# An approach to generating customized load-store architectures

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

**Guanglin Xu**

B.E., School of Software, Sun Yat-sen University Guangzhou

M.E., School of Software, Sun Yat-sen University Guangzhou

Carnegie Mellon University

Pittsburgh, PA

May 2023

# Acknowledgments

This work would not have been possible without the advice and support from my great advisors Prof. Franz Franchetti and Prof. James Hoe. I am very thankful for their time and effort along the unusually long journey of my PhD study. They have not only provided intelligent advice for research, but also generously taught me precious principles and lessons for life success. I owe all my success to them.

Special thanks to my thesis committee in providing important feedback for significant improvements for this thesis. The thesis committee is composed of Prof. Franz Franchetti (Co-Chair), Prof. James Hoe (Co-Chair), Prof. Tze Meng Low (CMU - ECE), Prof. Peter Milder (Stony Brook - ECE). In particular, Prof. Milder has offered me internal access to the Spiral DFT IP core generator and provided me tutorials so that I can understand the internal structure of state-of-the-art designs.

I would like to thank several people who have helped me a lot during my PhD study. In my early years, Prof. Tze Meng Low has taught me to question everything that's out there by steadily asking me "simple" questions like "what" and "why". Dr. Doru Thom Popovici, before becoming a doctor himself, has generously shared with me tons of hands-on experience with the Spiral system. Dr. Jing Huang has taught me how to design and debug in RTL. Special thanks to Prof. David Padua at UIUC who has hosted my half-year visit to UIUC. There are more people that I feel grateful to. They include Richard Veras, Yu Wang, Jiyuan Zhang, Zhipeng Zhao, Marie Nguyen, Joe Melber, Fazle Sadi, Qi Guo, Daniele Spampinato, Maia

Blanco, Shashank Obla, Chengyue Wang, and many more.

This work was partially sponsored by the Defense Advanced Research Projects Agency (DARPA) PERFECT program under agreement HR0011-13-2-0007 and the BRASS program under agreement FA8750-16-2-003. I'm thankful for their generous supports to fund this research.

This thesis was proposed at the beginning of the pandemic and has been delayed partially due to the pandemic. I am grateful to my wife Dr. Xiao Wang for her continuous love and support during the hard time. I am grateful to my daughter Stephanie and my son Echo for bringing me the endless joy.

Guanglin Xu

*Carnegie Mellon University*

*May 2023*

# Abstract

Automated design generation is increasingly used in hardware accelerators to effectively handle the large trade-off space between performance and resource utilization. Typically, a generator focuses on a specialized parallel architecture that fits a particular set of algorithms. As a result, it is difficult to extend the algorithmic support of generators. In contrast, a processor-like architecture, where the datapath is connected to memory ports with all computations sequenced by a controller, can provide much better flexibility. In this dissertation, I call such an architecture the "load-store architecture" and present an approach to hardware design generation from high-level specifications. The approach generates customized load-store architecture designs across multiple abstraction levels for algorithm generation, loop optimization, and hardware interpretation, respectively.

The proposed approach is inspired by the Spiral code generation framework and is realized by extending Spiral. In generating algorithms for customized load-store architectures at the data flow level, I present the importance of providing sufficient independent iterations to accommodate the long latency of customized pipelines. The generated algorithms are then translated to loop programs captured in an extensible DSL for hardware-oriented loop optimizations. I identify a computational pattern of imperfect loop nests and provide optimizations for reducing execution cycle counts and decreasing memory buffer utilization as well as arithmetic counts in address calculation. Finally, the optimized loops are interpreted to

register-transfer level designs in another hardware-extended DSL where local optimizations are employed.

I implement the approach by extending the open-source SPIRAL system. I demonstrate the flexibility of the system by generating designs for signal transforms including WHT and DFT, and the sorting operation. Experimental results show the benefit of hardware-oriented optimizations. In particular, the FFT IP cores generated with my approach are comparable to state-of-the-art designs. Despite further parallelization and hardware compilation efforts to be pursued, this dissertation has paved the way for generating competitive hardware designs with SPIRAL in a flexible manner.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Automated design generation is increasingly used in the design of hardware accelerators to handle the large trade-off space between performance and resource utilization. The Spiral framework focuses on automating the designs for digital signal transform kernels and other structured operations. Current Spiral framework is limited in algorithmic flexibility as it uses a streaming architecture that is only ideal for algorithms with uniform blocks in the data flow graph representation. In this thesis, I study embedding a customized load-store architecture, which is inspired by processor designs, into the Spiral framework. The updated Spiral framework could natively handle more complicated algorithms regardless of the uniformity.

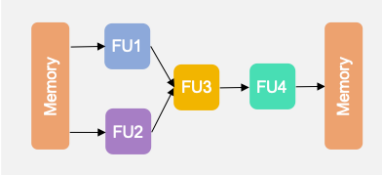
The proposed hardware generation flow starts from algorithm generation, then moves to program optimization and finally to hardware interpretation. The first stage sets up and solves a constraint program for producing algorithms matching the required architectural features. The second stage represents the generated algorithms in imperfect loop nest programs and uses pattern-based loop transformations that are enabled by the domain-specific language (DSL) in Spiral, to optimize

the programs. The final stage interprets the optimized programs into hardware designs as interconnected modules at the register-transfer level. The approach is tested for a compute pattern that process high-dimensional data cubes on a load-store architecture connected with a dual-ported memory. Imperfect loop nest programs conforming to the pattern are optimized for execution latency, RAM utilization, and arithmetic cost, supported by DSL extensions capturing relevant properties.

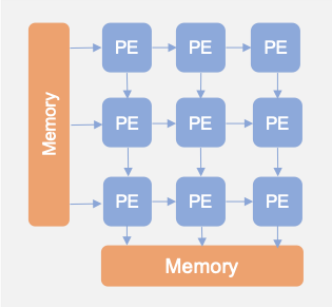
My results show that imperfect loop nest programs representing algorithms lacking ideal uniformity can now be natively implemented as hardware accelerators using the updated framework. This approach has been applied to Walsh-Hadamard transform, discrete Fourier transform and the bitonic sorter algorithm. In a case study of FFT accelerators, I show that by allowing for slight non-uniformity in the choice of algorithms, the execution latency in cycle counts is reduced and the SRAM utilization is identical compared to the carefully optimized streaming based designs.

## 1.1 Motivation

The conventional wisdom in the automatic generation of hardware accelerator designs has centered around customized parallel architectures for particular algorithms. The popular examples include streaming architectures (Figure 1.1a) and systolic arrays (Figure 1.1b). These architectures can achieve much higher computational throughput than what is possible in general processors by allowing direct communications between customized functional units or processing elements. However, these high-throughput architectures only fit an important while also limited set of algorithms. Besides, the highest throughput is not always required in application scenarios. Hence, a question raised naturally is: *can we tradeoff some throughput for algorithmic flexibility by targeting a more general architecture?*




(a) A streaming architecture composed of different functional units connected by streaming links.




(b) A systolic array architecture composed of homogenerous processing elements connected by neighboring communication links.

Figure 1.1: Algorithm-specific parallel architecture examples.

In this work, I attempt to answer this question by focusing on *customized load-store architectures* inspired by the design of general processors, as is shown in Figure 1.2. In a general processor, primitive arithmetic/logic operations are conducted in an ALU whose I/O ports are connected to a register file. The register file exchanges data with the memory system via load and store operations. A programmable controller devises the sequence of memory load/store operations and ALU operations such that arbitrary computations can be performed in a processor. In a customized load-store architecture, the memory load/store behavior is preserved for flexibility while both primitive operations in the datapath and compute sequences in the controller are highly customized to specific algorithms. The peak throughput of a load-store architecture depends on the complexity of the datapath and the number of memory ports connected to the datapath. Figure 1.3 shows two mature parallel paradigms known in processor designs that can be applied to customized load-store architectures, i.e. a SIMD style design and a multi-processing style design.

In the automatic generation of customized load-store architectures, an es-

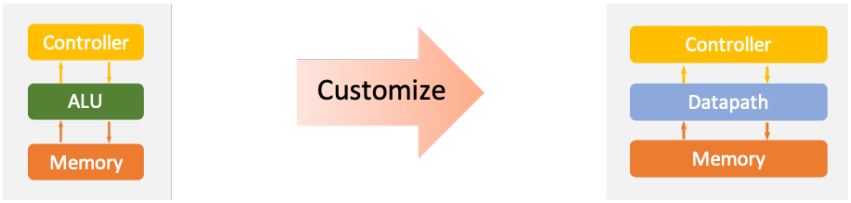
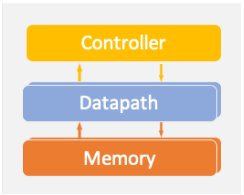
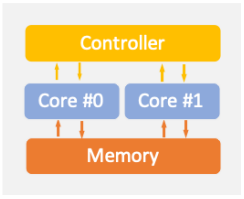



Figure 1.2: From processors to customized load-store architectures.




(a) A SIMD style load-store architecture.




(b) A multi-processing load-store architecture.

Figure 1.3: Scaling the processing throughput of load-store architectures.

sential task is to explore the large tradeoff space between performance and resource utilization. This is, however, challenging due to the mutual restriction between a wide range of algorithms and a large design space of architectures. For general processors, designing an efficient architecture and programming for the architecture are already complicated problems separately. In customized load-store architectures, the best choice in one domain depends on which choices are made in the other. Lacking a tool to reason about both sets of options at once, the cost of exploring the tradeoff space between performance and resource utilization is prohibitively high.

## 1.2 Spiral Code Generation Approach

The Spiral framework can contribute to the specialization of load-store architectures because it offers a unique method to reason about algorithms and architectures

simultaneously. In this way, the architecture is by construction optimized for the algorithm considered for hardware acceleration. The control of the customized architecture for the specific algorithm is generated in the meantime. As a result, the problems of designing customized load-store architectures and programming the architectures are solved in a unified framework.

SPIRAL was originally for automatically generating highly optimized software implementations of digital signal transforms and other structured algorithms for a landscape of commodity processor platforms. The core of SPIRAL is a constraint solver, which automatically derives algorithms fitting the specified architecture, and a multi-level rewriting system for program optimizations and code generation. The constraint solver captures algorithms, program transformations, and hardware in a unified formal system called the operator language (OL). A constraint problem is firstly setup by an expert user through defining the recursive specification of algorithms, the base cases that the hardware can efficiently process, and a set of architecture-aware rules that decompose a specification to base cases. Then, the problem is solved by applying the rule set recursively to the specification until termination, producing efficient algorithms for the architecture. Next, the derived algorithms are translated to *imperfect loop nest* programs captured by SPIRAL's $\Sigma$-OL language for loop optimizations. Finally, the optimized loop is translated to the icode representation for code generation. The code generation process using multi-level domain specific languages (DSLs) is shown by the left-most flow of Figure 1.4. In the past, SPIRAL has been successfully applied for generating high performance library code for novel architectures that are difficult to human programmers.

Though was initially for program generation for off-the-shelf processors, the constraint solver of the SPIRAL framework makes it suitable for algorithm-hardware

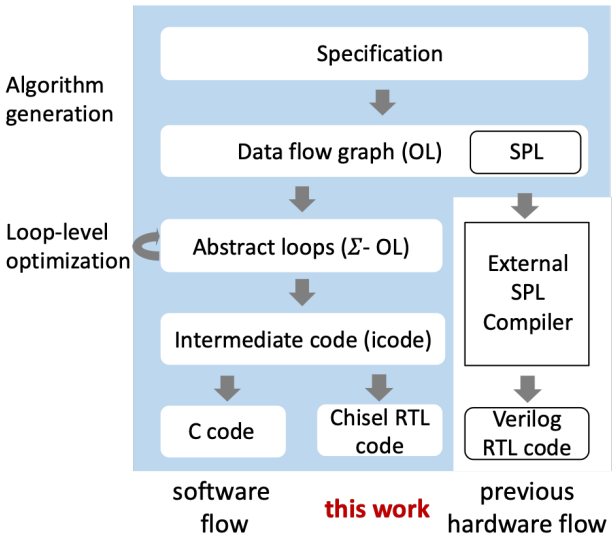
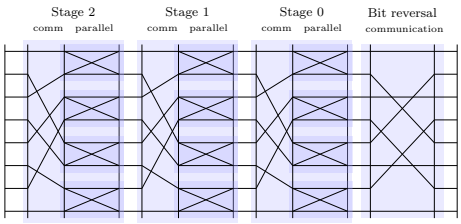


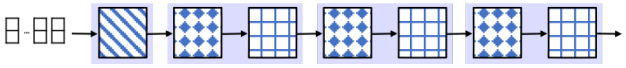

Figure 1.4: The SPIRAL code generation flows involving multi-level domain specific languages.

co-synthesis. In addition, the multi-level rewriting system can be extended for code generation at the register-transfer level (RTL). In the past, a SPIRAL approach for algorithm-hardware co-synthesis targeting transform kernels on a streaming architecture has proven to be successful. The approach produces hardware implementations comparable to hand-tuned designs and provides much more Pareto-optimal solutions in the tradeoff space between performance and resource utilization. In that work, the specification is decomposed to SPL[1] formulas describing the generated algorithm with streaming hardware parameters. Then, the SPL formula is translated by an external SPL compiler to produce Verilog RTL, as is shown by the right-most flow in Figure 1.4. Traversing the tradeoff space of the streaming architecture for an algorithm boils down to the vertical and horizontal foldings of the datapath mapped from the data flow graph representation, as illustrated by an example in Figure 1.5, thus it requires uniform blocks in the data flow graph that are executed stage by

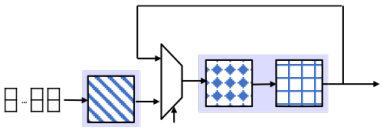
[1] a subset of OL for linear operators.




(a) Data flow graph of 8-point Pease FFT algorithm where the data flows from right to left. It is composed of an initial bit reversal permutation stage, followed by three stages with uniform geometry containing parallel computational blocks and a stride permutation.



(b) Vertically folded streaming datapath. The left most block filled with diagonal stripes represents streamed bit reversal permutation datapath. The solid diamond grid represents the computational datapath. The normal grid pattern represents the streamed stride permutation datapath.



(c) Horizontally folded streaming datapth.

Figure 1.5: Mapping a uniform data flow graph to streamed datapath. There exists a degree of freedom when folding datapath in both dimensions.

stage. As a result, this method works the best in domains where the algorithms exhibits ideal uniformity in the data flow graph representation.

Despite the earlier success, applying SPIRAL for customized load-store architecture is non-trivial due to 1) the numerous possible combinations of design variables exceeding what have been realized in commodity general-purpose processors, and 2) the extra considerations necessary in interpreting imperfect loop nest programs to efficient hardware implementations.

A load-store architecture can be specialized in controller, memory hierarchy and parallelism style, resulting in a large number of combinations which require non-trivial human intervention to setup the constraint solver of SPIRAL. In con-

troller designs, besides the instruction stream controller used in general purpose processors, a specialized load-store architecture can also have a simpler ROM-based or FSM-based controller. For memory sub-system designs, the options range from a single-level fast SRAM for IP core designs to a multi-level hierarchy involving main memory managed as scratchpad memory or by a cache protocol. The common parallelism style contains vectorization, symmetric multi-processing, and distributed memory. To handle a particular combination in SPIRAL, one has to develop rules for decomposing operations to base cases that can be directly handled in hardware. In addition, the architectural templates of irreducible operators and OL formulas need to be supplied to the framework for automatic hardware construction.

Algorithms generated from the constraint solver will be translated to the imperfect loop nest programs in $\Sigma$-OL and requires optimization for features not easily modeled in the constraint solving stage. In $\Sigma$-OL, the memory access indices are captured symbolically to enable loop analyses and transformations difficult to general compiler optimizers. In software code generation, SPIRAL has employed loop fusion to reduce data transfers between on-chip and off-chip memory and software pipelining to hide the latency. When interpreting loop programs to hardware implementations, other properties may also be considered. One such an example is the efficient mapping between the arrays used as intermediate buffers between loops and the hardware memory resource.

In summary, the SPIRAL framework has potential for customizing load-store architectures. However, it is nontrivial to address the large hardware design space and loop optimizations for hardware mapping.

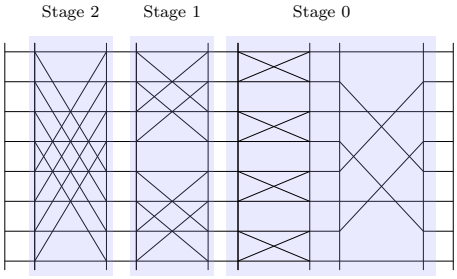
## 1.3 My Approach

This dissertation proposes an approach to generating and optimizing hardware implementations in specialized load-store architectures, through extending the SPIRAL framework. Compared to the existing streaming architecture in SPIRAL, the load-store architecture can inherently handle more complicated algorithms regardless of the uniformity in the data flow graph.

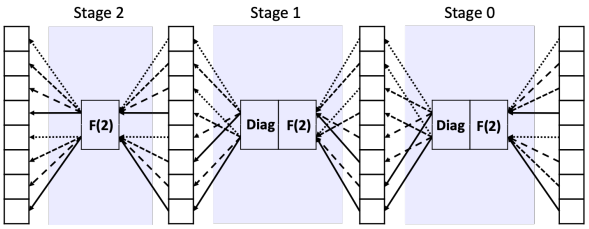
The proposed approach extends the DSLs in SPIRAL for hardware generation. It includes OL extensions for addressing the design space using the constraint solver, Σ-OL extensions for loop optimizations for hardware interpretation purpose, and icode extensions for modeling the interconnected RTL modules. The hardware icode is finally unparsed to Chisel RTL code. The resulting hardware generation flow is shown in the middle of Figure 1.4. Note that the hardware generation flow does not produce a separated instruction stream to drive the customized architecture. While this is sufficient to support simple ROM-based and FSM-based controller designs, it can be extended to incorporate other controller designs in the future.

Because successful specialization heavily relies on exploiting the properties of computation, this work has focused on a compute pattern that processes high-dimensional data cubes through indexing memory entries by multi-linear functions. This pattern covers the popular algorithms in computing the Walsh-hadamard transforms, discrete Fourier transforms, and sorting.

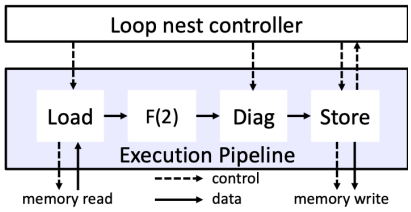
In the architecture dimension, this work has focused on a scalar load-store architecture with a flat on-chip memory that allows reading and writing one scalar data word in steady state. The parallelization can be achieved with moderate effort because the proposed extension is compatible with the parallelism paradigms in current SPIRAL.




(a) Generate an algorithm for 8-point DFT computation.




(b) Convert to iterative computing on memory.




(c) Implement using a load-store architecture.

Figure 1.6: The three steps in mapping the 8-point DFT computation to hardware designs with the proposed approach.

By focusing on a constrained compute patttern and architecture, this dissertation makes the first step in opening up the full power of SPIRAL for hardware generation. As an example, Figure 1.6 shows the representations at three abstraction levels of SPIRAL in decomposing the eight-point DFT computation to a customized load-store architecture design. At the first step, the extended constraint solver derives an algorithm for the architecture. Compared to the derived algorithm for streaming architecture in Figure 1.5 , the algorithm for load-store architecture treats data permutation in the form of memory indices. As a result, it allows different access patterns at each stage and merges the initial permutation to the first stage. Next, the data flow graph representation in OL formula is lowered to the imperfect loop nest program in Σ-OL that describes an iterative computation on memory buffers. Finally, the loop program is interpreted to the icode representation that captures the interconnected RTL modules of datapath and controller unit. The datapath is synthesized from the basic block specification of the loop program. The controller is synthesized from the loop nest structure of the program.

## 1.4 Contributions

The main contributions of this dissertation are:

1. A pattern-based approach to optimizing imperfect loop nest programs for load-store architectures. I focus on a pattern featuring static loop bounds, gather-compute-scatter shapes in the basic blocks, and multi-linear access patterns. This pattern can instantiate several important algorithms by configuring the iteration space, the memory indices, and the kernel operations correspondingly. Hardware-oriented optimizations including execution overlapping, reduced buffer allocation and simplified calculation of the multi-linear expres-

sions are developed for this pattern.

2. A code generation framework targeting the generation of customized load-store architectures using dual-ported memory. The framework is obtained by extending the Spiral approach. The Σ-OL language is extended for pattern-based loop optimizations. The icode language is extended for modeling the interconnected RTL modules internally and produces Chisel RTL code natively.

3. An implementation of the proposed approach in the Spiral system and the application of the proposed approach to Walsh-Hadamard transforms, discrete Fourier transforms, and bitonic sorters using a scalar load-store architecture connected to an on-chip dual-ported memory.

## 1.5 Limitations

It is worth noting that this work marks the first fundamental step in extending Spiral for generating customized load-store architectures while more efforts are required to uncover the full power of load-store architectures. First, this work has focused on algorithms with limited irregularity. Other algorithms may benefit more from the flexible nature of load-store architectures. Second, this work has constrained itself to a load-store architecture with a dual-ported memory. This limits the peak processing throughput to one data word per cycle. To scale the throughput, parallel techniques like SIMD and multicore can be employed which are compatible to the Spiral framework.

## 1.6 Thesis Outline

Chapter 2 presents the background of SPIRAL. The complete code generation flow is introduced which emphasizes the multi-level DSLs of OL, Σ-OL and icode. The existing SPIRAL hardware backend is also presented.

Chapter 3 explains the basic idea of synthesizing customized load-store architectures from imperfectly nested loop programs. The chapter discusses the major synthesis challenges and the proposed solutions.

Chapter 4 focuses on the development of a systematic approach to translating imperfect loop nest programs conforming to a pattern to load-store architecture with dual-ported memory, by extending the SPIRAL framework from algorithm generation to program optimization to hardware interpretation which covers the DSLs of OL, Σ-OL and icode.

Chapter 5 evaluates the effectiveness of the proposed approach. First, the flexibility is investigated via the cost of adding new algorithms to the framework. Second, the effectiveness of the program optimization for hardware generation is studied. Third, an FFT core generated with the proposed approach is compared against those produced by the existing SPIRAL hardware backend on a Xilinx FPGA device for performance and resource utilization.

Chapter 6 presents concluding remarks and future directions.

# Chapter 2

# The Spiral Code Generation Approach

Spiral is a code generation approach aiming to provide performance portability for well-defined, ubiquitously needed computational kernels across a wide range of computational devices. It provides a solid foundation for this thesis to generating hardware implementations using customized load-store architectures. The majority of this chapter explains the principles of Spiral, and its code generation flow targeting commodity processors which could be extended for synthesizing specialized load-store architectures. This chapter ends by briefing the previous hardware generation effort that focuses on a streaming architecture.

## 2.1 Performance Portability of Compute Kernels

Obtaining efficient implementations for complex algorithms on modern computers is challenging due to the complicated architecture features used for scaling performance from the initial stored-program computers, including the deep memory

hierarchy, the parallel computing paradigms including SIMD, multi-core, many-core, and distributed memory. The SPIRAL code generation approach is based on an observation that the mathematics backing ubiquitous computations is quite stable while the computer platforms change frequently and span a wide spectrum of computing power. SPIRAL aims to automatically generate implementations for well-defined computational specifications on given platforms, that are comparable to expert-tuned designs.

To address the problem of synthesizing highly efficient implementations for computational kernels, SPIRAL develops a formal framework that captures computational algorithms, computing platforms, and program transformations using a unified representation called *operator language* (OL). Then the problem is casted as a constrained optimization problem that is solved by multistage rewriting. The following three sections explains the SPIRAL approach centering around the three DSLs of the multistage rewriting flow. The three DSLs, from top to bottom, captures the data flow graphs (DFGs), abstract loops, and intermediate code as is shown in Figure 1.4. Along the generation flow, efficient implementations are obtained by lowering and optimizing the specifications step by step.

## 2.2 Architecture-aware Algorithm Generation in OL

SPIRAL focuses on computational kernels with recursive or iterative nature through which algorithmic variants can be obtained by different expansions of the kernel specification. The degrees of freedom in recursive breakdown span a large algorithm space. The first step of SPIRAL is deriving the "right" algorithms for a given computing platform. For doing that, OL is used to capture the algorithms, the hardware architectures, and the program transformations. A constraint solver is

setup by specifying the algorithm breakdown rules, architecture-specific breakdown rules and the base cases. Then, the solutions are obtained by recursively applying breakdown rules to the functional specification until fully expanded. Finally, an auto-tuning process evaluates the qualities of corresponding implementations to select the best solution from the multiple candidates.

### 2.2.1 Algorithm Abstraction

#### Specifications

SPIRAL captures the specifications of computational kernels as mathematical *operators* with unambiguous input/output behaviors that map vectors to vectors. The operators support taking multiple input vectors and producing multiple output vectors, utilizing multiple base types for the vectors, including fields $(\mathbb{R}, \mathbb{C}, GF(k))$, rings and semi-rings. In this work, the focus is on the kernels that map from one vector to another vector in the the common fields of $\mathbb{N}$, $\mathbb{R}$ or $\mathbb{C}$.

