orin configured to the maximum

Technology	TSMC 16nm
Chip Area	4.00 mm ²
Core Area	3.38 mm ²
IO Voltage	1.8 V
Core Voltage	0.8 V
Power Bumps	68
Signal Bumps	72
Power Domains	2

(a) Implementation Summary



(b) Area breakdown of SoC components.

Figure 4.20: Summary of 2mm $\times$ 2mm test chip implementation in a TSMC 16nm FinFET process. The area breakdown considers the annotated SoC components shown in Fig. 4.19.

power mode. The CPU program uses pyFFTW, which provides a Python wrapper for FFTW library calls, to execute the size N and N_p transforms in single-precision floating point using all CPU threads. The GPU-based implementation relies on calls to cuPy FFT routines which offers support for half-precision data types. Other FPGA-based accelerators for the SSCA target the AMD Versal VCK5000 [112] and ZCU118. A detailed comparison table with normalized metrics is shown below in Table 4.1. Comparing a normalized throughput metric that accounts for different SSCA parameters and precision, our implementation is 0.8 times lower than the Versal VCK5000, but about 1.5 times better than the ZCU118 and 6 times better than the Orin GPU. In terms of a normalized energy efficiency metric, the accelerator achieves a 16x improvement over the Versal design and is 128x better than the GPU implementation. Finally, combining into a normalized energy-delay product, our ASIC implementation is about 25 times better than the Versal FPGA

	Proposed	Versal	ZCU118	Orin GPU
Process	16 nm	7 nm	16 nm	8 nm
Frequency	1 GHz	–	–	1.3 GHz
Precision	FP16	FP32	FP32	FP16
FFT Size, N	2^{12}	2^{20}	2^{14}	2^{12}
FFT Size, N_p	2^6	2^6	2^7	2^6
Latency (ms)	0.20	114	5.65	1.19
Power (W)	0.39	8.00	–	10.76
Norm. Throughput	126.62	160.12	82.40	21.11
Norm. Energy/SCF	0.79	12.57	–	128.32
Norm. Energy-Delay	0.16	3.95	–	152.97

Table 4.1: ASIC implementation summary and comparisons

design. Our ASIC approach for accelerating the SSCA thus shows significant gains in normalized energy efficiency and EDP metrics over other state-of-the art approaches.

4.4.8 Discussion

Testing and Verification. Accelerator programs written against the library greatly enhance design verification and testing support. Configurations of the hardware can be tested individually and gradually built up into larger problems or applications, and at every step an accelerator program and set of test vectors can be inserted at different levels of design abstraction. During this tapeout process, the programs and test data were first verified with an accelerator software simulation model for functional correctness, while the same set of tests were then ported to bit-level representations for cycle-level RTL and gate-level simulation. Then, when testing baremetal programs running on the SoC, test data that was used for

accelerator verification could easily be printed to a C header file and compiled with the main driver program for SoC-level verification. These tools provide a unified framework that eases the verification cycles at different levels of abstraction and complexity, from higher-level architectural models to gate-level netlists and post-layout databases.

4.5 DFTs of Real Data

Many applications in scientific computing rely on partial differential equation (PDE) solvers using FFT-based spectral methods, which commonly perform computations on real-valued data where complex inputs are all zero. In practice, software and hardware implementations for real FFTs exploit this to reduce the number of operations and memory requirements for improved performance.

High performance FFT software libraries [25, 26, 28] offer specialized routines for real transforms as part of their standard API. Hardware implementations have taken a couple approaches for realizing the real FFT. One method [113–116] is to prune out unnecessary arithmetic operations in the butterflies by exploiting the conjugate symmetric property of real FFTs and the fact that half of the inputs are now zero. The other is an algorithmic approach in which a single real FFT can be realized using a half-sized complex FFT with a final post-processing step. This technique is used in a TI DSP [117]. In this work we take the latter approach, and our insight is that the existing hardware support for complex transforms only needs to be supplemented by a modified radix-2 butterfly in the codelet datapath with a specialized final pass over the data. The following sections derive this result and discuss how the codelet architecture and generation framework is extended for supporting real transforms.

4.5.1 Real FFT via Half-size Complex FFT

We begin by stating the property that a DFT of real-valued data produces conjugate symmetric outputs, which is used to derive the following real transform algorithm. More formally, given a sequence, $x[n]$, of length N consisting of real-valued data, the DFT of $x[n]$ has the following property:

$$X[k] = X^*[N - k] \quad (4.7)$$

where $X^*[N - k]$ is the complex conjugate of the sequence $X[N - k]$ and $n \in [0, N - 1]$. Note that indexing is performed modulo N .

Suppose we have the same sequence, $x[n]$, of length N split into its even-indexed data, $x_e[m]$, and odd-indexed data, $x_o[m]$, where $m \in [0, N/2)$.

$$x_e[m] = x[2m] \quad (4.8)$$

$$x_o[m] = x[2m + 1] \quad (4.9)$$

The even and odd sequences can be treated as a single complex sequence, $y[m] = x_e[m] + jx_o[m]$ which does not require any repacking since sequential real elements are already in complex interleaved format. Then, by linearity of the DFT, the transform of y is:

$$Y[l] = X_e[l] + jX_o[l], \quad l \in [0, N/2) \quad (4.10)$$

These two components, X_e and X_o , are exactly the two sequences that form the first stage radix-2 decomposition of the transform $X[k]$ on the original real se-

quence, $x[n]$:

$$X[k] = X_e[k] + e^{-j2\pi k/N} X_o[k], \quad (4.11)$$

$$X[k + N/2] = X_e[k] - e^{-j2\pi k/N} X_o[k], \quad k \in [0, N/2) \quad (4.12)$$

So once the even and odd transforms, X_e and X_o , in Eq. 4.10 are extracted from the transform Y of the complex signal, the real transform X can be reconstructed. Furthermore, only the first $N/2 + 1$ elements need to be computed due to the conjugate symmetry property of real-valued transforms.

To obtain the even and odd transforms, we first use the conjugate symmetric property to show that

$$\begin{aligned} Y^*[N/2 - k] &= X_e^*[N/2 - k] - jX_o^*[N/2 - k] \\ &= X_e[k] - jX_o[k]. \end{aligned} \quad (4.13)$$

Then, we can combine this result with the original expression for Y to get

$$X_e[k] = \frac{1}{2}(Y[k] + Y^*[N/2 - k]) \quad (4.14)$$

and

$$X_o[k] = -\frac{j}{2}(Y[k] - Y^*[N/2 - k]). \quad (4.15)$$

Substituting back into Eq. 4.11,

$$\begin{aligned}
X[k] &= \frac{1}{2}(Y[k] + Y^*[N/2 - k]) \\
&\quad + e^{-j2\pi k/N} \left(-\frac{j}{2}(Y[k] - Y^*[N/2 - k]) \right) \\
&= \left(\frac{1 - je^{-j2\pi k/N}}{2} \right) Y[k] + \left(\frac{1 + je^{-j2\pi k/N}}{2} \right) Y^*[N/2 - k] \tag{4.16}
\end{aligned}$$

The expression can be further manipulated by pulling out the element-wise multiplication with the complex valued constant, $\frac{1 - je^{-j2\pi k/N}}{2}$, and absorbing it into the final stage of the complex transform that produces Y . So the transform then becomes

$$Y_1[k] = \frac{1 - je^{-j2\pi k/N}}{2} Y[k] \tag{4.17}$$

$$X[k] = Y_1[k] + \left(\frac{1 - je^{j2\pi k/N}}{1 + je^{j2\pi k/N}} Y_1[N/2 - k] \right)^* \tag{4.18}$$

The interpretation of this result is that the real transform can be broken into two parts. First, a complex transform on the real sequence (reinterpreted as complex) with multiplication by complex-valued constants fused into the final stage. Then, results from the first stage (in Eq. 4.17) starting from $k = 1$ and $k = N/2 - 1$ are used pairwise, with the next pair loaded at increasing unit stride starting from $k = 1$ and decreasing unit stride from $k = N/2 - 1$. Each pair of inputs then follows the computation in Eq. 4.18.

Finally we note that $X[0]$ and $X[N/2]$ are both real and are simple to compute:

$$X[0] = \text{Re}(Y[0]) + \text{Im}(Y[0]) \tag{4.19}$$

$$X[N/2] = \text{Re}(Y[0]) - \text{Im}(Y[0]). \tag{4.20}$$

Thus we have shown how the first $N/2 + 1$ values of the real FFT can be derived from a half-size complex FFT.

Inverse Transform. Computing the inverse real transform of a conjugate symmetric signal, $X[k]$, follows a similar pattern to computing the forward real transform except the procedure occurs in reverse order. The two stages are:

$$X_1[k] = X[k] + \left(\frac{1 + je^{-j2\pi k/N}}{1 - je^{-j2\pi k/N}} X[N/2 - k] \right)^* \quad (4.21)$$

$$Y[k] = \frac{1 + je^{j2\pi k/N}}{2} X_1[k] \quad (4.22)$$

We leave out the full derivation but the idea is essentially the same. The even and odd transforms are recovered from the conjugate symmetric transform, X , and then combined to form Y . The final stage is a complex inverse FFT which results in the real transform result where the complex interleaved format is reinterpreted as sequential real data. Manipulating the expressions results in Eqs. 4.21 and 4.22 which follows the same computational pattern as the forward real FFT.