Classic digital signal processing examples of operators defined in SPIRAL are the Walsh-Hadamard transform (WHT),

$$\mathrm{WHT}_n : \mathbb{R}^n \to \mathbb{R}^n : \boldsymbol{x} \mapsto \left[\frac{1}{2^{n/2}}(-1)^{\sum_j k_j n_j}\right]_{0 \le k,l < n} \boldsymbol{x}, \tag{2.1}$$

and the discrete Fourier transform (DFT),

$$\mathrm{DFT}_n : \mathbb{C}^n \to \mathbb{C}^n : \boldsymbol{x} \mapsto \left[e^{2\pi i k l}\right]_{0 \le k,l < n} \boldsymbol{x}, \tag{2.2}$$

Non-linear operators are also supported, with an example of the sorting net-

work in ascending order,

$$\chi_n : \mathbb{N}^n \to \mathbb{N}^n : (a_i)_{0 \le i < n} \mapsto \left(a_{\sigma(i)}\right)_{0 \le i < n}, a_{\sigma(j)} \le a_{\sigma(k)} \text{ for } j \le k, \tag{2.3}$$

and in descending order,

$$\Theta_n : \mathbb{N}^n \to \mathbb{N}^n : (a_i)_{0 \le i < n} \mapsto \left(a_{\sigma(i)}\right)_{0 \le i < n}, a_{\sigma(j)} > a_{\sigma(k)} \text{ for } j \le k \tag{2.4}$$

Besides the above linear and non-linear kernels that are used in this thesis, SPIRAL has modeled many more kernels as OL operators. Examples include the discrete cosine transform [3], the wavelet transforms [4], the polar formatting synthetic aperture radar [5], the matrix-matrix multiplication [6], and the Viterbi decoder [7].

### Operator Language

SPIRAL employs data flow graphs (DFGs) to represent algorithms. The DFGs in SPIRAL are modeled with the operator language (OL) for convenient manipulation. Specifically, OL models the basic DFG fragments as *operators* and the meaningful shapes of fragments using *higher-order functions*.

The linear transform origin of SPIRAL has brought linear operators that can be represented as matrices. The specifications (2.1) and (2.2) are linear operators. The basic linear operators contains the the identity matrix

$$\mathrm{I}_n = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & \cdots & 1 \end{bmatrix}, \tag{2.5}$$

the stride permutation matrix $L^n_k$, which reads the input at stride and stores it at stride 1, defined by its corresponding permutation

$$L^n_k : i(\frac{n}{k}) + j \mapsto jk + i, \quad 0 \leq i < k, 0 \leq j < \frac{n}{k}, \tag{2.6}$$

the diagonal matrix $D_n$ generated by a function $diag(d_0, \cdots, , d_{n-1})$,

$$D_n = diag(d_0, \cdots, , d_{n-1}) = \begin{bmatrix} d_0 & 0 & \cdots & 0 \\ 0 & \ddots & \cdots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & \cdots & d_{n-1} \end{bmatrix} \tag{2.7}$$

which scales the input vector by multiplying to the weights.

In contrast to the linear operators presented above, non-linear operators cannot be represented as matrices but they are also clearly defined as shown in the specifications of (2.3) and (2.4).

Higher-order functions capture the shapes of DFG fragments that are essential to reason about efficient implementations. The direct sum $\oplus$ of operator $A$ and $B$

$$(A \oplus B) \cdot x = \begin{bmatrix} A & \\ & B \end{bmatrix} x \tag{2.8}$$

partitions the input vector into two sub-vectors to feed $A$ and $B$ separately and concatenates the resulted sub-vectors to form the result vector. The Kronecker

product $\otimes$ [8] of an identity matrix $I_n$ and an operator $A_m$

$$(I_n \otimes A_m) \cdot x = \begin{bmatrix} A_m & & & \\ & A_m & & \\ & & \ddots & \\ & & & A_m \end{bmatrix} x \tag{2.9}$$

repetitively applies $A_m$ to $n$ size-$m$ sub-vectors obtained by evenly divided the input vectors. Because the computations of each $A_m$ on sub-vectors are independent, this shape fits parallel computations. The Kronecker product $\otimes$ of $A_m$ and $I_n$

$$(A_m \otimes \mathrm{I}_n) \cdot x = \begin{bmatrix} a_{0,0}\mathrm{I}_n & \dots & a_{0,m-1}\mathrm{I}_n \\ \vdots & \ddots & \vdots \\ a_{m-1,0}\mathrm{I}_n & \dots & a_{m-1,m-1}\mathrm{I}_n \end{bmatrix} x \tag{2.10}$$

also performs repetitive operations except that the size-$m$ sub-vectors are obtained from the input vector with stride of $n$. In another word, the same $A_m$ operator is applied for neighboring data items thus can exploit data parallelism through vectorizing the computations. In non-linear cases where an intuitive matrix representation does not apply, the Kronecker product is defined formally in OL. For example, the Kronecker product of $\mathrm{I}_n$ and a non-linear operator $B_m(\cdot)$ is defined as

$$(\mathrm{I}_n \otimes B_m(\cdot))(x_0 \oplus \cdots \oplus x_{n-1}) = B_n(x_0) \oplus \cdots \oplus B_n(x_{n-1}) \tag{2.11}$$

using the direct sum of vectors.

Another important higher-order operator is composition. The composition $\circ$ of $A$ and $B$

$$(A \circ B)(x) = A(B(x)) \tag{2.12}$$

represents consecutive computational steps where the the input vector is firstly manipulated by operator $B$, then the result is processed by operator $A$. Sometimes the $\circ$ operator is omitted for simplicity while multiple operators can be visually separated, as we shall see in (2.15).

Besides the existing operators in current OL, the language is arbitrarily extensible as long as the extensions are well-defined and mathematically legal. In this work, we will slightly modify the sorting operator to add the sorting direction as a parameter to the operator.

### Breakdown Rules

In Spiral, algorithms describing how to compute a specification with a series of compute stages are encoded as breakdown rules. Note the difference between the algorithms modeled as breakdown rules and the fully specified algorithms obtained by the recursive application of breakdown rules to an input specification.

A breakdown rule matches a *non-terminal* (a not yet fully expanded kernel specification) and replaces the matched operator with the right-hand side of the rule. Spiral has defined more than 200 breakdown rules, part of which relevant to the specifications (2.1) - (2.4) are explained here.

**WHT.** The splitting WHT algorithm is expressed as

$$\mathrm{WHT}_{2^{k_1+k_2}} \rightarrow (\mathrm{WHT}_{2^{k_1}} \otimes \mathrm{I}_{2^{k_2}})\,(\mathrm{I}_{2^{k_1}} \otimes \mathrm{WHT}_{2^{k_2}}), \tag{2.13}$$

which encodes that the nonterminal $\mathrm{WHT}_{2^{k_1+k_2}}$ is translated into a right-hand side that involves new nonterminals $\mathrm{WHT}_{2^{k_1}}$ and $\mathrm{WHT}_{2^{k_2}}$. In this rule, the right-hand side nonterminals are of smaller power-of-2 sizes. Hence, the recursive application of rule will terminate when a terminal rule specifies how a minimal size $\mathrm{WHT}_2$ be

translated into an atomic OL operator which is called a "butterfly",

$$\mathrm{WHT}_2 \to F_2 \text{ with } F_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}. \tag{2.14}$$

**DFT.** Similarly, the nonterminal $\mathrm{DFT}_{mk}$ can be factorized with the general radix Cooley-Tukey FFT algorithm

$$\mathrm{DFT}_n \to (\mathrm{DFT}_k \otimes \mathrm{I}_m)\, T^n_m\, (\mathrm{I}_k \otimes \mathrm{DFT}_m)\, \mathrm{L}^n_k, \tag{2.15}$$

that translates the input to new nonterminals $\mathrm{DFT}_m$ and $\mathrm{DFT}_k$. This rule produces additional OL operators of stride permutation $\mathrm{L}^n_k$, and $T^n_m$ being a diagonal matrix defined as

$$T^n_m = \mathrm{diag}(d_0, \ldots, d_{n-1}), \text{ where } d_i = \omega_n^{\lfloor \frac{i}{m} \rfloor (i\%m)}. \tag{2.16}$$

The termination rule of DFT goes to a butterfly operator as in (2.14)

$$\mathrm{DFT}_2 \to F_2 \quad \text{with } F_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}. \tag{2.17}$$

**Sorting network.** The bitonic sorter algorithm for sorting a vector of power-of-2 sizes to ascending order is encoded in a breakdown rule

$$\chi_n \to M_{n,\chi} \left( \chi_{n/2} \oplus \Theta_{n/2} \right), \tag{2.18}$$

that produces nonterminals of a half size ascending sorter $\chi_{n/2}$, a half size descending sorter $\Theta_{n/2}$, and a bitonic merger of ascending order $M_{n,\chi}$. The bitonic merger can

be factorized with a rule

$$M_{n,\chi} \rightarrow \left(\mathrm{I}_2 \otimes M_{n/2,\chi}\right)\left(\chi_2 \otimes \mathrm{I}_{n/2}\right), \tag{2.19}$$

that involves $\Theta_2$ and a half-size merger. The descending sorter is factorized with a rule

$$\Theta_n \rightarrow M_{n,\Theta}\left(\chi_{n/2} \oplus \Theta_{n/2}\right), \tag{2.20}$$

similar to (2.18) except that a bitonic merger of descending order $M_{n,\Theta}$ is produced. The reversed order bitonic merger is then factorize by a rule similar to (2.19),

$$M_{n,\Theta} \rightarrow \left(\mathrm{I}_2 \otimes M_{n/2,\Theta}\right)\left(\Theta_2 \otimes \mathrm{I}_{n/2}\right). \tag{2.21}$$

Since sorting is non-linear, the terminal rules define the base case sorter with defining the behaviors explicitly, being the *minmax* and *maxmin* operation respectively, as shown in (2.22) and (2.23).

$$\chi_2 \rightarrow S_2 \quad \text{with } S_2 \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = \begin{bmatrix} \min(x_0, x_1) \\ \max(x_0, x_1) \end{bmatrix} \tag{2.22}$$

$$\Theta_2 \rightarrow \hat{S}_2 \quad \text{with } \hat{S}_2 \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = \begin{bmatrix} \max(x_0, x_1) \\ \min(x_0, x_1) \end{bmatrix} \tag{2.23}$$

### 2.2.2 Program Transformations

Section 2.2.1 has presented the formal representations in OL how a specification is computed through smaller problem sizes via breakdown rules. Earlier research like [9] has shown that the computations captured now in OL, including the stride permutations and the Kronecker products, can be converted to their mathematical

equivalence with different computational structures to address various computer platforms. SPIRAL casts the transformations enabled by mathematical equivalence as breakdown rules and employs hardware properties (to be explained in Section 2.2.3) to constrain the application of these breakdown rules.

First, the identity matrix is the tensor product of identity matrices

$$\mathrm{I}_{mn} \rightarrow \mathrm{I}_m \otimes \mathrm{I}_n \tag{2.24}$$

which can be used to represent loop tiling. The following example describes using (2.24) to tile the $mn$ iterations of applying the operator $A$ into $m$ iterations of $n$ computations of $A$:

$$\mathrm{I}_{mn} \otimes A \rightarrow \mathrm{I}_m \otimes (\mathrm{I}_n \otimes A)$$

Further, the two forms of Kronecker products (2.9),(2.10) can be mutually converted by introducing additional stride permutations.

$$\left(\mathrm{I}_m \otimes A^{n\times n}\right) \rightarrow \mathrm{L}_m^{mn}\left(A^{n\times n} \otimes \mathrm{I}_m\right)\mathrm{L}_n^{mn} \tag{2.25}$$

$$\left(A^{n\times n} \otimes \mathrm{I}_m\right) \rightarrow \mathrm{L}_n^{mn}\left(\mathrm{I}_m \otimes A^{n\times n}\right)\mathrm{L}_m^{mn} \tag{2.26}$$

Rule (2.25) specifies that the block parallel computations of $A_m$ can be equivalently computed as vectorized $A_m$ when performing stride permutations $\mathrm{L}_n^{mn}$ in advance and $\mathrm{L}_m^{mn}$ afterwards. Rule (2.26) specifies the reversed conversion with different parameters for stride permutations.

Finally, the point-to-point data layout transformations as stride permutations can be challenging in performance when permuting a large data set. In fact,

they can be performed in a way that exploits block granurarity.

$$\mathrm{L}^{kmn}_{n} \rightarrow \left(\mathrm{L}^{kn}_{n} \otimes \mathrm{I}_m\right)\left(\mathrm{I}_k \otimes \mathrm{L}^{mn}_{n}\right), \tag{2.27}$$

$$\mathrm{L}^{kmn}_{km} \rightarrow \left(\mathrm{I}_k \otimes \mathrm{L}^{mn}_{m}\right)\left(\mathrm{L}^{kn}_{k} \otimes \mathrm{I}_m\right), \tag{2.28}$$

The rules (2.27), (2.28) involves the Kronecker product of stride permutations. $\mathrm{I}_k \otimes \mathrm{L}^{mn}_{n}$ encodes block permutation of size $mn$ while $\mathrm{L}^{kn}_{k} \otimes \mathrm{I}_m$ encodes permutation of blocks of size $m$. These behaviors are more efficient in communication with memory or dedicated channels whose latency can be hidden by block transfers.

### 2.2.3 Hardware Abstraction

The architectural features that require data flow optimizations are captured in the formal framework. These features include memory organizations and parallel paradigms.

The hardware features are modeled through constraints on OL breakdown rules via hardware *tags*. A breakdown rule can be associated with tags on the left-side nonterminal, encoding a "right" way to expand the nonterminal such that the computations can be efficiently implemented in hardware platform with the prescribed properties in the tags.

By addressing the general data flow patterns and architectural features in OL, SPIRAL has provided general tagged breakdown rules to handle common OL formulas. For instance, the formula $A_m \otimes \mathrm{I}_n$ is inherently vectorizable and can be efficiently implemented in $v$-way short-vector SIMD platforms as shown in a rule

$$\underbrace{A_m \otimes I_n}_{\mathrm{vec}(v)} \rightarrow (A_m \otimes \mathrm{I}_{n/v}) \otimes_{vec(v)} \mathrm{I}_v \quad \text{with } v|n. \tag{2.29}$$

Rule (2.29) uses the equivalence of (2.24). The tag on the left side $\text{vec}(v)$ denotes $v$-way SIMD vectorization. On the right side, the underlying tag is removed and it means that the expanded OL formula has been restructured for SIMD execution. This is achieved by tagging a Kronecker product operator with the same tag $\text{vec}(v)$, which encodes the decision that this operator should be implemented as $v$-way SIMD operations in the code generation backend. The OL shape $A_m \otimes_{vec(v)} \mathrm{I}_v$ is a *base case* for SIMD hardware platform. The entire set of SIMD base cases can be found in [10].

Algorithm-specific tagged breakdown rules are important if direct matching on general OL patterns does not produce the optimal results. Sometimes, the combination of known OL patterns can be collaboratively optimized through ingeniously picking the right rules, from a large identified set of program transformation rules explained in Section 2.2.2, and applying them in a right way. For example, though there are independent SIMD tagged rules for $\mathrm{I}_m \otimes A^{n\times n}$ and $\mathrm{L}^{mn}_m$, the SIMD vectorization of the composition of the two produced by the Cooley-Tukey FFT factorization rule (2.15) can embrace OL stage reduction through careful program transformation. The FFT-specific tagged rule is

$$\underbrace{\left(\mathrm{I}_m \otimes A^{n\times n}\right) \mathrm{L}^{mn}_m}_{\text{vec}(\nu)} \rightarrow \underbrace{\left(\mathrm{I}_{m/\nu} \otimes \mathrm{L}^{n\nu}_{\nu} \left(A^{n\times n} \otimes \mathrm{I}_\nu\right)\right) \left(\mathrm{L}^{mn/\nu}_{m/\nu} \otimes \mathrm{I}_\nu\right)}_{\text{vec}(\nu)}, \quad \nu \mid m \tag{2.30}$$

By comparing Rule (2.30) and the results achieved by applying Rule (2.28) and 2.25 separately, the number of stages is reduced. This will make across-stage optimizations to be introduced in Section 2.3 feasible.

### 2.2.4 Algorithm Generation and Autotuning

With a unified formal framework for algorithms, hardware architectures, and program transformations, the problem of generating fully-expanded algorithms of computational kernels efficient for a given hardware platform is casted as a strongly constrained optimization problem that is solved by rewriting and search.

The constrained solver is setup by defining the algorithm space as a set of algorithmic breakdown rules, the hardware space as base cases, and the transformation space as hardware-tagged breakdown rules. It is a non-trivial task because it requires the combination of domain-knowledge and architecture expertise from human designers. To solve the constrained optimization problem, breakdown rules are applied recursively until all nonterminals are translated to terminals. The order of rule applications on nonterminals is called the *rule tree*, and by applying rules to the input OL specification with respect to the rule tree, we can obtain fully expanded OL formulas representing a flat DFG.

Because breakdown rules have degrees of freedom on how a nonterminal can be expanded, an input specification can be finally translated to many fully-expanded algorithms. To pick the optimal algorithms from the large candidate set, SPIRAL employs auto-tuning techniques to search within the design space. In the past, the combination of genetic search, dynamic programming, line search and exhaustive search have been used in the SPIRAL framework.

As an example, Figure 2.1 shows the algorithm generation and auto-tuning approach of how to find the most efficient algorithm of $\mathrm{DFT}_8$ for a 2-way vectorized processor. Hardware rules (left, red), algorithm rules (right, blue) and program transformation rules (center, grey) together span a search space (multi-colored oval). The given problem specification and hardware target give rise to the solution space

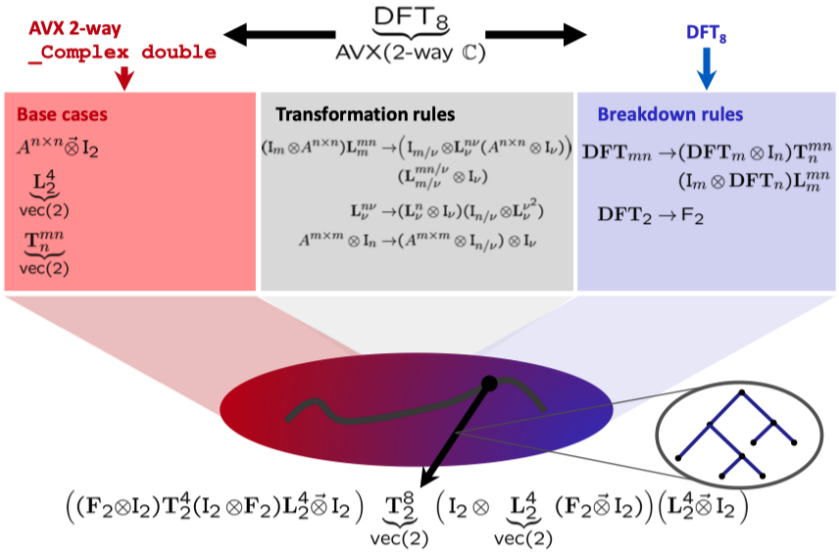



Figure 2.1: Algorithm generation as a constraint problem [1].

(black line) that is a subspace of the overall search space. Each point in the solution space (black dot) represents a DFG given as OL formula that optimized for the SIMD architecture. An auto-tuner walks through the solution space by evaluating the quality of each algorithm in implementations and finds the optimal solution out of the space.

## 2.3 Across-stage Optimization in Σ-OL

The algorithms generated with the constraint solver are flat DFGs containing multiple compute or data reorganization stages, each of which is optimized for the given compute platform. However, cross-stage optimization is inevitable to achieve efficiency on modern processors.

All commodity processor architectures share a same property that data resides in main memory and the processing core fetches data from memory, writes back

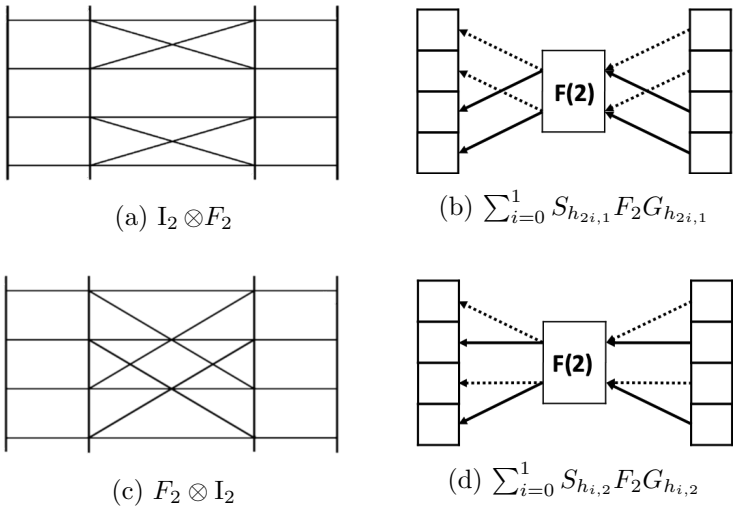
results to memory in an iterative manner. Moreover, the arithmetic / logical processing speed is usually much faster than data movements [11]. Consequently, efficient computations on modern processors tend to minimize data roundtrip. Specifically, it is important to merge neighboring stages in flat DFGs to combine computations and data reorganization on the same data set as much as possible. In addition, in case the data movement bandwidth falls behind the computational throughput, multi-buffering is essential to use bandwidth efficiently [12].

This section focuses on stage fusion for the flat DFGs. Since each stage exhibits repetitions and can be represented as a loop program, it is a loop fusion problem. SPIRAL introduces the Σ-OL language to make loops explicit and memory access patterns symbolically such that the difficult loop merging problems difficult to general compiler optimizers can be solved via a rewriting system.

### 2.3.1 Lowering DFGs to Loops

To represent repetitive operations in DFGs as a loop program, one has to provide an explicit memory abstraction, extract the kernel operation, and specifies how data is addressed in loading from memory to the kernel and storing from the kernel to memory. As in Spiral, a kernel always processes a vector, the load and store operations are gather and scatter operations [13] that addressing data in memory indirectly through an index vector.

A graphical representation of translating the two most common repetitive data flow patterns to loop programs are shown in Figure 2.2. The left-hand side shows the DFG. The right-hand side shows the corresponding loop program representation that uses stacked squares for memory, places the kernel operation to the center, gathers and scatters data prescribed by arrows. Note that although two




(a) $\mathrm{I}_2 \otimes F_2$ (b) $\sum_{i=0}^{1} S_{h_{2i,1}} F_2 G_{h_{2i,1}}$

(c) $F_2 \otimes \mathrm{I}_2$ (d) $\sum_{i=0}^{1} S_{h_{i,2}} F_2 G_{h_{i,2}}$

Figure 2.2: The translation between OL and Σ-OL.

memory arrays are visualized, they can be mapped to the same physical memory in actual implementations. The two examples have two iterations for the loop, the first is represented by dotted-line arrows and the second is represented by solid-line arrows. The corresponding OL and Σ-OL (will be explained soon) formulas are provided as subgraph captions. Figure 2.2a shows the DFG of the OL formula $\mathrm{I}_2 \otimes F_2$. Figure 2.2b shows the translated loop representations where the $F_2$ kernel is extracted and the unit stride pattern is used for gather and scatter. In Figure 2.2c and 2.2d, the translation is shown for the OL formula $F_2 \otimes \mathrm{I}_2$, which results in a stride-2 access pattern. From this example, we saw that the different data flow patterns with the same kernel can be normalized to the same loop representation only with varying access patterns.

### 2.3.2 Σ-OL

**Iterative sum.** Σ-OL is a superset of OL that introduces a few new constructs for the loop abstraction. The core operator is the *iterative sum* operator that captures

a loop,

$$\sum_{j=0}^{n-1} A_j, \tag{2.31}$$

where $j$ is an *induction variable* with range $n$. For linear operators, the iterative sum is formally defined as the summation of vectors produced at each iteration by the $A_j$ operator. The additions are never performed because $A_j$ is guaranteed to produce non-overlapping non-zero elements when $j$ iterates. For non-linear operators, the iterative sum is defined as non-overlapped iterative data processing from memory to memory. The non-overlapped behavior is guaranteed by forming $A_j$ with the gather and scatter operators parameterized by index mapping functions for indirect access.

**Index Mapping Function.** The gather and scatter of a subset of data elements in a vector is specified by an index sub-vector. SPIRAL employs index mapping functions to generate index sub-vectors from an integer interval. An integer interval is denoted by

$$\mathbb{I}_n = 0..., n-1.$$

An index mapping function $f$ with domain $\mathbb{I}_n$ and range $\mathbb{I}_N$ is denoted by

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

We use the short-hand notation $f^{n \to N}$ to refer to an index mapping function of the form $f : \mathbb{I}_n \to \mathbb{I}_N$.

The index mapping function can be arbitrarily complicated determined by the access pattern of the algorithms. A particular important index mapping function used in this thesis is the $h$ function parametersized by a base $b$ and a stride $s$ for

strided indexing,

$$h_{b,s} : \mathbb{I}_n \to \mathbb{I}_N;\ i \mapsto b + si. \tag{2.32}$$

**Gather operator.** Gather is a matrix operator in which its definition generates a matrix that produces its output vector by multiplying with the input vector. A Gather operator parameterized by an index mapping function $f^{n\to N}$ extracts a size-$n$ subvector from the size-$N$ input vector. In this work, a Gather operator is always associated with an $h^{n\to N}_{b,s}$ function for strided indexing. Let $e^n_k \in \mathbb{C}^{n\times 1}$ be the canonical basis vector with entry 1 in position k and entry 0 elsewhere. An index mapping function $h^{n\to N}_{b,s}$ generates the gather matrix ($[\cdot]^\top$ is the matrix transposition),

$$\mathrm{G}_{h^{n\to N}_{b,s}} := \left[e^N_{h(0)} \left| e^N_{h(1)} \right| \cdots \mid e^N_{h(n-1)}\right]^\top, \tag{2.33}$$

which implies that for two vectors $x = (x_0, ..., x_{N-1})^\top$ and $y = (y_0, ..., y_{n-1})^\top$,

$$y = G_{h^{n\to N}} x \iff y_i = x_{h(i)}.$$

The result is used for generating code from the Gather matrices.

**Scatter operator.** A Scatter and a Gather with the same index mapping function $h^{n\to N}$ are the transposition of each other. It represents transferring data entries of a sub-vector to the specified locations of the output vector. An index mapping function $h^{n\to N}_{b,s}$ generates the Scatter matrix

$$\mathrm{S}_{h^{n\to N}_{b,s}} := \left[e^N_{h(0)} \left| e^N_{h(1)} \right| \cdots \mid e^N_{h(n-1)}\right] \tag{2.34}$$

The definition (2.34) implies that for two vectors $x = (x_0, ..., x_{n-1})^\top$ and $y = (y_0, ..., y_{N-1})^\top$,

$$y = S_{h^{n\to N}} x \iff y_j = \begin{cases} x_i & \text{if } j = f(i) \\ 0 & \text{else} \end{cases}.$$

The result is used for generating code from the Scatter matrices.

**Example of Σ-OL expression.** Here we explain the Σ-OL expression that describes Figure 2.2b,

$$\sum_{i=0}^{1} (S_{h^{2\to 4}_{2*i,1}} F_2 G_{h^{2\to 4}_{2*i,1}}).$$

In this example, the loop containing two iterations is captured by the iterative sum Σ operator with maximal iteration 1 starting from 0, using an induction variable $i$. The unit stride access pattern is captured by the gather operator $G_{h^{2\to 4}_{2*i,1}}$ and the scatter operator $S_{h^{2\to 4}_{2*i,1}}$. The index mapping function $h_{2*i,1}$ produces a size-2 vector with unit stride for each input $i$. The matrix operator instances for the first iteration, i.e, $i = 0$, is

$$S_{h^{2\to 4}_{0,1}} F_2 G_{h^{2\to 4}_{0,1}} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix},$$

which extracts the first two data entries of the size-4 input vector to feed the $F_2$ butterfly operator, and finally places the size-2 sub-vector result on the first two entries of the output vector while setting 0s for the untouched elements.

**Other Σ-OL Operators.** Permutation is a special case of Gather with the constraint that the index mapping function must be bijective. Though any linear

permutation is permitted, in this work, we only deals with the stride permutation. A permutation matrix corresponding to its defining permutation $p^{n\to n}$ is written as

$$\text{perm}\left(p^{n\to n}\right) := \left[e^n_{p(0)} \left| e^n_{p(1)} \right| \cdots \mid e^n_{p(n-1)}\right]^\top.$$

Thus, the algorithm to implement gather matrices is used to implement permutation matrices.

The diagonal matrices of size $n \times n$ can be defined by a function $f^{n\to\mathbb{C}} : \mathbb{I}_n \to \mathbb{C}$,

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

**Translation between OL and Σ-OL.** With the newly introduced operators, the aforementioned two special geometries (2.9)(2.10) of the data flow graph captured by the Kronecker product can be lowered to Σ-OL expressions with the following two rewrite rules,

$$\mathrm{I}_m \otimes A_n \to \sum_{j=0}^{m-1} \mathrm{S}_{h^{n\to nm}_{nj,1}} A_n \mathrm{G}_{h^{n\to nm}_{nj,1}} \tag{2.35}$$

$$A_m \otimes \mathrm{I}_n \to \sum_{j=0}^{n-1} \mathrm{S}_{h^{m\to nm}_{j,n}} A_m \mathrm{G}_{h^{m\to nm}_{j,n}} \tag{2.36}$$

### 2.3.3 Loop Merging and Index Simplification

In practical Σ-OL expressions, the iterative sum operator is in composition with other operators that also process data iteratively. By merging as much as computations and data reorganization into a single iterative sum operator, the data roundtrip from memory to memory can be minimized. The loop merging is achieved in SPIRAL

through applying rewrite rules for Σ-OL expressions provided by human designers. This is possible because of the access pattern is made symbolic as index mapping functions. Further, the index mapping functions after loop merging can be simplified by utilizing the mathematical properties of the symbolic functions.

For instance, in FFT algorithms, the diagonal and permutation operators are the common neighbors of iterative sums in an expression, as shown in the following Σ-OL expression lowered from the OL formula of 4-point FFT

$$\left(\sum_{i=0}^{1}(S_{h_{i,2}^{2\to 4}}F_2G_{h_{i,2}^{2\to 4}})\right)\operatorname{diag}(f^{4\to\mathbb{C}})\left(\sum_{i=0}^{1}(S_{h_{2*i,1}^{2\to 4}}F_2G_{h_{2*i,1}^{2\to 4}})\right)\operatorname{perm}(p^{4\to 4}) \tag{2.37}$$

which contains four consecutive computational stages over the input vector, each of which can be implemented as a loop. The rewrite rules for merging Σ-OL expressions like this are explained as follows.

Other operators than the iterative sums can be incorporated to the neighboring iterative sum operator on the left side or on the right side. Rules

$$\left(\sum_{j=0}^{m-1}A_j\right)M \to \left(\sum_{j=0}^{m-1}A_jM\right), \tag{2.38}$$

$$M\left(\sum_{j=0}^{m-1}A_j\right) \to \left(\sum_{j=0}^{m-1}MA_j\right) \tag{2.39}$$

implement the distributivity law for moving operators inside iterative sums. After applying these rules to (2.37), permutations and diagonals can be paired with the gather or scatter operators.

The permutation can be merged into gather on the left so that another data

pass for permutation can be saved. Rule

$$\mathrm{G}_{rn\to N}\,\mathrm{perm}\left(\pi^{N\to N}\right) \to \mathrm{G}_{\pi\circ r}. \tag{2.40}$$

merges permutations into gather by composing the index mapping functions of permutation and gather.

The diagonal matrices can be swapped with the right-side scatter operation such that the diagonal operator can be performed in sub-vector granularity together with other computational kernels in the iterative sum. Rule

$$\mathrm{diag}\left(f^{N\to\mathbb{C}}\right)\mathrm{S}_{w^{n\to N}} \to \mathrm{S}_w\,\mathrm{diag}(f\circ w). \tag{2.41}$$

swaps diagonal with scatter by compositing the index mapping functions of diagonal and scatter.

By applying the above rewrite rules to (2.37), we can obtain the loop merged Σ-OL expressions. However, the merging may create complicated index mapping functions via composition. A particular set of rewrite rules in [2] can be applied to simplify the index mapping functions. Finally, the merged and simplified Σ-OL expression with only two stages is obtained:

$$\left(\sum_{i1=0}^{1}(S_{h_{i1,2}^{2\to4}}F_2G_{h_{i1,2}^{2\to4}})\right)\left(\sum_{i2=0}^{1}(S_{h_{2*i2,1}^{2\to4}}diag(f^{4\to\mathbb{C}}\circ h_{2*i2,1}^{2\to4})F_2G_{p^{4\to4}\circ h_{2*i2,1}^{2\to4}})\right). \tag{2.42}$$

## 2.4 Abstract Programs in icode

To enable loop code portability between various programming languages and application programming interfaces, SPIRAL introduces the *icode* abstraction to capture the syntax of C language or C-derived dialects like OpenCL. This also enables optimizations in basic blocks at higher level than programming languages.

### 2.4.1 Code Representation

The icode captures key constructs of common programming languages, including 1) values and types, 2)arithmetic and logic operations, 3)constants, arrays and scalar variables, and 4) assignments and control flow.

**Types and variables.** A sample of some types with the corresponding C constructs are listed in Table 2.1. The integer types can be mapped to C type in a one to one manner. The real type is mapped to `float` or `real` depending on the bit width parameter.

Table 2.1: icode constructs modeling the primitive types of C language

| icode constructs | Corresponding C code constructs |
|---|---|
| TInt | `int` |
| TUInt | `unsigned int` |
| TReal(width) | `float` / `double` |

Arrays are supported with a type specifying the size and the primitive scalar type. To address the scalar elements of an array, an `nth` object is added to extract the `nth`-th element from a variable of array type. The array type support is listed in Table 2.2.

A variable object can be created by specifying the name and type as presented in the third row of Table (2.4).

Table 2.2: icode constructs supporting C arrays

| Categories | icode | Corresponding C code constructs |
|---|---|---|
| Type | TArray(type,n) | n/a |
| Location | nth(v, idx) | `v[idx]` |

**Operators.** It is worth noting that the operators in icode are dynamic objects that can autonomously determine the type of results according to the types of the input. As a result, one operator object can model the same operation of various scalar types, array types and user-defined structures. The operations of arithmetic, relational, logical, bitwise and ternary types are listed in Table 2.3. For certain arithmetic operations such as addition and subtraction, the arbitrary number of operands are supported. The code `v1, ..., vn` represents $n$ operands from `v1` to `vn` divided by comma.

**Statements.** The statements in C code are modeled as *commands* in icode. Table 2.4 lists several important commands: the assignment, the compound statement, the variable declaration statement, and the for-loop statement.

**Extending types and operators.** Since icode is arbitrarily extensible, the constructs beyond the C language specification can always be created. User-defined data types and the corresponding operations can be modeled in icode. Table 2.5 presents the complex arithmetic extension where a complex data type and three arithmetic operators are addressed. The complex data type `complex_t` is a structure specified by the `typedef` statement of C and the definition will be incorporated in the final C code. The arithmetic operations add, sub, and mul on the complex type are implemented as user-defined C functions. Besides, the types and functions provided by the C standard library [14] or other API can be addressed in icode as well. Table 2.6 lists certain math functions of the C standard libarary suppported

Table 2.3: icode constructs modeling the operations of C language

| Types | icode constructs | Corresponding C code constructs |
|---|---|---|
| Arithmetic | add(v1, ..., vn) | `v1 + ... + vn` |
| Arithmetic | sub(v1, ..., vn) | `v1 - ... - vn` |
| Arithmetic | mul(v1, ..., vn) | `v1 * ... * vn` |
| Arithmetic | div(v1, ..., vn) | `v1 / ... / vn` |
| Arithmetic | imod(v1, v2) | `v1 % v2` |
| Relational | eq(v1, v2) | `v1 == v2` |
| Relational | neq(v1, v2) | `v1 != v2` |
| Relational | geq(v1, v2) | `v1 >= v2` |
| Relational | leq(v1, v2) | `v1 <= v2` |
| Relational | gt(v1, v2) | `v1 > v2` |
| Relational | lt(v1, v2) | `v1 < v2` |
| Logical | logic_and(v1, v2) | `v1 && v2` |
| Logical | logic_or(v1, v2) | `v1 \|\| v2` |
| Logical | logic_neg(v1) | `!v1` |
| Bitwise | bin_and(v1, v2) | `v1 & v2` |
| Bitwise | bin_or(v1, v2) | `v1 \| v2` |
| Bitwise | bin_xor(v1) | `v1 ^ v2` |
| Bitwise | lShift(v1,v2) | `v1 << v2` |
| Bitwise | rShift(v1,v2) | `v1 >> v2` |
| Ternary | cond(v1,v2,v3) | `(v1)?(v2):(v3)` |

Table 2.4: icode commands modeling statements of the C language

| icode constructs | Corresponding C code constructs |
|---|---|
| assign(loc, expr) | `loc = expr;` |
| chain(icode) | { `icode` } |
| decl([var("t1",TInt)], chain()) | `int t1;` { } |
| loop(v, domain, chain(<icode>)) | `for(v=0; v<domain; v++)` { `<icode>` } |

in current icode. Furthermore, icode has been extended in the past for capturing the instruction set extensions in the SIMD vectorized architecture such as Intel's SSE and AVX and the vectorized data types.

Finally, the short cuts of complicated operations can be modeled as a single

Table 2.5: Extending icode for complex arithmetic

| Categories | icode constructs | Spiral-defined C code constructs |
|---|---|---|
| Type | TComplex | `complex_t` |
| Arithmetic | add(v1, ..., vn) | `add(v1, ... ,vn)` |
| Arithmetic | sub(v1, ..., vn) | `sub(v1, ... ,vn)` |
| Arithmetic | mul(v1, ..., vn) | `mul(v1, ... ,vn)` |

Table 2.6: icode operators modeling the C math function library

| icode constructs | C math functions |
|---|---|
| log(v1) | `logf(v1)` or `log(v1)` |
| sqrt(v1) | `sqrt(v1)` |
| abs(v1) | `abs(v1)` |

icode operator to ease pattern matching. Table 2.7 lists the max and min operations that are realized with cond, geq, and leq icode operations.

Table 2.7: Short cut icode operators

| icode short cuts | the actual icode |
|---|---|
| max(v1,v2) | cond(geq(v1,v2),v1,v2) |
| min(v1,v2) | cond(leq(v1,v2),v1,v2) |

### 2.4.2 Code Generation

The Σ-OL expressions are converted to icode expressions with a set of rules. The rewriting process is implemented in a recursive descent translator where every Σ-OL operator is mapped to a specific icode object. A sample of the rules are listed in Table 2.8. Figure 2.3 shows the icode expressions for FFT(4) translated from the Σ-OL expression in (2.42). The generated icode can be finally *unparsed* (pretty-printed) for the target compiler.

Table 2.8: Translating Σ-OL constructs to code; $x$ denotes the input and $y$ the output vector. [2]

| |
|---|
| Code$(AB, y, x) \rightarrow$ Code$(B, t, x)$; Code$(A, y, t)$; |
| Code$(\Sigma_{j=0}^{k-1} A_j, y, x) \rightarrow$<br>`for(j=0; j<k; j++)` Code$(A_j, y, x)$; |
| Code$(G_{f^{n\rightarrow N}}, y, x) \rightarrow$<br>`for(j=0; j<n; j++) y[j] = x[f(j)];` |
| Code$(S_{f^{n\rightarrow N}}, y, x) \rightarrow$<br>`for(j=0; j<n; j++) y[f(j)] = x[j];` |
| Code$(\text{perm}_{p^{n\rightarrow n}}, y, x) \rightarrow$<br>`for(j=0; j<n; j++) y[j] = x[p(j)];` |
| Code$(\text{diag}_{p^{n\rightarrow \mathbb{C}}}, y, x) \rightarrow$<br>`for(j=0; j<n; j++) y[j] = f(j)*x[j];` |

### 2.4.3 Local Optimization

While the loop optimizations have been performed at OL and Σ-OL levels, the basic blocks can be optimized in icode. The main basic block optimizations like loop unrolling, array scalarization, constant folding, copy propagation, and common subexpression elimination have been implemented in SPIRAL.

## 2.5 Generating Streaming Hardware with Spiral

The formal framework for algorithm generation introduced in Section 2.2.1 can inherently go beyond the general processor architectures. In the past, a streaming architecture has been targeted in SPIRAL for generating high-throughput RTL implementations for linear transform kernels, including the discrete Fourier transforms (DFT), multi-dimensional DFT, real DFT and discrete sine/cosine transforms [15]. A streaming architecture continuously takes in data stream from the input ports and produces result stream on the output ports. It is typically composed of mul-

```
chain(
 loop(i2, 2, chain(
    loop(i4, 2, assign(nth(T1, i4))),
    chain(assign(t1, nth(T1, 0)),
          assign(t2, nth(T1, 1)),
          assign(nth(T2, 0), add(t1,t2)),
          assign(nth(T2, 1), sub(t1,t2))
    ),
    loop(i3, 2, assign(nth(T3, i3), nth(T2, i3))))
 ),
 loop(i1, 2, chain(
    loop(i4, 2, assign(nth(T1, i4))),
    chain(assign(t1, nth(T1, 0)),
          assign(t2, nth(T1, 1)),
          assign(nth(T2, 0), add(t1,t2)),
          assign(nth(T2, 1), sub(t1,t2))
    ),
    loop(i3, 2, assign(nth(T3, i3), nth(T2, i3))))
 )
)
```

Figure 2.3: The icode for FFT-4 translated from (2.42).

tiple internal stages, in a producer-consumer relationship, that performs streaming operations concurrently. Memory is utilized internally only when buffering is required in some streaming operations.

**Code generation flow.** The streaming hardware generation flow is presented at the right-most side of Figure 1.4. First, the architecture-aware algorithm generation process produces SPL formulas for streaming processing. Then, an external hardware compiler takes in specifications in SPL[1] and produces hardware datapath designs in Verilog RTL.

**Permutations.** In a streaming architecture, data reorganization is treated as explicit streamed operations that require internal storage buffers. The automated implementation of streamed permutations have been studied formally for arbitrary fixed-size permutations in [16] and for power-of-2 size permutations as bit permu-

[1]SPL is the predecessor of the OL language. Similar to OL, SPL captures algorithms in flat data flow graphs but limits to linear operators.

tations in [17].

The internal usage of memory in streamed permutation creates challenges in sharing the precious on-chip storage elements. For instance, when integrating the streaming core to a DRAM-based hardware acceleration environment, the replication of local memory is required though buffers have been implemented in the hardware core itself, as is shown in [18]. In contrast, a hardware core based on a load-store architecture can expose its local memory directly to the off-chip memory controller. Further, distinct kernels that impose challenge in functional unit sharing can still natively share the local memory.

**Datapath folding and limitations.** A major contribution of [15] is creating a large tradeoff space between throughput and area usage for streaming architectures. It is achieved by folding the datapath horizontally and vertically [15], as is shown in Figure 1.5. This requires the regularity in DFGs such that the folded operations are performed with identical datapath.

However, the folding method also limits the choice of algorithms for efficient implementations. For instance, a fully folded datapath design for FFT, implementing only one streamed butterfly kernels and one streamed permutation kernel, is locked to the iterative Pease algorithm, not to mention that the initial bit-reversal permutation has to be performed with another streamed permutation kernel. The load-store architecture studied in this work targets the low to medium throughput scenario and can support a wider range of algorithms.

**hardware extensions.** Nevertheless, the previous work in streaming architectures has provided precious experience in extending Spiral for hardware generation. First, the stream tags introduced to the constraint solver shows how a new architectural feature can be accommodated into Spiral. Second, it extends icode to

capture the fully pipelined datapath for RTL implementations. Each relevant icode construct is backed by a (parameterized) RTL module. The hardware extensions manifested the extensibility of the SPIRAL approach, which encourages this work for creating a more flexible hardware backend for SPIRAL.

## 2.6 Summary

This chapter introduces the background of the SPIRAL approach for code generation. SPIRAL employs OL to capture algorithms, hardware features, and program transformations in a unified formal framework, so as to generate architecture-specialized algorithms. The generated algorithms are then translated to the abstract loop representation in Σ-OL for loop merging and index simplification. Finally, optimized Σ-OL expressions are translated to the internal representation icode where basic block optimization is performed before unparsed to the executable program code.

The final section presents an overview of the previous SPIRAL hardware generator targeting a streaming architecture. The hardware extensions provide this work experiences and lessons for developing a new hardware generation backend of SPIRAL.

After introducing the background of the SPIRAL approach, the next chapter will explain the high level idea of flexible hardware generation targeting customized load-store architectures from imperfect loop nest programs.

# Chapter 3

# The Concepts of Load-store Architecture Synthesis

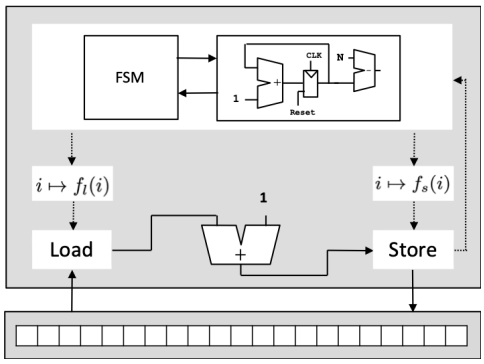
In this dissertation, the flexibility of hardware design generation is achieved by synthesizing loop programs to customized load-store architectures. This chapter explains the high-level ideas and introduces several challenges to overcome. We will begin with the mapping from a simple loop program to a basic load-store architecture. Then, we discuss a particular form of imperfect loop nest programs amenable to hardware acceleration. Afterwards, we elaborate the implementation flexibility of load-store architectures. Finally, three major challenges encountered in load-store architecture synthesis are discussed with the proposed solutions.

## 3.1 From Loop Programs to Load-store Architectures

Loops in high-level imperative programming languages such as C [19], are statement constructs that capture the repetition of other statements based on the structured programming paradigm [20]. The enclosed statements could be also loops or other

```
1 int i;
2 for (i=0; i<N; i++) {
3     B[i] = A[i] + 1;
4 }
```

(a) A for-loop program.




(b) The corresponding hardware design.

Figure 3.1: Mapping a simple for-loop program to customized load-store architecture

statements such as an assignment that performs an operation on information located in memory and store the results in memory for later use. In Figure 3.1a, the pseudo code shows a for-loop encoding the repetitive computations of adding integer 1 to each entry of a size-`N` array `A` and placing results in array `B`. Line 2 describes an integer variable `i` to be initialized to 0 and is incremented by 1 after each execution of the loop body enclosed by a pair of curly braces. The execution of the loop body is guarded by the condition `i<N`, and does include one assignment at Line 3 which computes one element of `B`. The nested structure of loops that are used in numerous practical algorithms will be introduced in the following sections.

In this thesis, we define the *load-store architecture* as a style of computational hardware organization that allows arbitrary access from the units of computations to memories. A basic load-store architecture is composed of a memory with one read port and one write port, a pipelined datapath connected to the memory, and a controller that manages the behavior of the datapath. In later sections, we will elaborate the more complicated designs that cover a wide range of computational patterns, throughput, and configurability.

A simple loop program like Figure 3.1a can be translated to a basic load-store

architecture design. Since the `N` iterations of additions are independent to each other, they can be executed concurrently in a customized pipelined datapath, as shown in Figure 3.1b. The pipelined datapath starts by loading a data element of array `A` from memory. It is followed by an addition to the input data and ended by storing the summation back to memory. The loading and storing operations require an address parameter provided by their dedicated address calculation functions with the current value of loop variables provided by the controller. The controller traverses the size-`N` iteration space by using a finite state machine, a counter and a comparator.

Despite the simplicity of the above example, it clearly shows that even though the load-store behavior resembles how a processor computes, the actual components of the architecture can be fully customized and simplified for hardware efficiency.

## 3.2 Imperfectly Nested Loops

This thesis focuses on a particular form of *imperfectly nested loops* that is amenable to hardware acceleration. The powerful expressibility of imperative programming languages allows arbitrarily complex computations to be specified, while only part of them are suitable for hardware acceleration. In fact, only a subset of loop programs are handled in the most advanced hardware compilation methods [21][22]. Compared to the general hardware compilers, our approach relies more on the "right" structures of computations for obtaining efficient results.

Imperfectly nested loop is a widely used term to describe program structures where loop statements are nested imperfectly. In a perfect way of nesting, each loop encloses another loop as the loop body except that the innermost loop contains a loop body with non-loop statements for computations. Since the perfect loop nest has only one basic block, it is usually regarded as a special case in hardware

```
1  for(i2 = 0; i2 <= 3; i2++) {
2      s13 = X[2*i2];
3      s14 = X[2*i2+1];
4      T1[2*i2] = s13+s14;
5      T1[2*i2+1] = s13-s14;
6  }
7  for(i1 = 0; i1 <= 1; i1++) {
8      for(i4 = 0; i4 <= 1; i4++) {
9          s21 = T1[i1+4*i4];
10         s22 = T1[i1+4*i4+2];
11         T2[2*i4] = s21+s22;
12         T2[2*i4+1] = s21-s22;
13     }
14     for(int i3 = 0; i3 <= 1; i3++) {
15         s29 = T2[i3];
16         s30 = T2[i3+2];
17         Y[i1+2*i3] = s29+s30;
18         Y[i1+2*i3+4] = s29-s30;
19     }
20 }
```

Figure 3.2: An imperfect loop nest program computing the 8-point Walsh-hadamard transform.

synthesis [23]. In this thesis, we focus on an imperfect loop nest structure inspired by SPIRAL, where each loop level allows either a single loop or multiple loops with producer-consumer relationship. Figure 3.2 shows a code block of imperfect loop nest programs with the desired structure. The top level contains loop-`i2` and loop-`i1` which use array `T1` as an intermediate buffer for data communication. Loop-`i1` contains the second level composition of loop-`i4` and loop-`i3` using intermediate buffer `T2`. As we can see, multiple basic blocks are allowed in imperfect loop nest programs.