4.5.2 Real Codelet Architecture

The overall dataflow is shown in Fig. 4.21. The first stage consists of a complex FFT fused with complex multiplication which is completely supported by the current codelet architecture. The second stage requires some modification in order to load inputs in the special pattern and support the computation. Since each pair of inputs comes from opposite ends of the array, we implement a set of registers to buffer the memory response data from each request. For a design with B banks of memory, the first B registers holds the set of data from sequentially increasing memory requests and a second set of B registers holds the data coming from the

end of the array. Since both ends of the array need to go through the computation in Eq. 4.18, we parallelize the computation in Eq. 4.18 for each sequence of B inputs across time. The codelet can already be configured to first perform the Diag multiplication with a complex-valued constant, so the other required modification is to support the calculation in Eq. 4.18.

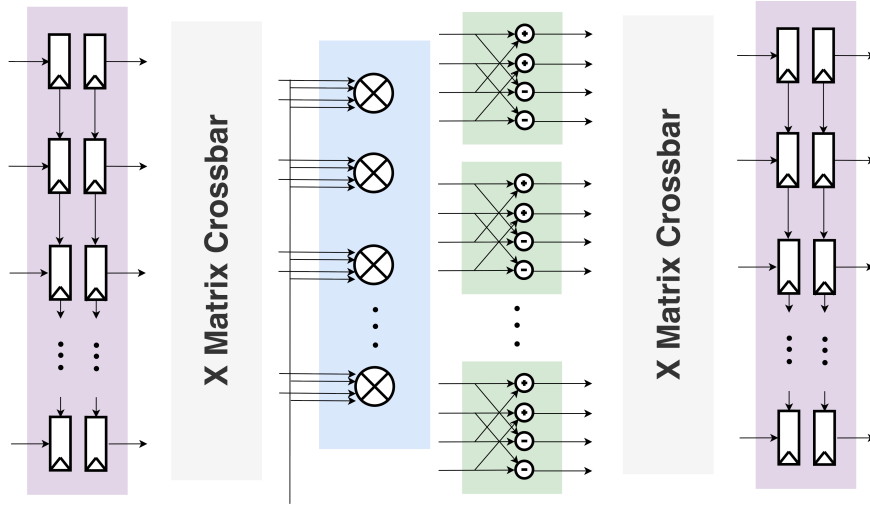


Figure 4.21: Real Codelet Datapath

Because we operate on pairwise inputs the structure of the radix-2 butterfly can be reused, where we simply extract the result from the adder and use a control signal that allows for only the second input to be conjugated on the way into the butterfly. The interface between the buffers and the Diag/modified butterflies is a custom crossbar with connections that only support the “X”-shape pattern required. Once the outputs from complex multiplication reach the butterfly structure, we retrieve the original inputs from the buffers to perform the addition. A similar crossbar is used once the computation is complete and is buffered in another

set of registers. We also double buffer on the input side to load the next sequence while the current one executes through the pipeline. All complex constants are precomputed and stored in memory.

4.5.3 Integration with Design Framework

Hardware. The modified radix-2 butterfly for real FFT support can be integrated into the existing design framework by simply treating it as another option in the Chisel package of generated FFT hardware instances. For a given design, we instantiate as many modified radix-2 butterflies as can be supported by the number of memory banks. Once the hardware codelet is configured for operating in ‘real’ mode, shown in Fig. 4.21, the buffers and custom crossbar are enabled. The butterflies can also be configured to perform the regular radix-2 operation for complex transforms.

Software. Targeting the real FFT codelet in software is also relatively easily to integrate into the toolchain. The real codelet is assigned an ID and if the current radix configuration of the batch is set to the ID then all data flows through the real codelet datapath. To support the reverse indexing scheme at unit stride, we require the CPU to pass the FFT size to the accelerator which will be stored in a control register.

To summarize, the real-valued transform first computes the half-size complex FFT in which the final stage fuses in multiplication with complex-valued constants. Finally, the modified radix-2 butterflies and permutation structure enables the final batch over the data to construct the real transform. We also showed how the inverse transform can use the same structures but with the computation performed in the reverse order.

Chapter 5

Performance Modeling and Discussion

The previous chapters showed how the proposed design approach can be implemented and the new design tools that were used for fabricating silicon testchips. This chapter concludes the main contributions of the thesis by first walking through the end-to-end execution of the LibraryX-ASIC example and finally showing some results from performance modeling.

5.1 Execution of LibraryX-ASIC Example

We now show how the accelerator code is compiled and executed on an instantiation of the accelerator shown in Fig. [3.9](#).

Runtime Compilation and Execution. LibraryX-ASIC compiles the generated code into a dynamic library and immediately links against it. This enables access of the generated functions using the user’s input and output buffers. The

```

1  #include <iostream>
2  #include <complex>
3  #include <vector>
4  #include "fftw3.h"
5  #include "LibraryX_ASIC.hpp"
6  using namespace std;
7  int main() {
8      int N = 64;
9      int sign = -1;
10     u_int f = FFTW_ESTIMATE;
11     vector<complex<float>> input(N);
12     vector<complex<float>> output(N,0.0);
13
14     buildInput(input);
15
16     //call is replaced with accelerator offload
17     //and executed
18     fftwf_plan p = fftwf_plan_dft_1d(N,
19         (fftwf_complex*)input.data(),
20         (fftwf_complex*)output.data(), sign, f);
21     fftwf_execute(p);
22
23     //output buffer contains accelerator result
24     checkOutput(output);
25
26     fftwf_destroy_plan(p);
27     return 0;
28 }

```

(a) Example FFT program.

```

1  #include "model.h"
2  #include "utils.h"
3  #include "accel.h"
4
5  Program dft_desc[7] = {
6      {CONFIGI,8,0,0,1,0,1,1,0,8},
7      {MEMI,MEM_IN,1,8,0xFF,LOCAL_MEM,0},
8      {MEMI,MEM_DIAG,1,8,0xFF,LOCAL_MEM,
9          LOCAL_MEM_REGION_SIZE};
10     {MEMI,MEM_OUT, 1, 8, 0xFF, LOCAL_MEM,0};
11     {CONFIGI,8,0,0,0,0,1,1,1,8};
12     {MEMI,MEM_IN,1,8,0xFF,LOCAL_MEM,0};
13     {MEMI,MEM_OUT,1,8,0xFF,LOCAL_MEM,0}
14 };
15
16 void dft64(float *Y, float *X) {
17     enter();
18     float *T23;
19     T23 = initTwiddles64();
20     dmaLoad(LOCAL_MEM, 0, 0, 8, 1, X, 8, 8, 8);
21     dmaLoad(LOCAL_MEM, 1, 0, 8, 1, T23, 8, 8, 8);
22     // Invoke Accelerator
23     executePlan(0, dft_desc);
24     dmaStore(LOCAL_MEM, 0, 0, 1, 8, Y, 8, 8, 8);
25     exit();
26 }
27
28

```

(b) SPIRAL generated accelerator code.

Figure 5.1: The FFT application in (a) written against FFTW will be executed on an FFT accelerator without user modification using LibraryX-ASIC. The output buffer will be populated as if nothing changed. The generated code in (b) uses information from SPIRAL’s FFT description include input ranges and input and output strides.

generated program in Fig. 5.1b consists of two parts: the host code defined on line 16 in function `dft64()` which runs on a controlling CPU and the accelerator code defined by the data structure `dft_desc` that is executed on the FFT hardware. To facilitate code generation targeting the accelerator, we have designed an API that enables data movement to/from on-chip local memory and main memory, accelerator invocation, and the FFT accelerator program itself. We outline the execution flow of the program on a high level abstract machine shown in Fig. 5.2. The three

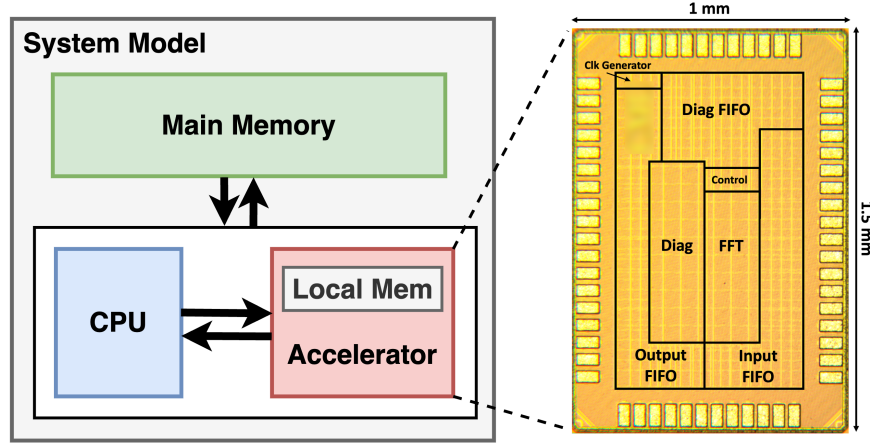


Figure 5.2: High level abstract system model of the CPU-accelerator system. The model consists of a CPU controller coupled to the accelerator, fast on-chip local memory, and main memory. The micrograph shows a silicon-verified FFT accelerator implementing a part of the architecture in Fig. 3.9.

main components are (1) main memory with a mechanism for data transfer, (2) the accelerator which interfaces to fast on-chip local memory, and (3) a controller CPU coupled to the accelerator.

Fig. 5.3 now walks through the execution of the generated program. The program first initiates input data transfer from main memory to the accelerator’s local memory with the function call to `dmaLoad()`. Once data transfer is complete, the accelerator is invoked from the CPU and passes a memory pointer to the base address of the descriptor array, `dft_desc`. The accelerator program is defined by this array that constructs the byte code that programs the accelerator. The accelerator then fetches, decodes, and executes all descriptors in the program. Finally, the accelerator signals completion to the CPU, and a DMA store request is issued to transfer output data from local memory back to main memory. At function call exit the accelerator output now exists in the output buffer.

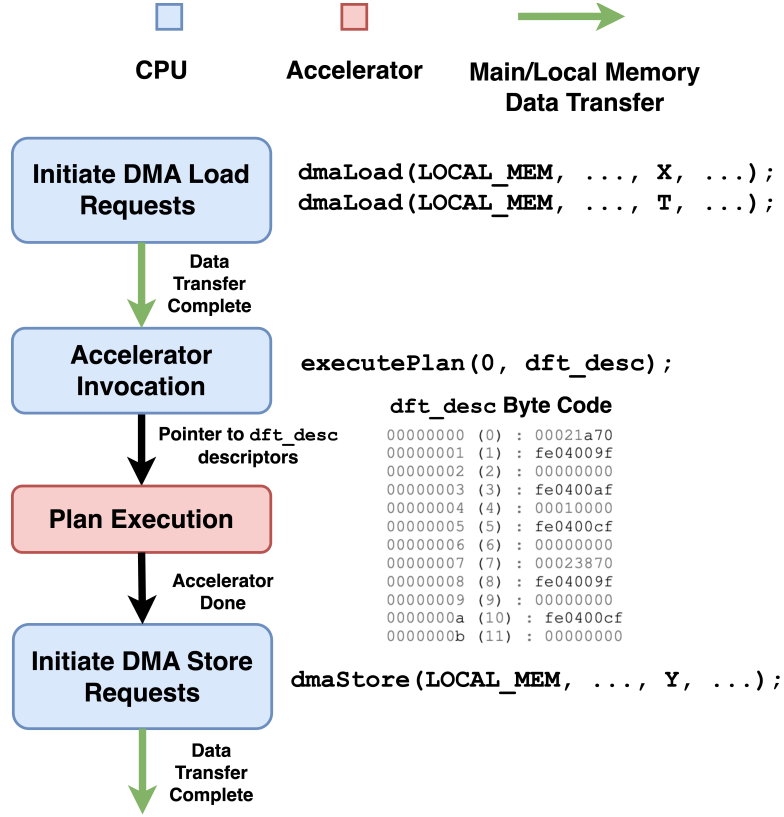


Figure 5.3: Flow chart demonstrating the execution of the generated accelerator program.

5.2 LibraryX-ASIC Results

Experimental Setup. We show preliminary results of LibraryX-ASIC for FFTs of various sizes against software implementations. CPU baseline results are obtained from running FFTW on a 20-core Intel Xeon E5-2698v4 with 256 GB of RDIMM DDR4. For GPU evaluation, we run cuFFT on an Nvidia H100. Accelerator performance results are based on a cycle-accurate accelerator model that is calibrated against real silicon measurements from test chips [?, 118] taped on a TSMC 28nm process. The performance model simulates the abstract machine

model shown in Fig. 5.3, where the controlling CPU is a single core of the Intel Xeon E5-2698 CPU, main memory consists of 256 GB of RDIMM DDR4, and the accelerator interfaces to 256 kB of banked, SRAM-based local memory. The FFT accelerator core accelerates a radix-8 twiddle codelet. For GPU comparisons, we target off-chip HBM3e DRAM with the same accelerator configuration.

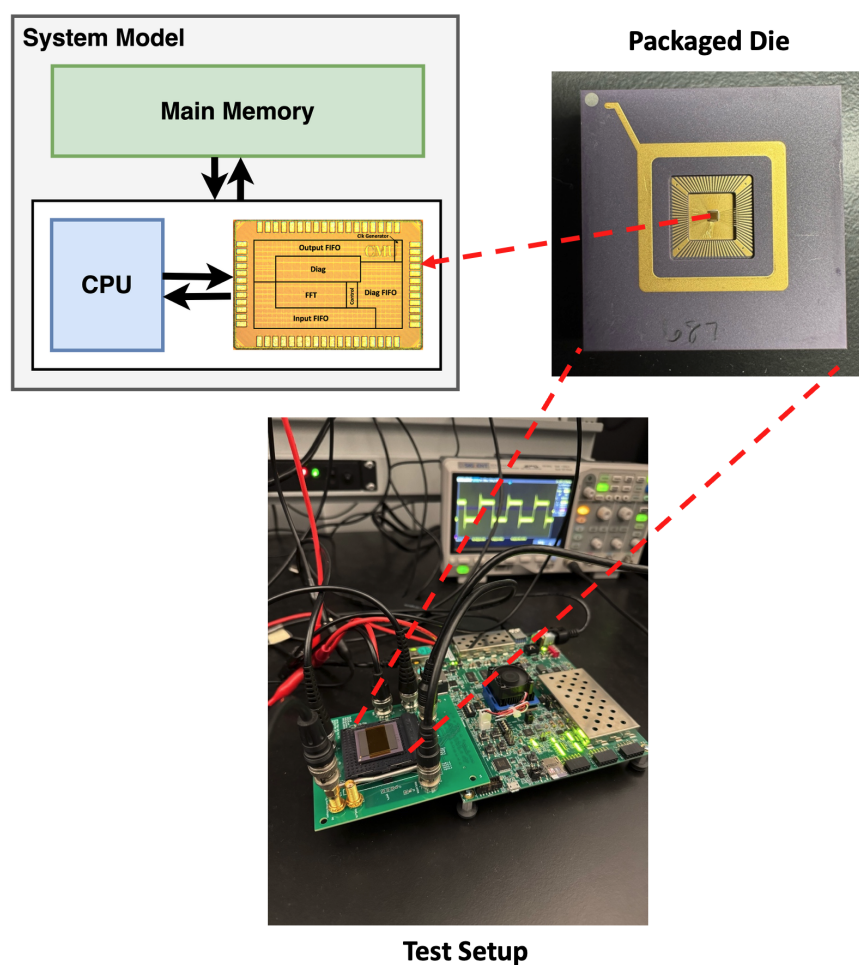


Figure 5.4: Test setup of custom FFT ASIC and integration of measured results into LibraryX-ASIC experimental setup.

Results. The execution times in microseconds are shown in Fig. 5.5 and 5.6 for power-of-8 1D complex FFTs ranging from 8 to 4096. LibraryX-ASIC targeting the CPU-accelerator system achieves speedups of 11x – 23x as compared to running FFTW on CPU only. The performance results demonstrate an order of magnitude improvement in execution time for a real FFT program running on the custom FFT ASIC through the LibraryX-ASIC framework.

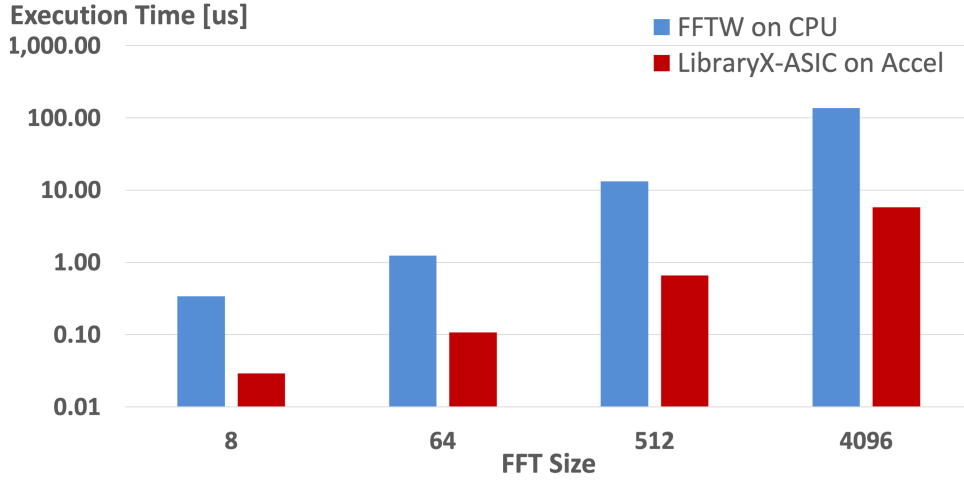


Figure 5.5: FFTW on CPU vs. LibraryX-ASIC on accelerator system.