The imperfect loop nest programs studied in this thesis includes certain important properties that make it suitable for hardware acceleration.

**Independent iterations.** The loop iterations at each loop level are independent to each other. Given the arbitrary depth of nesting and the imperfect nesting, this allows parallelism at various granularities. For instance, the repeti-

tive execution of the basic block of each innermost loop can be computed through pipelined parallelism. In contrast, the iterations of an outer loop can be computed through multiple processing elements in parallel.

**Regular basic block shape.** The statements at each basic block share an identical pattern. Each basic block loads a vector from a read buffer with calculated indices. Then the input vector is processed through an arbitrary operator to generate an output vector. Finally, the output vector is written to a write buffer with calculated indices. This pattern allows the concurrency of data access and computation in the pipelined datapath design of the load-store architecture.

**Static loop bounds.** The number of repetitions in all loops are statically determined. This results in the data-independent control flow that allows the controller and datapath to be decoupled with limited interactions in load-store architecture designs.

Even though the listed properties of imperfect loop nests appear to be restricted, loop programs satisfy those properties have been observed in high performance implementations of numerous algorithms, ranging from signal processing, media processing to machine learning. For example, the classic recursive Cooley-Tukey FFT algorithm views the input data as a two-dimensional tensor and computes with row-wise iterations followed by column-wise iterations [24]. Image encoding algorithms are usually designed as block-wise iterative operations on 2D pixels [25]. An analytical model of matrix-matrix multiplications suggested designing the kernel by calculating independent outer products in the innermost loop [26]. In fact, by considering an inherently parallelizable compute pattern for hardware synthesis, our approach separates the concerns of obtaining a "right" program specification and synthesizing efficient hardware implementations from the specification.

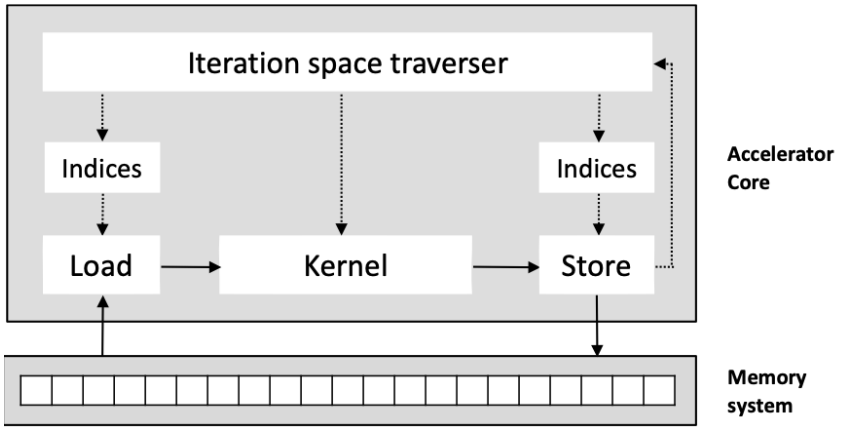



Figure 3.3: A basic load-store architecture.

## 3.3 Specialized Load-store Architectures

By allowing arbitrary access from units of computation to memory, a load-store architecture can be configured to solve general problems. While a processor achieves generality through an arithmetic logic unit and a programmable controller, a specialized load-store architecture must eliminate the high interpretation cost through component customizations. Figure 3.3 shows a basic load-store architecture to implement imperfect loop nest programs. Moreover, the parallel organizations in processors can be applied to specialized load-store architectures for scaling the computational throughput. This section explains the specialization of load-store architecture components and changes it brings to reasoning about efficiency.

### 3.3.1 Computation-specific Datapath

Building a dedicated hardware datapath to solve a specific computational problem is an essential step in hardware specialization. In mapping imperfect loop nests to load-store architectures, the basic block functionality can be implemented as the specialized pipelined datapath. A datapath for a specialized functionality can

be decomposed to cooperated hardware operators which form a directed acyclic graph (DAG). The operators will include data access, arithmetic and logic operators.

The protocols of connecting hardware operators vary depending on operator latencies. When latencies of any operators are statically determined, raw wires can be used to connect I/O ports between nodes of hardware operators in the DAG. When multiple flows of data exist in the DAG, additional flip-flop buffers could be necessary to guarantee consistent arrival time of data signals. The overall latency is then determined by the accumulated latencies along the signal paths.

However, when the latency of any operator is dynamically dependent on data, latency-insensitive protocols [27] such as the elastic circuit [28] can be used to connect hardware operators.

Since there are multiple basic blocks in an imperfect loop nest program, multiple hardware datapaths are required. Constructing a dedicated datapath for each basic block could be expensive in hardware resources when basic blocks are compute-intensive. Consequently, multiplexing the expensive hardware resources for non-overlapped basic blocks in execution is essential [29][30].

The dedicated datapath design for basic blocks in loops naturally generates a memory-memory architecture. By removing named registers out of the load-store architecture, it enables the native concurrency of data accesses and computations and simplifies the control. In the meantime, it is likely to result in a deep pipeline design that requires enough independent iterations to be executed continuously for efficient hardware utilization. This means that a software program with sufficient iterations for an ALU-based pipeline may not provide enough parallelism to the dedicated hardware pipeline. Hence, special consideration must be taken in crafting or selecting "right" loop programs for hardware synthesis.

### 3.3.2 Flexibility-driven Controller

The controller of a customized load-store architecture provides parameters to each iteration of basic blocks executed in the dedicated pipelined datapath, and manages the swapping of basic blocks by monitoring the completions of iteration executions. Such a controller can be implemented differently with respect to the requirement for control flexibility.

A controller for a fixed computation task can be implemented by hardware finite state machines (FSMs) with simple datapath. Figure 3.1b shows a design for controlling a single loop. Controllers for more complicated loops can be built with coordinated FSMs. Note that in traditional hardware compiler generated designs, control signals work at low-level and are required for every clock period for the entire time range of execution [31]. By contrast, a controller of a load-store architecture provides control signals at higher level for each iteration of basic block execution.

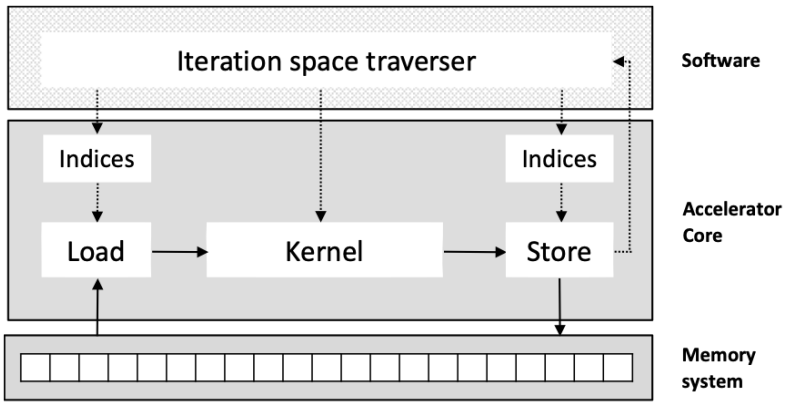
Another popular controller implementation stores the control information into a memory array addressed by a counter. It is possible to reprogram the controller by writing different sequences to the memory. One potential application is building a hardware design that can handle a various problem sizes by reconfiguring the control memory.

Further flexibility can be achieved by allowing software programs to produce control signals. It is possible because we decouple the controller and the datapath in the load-store architecture. This results in *partially hardened* designs. The emerging heterogeneous platforms with general purpose cores and specialized cores decoupled closely [32][33] offer an opportunity for such an implementation. In this use case, the overall design can be partitioned to the regular parallelism component and the irregular and frequently changed component for hardware and software implemen-

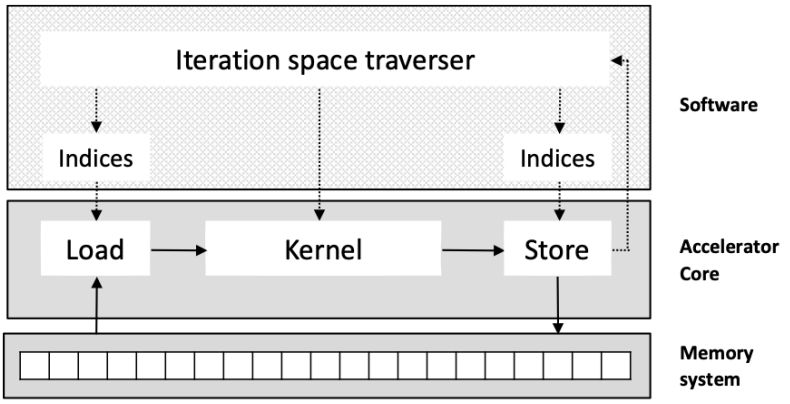
tation, respectively. The repetitive heavy arithmetic computations and bit-level manipulations [34][35] can overwhelm the fixed ALUs of processors, so is promising for hardware acceleration. In contrast, the less-frequently invoked functions could be potentially implemented as software and executed on the general purpose cores without slowing down the hardware components. The software components can additionally be reconfigured flexibly with much lower cost than the specialized hardware pieces to accommodate to the change of applications.

The potential benefits of partial hardening in mapping imperfect loop nests to load-store architectures are shown in three scenarios. First, not every component is executed the same speed in the design. As shown in Figure **??**, the load/store unit and the kernel datapath determines the throughput upper bound of the design. Because in our paradigm the kernel processes vectors, it only requires a new result from the space traverser for every few cycles, depending on the vector length to be processed. Hence, it does not harm the performance if these space are traversed in slower software with large enough vector size. Second, implementing the space traverser and indices calculations as software provides runtime configurability to the design. For example, an accelerator that solves arbitrary sizes of the same problem can be natively built with a soft iteration space traverser and the indices calculator. Third, softening the space traverser and provide the memory interface from the platform can minimize the states in the customized hardware, which facilitates the time-sharing execution environment in modern computers.

Different scenarios can require different modes of partial hardening in our framework. We show two examples in Figure 3.4. In the figures, we use dotted blocks to contain components implemented as software, and grey blocks to contain hardened components. Figure 3.4a shows a design that implement the space tra-




(a) Implementing the traverser in software.




(b) Implementing the traverser and indices in software.

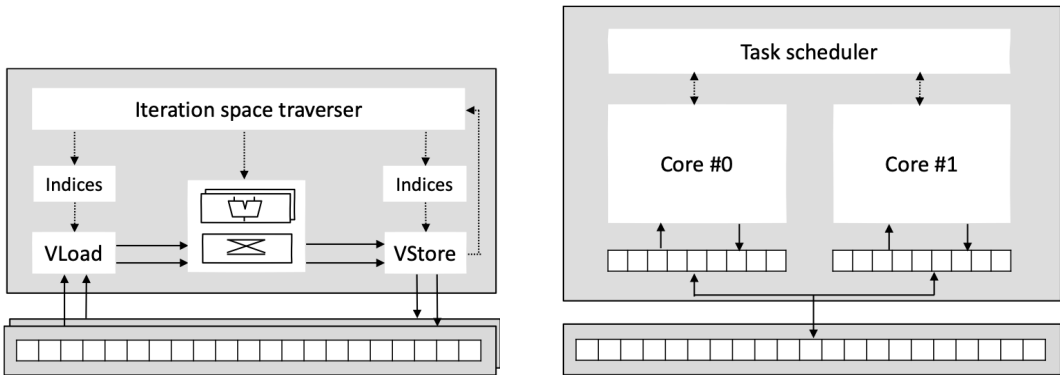
Figure 3.4: Possible partial hardening solutions.

verser in software with rest into hardware. Figure 3.4b describes a softer design that also calculates indices in software.

### 3.3.3 Throughput-driven Parallelism

Existing ideas of parallel architectures for general processors can be applied to our customized load-store architecture to scale the processing throughput. Our baseline architecture in Figure 3.3 resembles a vector processor like Cray-1 [36] because it can process one data item per cycle. For higher throughput, the baseline architecture can equip a vectorized load/store unit along with a SIMD vectorized kernel datapath to achieve multiples of the baseline throughput with mostly the same control, as shown in Figure 3.5a. The vectorized accelerator core must connect to multiple memory banks. The data shuffle circuit may be added to the vectorized kernel to enable local communications between the vector lanes. Another form of throughput scaling way is symmetric multi-processing (SMP) or shared-memory, as shown in Figure 3.5b. In this form, we duplicate the baseline accelerator core with local memory, and coordinates the multiple cores using a task scheduler. This form can exploit the enormous local memories for a large aggregated throughput. The vector and SMP can combine for a better performance efficiency in hardware. Though not presented graphically, the distributed memory architecture can also be applied potentially to accommodate high communication cost between cores when the cores reside in different chips.

Moreover, the load-store architecture design can serve as nodes in high throughput architectures like streaming or systolic array. The load-store architecture is powerful enough for delivering highly complicated computations to nodes of simple parallel organizations such that high throughput can be achieved for compli-




(a) A SIMD vectorized architecture. (b) A multicore architecture.

Figure 3.5: Load-store architecture parallelization.

cated computations.

## 3.4 The Challenges in Hardware Generation

In translating complicated imperfect loop nest programs to efficient load-store architecture implementations, one must have a systematic way to handle the complexity in the algorithm space and the hardware space. Moreover, the interpretation of loop nest programs shall go beyond the software programming syntax and support embedding extra information for more efficient hardware generation. This section explains three major challenges in hardware generation and how they can be resolved by extending the SPIRAL approach.

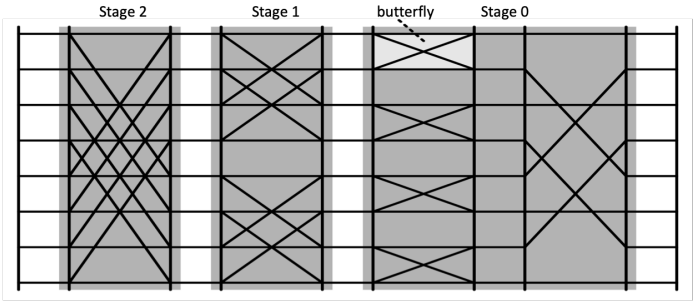
### 3.4.1 Program Generation

The creation of imperfect loop nest programs with desired properties for load-store architecture generation is non-trivial, as has been seen in previous intellectual efforts [24][26] in high performance computing. Moreover, the algorithm space of a given computational problem can be a large tradeoff space between computational

complexity, parallelism, memory access pattern, memory utilization, regularity, etc. It is possible that two algorithms conforming to our computational paradigm of the same problem result in distinct performance and resource utilization.

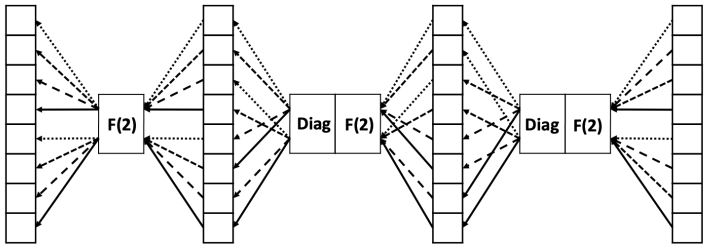
Characterizing existing algorithms and discovering new algorithms is a research field that requires profound domain knowledge, exceeding the scope of this dissertation. Fortunately, SPIRAL has addressed plentiful algorithms, through an architecture-aware algorithm generation process explained in Section 2.2. In the past, SPIRAL has addressed linear transforms, numerical linear algebra operations like the matrix-matrix multiply [6], polynomial evaluation, infinity norm, geofencing for unmanned aerial vehicles [37], the Viterbi decoder [7], polar formatting synthetic aperture radar [5], Euler integration, statistical $z$-test, wavelet transforms and JPEG2000 image compression [38], among many others. SPIRAL captures algorithms in data flow graphs using the OL formalism, which are then automatically translated to loops in $\Sigma$-OL, transformed through a powerful loop merging process (explained in Section 2.3.3) which produces imperfect loop nest programs with desired properties listed in Section 3.2. As an example, Figure 3.6 shows an 8-point FFT algorithm generated by SPIRAL and the computations captured in an imperfect loop nest using the $\Sigma$-OL language.

However, algorithms for high performance implementations on specialized load-store architectures could be different from those on general processors. A scenario of deep pipelines that demand more parallelism has been discussed in Section 3.3.1. Moreover, for parallel implementations, the hardware flexibility provides an opportunity for algorithm-hardware co-synthesis, instead of uni-directionally transforming algorithms for fixed parallel organizations. This means that hardware adaptation in the architecture-aware algorithm generation process is important to ob-




$$\Big(\mathrm{DFT}_2 \otimes \mathrm{I}_4\Big)\Big(\mathrm{T}_4^8\,(\mathrm{I}_2 \otimes (\mathrm{DFT}_2 \otimes \mathrm{I}_2))\Big)\Big((\mathrm{I}_2 \otimes (\mathrm{T}_2^4 (\mathrm{I}_2 \otimes \mathrm{DFT}_2)\,\mathrm{L}_2^4)\,\mathrm{L}_2^8\Big)$$

(a) A SPIRAL-generated FFT(8) algorithm represented in a data flow graph with the corresponding OL formula.




$$\Big(\sum_{i1=0}^{4} \mathrm{S}_{h_1}\,\mathrm{F}_2\,\mathrm{G}_{h_2}\Big)\Big(\sum_{i2=0}^{2}\sum_{i3=0}^{2} \mathrm{S}_{h_3}\,\mathrm{D}_{f_1}\,\mathrm{F}_2\,\mathrm{G}_{h_4}\Big)\Big(\sum_{i4}^{2}\sum_{i5}^{2} \mathrm{S}_{h_5}\,\mathrm{D}_{f_2}\,\mathrm{F}_2\,\mathrm{G}_{h_6}\Big)$$

(b) Iterative computing FFT(8) on memory and the corresponding Σ-OL expression.

Figure 3.6: Translating a SPIRAL-generated FFT algorithm to imperfect loop nest.

taining efficient hardware implementations.

### 3.4.2 Program Optimization

Due to different assumptions between general processors and customized load-store architectures, the structured imperfect loop nests need to be furthered optimized for hardware implementations, regardless of the abundant parallelism and regularity. The imperfect loop nests generated from SPIRAL are represented in Σ-OL introduced in Section 2.3.2. In this section, we discuss three optimizations for hardware generation that can be achieved by extending the Σ-OL representation.

### Efficient Buffer Allocation

In the Σ-OL representation, the read/write buffers of basic blocks are implicit. In the software generation flow of SPIRAL, the buffers are allocated when translating Σ-OL expressions to icode. The current buffer allocation scheme is designed based on a common assumption that main memory is cheap and the data transfer between on-chip and off-chip is managed by the cache sub-system automatically. Hence, it is reasonable that the scheme does not minimize buffer utilization. However, in hardware generation, the local memories are scarce resources, and the data transfer between on-chip and off-chip is performed explicitly. When processing on-chip data, the total buffer sizes makes a strong impact on resource utilization.

The basic idea of the buffer allocation scheme in current SPIRAL is allocating separated intermediate buffers between the compute stages. The overall program is specified an input buffer and an output buffer. When the computation is composed of stages, intermediate buffers are allocated. The intermediate buffer serves as the write buffer of the leading stage and the read buffer of the trailing stage. In this way, the number of intermediate buffers is the number of stages minus one. SPIRAL exploits in-place compute stages for reducing the intermediate buffers because in this case the read buffer can be safely reused as the write buffer. In an extreme case when all stages are in-place, the output buffer can replace the intermediate buffers and no extra intermediate buffers are required.

The buffer allocation scheme is demonstrated in pseudo code examples in Figure 3.7. In the pseudo code, the compute stages are listed vertically with respect to the execution order. Each stage is described with two lines: the first line identifies the stage with a name; the second line is indented and describes the read buffer and write buffer of the current stage, divided by a right arrow. An arbitrary compute

stage is named Compute_<Id> where <Id> is a natural number. The in-place stage must be a loop, and is named as Loop_inplace_<Id>. The input buffer to the sub-program is named $X$ and the output buffer is named $Y$. The newly allocated intermediate buffers are named as $T<Id>$.

On the left, Figure 3.7a demonstrates the buffer allocation result for four consecutive compute stages. At the first loop, an intermediate buffer *T0* is allocated as the write buffer for this stage and the read buffer for the next stage. Additional intermediate buffers are allocated at each loop except for *Loop_inplace_2* whose read buffer can be safely reused as the write buffer. The final stage Compute_3 writes the results to buffer $Y$.

On the right, Figure 3.7b shows an unusual situation with all in-place stages such that buffer $Y$ can serve as the write buffer for every stage, thus avoiding allocating dedicated intermediate buffers.

```
Compute_0
    X -> T0
Compute_1
    T0 -> T1
Loop_in_place_2
    T1 -> T1
Compute_3
    T1 -> Y
```

(a) Introducing intermediate buffers for non-in-place stages.

```
Loop_in_place_0
    X -> Y
Loop_in_place_1
    Y -> Y
Loop_in_place_2
    Y -> Y
Loop_in_place_3
    Y -> Y
```

(b) Reusing the output buffer for intensive in-place stages.

Figure 3.7: Allocating intermediate buffers in SPIRAL software generation.

By implementing the above scheme in a recursive-descent translator, allocating buffers for nested compositions is automated by using the read (write) buffer of

the parent stage as the input (output) buffer. Figure 3.8 presents a pseudo code example. In the pseudo code, the indentation depth denotes the depth of code blocks in the imperfectly nested loop program. The compute stage of compositions is denoted as Compose_<Id>, where the read/write buffer description is replaced by a deeper indented code block. In the example, the sub-program is composed of two stages: Compose_0 and Compose_1, using *T1* as the intermediate buffer. Compose_0 is composed of two stages: Compute_2 and Compute_3, using *T2* as the intermediate buffer. Compose_1 is composed of two stages: Compute_4 and Compute_5, using *T3* as the intermediate buffer.

```
Compose_0
  Compute_2
    X -> T2
  Compute_3
    T2 -> T1
Compose_1
  Compute_4
    T1 -> T3
  Compute_5
    T3 -> Y
```

Figure 3.8: Allocating intermediate buffers in SPIRAL software generation.

The reduction of buffers can be addressed in two aspects and the benefit may vary with different programs.

First, because the deep level buffers of different stages at the outer composition level do not overlap in execution, as is the *T2* and *T3* shown in Figure 3.8, they can be mapped to the same physical memory. This aspect does not require particular properties of the program, but the savings of buffers depends on the dis-

tribution of intermediate buffers at different stages. Those programs with balanced distribution of intermediate buffers along the compute stages benefit the most from this technique. In another extreme where the intermediate buffer used in one stage vastly overweigh the intermediate buffers of other stages, the benefit is negligible.

Second, the series of allocated intermediate buffers and the input and output buffers at the same composition level, as shown in Figure 3.7a, can be possibly replaced by swapping two buffers because the stage series are executed sequentially. However, the swapping strategy does not always help. A counter-example is that one extremely large intermediate buffer will call for the duplication of such a large buffer that could be possibly larger than the aggregated size of the rest of buffers. Hence, an effective buffer allocation scheme should consider the properties of the program.

In implementing an efficient buffer allocation scheme, the program properties required for the scheme can be addressed conveniently in Σ-OL. In addition, the language can be extended to captured the program structure of interest in the buffer allocation process. In Section 4.2.1, an optimized buffer allocation scheme based on the Σ-OL framework is presented.

#### Simplified Indices Calculation

There are some types of indices that could be computed with cheaper operations inductively [39]. Table 3.1 collects three examples from real applications, where the expensive multiplication, exponentiation and modulo operations can be lowered to addition, multiplication and counter operations, respectively. However, when performing these transformations in software loop code, newly introduced inductive computations are inserted between the nested loops, which can destroy the perfect

Table 3.1: Indices that can be simplified with inductive calculation.

| Loop variable | Index function | Inductive calculation | Application |
|---|---|---|---|
| | i * c0 | i=0; i = i + c0 | Cooley-Tukey FFT |
| i | $i^{c1}$ | i=1; i = i * c1 | Rader FFT [2] |
| | i mod c2 | i = repeat [0,..,c2-1] | Good-Thomas FFT [2] |

sub-nest structure. By extending the loop representation of Σ-OL, we can embed these computations into loops so that the perfect sub-nest structure is preserved to conform to the proposed imperfect loop nest paradigm.

#### Continuous Pipelining

When the data dependencies between adjacent perfect sub-nests are unknown, the shared datapath pipeline has to be drained before starting the execution of the trailing sub-nest to avoid data hazards. The dependency can be typically resolved either dynamically in execution time with expensive circuits [40] or statically through program analysis. The Σ-OL representation provides a powerful rule-based static analysis scheme that allows analyzing the symbolic index mapping functions to figure out the dependencies between sub-nests so that the trailing sub-nest can start execution earlier. The speedup upper bound of this optimization depends on the ratio of the pipeline latency over the iteration counts of the perfect sub-nest, arriving at the peak of 2x when the ratio equals to one.

### 3.4.3 Hardware Manipulation

To natively describe the spatial hardware designs without entering into the RTL abstraction level, we adopt icode to model the RTL modules and the connections between them to form a complete hardware design.

**RTL modules.** The RTL modules can be *hardware operators*, finite state

machines (FSMs), memory blocks, etc. A hardware operator manipulates the input data to produce the output data in the form of digital signals. FSMs and other RTL modules can be defined with arbitrary interface for flexible hardware interpretation.