Compared to cuFFT running on an H100 GPU, LibraryX-ASIC also provides up to an order of magnitude speedup at smaller size FFTs. Improved speedup against the GPU at larger FFT sizes can be achieved by scaling the number of hardened codelets in the FFT accelerator. These results demonstrate the LibraryX-ASIC framework automatically targeting a custom FFT accelerator through CPU and GPU FFT library calls.

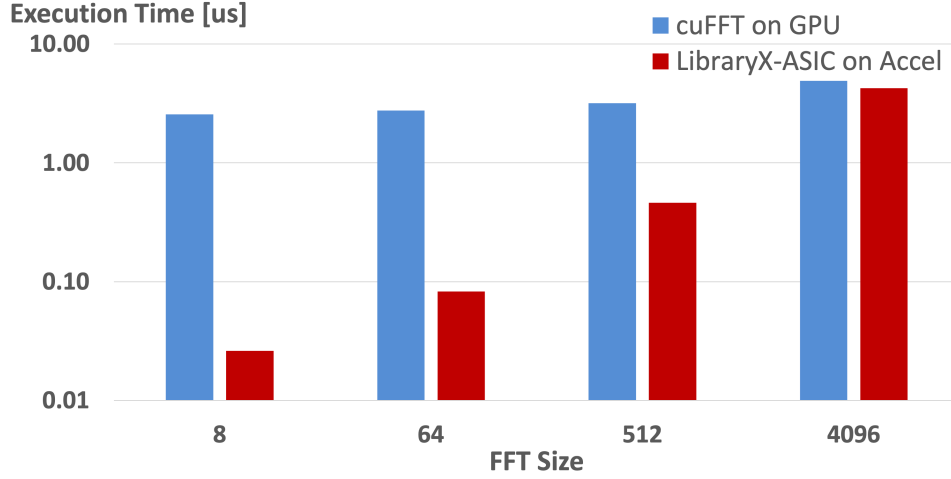


Figure 5.6: cuFFT on GPU vs. LibraryX-ASIC on accelerator system.

5.3 Discussion

The accelerator programs are constructed such that the codelet pipeline is saturated with no bubbles within a batch. Thus the only overhead comes from the startup cost to fill the pipeline and the cases in which the pipeline needs to be flushed between batches of codelet invocations. However, these costs are low and are amortized across many independent batches of codelet computations. We show a concrete breakdown for the execution of a complex 4096-point FFT in Fig. 5.7.

In this example, data is first moved from main memory to local accelerator memory and interleaved across the memory banks. Figure 5.7 then shows the execution breakdown of a 4096-point complex FFT which requires no intermediate round trips to memory since the entire vector fits in local on-chip memory and the transpose unit realizes the necessary data permutations. The program implements the four-step FFT algorithm using a square root decomposition, $4096 = 64 \times 64$, and each stage of vectorized 64-point FFTs is implemented iteratively with radix-8

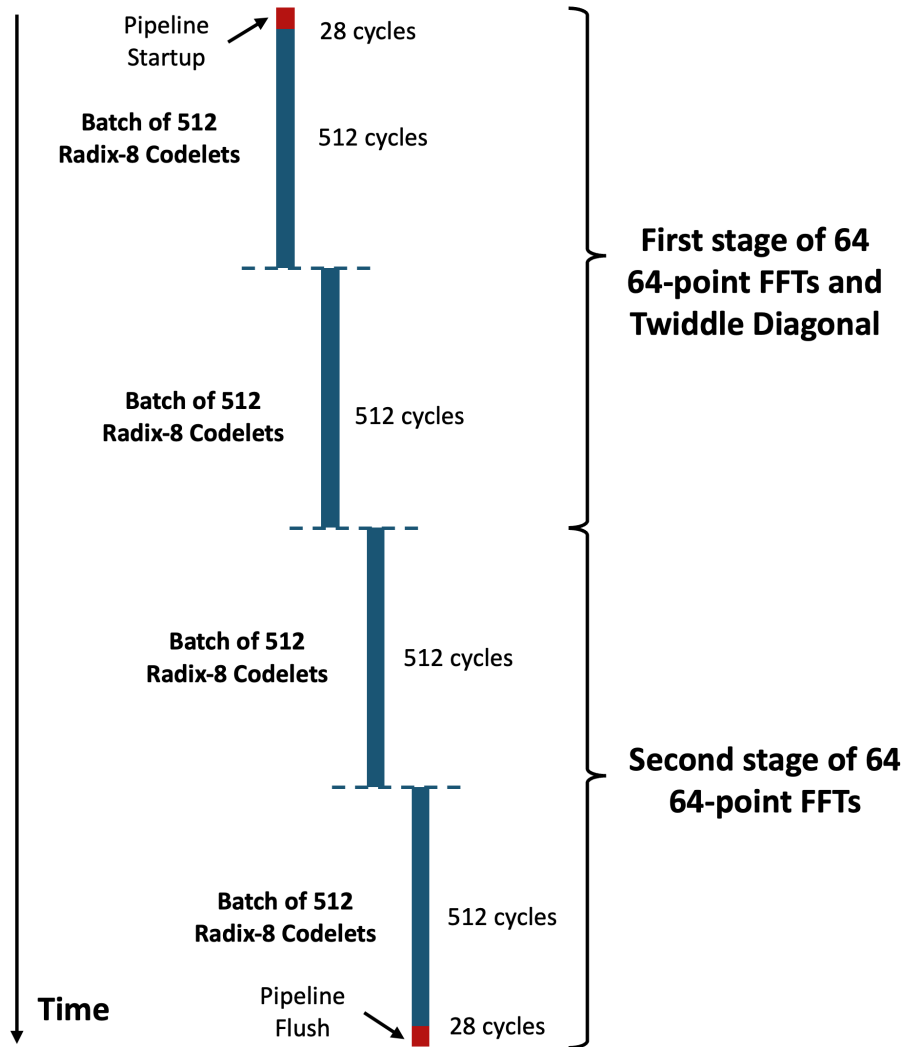


Figure 5.7: Breakdown of 4096 complex FFT executing on the accelerator.

codelets. The twiddle diagonal stage is fused into the final stage of the first set vector FFTs and after the final stage of vector FFTs, data is moved from local memory back to main memory.

From Fig. 5.7, the startup/flush cost of the radix-8 codelet pipeline is 28

cycles. However the entire program can be executed at full pipeline saturation since the codelet configuration does not change across batches, and a large number of independent codelets are batched together to ensure that data dependencies are resolved before the next required invocation. The startup and flush cost is therefore low, only 2.7% of the accelerator execution time for the 4096 complex FFT. For larger FFT sizes, we would expect that this cost is even further amortized. Thus the area overhead evaluation in Sec. 3.2.3 and the analysis here show that the cost of introducing flexibility through hardware codelets is small and our designs are comparable to the equivalent dedicated size FFT hardware.

Chapter 6

Concluding Remarks

As the current and next generations of systems continue to adopt new domain-specific hardware and chiplet-based solutions for energy efficiency, it is increasingly important for hardware designers to consider how to specialize for application domains. For HPC or HPEC systems, specialized FFT hardware has a strong upside as these systems can spend a significant fraction of time in FFT routines. The existing work in designing and optimizing custom FFT hardware for high throughput, streaming applications is relatively well understood. However this design approach of building standalone IP blocks breaks down in software-driven FFT-based applications where flexibility and programmability are the key for application developers.

The approach of this thesis draws from the FFTW insight that flexibility arises from expressing the FFT computation as a recursive composition of codelets. It first introduces a new approach for designing FFT-based hardware where only the codelets of the FFTW library are accelerated, leaving their composition to software. Rather than designing state machines and butterflies for a fixed set of

sizes or parameters, the focus of the accelerator design is on how it can effectively compose reconfigurable hardware codelets based on software specified parameters. This work then develops design tools to automatically realize the accelerators using the SPIRAL code generation system and an architectural template written in Chisel. FFT-based software programs can then be written or generated against a low-level library to target the accelerator, and hidden behind domain-specific library calls. Throughout the dissertation, we showcased silicon prototypes that applied and extended the approach for applications in fault tolerant hardware, signal intelligence, and scientific computing. Specifically, the work in this thesis shows the following:

- Develops a versatile approach for building FFT hardware where the abstraction of a hardware codelet gives an enormous amount of flexibility for composing FFT algorithms. The focus of the accelerator design shifts away from butterfly scheduling and state machine logic, and instead on how to design the architecture such that it interfaces nicely with software.
- Building off of the style of FFTW codelet computations allows us to exploit properties of the DFT. FFT-based computations such as the convolution can be realized at low hardware overhead or by reusing the same hardware components. A common theme is that element-wise products can be fused into stages of the DFT where there is no twiddle factor multiplication, expanding the scope beyond single transforms.
- Automated design tools are critical for domain-specific hardware prototyping and reducing design cycle times. Manual design of prime or higher radix butterflies is increasingly more complex and challenging, especially as the

signal flow loses the regularity of small radix-2 or radix-4 butterflies. Verification and testing strategies are enhanced through multiple levels of design abstraction and portable test vector generation.

- Demonstrates the efficacy of the approach for a few FFT workloads and applications and shows that the approach is extensible. We show how the generator incorporates new arithmetic techniques, and also identify common data and computational patterns which are integrated into the design framework. Results from performance modeling and silicon test chips show gains in energy efficiency and performance compared to software implementations targeting CPUs and GPUs.
- Fabricated silicon test chips and post-silicon measurements validate our design methodology and provide another level of characterization for performance, area, and power consumption. Measured silicon results can be used to calibrate architectural performance models.

6.1 Lessons Learned

This work focused primarily on developing new FFT hardware for accelerating applications that rely on high performance FFT software libraries. There are some interesting points of discussion and lessons learned from this approach.

- Some of the ideas and techniques presented in this thesis could also be extended to problems in other domains. For example, stencil-based computations in scientific computing also heavily rely on software libraries, and the fixed stencil pattern could be implemented with a structure similar to the specialized buffer introduced for unit strided FFT transforms.

- The idea of accelerating part of the FFTW library introduces a new hardware component for high performance FFT software, where the purely software-based implementation serves as the baseline. If the accelerator supports the codelets required by the plan, they are executed in custom hardware. Although this was the case for most of the examples in this thesis, unsupported codelet sizes can still be performed in software and any arbitrary size FFT can be computed using Bluestein’s FFT algorithm [119] (which can go up to the next power-of-2 FFT size).
- The selection of codelet sizes to accelerate in hardware should draw on insight from the application developer. Powers-of-2 FFT sizes can be covered by radix-2, 4, and 8 codelets. For scientific computing, including prime sizes such as 3, 5, and 7 already covers a wide range of sizes used by PDE solvers.
- Scaling the size of the codelet results in asymptotic operation count and the hardware codelet grows quickly in size. A radix-8 FFT contains 52 adders and 4 constant multipliers, and the Diag contains 16 adders and 32 multipliers. At radix-16, the FFT has 148 adders and 28 constant multipliers and the Diag operations are doubled, approaching the single-precision operation count supported by a compute core on modern high performance GPUs. The larger kernels also require scaling the number of memory banks to support the data bandwidth, suggesting an upper limit to the kernel size. For other reconfigurable accelerators such as CGRAs, one approach could be to have small hardware codelets for each processing element in the array rather than mapping the entire signal flow to the array.
- In software-driven HPC applications, the goal is to return from the call

to the library or PDE solver at the minimal latency. To achieve this, it is important to maintain pipeline saturation during steady state operation with small overheads for start-up and flush. A fast decoder was crucial for having the next codelet configuration ready for execution as soon as the previous one was completely issued. Descriptors were relatively simple to implement at low overhead with only a few parameters, avoiding the need for a full compiler since a basic assembler was enough. Finally, the data permutation performed by the transposers ensured that batches could continuously stream through the kernel to maintain high utilization. We would therefore expect that our approach is as efficient as the dedicated problem size FFT hardware given the same ALUs and memory interface.

6.2 Future Work

In this section we discuss some possible directions for future work.

- The approach of accelerating part of a standard software library to design domain-specific hardware can be explored for other domains.
- Hardware support for other FFT algorithms such as the prime factor algorithm, which requires a permutation of the input/output elements that is not supported by a transpose.
- Expand the real FFT support to odd size real transforms. The current approach relies on the first stage radix-2 decomposition which cannot be applied to odd problem sizes.
- Evaluate the accelerator in the broader context of end-to-end applications

and as a potential domain-specific chiplet. For example, scientific computing applications such as WarpX [13] or P3M [15], where FFT-based spectral methods are part of a larger simulation or solver. Since FFTs are generally on the boundary between memory and compute-bound, it would be also be interesting to investigate the tradeoffs between silicon area, energy efficiency, performance, and FFT workload usage.

- Integrating the idea of an interval arithmetic-based FFT accelerator into a full system-on-chip that is hardened by design. There is potential to expand the technique for error correction and to other domain-specific accelerators.

Bibliography

- [1] B. Boashash, N. A. Khan, and T. Ben-Jabeur, “Time–frequency features for pattern recognition using high-resolution tfds: A tutorial review,” *Digital Signal Processing*, vol. 40, pp. 1–30, 2015. 1.1
- [2] L. Yuan, J. Sun, L. Quan, and H.-Y. Shum, “Image deblurring with blurred/noisy image pairs,” in *ACM SIGGRAPH 2007 Papers*, ser. SIGGRAPH ’07. New York, NY, USA: Association for Computing Machinery, 2007, p. 1–es. [Online]. Available: <https://doi.org/10.1145/1275808.1276379> 1.1
- [3] G. Sharma, K. Umapathy, and S. Krishnan, “Trends in audio signal feature extraction methods,” *Applied Acoustics*, vol. 158, p. 107020, 2020. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0003682X19308795> 1.1
- [4] M. S. Hansen and P. Kellman, “Image reconstruction: An overview for clinicians,” *Journal of Magnetic Resonance Imaging*, vol. 41, no. 3, pp. 573–585, 2015. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1002/jmri.24687> 1.1

- [5] S. Matej, J. Fessler, and I. Kazantsev, “Iterative tomographic image reconstruction using fourier-based forward and back-projectors,” *IEEE Transactions on Medical Imaging*, vol. 23, no. 4, pp. 401–412, 2004. [1.1](#)
- [6] H. Moulinec and P. Suquet, “A fft-based numerical method for computing the mechanical properties of composites from images of their microstructures,” in *IUTAM Symposium on Microstructure-Property Interactions in Composite Materials*, R. Pyrz, Ed. Dordrecht: Springer Netherlands, 1995, pp. 235–246. [1.1](#)
- [7] J.-L. Vay, I. Haber, and B. B. Godfrey, “A domain decomposition method for pseudo-spectral electromagnetic simulations of plasmas,” *Journal of Computational Physics*, vol. 243, pp. 260–268, 2013. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0021999113001873> [1.1](#)
- [8] A. V. Oppenheim, R. W. Schaffer, and J. R. Buck, *Discrete-Time Signal Processing (2nd Ed.)*. USA: Prentice-Hall, Inc., 1999. [1.1](#), [2.1](#)
- [9] A. Canning and D. Raczkowski, “Scaling first-principles plane-wave codes to thousands of processors,” *Computer Physics Communications*, vol. 169, no. 1, pp. 449–453, 2005, proceedings of the Europhysics Conference on Computational Physics 2004. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0010465505001967> [1.1](#)
- [10] D. F. Martin and P. Colella, “A cell-centered adaptive projection method for the incompressible euler equations,” *J. Comput. Phys.*, vol. 163, no. 2, p. 271–312, Sep. 2000. [Online]. Available: <https://doi.org/10.1006/jcph.2000.6575> [1.1](#)

- [11] X. Ding, X. Zhang, J. Han, and G. Ding, “Scaling up your kernels to 31x31: Revisiting large kernel design in cnns,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2022, pp. 11 963–11 975. 1.1
- [12] Y. Li, T. Cai, Y. Zhang, D. Chen, and D. Dey, “What makes convolutional models great on long sequence modeling?” *arXiv preprint arXiv:2210.09298*, 2022. 1.1
- [13] J.-L. Vay, A. Almgren, J. Bell, L. Ge, D. Grote, M. Hogan, O. Kononenko, R. Lehe, A. Myers, C. Ng, J. Park, R. Ryne, O. Shapoval, M. Thévenet, and W. Zhang, “Warp-x: A new exascale computing platform for beam-plasma simulations,” *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 909, pp. 476–479, 2018, 3rd European Advanced Accelerator Concepts workshop (EAAC2017). [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0168900218300524> 1.1, 6.2
- [14] M. Valiev, E. Bylaska, N. Govind, K. Kowalski, T. Straatsma, H. Van Dam, D. Wang, J. Nieplocha, E. Apra, T. Windus, and W. de Jong, “Nwchem: A comprehensive and scalable open-source solution for large scale molecular simulations,” *Computer Physics Communications*, vol. 181, no. 9, pp. 1477–1489, 2010. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0010465510001438> 1.1
- [15] J. Yuan, H. S. Antila, and E. Luijten, “Particle-particle particle-mesh algorithm for electrolytes between charged dielectric interfaces,” *The Journal of*