The icode representation of each RTL module serves as the functional specification to an RTL code generator. The hardware operators and other RTL module are modeled differently.

RTL modules other than hardware operators are modeled as icode types, specified by its name and possibly some parameters, which is mapped to an RTL template by the generator. The RTL module types can be instantiated with a special icode command and assigned to a variable for interconnection.

A hardware operator is modeled in icode similar to the existing software operators, specified by a name, one or multiple input ports with type information. Any hardware operators always have one output, whose type is derived from the input types. By extending the types in icode, a hardware operator can manipulate digital signals representing distinct data types from integer to floating point, real to complex, scalar to vector, raw data to decoupled types[1].

The RTL generation of modules can be arbitrarily complex, possibly with extra specifications. Logic simplification is possible with the functional specification. Figure 3.9 shows an example where a multiplexor implementation can be simplified if identical bits exist in all inputs. The timing specification could improve the circuit efficiency of RTL designs. The hardware platform information may be essential for efficient resource binding especially in FPGAs, where the macro DSP blocks and memories are necessary for higher hardware efficiency. The mature hardware compilation methods such as the constrained scheduling [41] and resource binding

[1] A decoupled type encapsulates the data payload with valid and ready signals, which is a common protocol to build latency-insensitive hardware.

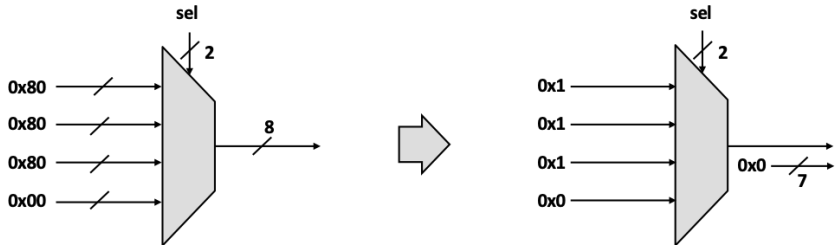



Figure 3.9: Simplify the multiplexor logic with identical bits

of expressions can be employed in our RTL generator. Existing hardware IP blocks can be encapsulated as hardware operators or other RTL modules in our framework.

**Module connections.** We use icode commands to model the connections between RTL modules. The existing assign command in icode connects the output ports of the automatically instantiated hardware operator module to the assigned hardware variable that models an RTL wire. Hence, the operator graph representing by a list of assign commands can be translated to the interconnected hardware operator modules in RTL. Other RTL modules are firstly instantiated with an explicit command and then assigned to a hardware variable, with which the connection can be modeled by a special connection command. The I/O ports to be connected are encoded in the connection command.

## 3.5 Summary

This chapter introduces a high-level idea of translating imperfect loop nest programs to customized load-store architecture designs. The algorithmic flexibility of the proposed hardware generation approach is assured by the expressibility of imperfect loop nest programs and the implementation flexibility of load-store architectures. The key challenge is how to achieve hardware efficiency through component customization of the load-store architecture. We have addressed problems of program

generation, optimization and hardware manipulation, which can be solved by extending the SPIRAL approach.

Given the high level idea of flexible hardware generation, the next chapter will explain how to systematically extend SPIRAL for synthesizing scalar load-store architectures.

# Chapter 4

# Extending Spiral for Generating Specialized Load-store Architectures

The previous chapter has presented a large design space of algorithms and hardware in load-store architecture synthesis. To show a systematic method of extending Spiral for load-store architecture generation, this chapter will focus on a more restricted compute pattern of imperfect loop nest programs and the scalar load-store architecture.

## 4.1 The Multi-linear Paradigm and the Scalar Load-store Architecture

In this work, we introduce a computational paradigm that supports some important and classic algorithms of the Spiral framework. The imperfect loop nest programs

conforming to this paradigm are mapped to an elementary fully-hardened load-store architecture that at most loads and stores one data entries from the memory interface.

**The multi-linear paradigm.** This section imposes additional property requirements for the general computational paradigm of imperfect loop nest programs in the iteration space, the indices patterns, and the kernel operations:

1. All loop bounds of the loop nest are constant integers.

2. In the basic blocks, the memory gather and scatter operations are directed by multi-linear functions of loop variables.

3. Within every basic blocks, the kernel operations, potentially parameterized, are identical.

The above paradigm is called the *multi-linear paradigm* in this thesis. It captures a lot of algorithms that orchestrate computations as iterative data manipulations on a high-dimensional data cube. In the next section, we will show how to generate algorithms conforming to this paradigm for three fundamental computational problems.

**The scalar load-store architecture.** This work targets an elementary load-store architecture that is fully hardened and is only able to load/store at most one memory data word at the steady state. We refer to this design the *scalar load-store architecture*.

The mapping from the multi-linear paradigm to the scalar load-store architecture is depicted in Figure 4.1. The iteration space traversal is performed in a hardened loop controller. An execution pipelined is implemented to load data from memory, perform the kernel operation, and store the results back to memory. The

kernel operations of different basic blocks are the same, as required by the multi-linear paradigm. The parameters for each basic block is provided from the loop controller to the pipeline components for functional correctness. According to the paradigm, for each iteration, a data vector is gather from the memory with multiple scalar reads, processed with the kernel implementations, then finally scattered back to the memory with multiple scalar writes.

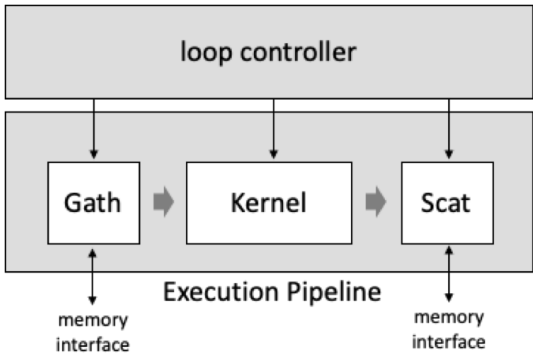



Figure 4.1: Implement on a scalar load-store architecture.

## 4.2 Optimizing Programs of the Multi-linear Paradigm

This section discusses the optimization opportunities in mapping the multi-linear paradigm of imperfect loop nest programs to the load-store architecture. The optimizations exploit the properties of the multi-linear paradigm for reduced resource utilization or improved performance. Three optimizations based on static analysis are discussed in the following subsections, with increasing requirements for the programs.

### 4.2.1 Efficient Buffer Allocation

In an imperfectly nested loop, the computation is divided into several stages, thus intermediate buffers are required for data exchange between stages.

When mapping an imperfect loop nest program to hardware designs on a load-store architecture, each buffer occupies a certain number of entries in the memory. Thus, reducing the total buffer entries can reduce memory utilization in hardware.

In the multi-linear paradigm, the gather and scatter operations of every basic blocks have a static number of entries to load and store on a fixed size data set. As a result, the buffers required for the computations can be analyzed statically.

#### Reusing Deep Level Buffers

In the multi-linear paradigm, the buffer sizes allocated at the deep levels of each stage of shallow level can be determined statically. Since the different stages of shallow level do not overlap in execution, the intermediate buffers used in deep levels can be shared by other stages of the shallow level. Figure 4.2 compares the allocation schemes in multi-level compositions between the SPIRAL software generator and the proposed hardware generator. On the left side, Figure 4.2a replicates Figure 3.7b for the current SPIRAL. On the right side, Figure 4.2b shows the allocation scheme for hardware generation, where the buffer *T2* used in the first level-2 composition is reused in the second level-2 composition, removing the additional *T3* in Figure 4.2a. Practically, *T2* and *T3* may not be the same sizes, then the larger size represents the required buffer size in the deep levels. This strategy stays valid when more than two levels are involved.

```
Compute_0
  Compute_2
    X -> T2
  Compute_3
    T2 -> T1
Compute_1
  Compute_4
    T1 -> T3
  Compute_5
    T3 -> Y
```

(a) Always allocating new buffers at deep levels.

```
Compute_0
  Compute_2
    X -> T2
  Compute_3
    T2 -> T1
Compute_1
  Compute_4
    T1 -> T2
  Compute_5
    T2 -> Y
```

(b) Reusing deep level intermediate buffers.

Figure 4.2: SPIRAL software versus hardware: Allocating intermediate buffers at deep level composition.

### Swapping Buffers of Uniform Size

For allocating buffers within a composition level for reduced buffer utilization, the buffer swapping idea mentioned in 3.4.2 is studied in this section by considering particular program properties with uniform buffer sizes in the intermediate, output, and input buffers. For programs with these properties, swapping can assure fewer buffer utilization compared to the default strategy of SPIRAL software generator as discussed in Section 3.4.2.

When the input, intermediate and output buffers are of the same size, the swapping can be performed between the input buffer and the output buffer, without additional intermediate buffers. This means that the initial data in the input buffer will be overwritten. Such behavior could be potentially prohibited by the application in case the integrity of the original input data is required. In this case, the in-place

trick used in software SPIRAL can help avoid swapping and input buffer overwritten when all stages except the first one are in-place. Nevertheless, in deeply nested loop programs, the input buffer of a composition is mostly an intermediate buffer of the previous composition level that can be safely overwritten. The two situations with or without input buffer overwritten are presented in Figure 4.3.


(a) Swapping between the input and output buffer.

(b) Avoid swapping and input buffer overwritten with in-place loops.

Figure 4.3: Swapping the input/output buffers when input/output/intermediate buffers are of uniform size.

Though the uniform size of input/intermediate/output buffers avoids the intermediate buffers completely, such a constraint is restricted. By loosening the constraints, the lower bound of intermediate buffer utilization can be guaranteed to be one or two.

When only the intermediate and the output buffers are of the same size, the number of intermediate buffers required for swapping can be performed between one intermediate buffer and the output buffer, except a case that the output buffer can not be the final buffer without the second intermediate buffer.

```
Compute_0
    X -> Y
Loop_in_place_1
    Y -> Y
Loop_in_place_2
    Y -> Y
```

(a) All in-place intermediate stages.

```
Compute_0
    X -> T
Compute_1
    T -> Y
Compute_2
    Y -> T
Compute_3
    T -> Y
```

(b) Even number of stages.

```
Compute_0
    X -> T
Loop_in_place_1
    T -> T
Compute_2
    T -> Y
```

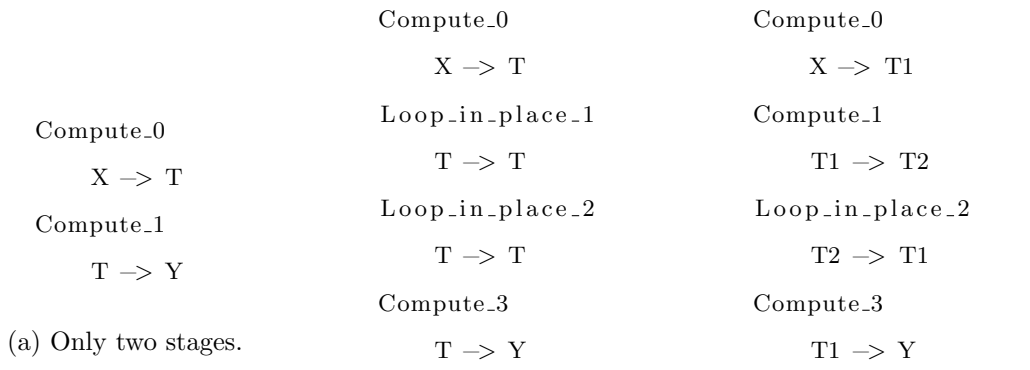
(c) Odd number of stages with in-place stage(s).

```
Compute_0
    X -> T1
Compute_1
    T1 -> T2
Compute_2
    T2 -> Y
```

(d) Odd number of stages without in-place stage(s).

Figure 4.4: Allocating buffers when output/intermediate buffers are the same size.

When only the intermediate buffers are of the same size, a single intermediate buffer is sufficient when there are only two stages of computations (shown in 4.5a), or when all intermediate stages are in-place (shown in 4.5b). Otherwise, the swapping between two intermediate buffers are necessary, as shown in 4.5c




(a) Only two stages.

(b) All in-place intermediate stages.

(c) Exist non-in-place stage(s).

Figure 4.5: Allocating buffers when only intermediate buffers are the same size.

Since there is only one memory address space in the load-store architecture, when mapping the multi-linear paradigm programs to the load-store architecture, the allocation of buffers must be realized with address offsets for each gather and scatter operation. In the existing Σ-OL formalism, buffer allocation is implicit to the loop nest and thus will be captured explicitly with an extension. The next section will introduce the OL extensions that makes the address offset explicit.

### 4.2.2 Simplified Calculations of Multi-linear Functions

The multi-linear paradigm utilizes the multi-linear expressions of loop variables in indexing memory entries for loading and storing, whose computation can be potentially simplified with common compiler optimization techniques. However, one drawback of a common optimization is breaking the loop nest structure. Thus the simplifications need to be handled specifically.

A multi-linear expression can be captured by a function

$$f(j_0, j_1, \ldots j_{n-1}) = c_0 j_0 + c_1 j_1 \cdots + c_{n-1} j_{n-1},$$

where $c_k, k \in [0, n-1]$ are natural numbers and $j_k, k \in [0, n-1]$ are all loop variables.

A direct computation of a size-$n$ multi-linear expression requires $n$ multiplications and $n-1$ additions. The computational cost could be reduced with two well-known techniques: inductive computation and bit manipulation. The inductive computation is a general technique that lowers the sums of multiplications of loop variables to accumulations given that the loop variable values increase by one at each iteration. The bit operation requires certain properties of the constant factors but provides a cheaper solution. It converts multiplications to bit shifts when $c_k$ is a power-of-2 number. If all $c_k$ are power-of-2 numbers and each operand of the summation does not overlap in the bit fields of the final sum, the whole computation can be implemented with bit shifts and bit-wise-OR operation.

Figure 4.6 shows the three approaches in calculating two size-2 multi-linear expressions in an imperfectly nested loop program. The computational costs decrease from left to right. The bit manipulation approach requires zero arithmetic cost. In inductive computation, a variable is initialized before starting each loop execution and its value is increased by a constant value. In bit manipulation, this example shows an ideal situation where all constant factors are power-of-2 numbers. Thus the whole multi-linear expressions are replaced by bit shifts and bit-wise-OR operations. Otherwise, the bit manipulation technique can be only partially applied.

```
for (i=0;i<4;i++)
  for (j=0;j<2;j++)
    idx1=2*i+j;
    ...
    ...
    ...
    ...
  for (k=0;k<2;k++)
    idx2=i+4*k;
    ...
    ...
    ...
    ...
```

(a) Direct computation.

```
idx1_tmp=0;
idx2_tmp=0;
for (i=0;i<4;i++)
  idx1=idx1_tmp;
  for (j=0;j<2;j++)
    ...
    idx1+=1;
  idx2=idx2_tmp;
  for (k=0;k<2;k++)
    ...
    idx2+=4;
  idx1_tmp+=2;
  idx2_tmp+=1;
```

(b) Inductive computation.

```
for (i=0;i<4;i++)
  for (j=0;j<2;j++)
    idx=i<<1 | j;
    ...
    ...
    ...
    ...
  for (k=0;k<2;k++)
    idx=i | k<<2;
    ...
    ...
    ...
    ...
```

(c) Bit manipulation.

Figure 4.6: Three approaches for computing multi-linear expressions.

The challenge in simplifying the computation of multi-linear expressions is that the inductive calculation expressions inserted between the loop nest makes the program more imperfect, reverse the effort of the multi-linear paradigm in internalizing the computations to the inner most loops for easier pipelined implementations. The next section will explain the Σ-OL extension that enables computation embedding in loops.

### 4.2.3 Overlapped Execution across Perfect Sub-nests

The multi-linear paradigm imposes identical kernel operations between basic blocks, which shapes a particular access pattern amenable to static dependency analysis between the perfect sub-nest. The result can enable the overlapped execution across the boundary of perfect sub-nests for lower execution latency while the hardware

pipeline is implemented properly.

### Dependency Analysis

We analyze the dependencies between two neighboring perfect sub-nests that access the data entries linearly in constant size vectors with different fixed strides for read and write, respectively.

Table 4.1 shows such an access pattern of stride 1, 2, and 4 for size-8 buffer and size-2 kernel. Each row represents an iteration. At each iteration, two data entries are accessed. For each stride, the left columns list the values of the relevant loop variables. The right columns list the values of two strided indices with the corresponding multi-linear index functions described in the heading row. As we can see, for each stride, every entry of the size-8 buffer is accessed exactly once. Stride-4 accesses the entry pair serially with the largest possible stride. Stride-2 partitions all data entries into two groups evenly and within each group accesses each pair with stride of 2 serially, as shown by the dashed line in the table. Stride-1 partitions entries into four groups evenly and within each group accesses each pair with unit stride. When comparing the access patterns between unit stride and stride-2, the access orders for each half data entries are equivalent.

Table 4.1: The indices for accessing a size-2 vector from a size-8 buffer with various strides.

| Iteration | unit stride | | | stride-2 | | | | stride-4 | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | var | indices | | var | | indices | | var | indices | |
| | $i$ | $2i$ | $2i+1$ | $i$ | $j$ | $4i+j$ | $4i+j+2$ | i | $i$ | $i+4$ |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 2 | 0 | 0 | 4 |
| 1 | 1 | 2 | 3 | 0 | 1 | 1 | 3 | 1 | 1 | 5 |
| 2 | 2 | 4 | 5 | 1 | 0 | 4 | 6 | 2 | 2 | 6 |
| 3 | 3 | 6 | 7 | 1 | 1 | 5 | 7 | 3 | 3 | 7 |

To analyze the dependencies using the access pattern, we assume a general buffer size $n$ and kernel size $k$. The maximal stride is $n/k$ and the mimimal stride is 1. For any strides $s$, the $n$ data entries are partitioned into $\frac{n}{ks}$ even sub-groups. Given the write stride $s_w$ and the read stride $s_r$, analyzing the dependencies on the leading group of $k * \max(s_w, s_r)$ entries is sufficient because the trailing groups follow the same pattern.

We set $s_l$ and $s_s$ as the larger and smaller stride between $s_w$ and $s_r$, respectively. The indices for the leading sub-group of the larger stride is shown in Table 4.2. For the same iteration, the values increase by $s_l$ for each of the next location. For the same location of the indices, the values increase by 1 for each of the next iteration.

Table 4.2: The indices for larger stride.

| Iteration | 1st idx | $\cdots$ | k-th idx |
|---|---|---|---|
| 0 | 0 | $\cdots$ | $(k-1)s_l$ |
| $\ldots$ | $\cdots$ | $\cdots$ | $\cdots$ |
| $\ldots$ | $\cdots$ | $\cdots$ | $\cdots$ |
| $s_l - 1$ | $s_l - 1$ | $\cdots$ | $ks_l - 1$ |

The indices of the smaller stride covering the sub-group of the larger stride is shown in Table 4.3. It requires multiple sub-groups of the smaller stride to cover a sub-group of the larger stride, as described by the dashed lines in the table. The parameter $d$ in the table specifies the d-th sub-group, ranging from 0 to $\frac{n}{s_s k} - 1$. For the same iteration, the values increase by $s_s$ for each of the next location. For the same location of the indices, the values increase by 1 for each of the next iteration within the sub-group and increase by $ks_s$ in the sub-group granularity.

Assuming an execution pipeline that can overlap the iterations between the neighboring perfect sub-nests. We define a write iteration $T$, ranging from 0 to

Table 4.3: The indices for smaller stride.

| Iteration | 1st idx | $\cdots$ | k-th idx |
|---|---|---|---|
| 0 | 0 | $\cdots$ | $(k-1)s_s$ |
| $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ |
| $s_s-1$ | $s_s-1$ | $\cdots$ | $ks_s-1$ |
| $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ |
| $ds_s+0$ | $kds_s+0$ | $\cdots$ | $\cdots$ |
| $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ |
| $(d+1)s_s-1$ | $\cdots$ | $\cdots$ | $k((d+1)s_s-1)+k-1$ |
| $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ |

$s_l-1$, whose completion can enable the read iterations to be executed in the same pipeline without data hazard. One has to guarantee that all the write indices at and after iteration $T$ are larger than the corresponding read iterations.

Table 4.4: A schedule for small write stride and large read stride.

| Iteration | | Write indices | | | Read indices | | |
|---|---|---|---|---|---|---|---|
| write | read | 1st | $\cdots$ | k-th | 1st | $\cdots$ | k-th |
| 0 | | 0 | $\cdots$ | $(k$-1$)s_s$ | | | |
| $\cdots$ | | $\cdots$ | $\cdots$ | $\cdots$ | | | |
| $s_s$-1 | | $s_s$-1 | $\cdots$ | $ks_s$-1 | | | |
| $\cdots$ | | $\cdots$ | $\cdots$ | $\cdots$ | | | |
| $ds_s$+0 | 0 | $kds_s+0$ | $\cdots$ | $\cdots$ | 0 | $\cdots$ | $(k$-1$)s_l$ |
| $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ |
| $(d$+1$)s_s$-1 | $\cdots$ | $\cdots$ | $\cdots$ | $k((d$+1$)s_s$-1$)+k$-1 | $\cdots$ | $\cdots$ | $\cdots$ |
| $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ |
| | $s_l$-1 | | | | $s_l$-1 | $\cdots$ | $ks_l$-1 |

When the write stride is smaller than the read stride, i.e., $s_w = s_s$ $s_r = s_l$, a schedule is shown in Figure 4.4 where each row represents a scheduling step. On the completion of the write iteration $ds_s$ that fills the beginning indices of a sub-group, the read iterations start to execute. As long as all write indices at write iteration $ds_s$ are larger than all read indices at read iteration 0, the same property can be

guarantee for the trailing iterations because the index values of the smaller stride increase at the same speed or faster than the index values of the larger stride. The condition can be captured by a formula

$$\begin{cases} T = ds_s \\ kds_s > (k-1)s_l \end{cases}. \tag{4.1}$$

Solving the inequality gives us a result about $T$,

$$T > \frac{(k-1)}{k} s_l. \tag{4.2}$$

Table 4.5: A schedule for large write stride and small read stride.

| Iteration | | Write indices | | | Read indices | | |
|---|---|---|---|---|---|---|---|
| write | read | 1st | $\cdots$ | k-th | 1st | $\cdots$ | k-th |
| 0 | | 0 | $\cdots$ | $(k\text{-}1)s_l$ | | | |
| $\cdots$ | | $\cdots$ | $\cdots$ | $\cdots$ | | | |
| $\cdots$ | 0 | $\cdots$ | $\cdots$ | $\cdots$ | 0 | $\cdots$ | $(k\text{-}1)s_s$ |
| $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ |
| $\cdots$ | $s_s\text{-}1$ | $\cdots$ | $\cdots$ | $\cdots$ | $s_s\text{-}1$ | $\cdots$ | $ks_s\text{-}1$ |
| $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ |
| $\cdots$ | $ds_s\text{+}0$ | $\cdots$ | $\cdots$ | $\cdots$ | $kds_s + 0$ | $\cdots$ | $\cdots$ |
| $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ |
| $s_l\text{-}1$ | $(d\text{+}1)s_s\text{-}1$ | $s_l\text{-}1$ | $\cdots$ | $ks_l\text{-}1$ | $\cdots$ | $\cdots$ | $k((d\text{+}1)s_s\text{-}1)\text{+}k\text{-}1$ |
| | $\cdots$ | | | | $\cdots$ | $\cdots$ | $\cdots$ |

When the write stride is larger than the read stride, i.e., $s_w = s_l$ $s_r = s_s$, a schedule is shown in Figure 4.5. On the completion of the write iteration $T$, the read iterations start to execute such that all write indices of the final write iteration are larger than all read indices of the corresponding read iteration $s_l - 1 - T$. This guarantees the same property for the previous corresponding iterations because the read indices decrease at the same speed or faster when reverse-counting the

iterations. The condition can be captured by a formula

$$s_l - 1 > k(s_l - 1 - T) + k - 1 \tag{4.3}$$

Solving the inequality gives us a result about $T$,

$$T > \frac{(k-1)}{k} s_l. \tag{4.4}$$

The results for both situations appear to be the same. Because $T$ is an integer, the results indicate that when the conditions are met, as long as the $\frac{(k-1)}{k} s_l +$ 1-th write iteration completes, it is safe to start the execution of the next perfect sub-nest in the same pipeline without data hazard.

**Hardware Requirements**

The overlapped execution across perfect sub-nest involves the switch of basic blocks in the hardware pipeline. Though the multi-linear paradigm requires identical kernel operations between basic blocks, the hardware implementations of gather, kernel and scatter operations for each basic still vary when they are parameterized differently. The common implementation schemes for resource sharing between basic blocks employ multiplexors that only allow one state of the implementations, and thus prohibits overlapped execution across basic blocks. To remove the restriction in the load-store architecture, the multiplexors can be migrated from the execution pipeline to the loop controller, such that the execution pipeline does not require reconfiguration in basic block switching to allow iterations from different perfect sub-nests to be overlapped.

**Language Support**

To encode the aforementioned static analysis results in the program, the next section will introduce a Σ-OL language extension for capturing the perfect sub-nests with explicit index functions and the iteration sequence number for synchronization.