- Chemical Physics*, vol. 154, no. 9, p. 094115, 03 2021. [1.1](#), [6.2](#)
- [16] M. Cerna and A. F. Harvey, “The fundamentals of fft-based signal analysis and measurement,” *National Instruments, Junho*, vol. 54, 2000. [1.1](#)
- [17] S. Liu and D. Liu, “A high-flexible low-latency memory-based fft processor for 4g, wlan, and future 5g,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 3, pp. 511–523, 2019. [1.1](#)
- [18] X.-Y. Shih, H.-R. Chou, and Y.-Q. Liu, “Vlsi design and implementation of reconfigurable 46-mode combined-radix-based fft hardware architecture for 3gpp-lte applications,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 1, pp. 118–129, 2018. [1.1](#)
- [19] Y. Guo, Z. Wang, Q. Hong, H. Luo, X. Qiu, and L. Liang, “A 60-mode high-throughput parallel-processing fft processor for 5g/4g applications,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 31, no. 2, pp. 219–232, 2023. [1.1](#)
- [20] M. Mathieu, M. Henaff, and Y. LeCun, “Fast Training of Convolutional Networks through FFTs,” 2013. [Online]. Available: <https://arxiv.org/abs/1312.5851> [1.1](#)
- [21] T. Abtahi, C. Shea, A. Kulkarni, and T. Mohsenin, “Accelerating convolutional neural network with fft on embedded hardware,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 9, pp. 1737–1749, 2018. [1.1](#)
- [22] N. Kovachki, Z. Li, B. Liu, K. Azizzadenesheli, K. Bhattacharya, A. Stuart, and A. Anandkumar, “Neural operator: Learning maps

- between function spaces with applications to pdes,” *Journal of Machine Learning Research*, vol. 24, no. 89, pp. 1–97, 2023. [Online]. Available: <http://jmlr.org/papers/v24/21-1524.html> 1.1
- [23] Z. Li, N. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. Stuart, and A. Anandkumar, “Fourier neural operator for parametric partial differential equations,” 2021. [Online]. Available: <https://arxiv.org/abs/2010.08895> 1.1
- [24] S. S. Bhattacharyya, E. F. Deprettere, R. Leupers, and J. Takala, *Handbook of signal processing systems*. Springer, 2013. 1.1, 1.2.2
- [25] M. Frigo and S. G. Johnson, “The design and implementation of FFTW3,” *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005, special issue on “Program Generation, Optimization, and Platform Adaptation”. 1.2.1, 1.3, 2.2, 2.4, 3.3.3, 4.5
- [26] Intel, “oneAPI Math Kernel Library.” [Online]. Available: <https://software.intel.com/mkl> 1.2.1, 4.5
- [27] Nvidia, “cuFFT.” [Online]. Available: <https://developer.nvidia.com/cufft> 1.2.1
- [28] AMD, “rocofft.” [Online]. Available: <https://rocm.docs.amd.com/projects/rocFFT/en/latest/> 1.2.1, 4.5
- [29] IBM, “Engineering and Scientific Subroutine Library.” [Online]. Available: <https://www.ibm.com/docs/en/essl/7.1> 1.2.1
- [30] Apple, “vDSP.” [Online]. Available: <https://developer.apple.com/documentation/accelerate/fast-fourier-transforms> 1.2.1

- [31] F. Franchetti, D. G. Spampinato, A. Kulkarni, D. Thom Popovici, T. M. Low, M. Franusich, A. Canning, P. McCorquodale, B. V. Straalen, and P. Colella, “FFTX and SpectralPack: A First Look,” in *2018 IEEE 25th International Conference on High Performance Computing Workshops (HiPCW)*, 2018, pp. 18–27. [1.2.1](#)
- [32] F. Franchetti, T. M. Low, D. T. Popovici, R. M. Veras, D. G. Spampinato, J. R. Johnson, M. Püschel, J. C. Hoe, and J. M. F. Moura, “Spiral: Extreme performance portability,” *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1935–1968, 2018. [1.2.1](#), [1.2.3](#), [2.1.5](#), [3.2](#), [3.2.1](#), [3.4](#), [4.3.1](#)
- [33] S. Rao, A. Kutuluru, P. Brouwer, S. McMillan, and F. Franchetti, “Gbtlx: A first look,” in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, 2020, pp. 1–7. [1.2.1](#)
- [34] N. Zhang and F. Franchetti, “Code generation for cryptographic kernels using multi-word modular arithmetic on gpu,” ser. CGO ’25. New York, NY, USA: Association for Computing Machinery, 2025, p. 476–492. [Online]. Available: <https://doi.org/10.1145/3696443.3708948> [1.2.1](#)
- [35] N. Zhang, A. Ebel, N. Neda, P. Brinich, B. Reynwar, A. G. Schmidt, M. Franusich, J. Johnson, B. Reagen, and F. Franchetti, “Generating high-performance number theoretic transform implementations for vector architectures,” in *2023 IEEE High Performance Extreme Computing Conference (HPEC)*, 2023, pp. 1–7. [1.2.1](#)
- [36] H. Mankad, S. Rao, P. Colella, B. Van Straalen, and F. Franchetti, “Protox: A first look,” in *2023 IEEE High Performance Extreme Computing Conference (HPEC)*, 2023, pp. 1–6. [1.2.1](#)

- [37] P. Milder, F. Franchetti, J. C. Hoe, and M. Püschel, “Computer generation of hardware for linear digital signal processing transforms,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 17, no. 2, Apr. 2012. [Online]. Available: <https://doi.org/10.1145/2159542.2159547> 1.2.1, 1.2.2, 1.2.3
- [38] M. Zuluaga, P. A. Milder, and M. Püschel, “Computer generation of streaming sorting networks,” in *Design Automation Conference (DAC)*, 2012, pp. 1245–1253. 1.2.1
- [39] G. Xu, J. C. Hoe, and F. Franchetti, “Flexible hardware accelerator design generation with spiral,” in *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, 2022, pp. 1–7. 1.2.1, 1.2.3
- [40] M. Garrido, “A survey on pipelined fft hardware architectures,” *Journal of Signal Processing Systems*, vol. 94, no. 11, pp. 1345–1364, 2022. 1.2.2
- [41] F. Qureshi, M. Garrido, and O. Gustafsson, “Unified architecture for 2, 3, 4, 5, and 7-point dfts based on winograd fourier transform algorithm,” *Electronics Letters*, vol. 49, no. 5, pp. 348–349, 2013. [Online]. Available: <https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/el.2012.0577> 1.2.2, 2.4.1
- [42] F. Qureshi, M. Ali, and J. Takala, “Multiplierless reconfigurable processing element for mixed radix-2/3/4/5 ffts,” in *2017 IEEE International Workshop on Signal Processing Systems (SiPS)*, 2017, pp. 1–6. 1.2.2, 2.4.1
- [43] A. Wang, J. Bachrach, and B. Nikolić, “A generator of memory-based, runtime-reconfigurable $2^n 3^m 5^k$ fft engines,” in *2016 IEEE International*

- Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2016, pp. 1016–1020. [1.2.2](#), [2.4.1](#)
- [44] S. H. Mirfarshbafan, S. Taner, and C. Studer, “Smul-fft: A streaming multiplierless fast fourier transform,” *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 68, no. 5, pp. 1715–1719, 2021. [1.2.2](#)
- [45] B.-C. Lin, Y.-H. Wang, J.-D. Huang, and J.-Y. Jou, “Expandable mdc-based fft architecture and its generator for high-performance applications,” in *23rd IEEE International SOC Conference*, 2010, pp. 188–192. [1.2.2](#)
- [46] V. M. Milovanović and M. L. Petrović, “A highly parametrizable chisel hcl generator of single-path delay feedback fft processors,” in *2019 IEEE 31st International Conference on Microelectronics (MIEL)*, 2019, pp. 247–250. [1.2.2](#)
- [47] F. Serre and M. Püschel, “A dsl-based fft hardware generator in scala,” in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2018, pp. 315–3157. [1.2.2](#)
- [48] G. Xu, J. C. Hoe, and F. Franchetti, “Flexible hardware accelerator design generation with spiral,” in *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, 2022, pp. 1–7. [1.2.2](#)
- [49] C.-L. Yu, K. Irick, C. Chakrabarti, and V. Narayanan, “Multidimensional dft ip generator for fpga platforms,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 58, no. 4, pp. 755–764, 2011. [1.2.2](#)

- [50] S. Winograd, “On computing the discrete fourier transform,” *Mathematics of Computation*, vol. 32, no. 141, pp. 175–199, 1978. [Online]. Available: <http://www.jstor.org/stable/2006266> 1.2.2
- [51] Xilinx, “Fast fourier transform v9.1,” 2022. [Online]. Available: https://www.xilinx.com/support/documents/ip_documentation/xfft/v9_1/pg109-xfft.pdf 1.2.2
- [52] Intel, “Unified fft intel fpga ips user guide,” 2024. [Online]. Available: <https://www.intel.com/content/www/us/en/docs/programmable/683366/21-1-1-0-0/about-the-unified-fft-intel-fpga-ips.html> 1.2.2
- [53] Siemens EDA, “Catapult high-level synthesis and verification.” [Online]. Available: <https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis> 1.2.2, 1.2.3
- [54] A. Krishnakumar, U. Ogras, R. Marculescu, M. Kishinevsky, and T. Mudge, “Domain-specific architectures: Research problems and promising approaches,” *ACM Trans. Embed. Comput. Syst.*, vol. 22, no. 2, Jan. 2023. [Online]. Available: <https://doi.org/10.1145/3563946> 1.2.3
- [55] AMD, “Vitis high-level synthesis user guide,” 2025. [Online]. Available: <https://docs.amd.com/r/en-US/ug1399-vitis-hls/Introduction> 1.2.3
- [56] Intel, “Intel® oneapi dpc++/c++ compiler handbook for fpgas,” 2025. [Online]. Available: <https://www.intel.com/content/www/us/en/docs/oneapi-fpga-add-on/developer-guide/2025-0/fpga-handbook.html> 1.2.3
- [57] R. Venkatesan, Y. S. Shao, M. Wang, J. Clemons, S. Dai, M. Fojtik, B. Keller, A. Klinefelter, N. Pinckney, P. Raina, Y. Zhang, B. Zimmer, W. J.