## 4.3 Σ-OL Extensions for Program Optimization

To enable the optimizations introduced in the previous section for the multi-linear paradigm of imperfect loop nest programs, the Σ-OL formalism is extended. First, two new Σ-OL constructs are introduced to capture the properties essential to optimizations. Second, several rewrite rules and compiler passes are added for transforming the imperfectly nested loop programs.

### 4.3.1 Hardware Formula Constructs

We introduce two new formula constructs to Σ-OL, with the first modeling loops with embedded computations and the second capturing the perfect sub-nest with the essential parameters.

Loops with embedded computations offloaded from basic blocks can preserve the loop nest structure in the inductive calculation of multi-linear expressions. Because these embedded computations are control-flow-irrelevant, such code motion preserves the functional correctness. We extend the loop symbol $\sum$ by adding the computation of a *companion variable* $v$ initialized to $v_0$ and updated using a function $f(v)$ as shown in the expression

$$\sum_{\substack{i=0 \\ \{v=v_0;v=f(v)\}}}^{N} A$$

for arbitrary operator $A$ iterating through a loop of size $N$ using the iterator variable $i$. It is legal to have zero or multiple companion variables in a single loop.

A perfect sub-nest structure encapsulates an iteration space and a basic block. All its iterations are computed independently with each other. A perfect sub-nest produce data for its trailing perfect sub-nest to consume, thus needs to synchronize with the trailing sub-nest to avoid data hazard. To accommodate with the single address space of the memory system, the memory offsets for gather and scatter need to be provided for each perfect sub-nest. We capture the perfect sub-nest with a parameterized *PerfNest* wrapper

$$\underset{\substack{f_s,f_g\\ \delta,o_s,o_g}}{\text{PerfNest}} \left( \sum_{\substack{i=0\\ \{v=f(i)\}}}^{N} A_i \right)$$

which denotes $f_g$ ($f_s$) the gather (scatter) index functions, $\delta$ the iteration sequence number for synchronizing the trailing sub-nest, and $o_g$ ($o_s$) the gather (scatter) buffer offsets for the basic block operations. At an early stage where the offsets are undetermined, they are denoted as $\emptyset$.

### 4.3.2 Formula-based Program Analysis and Transformation

We develop several rewrite rules and the several syntax tree visitors to improve latency and resource optimization of the input program when translating to hardware designs. For each rewrite rule, Spiral's rewrite system matches the left side of a rule against a given formula and replaces the matched expression by the right-hand side of the rule. A syntax tree visitor can traverse the loop nest program more flexibly to collect broader information for more complicated analyses and transformations.

### Identifying Perfect Sub-nests

The perfect sub-nest of the input program can be detected with two rewrite rules. Rule (4.5) annotates an inner most loop with the perfect sub-nest wrapper. The index functions of gather $G$ and scatter $S$ are copied to the parameter fields. The iteration sequence number for synchronization is initialized conservatively such that the final iteration of the size-$N$ loop must complete before the trailing sub-nest starts computation. The memory offsets are not determined yet, thus are initialized to $\emptyset$. In this rule, the canonical loop is converted to a loop supporting embedded computations, even though with empty companion variables temporarily.

$$\sum_{i=0}^{N} \mathrm{S}_{f_s}\, A_i\, \mathrm{G}_{f_g} \rightarrow \underset{N,\emptyset,\emptyset}{\overset{f_s,f_g}{\mathrm{PerfNest}}} \left( \sum_{\substack{i=0 \\ \{\}}}^{N} \mathrm{S}_{f_s}\, A_i\, \mathrm{G}_{f_g} \right) \tag{4.5}$$

Rule (4.6) moves the outer loop into the perfect sub-nest wrapper. Besides moving the outer loop inside the wrapper, the iteration sequence number for synchronization is updated by multiplying to the domain of the outer loop so that the new parameter still points to the final iteration of the updated sub-nest. Other parameters remain unchanged.

$$\sum_{i_k=0}^{N_k} \underset{\delta,o_s,o_g}{\overset{f_s,f_g}{\mathrm{PerfNest}}} \left( \left( \sum_{\substack{i_j=0 \\ \{\}}}^{N_j} A_{i_j} \right) \right) \rightarrow \underset{N_k\delta,o_s,o_g}{\overset{f_s,f_g}{\mathrm{PerfNest}}} \left( \sum_{\substack{i_k=0 \\ \{\}}}^{N_k} \left( \sum_{\substack{i_j=0 \\ \{\}}}^{N_j} A_{i_j} \right) \right) \tag{4.6}$$

After all perfect sub-nests have been completely detected, the residual canonical loops are converted to loops supporting embedded computations using Rule (4.7)

$$\sum_{i=0}^{N} A_i \rightarrow \sum_{\substack{i=0 \\ \{\}}}^{N} A_i \tag{4.7}$$

**Specifying Iterations for Synchronization**

The results of the static dependency analysis in Section 4.2.3 can be added to the *PerfNest* parameters to allow the early start of iterations from the trailing perfect sub-nest in the pipeline when possible. In Rule (4.8), it first captures the required conditions to the program in the $h$ function of the producer scatter and the consumer gather, then adds the iteration sequence number for synchronization to the *PerfNest* parameter.

$$\underset{\substack{f_s,h_{N,k,b,s} \\ \delta,o_s,o_g}}{\text{PerfNest}}\left(A\right) \underset{\substack{h_{N,k,b^*,s^*,f_g} \\ \delta^*,o_s^*,o_g^*}}{\text{PerfNest}}\left(A^*\right) \rightarrow \underset{\substack{f_s,h_{N,k,b,s} \\ \delta,o_s,o_g}}{\text{PerfNest}}\left(A\right) \underset{\substack{h_{N,k,b^*,s^*,f_g} \\ \frac{k-1}{k}\max(s,s^*),o_s^*,o_g^*}}{\text{PerfNest}}\left(A^*\right) \tag{4.8}$$

**Inductive Computations of Multi-linear Expressions**

With the loop symbol with embedded computations, the multi-linear expressions in the basic blocks can be computed inductively without breaking the loop nest structure. The program transformation are performed in two steps: the first step offloads the multi-linear functions from the basic blocks to the inner most loop; the second step propagates the inductive calculation steps to each relevant loop level.

The first step of basic block computation offloading is specified in Rule (4.9). It offloads the multi-linear expressions from an arbitrary basic block operator to the innermost loop, after which the basic block should reference the results via the companion variable $v_k$. In the loop, $v_k$ is initialized to a truncated multi-linear expression without the term containing the innermost loop variable, and is

added the constant factor of the removed term after executing each iteration. In implementations, a concrete formula pattern need to be instantiated to the SPIRAL system so that a simple pattern match can call for this rule.

$$\sum_{\substack{i_k=0 \\ \{\}}}^{N} \mathrm{A}_{(c_k * i_k + \cdots)} \rightarrow \sum_{\substack{i_k=0 \\ \{v_k=(\cdots), v_k += c_k\}}}^{N} \mathrm{A}_{v_k} \tag{4.9}$$

Rule (4.10), (4.11), (4.12) propagate the loop-embedded accumulations to the outer loop. The principle is similar to Rule (4.9). Rule (4.10) applies to the perfect sub-nest inside the PerfNest wrapper. Rule (4.11) and (4.12) handles the imperfect loop nest composed of multiple perfect sub-nests and imperfect sub-nest, respectively. Finally, the multi-linear expression without any terms degenerate to value zero.

$$\sum_{\substack{i_j=0 \\ \{\}}}^{N_j} \left( \sum_{\substack{i_k=0 \\ \{v_k=(c_j * i_j + \cdots), v_k += c_k\}}}^{N_k} A_{v_k} \right) \rightarrow \sum_{\substack{i_j=0 \\ \{v_j=(\cdots), v_j += c_j\}}}^{N_j} \left( \sum_{\substack{i_k=0 \\ \{v_k=v_j, v_k += c_k\}}}^{N_k} A_{v_k} \right) \tag{4.10}$$

$$\begin{aligned} &\sum_{\substack{i_j=0 \\ \{\}}}^{N} \left( \cdots \underset{\substack{f_s, f_g \\ \delta, o_s, o_g}}{\mathrm{PerfNest}} \left( \sum_{\substack{i_k=0 \\ \{v_k=(c_j * i_j + \cdots), v_k += c_k\}}}^{N_k} A_{v_k} \right) \cdots \right) \\ &\rightarrow \sum_{\substack{i_j=0 \\ \{v_j=(\cdots), v_j += c_j\}}}^{N_j} \left( \cdots \underset{\substack{f_s, f_g \\ \delta, o_s, o_g}}{\mathrm{PerfNest}} \left( \sum_{\substack{i_k=0 \\ \{v_k=v_j, v_k += c_k\}}}^{N_k} A_{v_k} \right) \cdots \right) \end{aligned} \tag{4.11}$$

$$\begin{aligned}
&\sum_{\substack{i_j=0\\ \{\}}}^{N}\left(\cdots\left(\sum_{\substack{i_k=0\\ \{v_k=(c_j*i_j+\cdots),v_k+=c_k\}}}^{N_k} A_{v_k}\right)\cdots\right)\\
&\rightarrow \sum_{\substack{i_j=0\\ \{v_j=(\cdots),v_j+=c_j\}}}^{N_j}\left(\cdots\left(\sum_{\substack{i_k=0\\ \{v_k=v_j,v_k+=c_k\}}}^{N_k} A_{v_k}\right)\cdots\right)
\end{aligned} \tag{4.12}$$

**Bit Manipulations of Multi-linear Expressions**

The applications of Rule (4.9-4.12) can remove all multiplicative operations. The result can be further optimized when the constant factors of the multi-linear expressions are power-of-2 numbers. In this case, the mathematical identity $a * 2^q = a\texttt{<<}q$ allows us to calculate the terms or the entire multi-linear expression with cheap binary bit mapping operations. It is also performed in two steps: the first step converts the accumulations to bit shift operations in the outer-most loop; the second step propagates the bit operations inward the loop nest.

At the first step, Rule (4.13) converts zero-initiated accumulations of power-of-2 numbers $2^q$ to the bit shift operation $i\texttt{<<}q$.

$$\sum_{\substack{i=0\\ \{v=0;v+=2^q\}}}^{N} A_{i,v} \rightarrow \sum_{\substack{i=0\\ \{v=i\texttt{<<}q\}}}^{N} A_{i,v} \tag{4.13}$$

Rule (4.14-4.16) propagate the bit operations inward the perfect and imperfect sub-nest. The conversion is valid when the bit shift of the inner loop does not overlap with the bit fields manipulated by the outer loop, i.e. $[p, p+\log_2 N_i) \cap [q, q+\log_2 N_j) = \emptyset$. In this case, the embedded calculation of the outer loop is removed and merged to the inner loop with a bitwise-or operation.

$$\sum_{\substack{i=0\\{v_i=i<<p}}}^{N_i} \left( \sum_{\substack{j=0\\{v_j=v_i;v_j+=2^q}}}^{N_j} A_i \right) \rightarrow \sum_{\substack{i=0\\{}}}^{N_i} \left( \sum_{\substack{j=0\\{v_j=i<<p|j<<q}}}^{N_j} A_i \right), \tag{4.14}$$

$$\begin{aligned} &\sum_{\substack{i=0\\{v_i=i<<p}}}^{N_i} \left( \cdots \underset{\substack{f_s,f_g\\ \delta,o_s,o_g}}{\text{PerfNest}} \left( \sum_{\substack{j=0\\{v_j=v_i;v_j+=2^q}}}^{N_j} A_i \right) \cdots \right) \\ &\rightarrow \sum_{\substack{i=0\\{}}}^{N_i} \left( \cdots \underset{\substack{f_s,f_g\\ \delta,o_s,o_g}}{\text{PerfNest}} \left( \sum_{\substack{j=0\\{v_j=i<<p|j<<q}}}^{N_j} A_i \right) \cdots \right), \end{aligned} \tag{4.15}$$

$$\sum_{\substack{i=0\\{v_i=i<<p}}}^{N_i} \left( \cdots \left( \sum_{\substack{j=0\\{v_j=v_i;v_j+=2^q}}}^{N_j} A_i \right) \cdots \right) \rightarrow \sum_{\substack{i=0\\{}}}^{N_i} \left( \cdots \left( \sum_{\substack{j=0\\{v_j=i<<p|j<<q}}}^{N_j} A_i \right) \cdots \right), \tag{4.16}$$

**Reduced Buffer Allocation**

The three strategies for reduced buffer allocation are implemented in a Σ-OL syntax tree visitor for assigning gather and scatter address offsets to all perfect sub-nests.

The first perfect sub-nest gathers from offset zero. Assume that the input buffer size is $N$, the sub-nest scatters to offset 0 or $N$, depending on whether the basic block is in-place computation or not. The visitors maintains the most recent offset and the buffer size so that it can assign a correct offset to other perfect sub-nests in the execution order. The final buffer offset and the total memory size is collected for hardware generation.

#### Collecting Basic Blocks Parameters

As all basic blocks are required to perform the same gather, scatter and kernel operations, potentially with different parameters. We need to extract and manage these parameters so that the hardware generator can select the correct parameters for basic block execution.

This process requires the pre-registration of each basic block operator and the PerfNest construct for the pattern match shape, a function to extract parameters. It traverses the loop nest program in execution order. When a PerfNest construct is visited, the read and write offsets are extracted. The parameters extracted from each basic block operator depends on the individual registered function for parameter extractions. The parameters of each operator for all basic blocks forms a two-dimensional array.

## 4.4 Generating RTL Designs

The idea of our hardware generation backend is to decompose the hardware construction problem as selecting building blocks and connecting them. We have created the initial set of hardware building blocks in the icode library. Our backend can currently synthesize the hardware loop nest controllers and the feed-forward pipelined datapath in Chisel RTL language from icode. As is shown in Figure (4.7) for an 8-point WHT design, the loop nest controller is constructed as coordinating finite state machines of single loop controllers and loop composers. The datapath is constructed as a directed acyclic graph of primitive hardware operators. Since our framework is built with extensibility, further extensions of the building blocks and more advanced synthesis techniques can be added to the framework.

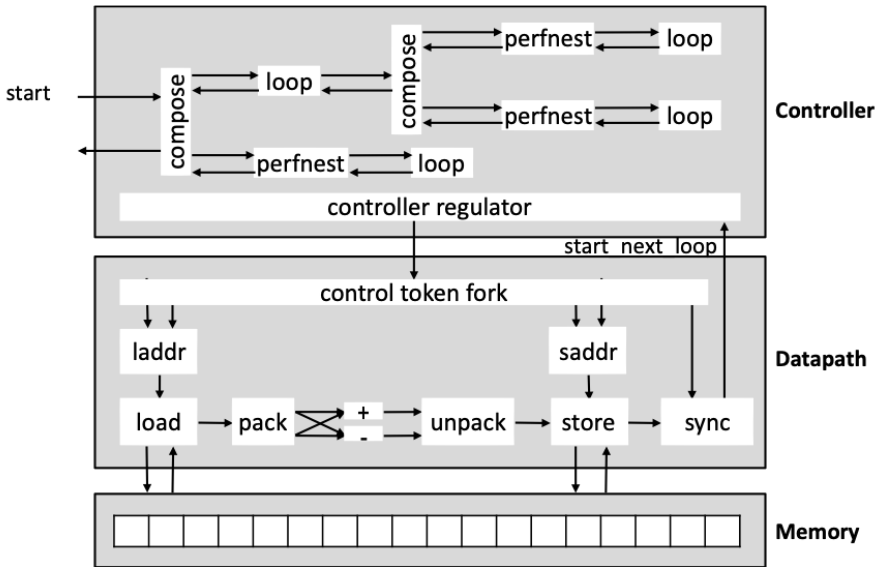
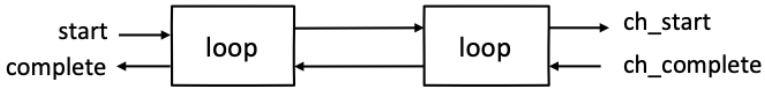
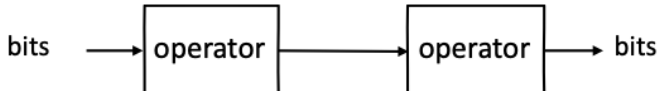



Figure 4.7: Implement an 8-point WHT hardware.

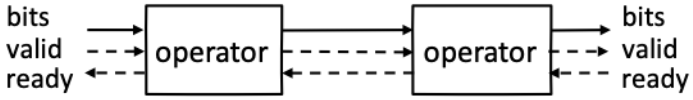
Two types of interface protocols are employed in connecting the building blocks, as shown in Figure 4.8. In the loop nest controller, FSMs are connected with raw signals (Figure 4.8a). In the datapath, the hardware operators are connected either with a raw interface (Figure 4.8b) or with a decoupled interface that associates the ready / valid signal to the raw signals (Figure 4.8c). The decoupled interface is latency-insensitive and allows distributed control of the datapath, thus provides flexibility in hardware construction at the cost of token overhead. We allow raw interface and decoupled interface simultaneously in our backend to enable the co-exist of efficiency and flexibility.




(a) The raw interface in loop nest controller.




(b) The raw interface in datapath.




(c) The decoupled interface in datapath.

Figure 4.8: The two interface protocols used in backend.

The optimized Sigma-OL expressions can be converted to hardware designs in icode through a translation algorithm based on a hierarchical visitor algorithm to the syntax tree.

### 4.4.1 Hardware icode Constructs

In our extensions to the icode library, except for a limited set of extensions dedicated for loop nest controllers, other extensions are mainly used in datapath design where some general components are also used in controller construction.

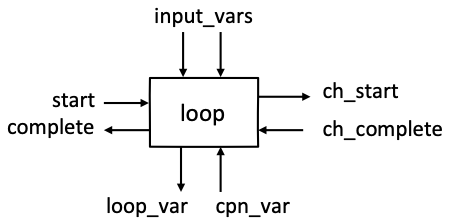
#### Controller Extensions

To describe the coordinating FSMs design of the loop nest controllers, we have added three FSM types, two commands and a special operator, as listed in Table 4.6. The three FSM types models the single loop, the loop composer, and the perfect sub-nest wrapper, respectively, whose I/O ports are shown in Figure 4.9. The

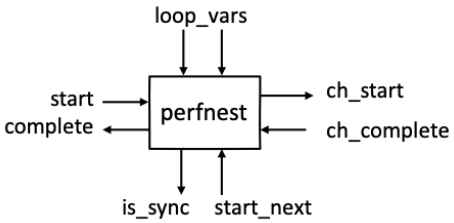
I/O ports of the instantiated FSM modules are connected using the new commands *loop_ctrl_connect* and *loop_io_connect*. The former command connects the *ch_start / ch_complete* signals of an FSM to the *start /complete* signals of its children FSMs. The latter command connects the loop variables / companion variables between two FSMs that exist a producer-consumer relationship in the loop embedded computation expressions. A special operator *trigger_bb* models an FSM that takes in the start signals of all perfect sub-nests and produce a basic block activation signal.

Table 4.6: The icode extensions dedicated for loop nest controllers.

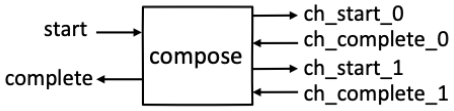
| Category | name | descriptions |
|---|---|---|
| type | TLoop( v,range,embed_calc) | Model an FSM that issues all iterations of a loop and calculate the embedded expressions. |
| type | TCompose( n) | Model an FSM that sequentialize the execution of n children FSMs. |
| type | TPerfNest( is_sync_expr) | Model an FSM that triggers the execution of perfect sub-nest and indicates synchronization. |
| command | loop_io_connect( src_loop, dst_loop) | Connect the I/O ports of loop variables between loop FSMs. |
| command | loop_ctrl_connect( src_mod, dst_mod) | Connect the I/O ports of control signals between loop controller FSMs. |
| operator | trigger_bb([perfnest_1, .., perfnest_n]) | Produce the basic block activation signal by monitoring the start signal of perfect sub-nest. |




(a) The loop FSM.




(b) The perfect nest FSM.




(c) The loop composer FSM.

Figure 4.9: The I/O descriptions of loop controller FSMs.

### Datapath Extensions

To describe the uni-rate streaming datapath, we have extended the icode library. Our extension includes types, commands, operators and location descriptors. The new operators are the key constructions of our extension while the new types, commands and location descriptors, as shown in Table 4.7, supports the new operators.

Three new types are introduced to capture the arbitrary precision type, the decoupled token type, and a RTL module generation type. The *TUIntAP(nbits, max)* type specifies an unsigned integer type with a given bit width and the maximal possible value. The *TDecoupled(t)* encapsulates a normal icode type and associates the ready and valid flags to the type payload. Operators processing data of these new types can support bit-precise input/output and the decoupled interface natively. The *TRTLModGen(name,input,output,code)* models an RTL module describing the input, output and the internal design using icode, which is more general than the controller FSM types dedicated for the loop nest controller.

Two new commands of our extension supports the hardware semantic. The *instantiate* command instantiates a type of any RTL modules, including the loop controller FSM modules so that it can be assigned to a variable of a corresponding type. The *define* command is in charge of the definition of types and operators constructed on demand.

The new location descriptors allow us to reference variable in particular environments. The

Table 4.7: The icode extensions of types, commands and location descriptors.

| Category | name | descriptions |
|---|---|---|
| type | TUIntAP(nbits, max) | An arbitrary precision type with specified maximal value. |
| type | TDecoupled(t) | A token type encapsulating ready/valid signals. |
| type | TRTLModGen(name, input,output,code) | Construct an RTL module with icode. |
| command | instantiate | Instantiate a type of RTL module. |
| command | define | Define a list of normal icode types of RTL . modules. |
| location | field(v,fvar) | Reference to a field fvar of a structured type. variable v I/O variable |
| location | iovar(v) | Reference to an I/O variable. |
| location | modiovar(v,vmod) | Reference to an I/O variable of an instantiated RTL module. |
| location | valid_tk(tk) | Reference to the valid flag of a token variable. |
| location | ready_tk(tk) | Reference to the ready flag of a token variable. |
| location | bits_tk(tk) | Reference to the payload of a token variable. |

Compared to the current icode operators for software generation, the hardware icode operators require hardware-specific properties. When composing hardware designs with icode operators, the *delay* in clock cycles between the output data and the input data of each operator must be specified so that buffers can be inserted in proper locations to assure the functional correctness and full throughput. Furthermore, when the decoupled interface is employed in operators, the *token scaling*

*ratio* – a fractional ratio of the number of tokens produced over each token consumption, is essential to guarantee the consistency between token production and consumption of the datapath. Currently, we only handle operators with constant delays and token scaling ratios. More dynamism can be supported in the future with proper FIFOs inserted to the design.

The common integer arithmetic, logical and relational operators already inside the icode library are used as hardware operators after setting their delay to zero and the token scaling ratio to one. We further extend the icode operators in three groups: token regulation operators, memory operators and other customized operators.

The token regulation operators listed in Table 4.8 assure the consistent token production and consumption. The *tk_pack* and *tk_unpack* operators bridges the gap between the scalar memory words and the vector data processed in Σ-OL kernels. The *tk_range* operator can drive iterative computations with its output tokens in the absence of control flow commands. The *tk_buf* operator works like a register in non-decoupled interface operators. The *tk_rpt* operator duplicates the input token multiple times to a single consumer. The *tk_fork* operator duplicates the input token to multiple consumers.

Table 4.8: The icode extensions of token regulation operators.

| Operator | Delay | Ratio | Descriptions |
|---|---|---|---|
| tk_pack(tk, n) | $n$ | $\frac{1}{n}$ | Accumulate $n$ input tokens to produce a token of size-n array type. |
| tk_unpack(tk) | 0 | $n$ | Convert a size-n array typed token to $n$ output tokens. |
| tk_range(tk, n) | 0 | $n$ | Produce a token stream with values range from 0 to n-1 for each input token. |
| tk_buf(tk) | 1 | 1 | Delay a token for one clock cycle. |
| tk_rpt(tk, n) | 1 | $n$ | Repeat each input token n times to the output. |
| tk_fork(tk, n) | 0 | 1 | Duplicate each input token to n users. |

The memory operators need to bind with a specific platform that provides a memory interface. It is worth noting that *mem_read* and *mem_write* operators are typed operators such that the operators can adapt to the bit width change of address and data. In contrast to a software write operator, our *mem_write* for hardware returns an acknowledgement token so that it is straightforward to synchronize at the completion of memory stores for a target iteration. The *lktable* operator supports lookup table implemented using read-only memory.

Table 4.9: The icode extensions of memory operators.

| Operator | Delay | Ratio | Descriptions |
|---|---|---|---|
| mem_read(baddr, idx, type) | 1 | 1 | Read a typed data from memory at base address baddr and with index idx. |
| mem_write(baddr, idx, wdata) | 1 | 1 | Write a typed data to memory at base address baddr and with index idx. |
| lktable(entries, loc) | 1 | 1 | Read an entry from the ROM with index loc. |

Other customized operators listed in Table 4.10 are created to facilitate the construction of hardware designs. The *mux* operator models the common multiplexors in digital circuit. The *accum* operator exploits the token flexibility to perform accumulations without control flow commands. The *bit_maps* operator can replace some expensive arithmetic operations for power-of-2 numbers. The *add_sub* is an operator usually employed in the low-level IR of hardware compilers for resource optimization. Is it explicitly defined in our library so that this technique can be used manually. No that the delay of any arithmetic operators depends on the platform and the type of data to be processed by the operator. The *comb_blk* operator can encapsulate an arbitrary pipelined combinational raw data typed datapath with decoupled interface.

Table 4.10: The icode extensions of other customized operators.

| Operator | Delay | Ratio | Descriptions |
|---|---|---|---|
| mux(vsel, [v1, .., vn]) | 0 | 1 | Select an output from the input variables. |
| accum(v, stride, n) | 0 | $n$ | Accumulate stride to the v token n times. |
| bit_maps(vars, bfields) | 0 | 1 | Maps variable fields to the bit field locations. |
| addsub(a,b,sel) | TBD | 1 | Add or subtract two inputs depending on sel. |
| comb_blk(in,code,delay) | delay | 1 | Take in decoupled data and process internally with the icode specification. |

### 4.4.2 Synthesize RTL code from Σ-OL

This work provides two independent synthesis flow for the controller and the datapath.

#### Controller Synthesis

We obtain the spatial design of loop nest controllers in two steps. First, we translate the Σ-OL loop nest to a coordinating FSM design in icode using the rules described in Table 4.11. Second, we generate the accessory components of the controller.

The translation between loop nest of Σ-OL and of icode is performed with a hierarchical visitor of the syntax tree. The visitor maintains a list of icode generated in this process. When visiting a, the assignment of the instantiation command of the corresponding node is added to the list. Then the children nodes are visited. Afterwards, the control connections and data connections between parent node and children are added to the list.

Table 4.11: Synthesize the coordinating FSMs in icode for loop nest controllers.

| Σ-OL | icode |
|---|---|
| Compose | assign(var_compose, instantiate(TCompose(nch)) |
| | loop_ctrl_connect(var_compose, children) |
| PerfNest | assign(var_perfnest, instantiate(TPerfNest(is_sync_expr))) |
| | loop_ctrl_connect(var_perfnest, children) |
| | loop_io_connect(var_perfnest, any_source_loop) |
| $\sum_{i=0}^{N}$ | assign(var_loop, instantiate(TLoop(cpnv))) |
| | loop_ctrl_connect(var_loop, children) |
| | loop_io_connect(var_loop, any_source_loop) |

The controller design is not completed until the accessory components are generated. First, to activate the right basic block in execution, we use the *trigger_bb* operator to monitor the start signal of each perfect sub-nest FSM, whose output indicates which basic block is being executed. Second, all variables captured in the basic block unifying process as parameters will be multiplexed to become the payload component of the control token, selected by the basic block activation signal. Finally, simple gate logic and wire connections assures the control token can be constructed and interface with the datapath correctly.

### Datapath Synthesis

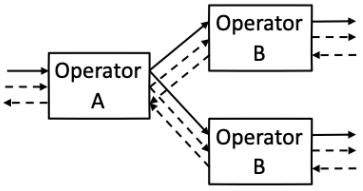
The datapath design in icode is synthesized from Σ-OL expressions in two steps. First, each Σ-OL construct is mapped to the corresponding icode expressions. Second, several compiler passes are employed to transform the icode with complete information for RTL code generation.

Table 4.12 shows the current rewrite rules we have implemented, where gather and scatter operations, arbitrary non-parameterized OL kernels, the diagonal operator, and the size-2 DFT operations are handled. For arbitrary OL kernels, an *ol2icode* function translates the OL specification to icode that processes data vectors such that a *comb_blk* operator can be constructed. Although we haven't implemented an automated throughput-constrained basic block scheduler for reducing resource utilization of *comb_blk* operators, we provide an example of 2-point DFT operations for using the extensible operations to achieve the same effect.

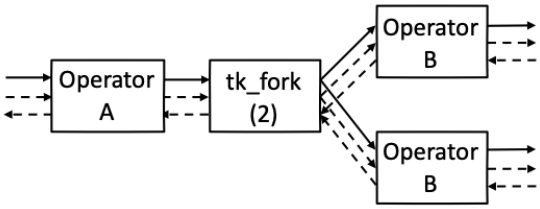
Table 4.12: Synthesize the datapath in icode.

| Σ-OL | icode |
|---|---|
| $G_{h_{N,n,b,s}}$ | assign(y, mem_read(roffset, y.t.t, accum(b, s, n))) |
| $S_{h_{N,n,b,s}}$ | assign(y, mem_write(woffset, accum(b, s, n), x)) |
| OLContainer($t_n$) | assign(in, pack_tk(x, n))<br>assign(y, unpack(comb_blk(x, ol2icode(t), delay))) |
| $Diag_f$ | assign(y, x * lktable(entries, f()) |
| $F_2$ | assign(in, tk_pack(x,2))<br>assign(y, addsub(nth(in,0), nth(in,1), tk_range(in,2)=0)) |

The translated icode requires several compiler transformations before being unparsed to the actual RTL designs. First, as illustrated in Figure 4.10 the icode expressions using the decoupled interface requires the variables referenced multiple times to be explicitly duplicated using the *tk_fork* operator. Second, as shown in Figure 4.11, the icode needs to be inspected for the case that operators process tokens at different rates, in which the *tk_rpt* operator needs to be inserted. Finally, the icode expressions with determined total delay requires inserting elastic buffers

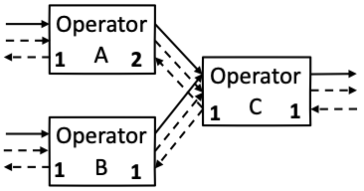



(a) Before transformation.

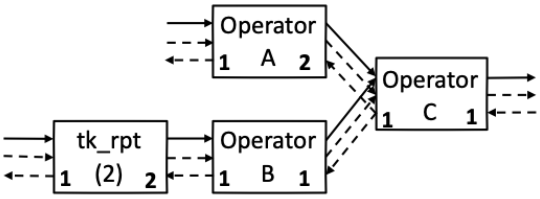



(b) After transformation.

Figure 4.10: Transform icode with explicit token duplication.




(a) Before transformation.




(b) After transformation.

Figure 4.11: Transform icode with explicit temporal token duplication.

or registers depending on whether the decoupled interface is used or not. This transformation assures functional correctness and the full throughput of the pipeline. The algorithms for performing these transformations work similarly. The icode is first split to assignments to a variable of a single operator using variable arguments. Then the input variables are set the initial metric of interest. We inspect each assignment in data flow order at which the input variable are inspected for the metric and determines if the helper operators needed to be inserted or not. The algorithm completes when the last assignment of the data flow has been processed.

### Unparse icode to Chisel RTL Code

To translate our hardware designs in icode to Chisel RTL code, we have created a Chisel RTL code library that provides the mapping targets of each icode type,

operator, command and location descriptor. The process is straightforward with a recursive-descent translator.

## 4.5 Algorithm Generation for Load-store Architectures

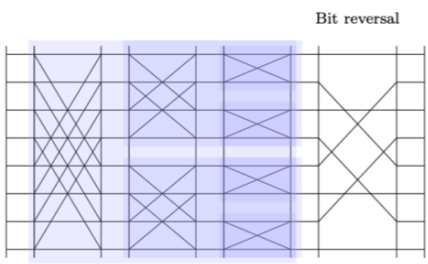
In this work, a few algorithms supported in the SPIRAL framework are selected for the WHT, DFT and bitonic sorting problems because they inherently match or can be slightly modified to match the multi-linear paradigm. This section discusses the decision of algorithms for scalar load-store architectures. For each algorithm, the breakdown rules can expand the specifications into OL formulas, which is translated to Σ-OL expressions using SPIRAL, and then the loop fusion optimization [2] is performed to obtained the Σ-OL expressions conforming to the multi-linear paradigm.

### 4.5.1 DFT Algorithms

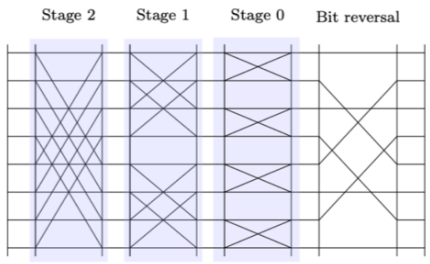
There are a bunch of algorithms for calculating DFTs, with various data flow geometries that can satisfy different architectures. For a scalar load-store architecture with a single level dual-ported memory, a reasonable goal is to maximize the utilization rate of the scalar pipeline. As a result, an algorithm that provides the most parallelism is the best.

#### Analysis

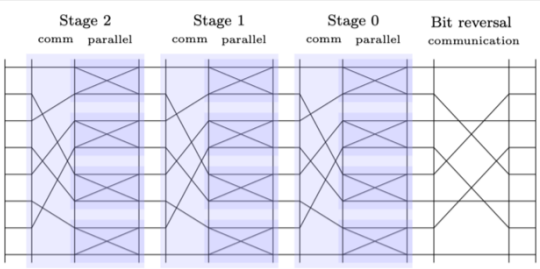
Here, we discuss the considerations between three famous FFT algorithms, i.e. the recursive Cooley-Tukey FFT, the iterative Cooley-Tukey FFT and the Pease FFT. Though the illustrative data flow graphs in Figure 4.12 are for size-8 FFT, the pattern applies for general radix-$r$ sizes. The three algorithms in mathematics can be reduced to the same Cooley-Tukey decomposition, but provide different orders




(a) Recursive Cooley-Tukey FFT.




(b) Iterative Cooley-Tukey FFT.




(c) Pease FFT.

Figure 4.12: Classic algorithmic candidates in data flow graphs for FFT (N=8).

for calculating the basic butterfly operations. A shared property of all data flow graphs is the separation between an accumulated bit reversal permutation stage and the computation stage.

In recursive Cooley-Tukey FFT shown in Figure 4.12a, the order of computation is different as emphasized by the shading. It can be seen that the butterfly operations with its input ready are not computed as early as possible. As a result, the recursive algorithm is not the best for scalar load-store architectures.

In contrast, both the iterative Cooley-Tukey FFT shown in Figure 4.12b and the Pease FFT shown in Figure 4.12c decomposes computations into several data-dependent stages, thus exposes more parallelism than the recursive algorithm. In particular, the Pease algorithm is favored by streaming implementations because it provides identical geometry in each stage with parallel butterfly operations and

stride permutations. However, the automated generation of load-store architectures in this work prefers the *iterative Cooley-Tukey FFT* because it offers more opportunities for across-stage execution overlapping (see Section 4.2.3) by not enforcing an identical geometry between stages.

### Iterative FFT Breakdown Rules

The iterative radix-$r$ Cooley-Tukey FFT algorithm is given in Formula 4.17. By applying the algorithm to eight-point DFT and given the definition of two-point DFT in Formula 4.18, the obtained algorithm is shown in Formula 4.19. Note that the initial bit reversal permutation is merged to the computational stages.

$$\mathrm{DFT}_{r^\ell} \rightarrow \left( \prod_{i=0}^{\ell-1} D_i^{r^\ell} \left( I_{r^i} \otimes \mathrm{DFT}_r \otimes I_{r^{\ell-i-1}} \right) \right) R_r^{r^\ell} \tag{4.17}$$

$$\mathrm{DFT}_2 = F_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{4.18}$$

$$\begin{aligned} \mathrm{DFT}_8 = & \left( \sum_{i_1=0}^{4} S_{h_{i_1,4}} D_{f_3} F_2 G_{h_{i_1,4}} \right) \left( \sum_{i_2=0}^{2} \sum_{i_3=0}^{2} S_{h_{i_3+4i_2,2}} D_{f_2} F_2 G_{h_{i_3+4i_2,2}} \right) \\ & \left( \sum_{i_4=0}^{2} \sum_{i_5=0}^{2} S_{h_{2i_5+4i_4,1}} D_{f_1} F_2 G_{h_{2i_5+i_4,4}} \right) \end{aligned} \tag{4.19}$$

### Recursive FFT Breakdown Rules

Though not for performance, the classic recursive Cooley-Tukey FFT is also implemented in this work to study the effect of pattern-based loop optimizations. The classic recursive Cooley-Tukey FFT algorithm defined in Formula 4.20. After recursively applying the algorithm to eight-point DFT, the obtained algorithm does

not conform to the multi-linear paradigm by missing a diagonal operator on the left most basic block. As a result, a special diagonal operator $D_0$ equalizing with the identity operator by multiplying 1s to all input data is inserted to the left most basic block. Based on the property of twiddle factors, $D_0$ is constructed by indexing to the first element of the twiddle factor lookup table. The resulting algorithm is shown in Formula 4.21 which conforms to the multi-linear paradigm.

$$\mathrm{DFT}_{mk} \rightarrow (\mathrm{DFT}_k \otimes \mathrm{I}_m)\, T_m^{mk}\, (\mathrm{I}_k \otimes \mathrm{DFT}_m)\, \mathrm{L}_k^{mk} \tag{4.20}$$

$$\begin{aligned} \mathrm{DFT}_8 = & \left( \sum_{i=0}^{2} \left( \sum_{k=0}^{2} S_{h_{i+2k,4}} D_0 F_2 G_{h_{k,2}} \sum_{l=0}^{2} S_{h_{2l,1}} D_{f_2} F_2 G_{h_{i+2l,4}} \right) \right) \\ & \left( \sum_{j=0}^{4} S_{h_{2j,1}} D_{f_1} F_2 G_{h_{j,4}} \right) \end{aligned} \tag{4.21}$$

**Discussion: Twiddle Factor Scalability**

In expanded algorithms of FFTs, we have made an assumption that the diagonal operator that multiplying the twiddle factors to input data will be implemented as a hardware block using an internal lookup table. However, this will limit the feasible problem sizes in hardware design where a main memory is connected to the load-store architecture and the large capacity of main memory should have stored the large twiddle factor tables instead. On the other hand, using on-chip ROMs for twiddle factors can save the precious memory bandwidth. Hence, both methods worth handling properly in a design generator.

The on-chip ROM-based method could fit larger problem sizes by compressing the twiddle factors by leveraging domain knowledge. The twiddle factors by

definition is the N-th root-of-unity. The left side of Figure 4.13 shows the definition of the $k$-th entry of the twiddle factors for $n$-point FFT as $\omega_n^k$. It is a complex number generated by sinusoids. The right side of Figure 4.13 shows a root-of-unity in complex number plane. One well known trick is that by leveraging the symmetry of sinusoids, only eighth of the original size need to be stored in the lookup table. In this way, the other part of the table can be re-directed to the stored portion of the table without accuracy loss.

$$\omega_n^k = e^{\frac{2\pi k i}{n}} = \cos\frac{2\pi k}{n} + i\sin\frac{2\pi k}{n}$$

*Known trick: compressing to 1/8*

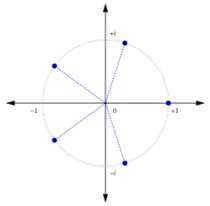


Figure 4.13: Known trick: compressing twiddle factors with symmetry.

Another method for compression is based on the periodicity of twiddle factors. Figure 4.14 shows the derivation process of twiddle table decomposition from size-$mn$ to size-$n$ and size-$m$. The decomposition introduces an extra multiplication between entries of the two sub-tables. In this way, the accuracy of the final twiddle factors is tunable by specifying the precision of the multiplier. When applying this technique, much larger problem sizes can be supported with limited capacity of on-chip ROMs.

$$\omega_{mn}^k = \omega_{mn}^{\lfloor \frac{k}{m} \rfloor m + (k \bmod m)} = \omega_{mn}^{\lfloor \frac{k}{m} \rfloor m} \omega_{mn}^{k \bmod m} = \omega_n^{\lfloor \frac{k}{m} \rfloor} \omega_{mn}^{i \bmod m}$$

Figure 4.14: Decomposing size-m*n table to size-n table and size-m table.

If the twiddle factor lookup table size still exceeds the ROM capacity, a general solution that loads data entries from the main memory is available as shown in Figure 4.15. In the design, the diagonal operator shares the memory read interface

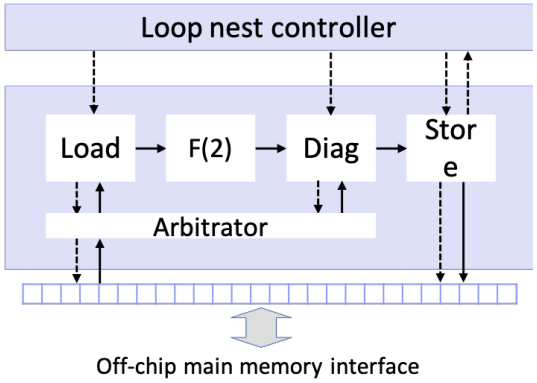



Figure 4.15: A general solution that loads twiddle factors from main memory.

with the load unit through an arbitrator module. The arbitrator module determines the right of access for each request and is expected to be configured to efficiently utilize the memory bandwidth. However, such a design will throttle the speed of data loading and will then limit the speed of data storing even though a dedicated write port is monopolized by the store unit.

### 4.5.2 WHT and Sorting

For WHT and sorting, I temporarily implemented the recursive algorithms for demonstration process. An efficient implementation like FFT will require similar effort to identify the proper algorithms from numerous candidates to best saturate the scalar customized pipeline. A performance analysis of WHT algorithms can be found in [42].

**Recursive WHT.** A recursive WHT algorithm defined in Formula 4.22 is incorporated in this work. By recursively applying the breakdown rule to eight-point WHT and given the definition of two-point WHT in Formula 4.23, one obtained algorithm is shown as a Σ-OL expression in Formula 4.24. In the generated algorithm, the multi-linear pattern of load and store is represented by the multi-linear functions

of the base of each $h$ function of gather and scatter. The kernel operations are all 2-point DFT operations.

$$\mathrm{WHT}_{mk} \rightarrow (\mathrm{WHT}_m \otimes \mathrm{I}_k)\,(\mathrm{I}_m \otimes \mathrm{WHT}_k) \tag{4.22}$$

$$\mathrm{WHT}_2 = F_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{4.23}$$

$$\mathrm{WHT}_8 = \left(\sum_{i=0}^{2}\left(\sum_{k=0}^{2} S_{h_{i+2k,4}} F_2 G_{h_{k,2}} \sum_{l=0}^{2} S_{h_{2l,1}} F_2 G_{h_{i+4l,2}}\right)\right)\left(\sum_{j=0}^{4} S_{h_{2j,1}} F_2 G_{h_{2j,1}}\right) \tag{4.24}$$

**Bitonic sorter.** The final algorithm studied in this work solves the problem of obtaining a sorted list of data, called the bitonic sorter. In contrast to the previous algorithms, bitonic sorter is a non-linear operation. As suggested by the name, a bitonic sort algorithm divides the data list into two sub-list, sorts them with reversed direction, then merges the two sorted sub-list. The algorithm defined in SPIRAL software flow hardcoded the direction of the sub-problems and utilize a direct sum operator to combine the two sub problems, which does not conform to the multi-linear paradigm. To enhance the regularity of the algorithm representation, we define a new OL construct $\chi\Theta_{n,d}$ of sorting that adds the sorting direction as a boolean parameter $d$, representing ascending sorting value true and descending sorting with value false. Then the same algorithm is defined with the new constructs in Formula 4.25 and 4.26.

$$\chi\Theta_{n,d} \rightarrow M_{n,d}\left(\mathrm{I}_2 \otimes_l \chi\Theta_{n/2,l=0}\right) \tag{4.25}$$

$$M_{n,d} = \left(\mathrm{I}_2 \otimes M_{n/2,d}\right)\left(\chi\Theta_{2,d} \otimes \mathrm{I}_{n/2}\right) \tag{4.26}$$

$$M_{1,d} = \begin{bmatrix} 1 \end{bmatrix} \tag{4.27}$$

$$\chi\Theta_{2,d}\begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = K_d \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = \begin{bmatrix} \text{if } d\ min(x_0, x_1) \text{ else } max(x_0, x_1) \\ \text{if } d\ max(x_0, x_1) \text{ else } min(x_0, x_1) \end{bmatrix} \tag{4.28}$$

As shown in Formula 4.27, a one-point merger is an identity, thus will be removed at the end of algorithm expansion. By applying the algorithm to a size-8 data list and providing the building blocks of sorter in Formula 4.28, an ascending bitonic sort algorithm is obtained in Formula 4.29.

$$\begin{aligned}
&\chi\Theta_{8,true} = \\
&\sum_{i_3=0}^{2}\left(\sum_{i_5=0}^{2} S_{h_{4i_3+2i_5,1}}\,\mathrm{K}_{true}\sum_{i_6=0}^{2} S_{h_{i_6,2}}\,\mathrm{K}_{true}\,G_{h_{i_6+4i_3,2}}\right) \\
&\left(\sum_{i_4=0}^{4} S_{h_{i_4,1}}\,\mathrm{K}_{true}\,G_{h_{i_4,1}}\right) \\
&\sum_{i_1=0}^{2}\left(\sum_{i_7=0}^{2} S_{h_{2i_7+4i_1,1}}\,\mathrm{K}_{i_1=0}\,G_{h_{2i_7,1}}\sum_{i_8=0}^{2} S_{h_{i_8,2}}\,\mathrm{K}_{i_1=0}\,G_{h_{i_8,2}}\sum_{i_2=0}^{2} S_{h_{2i_2,1}}\,\mathrm{K}_{i_2=0}\,G_{h_{4i_1+2i_2,1}}\right)
\end{aligned} \tag{4.29}$$

As explained above, we have formally introduced four algorithms conforming to the multi-linear paradigm. The WHT algorithm and the iterative FFT algorithm inherently conform to the paradigm, while the recursive FFT algorithm and the bitonic sorting algorithm require slight modification to meet the paradigm requirement.

## 4.6 Summary

This chapter explains the extensions to the SPIRAL framework for addressing the multi-linear paradigm of imperfect loop nest programs for hardware generation targeting the customized scalar load-store architecture. The extensions cover the full flow from algorithm generation to program optimization to hardware interpretation, extended the OL, Σ-OL and icode DSLs of SPIRAL, respectively.

The DSL extensions at the three abstraction levels are made to capture either the properties of the paradigm or the structures of hardware designs. In OL, the sorting operation with the direction parameter enables the unified kernel operation for the paradigm. In Σ-OL, the perfect sub-nest construct allows this special program structure to be analyzed and optimized through simple pattern matching in the form of term rewriting rules. Another Σ-OL extension is the loop with embedded computations, which captures the optimized program with loop-inductive computations in a way that conforms to the multi-linear paradigm. In icode, new types of precision-precise integers and the decoupled interface make the hardware features explicit; new operators and commands allow the modeling of arbitrary RTL modules and their connections.

While the multi-linear paradigm imposes some properties of the program, it still provides instantiation flexibility in the iteration space, the multi-linear gath-

er/scatter pattern and the kernel specification such that many different algorithms beyond the WHT, DFT and bitonic sorter can be supported.

# Chapter 5

# Evaluation

In this chapter, we employ multiple dimensions of metrics to evaluate the effectiveness of the proposed approach. They include the effectiveness of pattern-based loop optimizations for hardware, and the quality of designs of FFT cores compared to a state-of-the-art DFT IP generator. The metrics are obtained at different steps of the evaluation flow as is shown in Figure 5.1, where the Spiral generated designs are compiled to Verilog RTL code and then simulated cycle-accurately, synthesized and place-and-routed to FPGA implementations. The details are elaborated in the following sections.

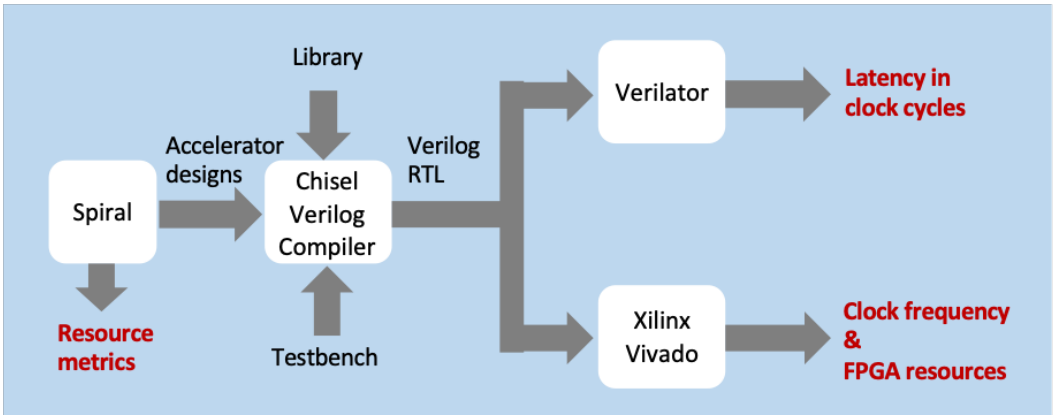



Figure 5.1: Obtaining experimental data from the Spiral-generated designs.

## 5.1 Effectiveness of Hardware Optimizations

### 5.1.1 Methodology

To evaluate the effectiveness of hardware optimizations perform at Σ-OL, we have generated algorithms for WHT, DFT and bitonic sorter and implemented the load-store architecture with our approach. The baseline implementations exclude all hardware optimizations in the Σ-OL level. The optimized implementations are optimized in Σ-OL for latency, buffer resources and the arithmetic units for calculating multi-linear expressions. Each dimension could be optimized with different depth, and we compare the results from the baseline to the deepest optimization.