- Dally, J. Emer, S. W. Keckler, and B. Khailany, “Magnet: A modular accelerator generator for neural networks,” in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2019, pp. 1–8. [1.2.3](#)
- [58] J.-F. Schulte, B. Ramhorst, C. Sun, J. Mitrevski, N. Ghielmetti, E. Lupi, D. Danopoulos, V. Loncar, J. Duarte, D. Burnette, L. Laatu, S. Tzelepis, K. Axiotis, Q. Berthet, H. Wang, P. White, S. Demirsoy, M. Colombo, T. Aarrestad, S. Summers, M. Pierini, G. D. Guglielmo, J. Ngadiuba, J. Campos, B. Hawks, A. Gandrakota, F. Fahim, N. Tran, G. Constantinides, Z. Que, W. Luk, A. Tapper, D. Hoang, N. Paladino, P. Harris, B.-C. Lai, M. Valentin, R. Forelli, S. Ogrenici, L. Gerlach, R. Flynn, M. Liu, D. Diaz, E. Khoda, M. Quinnan, R. Solares, S. Parajuli, M. Neubauer, C. Herwig, H. F. Tsoi, D. Rankin, S.-C. Hsu, and S. Hauck, “hls4ml: A flexible, open-source platform for deep learning acceleration on reconfigurable hardware,” 2025. [Online]. Available: <https://arxiv.org/abs/2512.01463> [1.2.3](#)
- [59] H. Ye, H. Jun, H. Jeong, S. Neuendorffer, and D. Chen, “Scalehls: a scalable high-level synthesis framework with multi-level transformations and optimizations: invited,” in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, ser. DAC ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1355–1358. [Online]. Available: <https://doi.org/10.1145/3489517.3530631> [1.2.3](#)
- [60] Y. Zhang, X. Zhang, P. Xu, Y. Zhao, C. Hao, D. Chen, and Y. Lin, “Autoai2c: An automated hardware generator for dnn acceleration on both fpga and asic,” *IEEE Transactions on Computer-Aided*

- Design of Integrated Circuits and Systems*, 2024. [Online]. Available: <https://ieeexplore.ieee.org/document/10508183> 1.2.3
- [61] C. Wang, J. Zhang, Y. Wu, and J. Cong, “Hp-fft: A general high-performance fft generator using high-level synthesis,” in *2025 IEEE 33rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2025, pp. 19–23. 1.2.3
- [62] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, “Chisel: constructing hardware in a scala embedded language,” ser. DAC ’12. New York, NY, USA: Association for Computing Machinery, 2012, p. 1216–1225. [Online]. Available: <https://doi.org/10.1145/2228360.2228584> 1.2.3, 2.4, 3.2.1
- [63] C. Papon and Y. Xiao, “SpinalHDL.” [Online]. Available: <https://github.com/SpinalHDL/SpinalHDL> 1.2.3
- [64] P. Hanrahan, “magma.” [Online]. Available: <https://github.com/phanrahan/magma> 1.2.3
- [65] K. Koul, J. Melchert, K. Sreedhar, L. Truong, G. Nyengele, K. Zhang, Q. Liu, J. Setter, P.-H. Chen, Y. Mei, M. Strange, R. Daly, C. Donovanick, A. Carsello, T. Kong, K. Feng, D. Huff, A. Nayak, R. Setaluri, J. Thomas, N. Bhagdikar, D. Durst, Z. Myers, N. Tsiskaridze, S. Richardson, R. Bahr, K. Fatahalian, P. Hanrahan, C. Barrett, M. Horowitz, C. Torng, F. Kjolstad, and P. Raina, “Aha: An agile approach to the design of coarse-grained reconfigurable accelerators and compilers,” vol. 22, no. 2, Jan. 2023. [Online]. Available: <https://doi.org/10.1145/3534933> 1.2.3

- [66] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton, P. Rigge, C. Schmidt, J. Wright, J. Zhao, Y. S. Shao, K. Asanović, and B. Nikolić, “Chipyard: Integrated design, simulation, and implementation framework for custom socs,” *IEEE Micro*, vol. 40, no. 4, pp. 10–21, 2020. [1.2.3](#)
- [67] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, P. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, “The rocket chip generator, technical report ucb/eecs-2016-17,” EECS Department, University of California, Berkeley, Tech. Rep., April 2016. [1.2.3](#)
- [68] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic, “Sonicboom: The 3rd generation berkeley out-of-order machine,” May 2020. [1.2.3](#)
- [69] “VexRiscv.” [Online]. Available: <https://github.com/spinalhdl/vexriscv> [1.2.3](#), [4.4.5](#)
- [70] H. Genc, S. Kim, A. Amid, A. Haj-Ali, V. Iyer, P. Prakash, J. Zhao, D. Grubb, H. Liew, H. Mao, A. Ou, C. Schmidt, S. Steffl, J. Wright, I. Stolica, J. Ragan-Kelley, K. Asanovic, B. Nikolic, and Y. S. Shao, “Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration,” in *Proceedings of the 58th Annual Design Automation Conference (DAC)*, 2021. [1.2.3](#)
- [71] P. Mohan, O. Atli, O. O. Kibar, and K. Mai, “A top-down design methodology for synthesizing fpga fabrics using standard asic flow,”

- in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 313. [Online]. Available: <https://doi.org/10.1145/3373087.3375383> 1.2.3
- [72] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, “Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines,” in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 519–530. [Online]. Available: <https://doi.org/10.1145/2491956.2462176> 1.2.3
- [73] A. Sabne, “Xla : Compiling machine learning for peak performance,” 2020. 1.2.3
- [74] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, “Tvm: an automated end-to-end optimizing compiler for deep learning,” ser. OSDI'18. USA: USENIX Association, 2018, p. 579–594. 1.2.3
- [75] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe, “Graphit: a high-performance graph dsl,” *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, Oct. 2018. [Online]. Available: <https://doi.org/10.1145/3276491> 1.2.3
- [76] O. Hsu, M. Strange, R. Sharma, J. Won, K. Olukotun, J. S. Emer, M. A. Horowitz, and F. Kjolstad, “The sparse abstract machine,” in *Proceedings*

- of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 710–726. [Online]. Available: <https://doi.org/10.1145/3582016.3582051> 1.2.3
- [77] J. Woodruff, J. Armengol-Estapé, S. Ainsworth, and M. F. P. O’Boyle, “Bind the gap: compiling real software to hardware fft accelerators,” ser. PLDI 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 687–702. [Online]. Available: <https://doi.org/10.1145/3519939.3523439> 1.2.3
- [78] J. W. Cooley and J. W. Tukey, “An Algorithm for the Machine Calculation of Complex Fourier Series,” *Mathematics of Computation*, vol. 19, no. 90, pp. 297–301, 1965. 2.1
- [79] Z. Gong, N. Zhu, M. Ngaw, J. Rivera, L. Tang, E. Tang, H. Mankad, and F. Franchetti, “Interval arithmetic-based fft for large integer multiplication,” in *Proceedings of the 26th IEEE High Performance Extreme Computing Conference (HPEC)*, 2022. 2.1.2
- [80] F. Franchetti, M. Puschel, Y. Voronenko, S. Chellappa, and J. M. Moura, “Discrete fourier transform on multicore,” *IEEE Signal Processing Magazine*, vol. 26, no. 6, pp. 90–102, 2009. 2.1.7
- [81] F. Franchetti and M. Puschel, “Short vector code generation for the discrete fourier transform,” in *Proceedings International Parallel and Distributed Processing Symposium*, 2003, pp. 10 pp.–. 2.1.7

- [82] L. Tang, S. Chen, K. Harisrikanth, G. Xu, K. Mai, and F. Franchetti, “A high throughput hardware accelerator for fftw codelets: A first look,” in *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, 2022, pp. 1–7. [2.2](#), [4.3.1](#)
- [83] S. Rao, L. Tang, and F. Franchetti, “Libraryx-asic: A first look,” in *2025 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2025, pp. 205–208. [3.3.3](#)
- [84] P. Mantovani, D. Giri, G. Di Guglielmo, L. Piccolboni, J. Zuckerman, E. G. Cota, M. Petracca, C. Pilato, and L. P. Carloni, “Agile soc development with open esp,” in *Proceedings of the 39th International Conference on Computer-Aided Design*, ser. ICCAD ’20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: <https://doi.org/10.1145/3400302.3415753> [3.5.1](#)
- [85] L. Tang, V. Kumar, M. Ngaw, S. Singh, D. Nadkarni, L. Tummala, K. Mai, and F. Franchetti, “Towards an algorithm-based approach for soft error tolerance using interval arithmetic,” in *2025 IEEE High Performance Extreme Computing Conference (HPEC)*, 2025, pp. 1–7. [4](#)
- [86] D. M. Fleetwood, “Radiation effects in a post-moore world,” *IEEE Transactions on Nuclear Science*, vol. 68, no. 5, pp. 509–545, 2021. [4.1](#)
- [87] D. Kobayashi, “Scaling trends of digital single-event effects: A survey of seu and set parameters and comparison with transistor performance,” *IEEE Transactions on Nuclear Science*, vol. 68, no. 2, pp. 124–148, 2021. [4.1](#)

- [88] H. D. Dixit, S. Pendharkar, M. Beadon, C. Mason, T. Chakravarthy, B. Muthiah, and S. Sankar, “Silent data corruptions at scale,” 2021. [Online]. Available: <https://arxiv.org/abs/2102.11245> 4.1
- [89] R. Bonderson, “Training in turmoil: Silent data corruption in systems at scale,” in *International Test Conference Silicon Lifecycle Management Workshop*, 2021. 4.1
- [90] P. H. Hochschild, P. Turner, J. C. Mogul, R. Govindaraju, P. Ranganathan, D. E. Culler, and A. Vahdat, “Cores that don’t count,” in *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2021, pp. 9–16. 4.1
- [91] O. Atli, “Soft error analysis and design space exploration of radiation-hardened system-on-a-chip platforms,” Ph.D. dissertation, Carnegie Mellon University, 2024. 4.1
- [92] J.-M. Daveau, A. Blampey, G. Gasiot, J. Bulone, and P. Roche, “An industrial fault injection platform for soft-error dependability analysis and hardening of complex system-on-a-chip,” in *2009 IEEE International Reliability Physics Symposium*, 2009, pp. 212–220. 4.1
- [93] K.-H. Huang and J. A. Abraham, “Algorithm-based fault tolerance for matrix operations,” *IEEE Transactions on Computers*, vol. C-33, no. 6, pp. 518–528, 1984. 4.1, 4.1.1
- [94] A. Reddy and P. Banerjee, “Algorithm-based fault detection for signal processing applications,” *IEEE Transactions on Computers*, vol. 39, no. 10, pp. 1304–1308, 1990. 4.1.1

- [95] B. Shim and N. R. Shanbhag, “Energy-efficient soft error-tolerant digital signal processing,” *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 14, no. 4, p. 336–348, Apr. 2006. [Online]. Available: <https://doi.org/10.1109/TVLSI.2006.874359> 4.1.1
- [96] J.-Y. Jou and J. Abraham, “Fault-tolerant fft networks,” *IEEE Transactions on Computers*, vol. 37, no. 5, pp. 548–561, 1988. 4.1.1
- [97] S.-J. Wang and N. Jha, “Algorithm-based fault tolerance for fft networks,” *IEEE Transactions on Computers*, vol. 43, no. 7, pp. 849–854, 1994. 4.1.1
- [98] K. Zhao, S. Di, S. Li, X. Liang, Y. Zhai, J. Chen, K. Ouyang, F. Cappello, and Z. Chen, “Ft-cnn: Algorithm-based fault tolerance for convolutional neural networks,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 7, pp. 1677–1689, 2021. 4.1.1
- [99] J. Kosaian and K. V. Rashmi, “Arithmetic-intensity-guided fault tolerance for neural network inference on gpus,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC ’21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: <https://doi.org/10.1145/3458817.3476184> 4.1.1
- [100] G. Bosilca, R. Delmas, J. Dongarra, and J. Langou, “Algorithm-based fault tolerance applied to high performance computing,” *Journal of Parallel and Distributed Computing*, vol. 69, no. 4, pp. 410–416, 2009. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0743731508002141> 4.1.1

- [101] Z. Chen and J. Dongarra, “Algorithm-based fault tolerance for fail-stop failures,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, no. 12, pp. 1628–1641, 2008. [4.1.1](#)
- [102] L. Narmour, S. Derrien, and S. Rajopadhye, “Automatic algorithm-based fault tolerance (aabft) of stencil computations,” in *2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2023, pp. 187–198. [4.1.1](#)
- [103] Z. Chen, “Online-abft: an online algorithm based fault tolerance scheme for soft error detection in iterative methods,” in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP ’13. New York, NY, USA: Association for Computing Machinery, 2013, p. 167–176. [Online]. Available: <https://doi.org/10.1145/2442516.2442533> [4.1.1](#)
- [104] R. E. Moore, *Methods and applications of interval analysis*. SIAM, 1979. [4.1.2](#)
- [105] D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, and F. Védryne, “Towards an industrial use of fluctuat on safety-critical avionics software,” in *Formal Methods for Industrial Critical Systems*, M. Alpuente, B. Cook, and C. Joubert, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 53–69. [4.1.2](#)
- [106] F. Franchetti, T. M. Low, S. Mitsch, J. P. Mendoza, L. Gui, A. Phaosawasdi, D. Padua, S. Kar, J. M. Moura, M. Franusich, J. Johnson, A. Platzer, and M. M. Veloso, “High-assurance spiral: End-to-end guarantees for robot and

- car control,” *IEEE Control Systems Magazine*, vol. 37, no. 2, pp. 82–103, 2017. 4.1.2
- [107] P. Breiding, K. Rose, and S. Timme, “Certifying zeros of polynomial systems using interval arithmetic,” *ACM Trans. Math. Softw.*, vol. 49, no. 1, Mar. 2023. [Online]. Available: <https://doi.org/10.1145/3580277> 4.1.2
- [108] J. Rivera, F. Franchetti, and M. Püschel, “An interval compiler for sound floating-point computations,” in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2021, pp. 52–64. 4.1.2
- [109] —, “A compiler for sound floating-point computations using affine arithmetic,” in *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2022, pp. 66–78. 4.1.2
- [110] R. Roberts, W. Brown, and H. Loomis, “Computationally efficient algorithms for cyclic spectral analysis,” *IEEE Signal Processing Magazine*, vol. 8, no. 2, pp. 38–49, 1991. 4.4.2
- [111] April, Eric, “On the Implementation of the Strip Spectral Correlation Algorithm for Cyclic Spectrum Estimation,” 1994. [Online]. Available: <https://publications.gc.ca/site/eng/9.942572/publication.html> 4.4.2
- [112] J. Li, “High-speed implementations of spectral correlation analysis techniques,” Ph.D. dissertation, 2025. [Online]. Available: <https://hdl.handle.net/2123/33865> 4.4.7
- [113] C. Eleftheriadis and G. Karakonstantis, “Energy-efficient fast fourier transform for real-valued applications,” *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 69, no. 5, pp. 2458–2462, 2022. 4.5

- [114] Z. Qian and G. Gan, “Accelerating real-valued fft on cpu-fpga platforms,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 43, no. 8, pp. 2532–2536, 2024. 4.5
- [115] M. Ayinala, Y. Lao, and K. K. Parhi, “An in-place fft architecture for real-valued signals,” *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 60, no. 10, pp. 652–656, 2013. 4.5
- [116] M. Garrido, K. K. Parhi, and J. Grajal, “A pipelined fft architecture for real-valued signals,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 56, no. 12, pp. 2634–2643, 2009. 4.5
- [117] Texas Instruments, “Implementing Fast Fourier Transform Algorithms of Real-Valued Sequences With the TMS320 DSP Platform,” 2001. [Online]. Available: <https://www.ti.com/lit/an/spra291/spra291.pdf> 4.5
- [118] L. Tang, S. Chen, K. Harisrikanth, G. Xu, F. Franchetti, and K. Mai, “A 1.19ghz 9.52gsamples/sec radix-8 fft hardware accelerator in 28nm,” in *2024 IEEE Hot Chips 36 Symposium (HCS)*, 2024, pp. 1–1. 5.2
- [119] L. Bluestein, “A linear filtering approach to the computation of discrete fourier transform,” *IEEE Transactions on Audio and Electroacoustics*, vol. 18, no. 4, pp. 451–455, 1970. 6.1