We only investigate the latency of iterative FFTs because it is the only algorithm in this thesis that provides enough parallelism. The base implementation forces a perfect sub-nest to complete all memory write operations of all iterations before the next perfect sub-nest can start. The optimized implementation exploits the multi-linear access pattern for static analysis and identifies the iteration for synchronization whose completion unlocks the start of the next perfect sub-nest.

To obtain the execution latency, we have crafted an accelerator wrapper in Chisel RTL [43] and a testbench using the Chisel iotesters 2.10 [44] API. The iotesters framework compiles the Chisel RTL code to Verilog RTL code, then invoke the Verilator 3.904 simulator [45] to compiled the Verilog RTL code to C++ code which is further compiled and executed for cycle-accurate simulation. This setup initializes the input data in memory through the input data ports and retrieves the result data from memory via the output data ports. We report the lapse of clock cycles in the testbench between the time all input data has been loaded to the memory and the time when the resulting data is ready to be retrieved from the memory.

To obtain the buffer utilization of the generated designs, we instrument the buffer allocation process of SPIRAL to report the aggregated number of data entries in the memory. The buffer allocation processes are implemented as four separated passes with different levels of optimizations. Level-0 reserves the dedicated input and output buffers and allocates intermediate buffers for every composition of loops. It serves as the baseline. Level-1 can exploit in-place calculations in loop compositions to reuse the read buffer as write buffer. Level-2 simulates the dynamic temporary buffer allocation scheme by recycling the buffer entries used in deeper loop nest that have completed computation. Level-3 is the enhanced version of level-2 that also considers the in-place computations for further buffer savings.

To investigate the arithmetic cost of calculating multi-linear expressions, we count the total bits of adders and multipliers in the $\Sigma$-OL expressions. The baseline implementation calculates the expression directly, using considerable number of adders and multipliers. We divide the optimizations into two, where the first utilizes inductive calculations to avoid multipliers and the second further exploits the power-of-2 constant factors to replace adders as bit mapping operations.

### 5.1.2 Latency

Figure 5.2 shows the latency (in clock cycles) of the generated radix-2 FFT cores ranging from size of 8 to size of 2048 that compares the conservative and the precise dependency management schemes. The secondary axis presents the speedup of the precise scheme over the conservative scheme because the latency grows fast as the problem size doubles and makes it difficult to show the latency of small FFT sizes in the primary axis. The average speedup is 1.16x. The speedup increases at the beginning as the problem sizes grow from size of 8, arrives the peak of 1.41x at size of 64, and decreases when the problem sizes grow larger. The speedup increase in smaller size range is attributed to the growing parallelism given by the increasing number of independent iterations in the perfect sub-nest. Because the implemented datapath pipeline has a latency of 43 cycles, for very small size such as 4, the latency difference between precise and conservative dependency management is negligible. At size of 64, the number of independent iterations at each perfect sub-nest is slightly larger than the pipeline latency so that most of the pipeline bubbles can be avoided as long as the iteration for synchronization does not reside at the very end of the perfect sub-nest. The speedup falls when the size getting even larger, because in these situations, the cost of pipeline draining is relatively small compared to the total number of independent iterations at each perfect sub-nest.

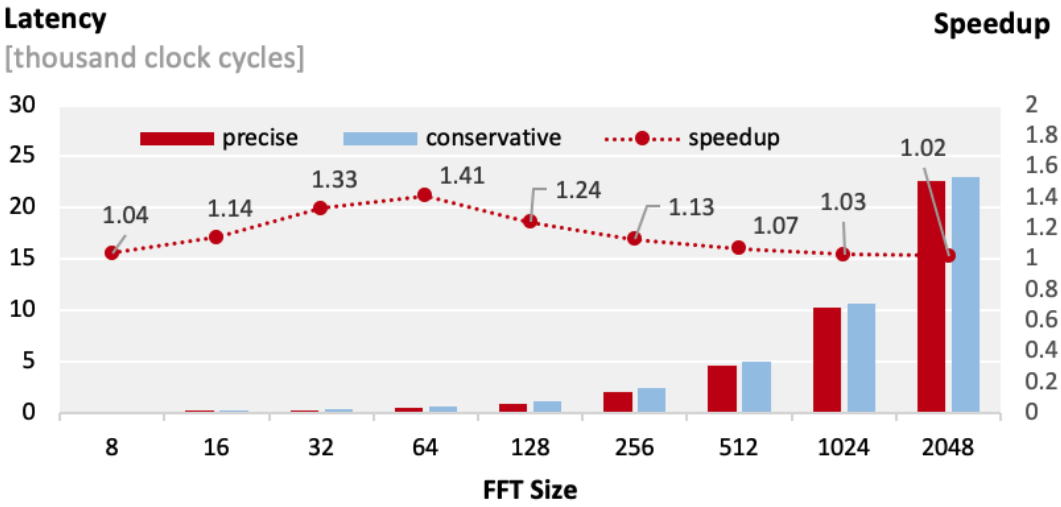



Figure 5.2: The latency comparison between two dependency management strategies for iterative FFTs.

Though the theoretical speedup upper bound can not exceed 2x as analyzed in Chapter 3 and only happens for problem sizes at the same scale of the fixed pipeline latency, this optimization is still beneficial for two reasons. First, the analysis is performed statically and does not incur additional hardware implementation cost. Second, because the future throughput scaling of the architecture will reduce the number of independent iterations for the same problem sizes, the problem sizes that obtain the most benefit from this optimization will also increase.

### 5.1.3 Buffer Utilization

The effect of the proposed buffer allocation schemes in Section 4.2.1 is evaluated by compared against the baseline buffer allocation scheme of current SPIRAL for WHTs, FFTs and bitonic sorters. In the proposed scheme, three techniques are employed, with increasing requirements of the program properties. To evaluate each technique, three implementations have been created that employs the three

techniques accumulatively. The descriptions of the implementations are listed in Table 5.1.

Table 5.1: The implementations of four buffer allocation schemes

| Name | Techniques employed (accumulated) |
|---|---|
| baseline | The scheme used in SPIRAL software generation. |
| hierarchical | Reuses deep level intermediate buffers hierarchically. |
| swapping | swaps uniform size buffers. |
| overwritten | allows overwriting the data of the input buffer. |

The different buffer allocation schemes can lead to various memory utilization. In all experiments, the result of the implemented scheme is normalized to the baseline scheme.

Figure 5.3 presents the allocation scheme comparison for iterative FFTs across the size from 8 to 1024. The iterative FFT algorithm is composed of several loop stages accessing the full data. Furthermore, in-place calculations exist in all but the first stage, thus the intermediate buffers can be replaced by the output buffer in all implementations. This result can not be improved with all three techniques proposed in this work. Consequently, the four implementations achieve the same memory utilization.

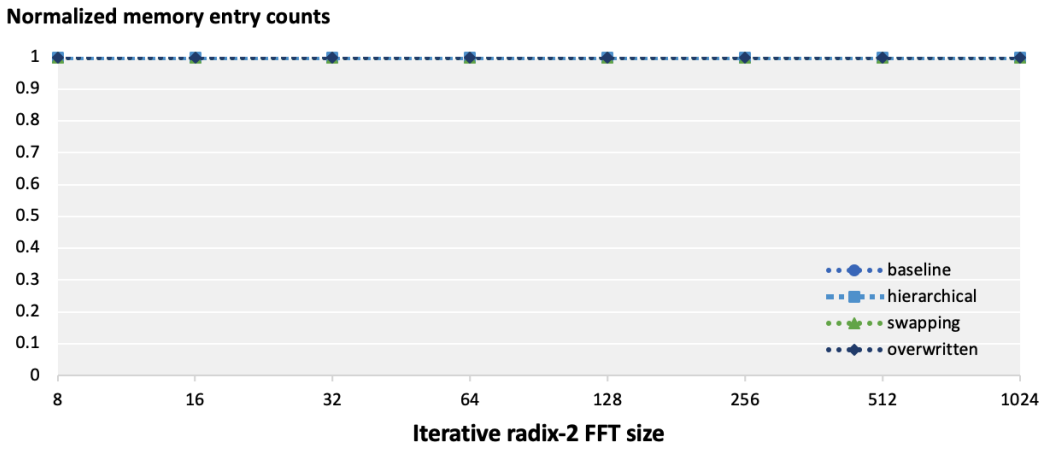



Figure 5.3: The comparison of buffer utilization between different allocation strategies for iterative FFTs.

Figure 5.4 presents the comparison for the recursive Cooley-Tukey algorithms of WHT and FFT from size-8 to size-1024. The recursive algorithms are numerous which all produce deeply nested loop programs. The algorithm decomposition that tends to create balanced rule trees is selected. In this plot, the *hierarchical* and *swapping* overlap perfectly. The benefit of the *hierarchical* implementation is moderate because at each level, the computation is factorized into two stages of much smaller sizes which derives small intermediate buffer sizes. The swapping technique failed to reduce the memory utilization further because in the factorization into two stages with a smaller size intermediate buffer, swapping does not apply. The *overwritten* implementation reduces the memory utilization from almost 1/3 by allowing the data of the input buffer to be overwritten. This removes the largest intermediate buffer the same as the problem size in the top-level decomposition while preserving the input and output buffer.

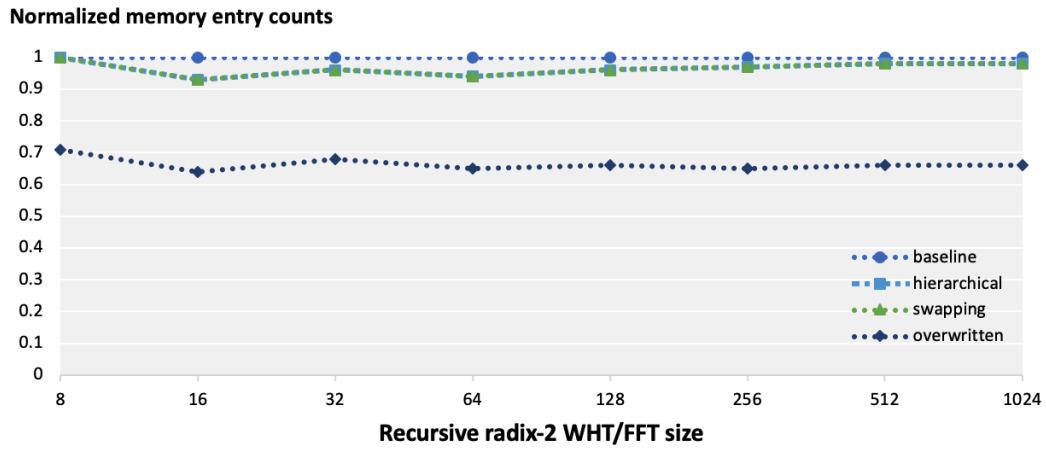



Figure 5.4: The comparison of buffer utilization between different allocation strategies for recursive WHTs/FFTs.

Figure 5.5 shows the comparison for the bitonic sorter algorithm from size-8 to size-1024. This is a recursive algorithm where each breakdown step factorizes the computation into three stages with the middle one being in-place. The plot shows overlapped scatter dots for *hierarchical* and *swapping*, both of which reduces the memory utilization considerably. The benefit of the *hierarchical* scheme comes from the deep nested level that can reuse intermediate buffers. The *swapping* scheme failed to reduce the utilization further because the input buffer at each level are all the input buffer of the program, which prohibits swapping. When overwriting the input buffer is allowed in the *overwritten* scheme, the memory utilization continues to drop by about 20%.

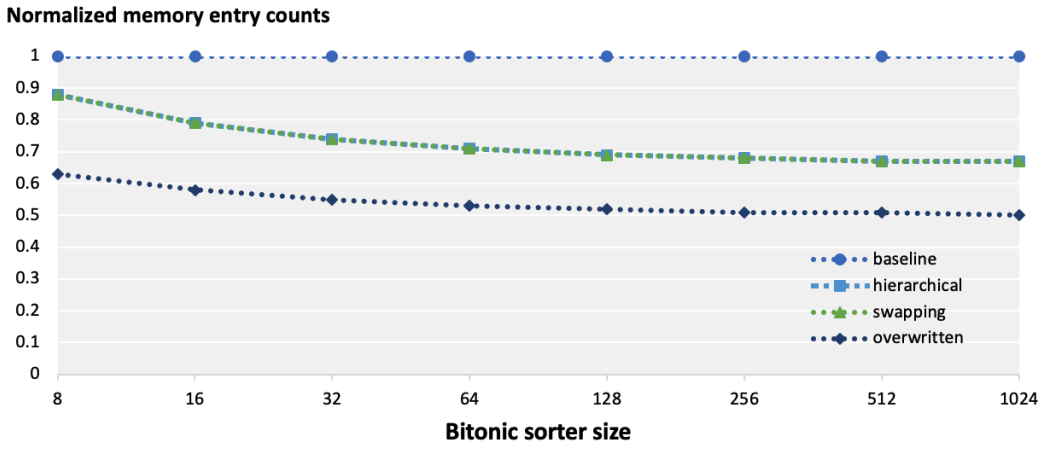



Figure 5.5: The comparison of buffer utilization between different allocation strategies for bitonic sorters.

### 5.1.4 Optimizing Multi-linear Expressions

Then, Table 5.2 5.3 5.4 5.5 show the accumulated bits of the multiplication and addition operations used for calculating the multi-linear expressions for a range of problem sizes of WHT, DFT and bitonic sorters, implemented with three different optimizations. Table 5.2 5.3 5.4 show the similar results for radix-2 algorithms of WHT, DFT and bitonic sorter. They all succeessfully remove all multipliers and adders because the constant factors of the multi-linear expressions of radix-2 algorithms are all power-of-2 numbers, which can be implemented through bit mappings.

Figure 5.5 shows the result for the radix-3 FFT where most constant factors are not power-of-2 numbers unless the potential constant factor 1. In this situation, the bit mapping scheme optimizes the inductive calculation implementation slightly by replacing the accumulations of stride-1 into bit mappings.

Table 5.2: The comparison of total arithmetic bits between different implementations of multi-linear expressions for recursive radix-2 WHTs.

| Problem | direct | | induction | | bitmap | |
|---|---|---|---|---|---|---|
| size | mul | add | mul | add | mul | add |
| 8 | 16 | 7 | 0 | 12 | 0 | 0 |
| 16 | 28 | 19 | 0 | 27 | 0 | 0 |
| 32 | 42 | 27 | 0 | 42 | 0 | 0 |
| 64 | 59 | 35 | 0 | 62 | 0 | 0 |
| 128 | 76 | 45 | 0 | 86 | 0 | 0 |
| 256 | 96 | 55 | 0 | 113 | 0 | 0 |

Table 5.3: The comparison of total arithmetic bits between different implementations of multi-linear expressions for iterative radix-2 FFTs.

| Problem | direct | | induction | | bitmap | |
|---|---|---|---|---|---|---|
| size | mul | add | mul | add | mul | add |
| 8 | 21 | 15 | 0 | 18 | 0 | 0 |
| 16 | 53 | 29 | 0 | 45 | 0 | 0 |
| 32 | 97 | 47 | 0 | 83 | 0 | 0 |
| 64 | 153 | 69 | 0 | 132 | 0 | 0 |
| 128 | 221 | 95 | 0 | 192 | 0 | 0 |
| 256 | 301 | 125 | 0 | 263 | 0 | 0 |

Table 5.4: The comparison of total arithmetic bits between different implementations of multi-linear expressions for Bitonic sorters.

| Problem | direct | | induction | | bitmap | |
|---|---|---|---|---|---|---|
| size | mul | add | mul | add | mul | add |
| 8 | 32 | 16 | 0 | 28 | 0 | 0 |
| 16 | 70 | 32 | 0 | 72 | 0 | 0 |
| 32 | 128 | 55 | 0 | 142 | 0 | 0 |
| 64 | 210 | 86 | 0 | 243 | 0 | 0 |
| 128 | 320 | 126 | 0 | 380 | 0 | 0 |
| 256 | 462 | 176 | 0 | 558 | 0 | 0 |

Table 5.5: The comparison of total arithmetic bits between different implementations of multi-linear expressions for iterative radix-3 FFTs.

| Problem | direct | | induction | | bitmap | |
|---|---|---|---|---|---|---|
| size | mul | add | mul | add | mul | add |
| 27 | 39 | 20 | 0 | 32 | 0 | 24 |
| 81 | 95 | 41 | 0 | 75 | 0 | 64 |
| 243 | 165 | 70 | 0 | 136 | 0 | 122 |
| 729 | 260 | 100 | 0 | 218 | 0 | 201 |
| 2187 | 374 | 143 | 0 | 318 | 0 | 298 |
| 6561 | 503 | 184 | 0 | 431 | 0 | 407 |

## 5.2 Quality of Designs: FFTs on FPGA

This section evaluates the quality of the generated designs on an FPGA device using the proposed approach in a case study of FFT. The cycle counts, peak frequency and resource utilization are compared against a state-of-the-art implementation. Then the controller design of the proposed approach is investigated for the peak frequency and resource utilization of a range of problem sizes.

### 5.2.1 Methodology

The iterative radix-2 FFT algorithm is chosen to produce the FFT core designs for evaluation. Note that this work has focused on the scalar load-store architecture which provides a data rate of one data word per cycle. The generated design matches the operation rate to the data rate. The generated Chisel RTL designs of the proposed approach is encapsulated in an accelerator wrapper that initializes the input data in memory through the input data ports and retrieves the result data from memory via the output data ports. The overall Chisel RTL code is translated to Verilog RTL code through the Chisel compiler.

The baseline designs are generated by the Discrete Fourier Transform Verilog

IP Generator [46] (hereafter referred to as DFTgen) based on the current SPIRAL hardware backend technology. Since DFTgen scales the architecture better than the current implementation of the proposed approach, an architecture configuration is chosen to generate a design with scalar throughput: iterative architecture, radix-2, streaming width of 1. The bit-permutation method is selected. In the web interface of DFTgen, the minimal streaming width is two. Hence, the design using streaming width of 1 is obtained by using the backend interface of DFTgen with the help from the author. The obtained design is depicted in Figure 5.6. The design is composed of three main components, the twiddle factor module, the half-rate F(2) module, and a temporal permutation module. The tool automatically generate these modules to provide streaming rate of one word per cycle by using reasonable resources. In this scenario, the permutation module is merely a RAM that is written and read through indices generated dynamically with a function.

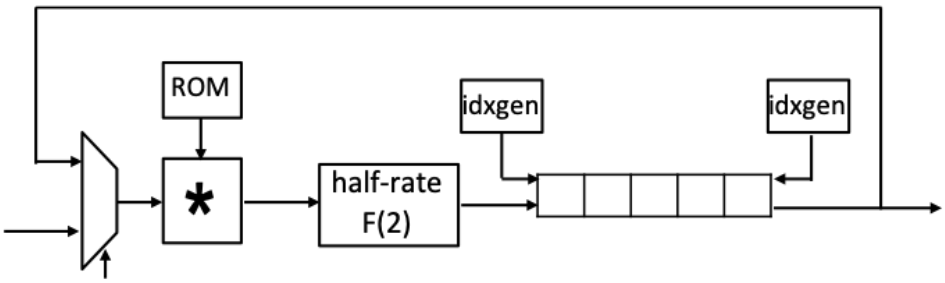



Figure 5.6: The baseline design from DFTgen.

Other configurations are employed to improve the similarity between the designs from two approaches. The single precision floating point data format is used for both approaches. The floating point arithmetic operators in the proposed approach are mapped to the same RTL modules instantiated by DFTgen designs. Both designs maps the memory and the twiddle factor lookup table to the block

RAM resources of the FPGA.

The FPGA experiments done target the Xilinx Virtex-7 xc7z045ffg900-2 FPGA. All FPGA synthesis is performed using Xilinx Vivado 2016.2, and the area and timing data shown in the results are extracted after the final place and route are complete.

**Latency.** The latency of the designs produced with the proposed method is obtained through RTL simulation, as described in Section 5.2.1. The latency of DFTgen designs is reported by the generator and is defined as the lapse clock cycles between the time the first input data sample is streamed into the FFT core and the time when the first output data sample is streamed out of the FFT core.

**FPGA resource metric.** The FPGA resource metric is described by the utilization of four types of resources, namely the lookup tables (LUTs), the flip-flops (FFs), the DSPs, the block RAMs (BRAMs). The LUTs are the general reconfigurable resources on FPGA, which can simulate the arbitrary gate logic through configuring the bits of the lookup tables. The FFs are register resources provided by the FPGA device. The Xilinx Virtex-7 FPGAs contain dedicated arithmetic units called DSP slices. DSP slices contain hard multipliers, accumulators, registers, and interconnect. The multipliers in these slices are used in floating point multiplication. The single precision floating point multiplier each use two DSP slices. The floating point adder is mapped to LUTs and FFs. Xilinx FPGAs provide two types of memories: BRAM or distributed RAM. BRAMs are 36kb dedicated hard memories built into the FPGA. The Virtex-7 contains 545 BRAMs. Memory structures can also be constructed with the FPGA's LUTs which are normally used as logic. In our experiments, all the RAMs and ROMs are mapped to BRAM resource.

### 5.2.2 Latency

Figure 5.7 shows the latency comparison between the FFT cores of our approach and of DFTgen. For the FFT size range from 16 to 2048, this work is in average 15% faster than DFTgen. The key reason is in the algorithm. While the initial bit reversal permutation is separated from the $\log N$ FFT stages in DFTgen, it is merged in our approach to the FFT stages. The secondary reason is across-stage execution overlapping, as evaluated in Section 5.2.2, which is presented by the more speedup of smaller sizes.

Note that the latency in clock cycles is not the ultimate metric to compare the execution speed because the final performance also depends on the clock frequency. We will evaluate the frequency latter in this section.

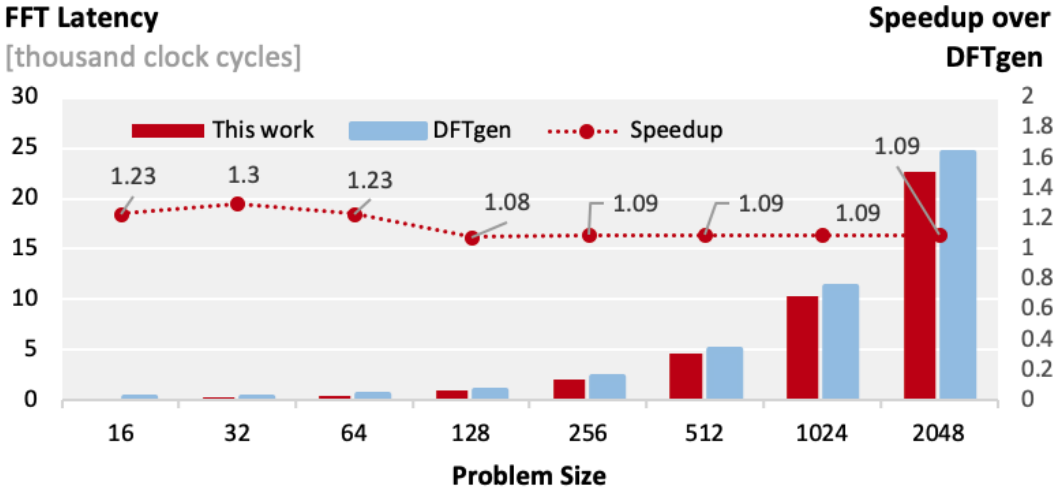



Figure 5.7: This work vs. DFTgen: latency comparison.

### 5.2.3 Resource Utilization

Figure 5.8 5.9 5.10 show the resource utilization of LUTs, FFs and BRAMs on Xilinx FPGA for FFT sizes ranging from 128 to 2048. The primary axis presents

the number of resources for the this work and DFTgen. The secondary axis presents the ratio of this work over DFTgen. This work uses averagely 14% more LUTs than DFTgen. The ratio increases slowly as the FFT sizes go larger. Since the datapath of different sizes stays mostly the same for both designs, it is possibly due to the implementation cost of the controller used in our approach. This work uses averagely 5% more FFs than DFTgen with similar trend as LUT utilization with respect to the problem sizes possibly due to the same reason. In contrast, the proposed approach utilize fewer BRAMs, in average 70% of DFTgen. The BRAMs are used exclusively for the memory modules and the twiddle factor lookup tables in both approaches. The twiddle factor lookup table are at the same size in both approaches. In the proposed approach, the required memory entires are twice of the problem sizes. In DFTgen, the number is fourfold of the problem sizes because the initial permutation module does not share the memory with the stride permutation module in the FFT compute stage.

It is worth noting that newer literatures have reported extra optimizations to the permutation cores [47] and their use in streaming FFTs [48]. In particular, [48] reports half of the RAM bank utilization compared to the DFTgen baseline used in this study, by sharing the memory block between bit-reversal permutation and stride permutation. This matches the number in our load-store architecture design. However, while it is a native design with a load-store architecture, applying the sharing technique for streaming designs requires special rewiring of the streaming blocks with extra control. Besides, the optimization [48] does not change the cycle counts studied in the previous sub-section because the bit reversal permutation is still treated as a separate stage. Nonetheless, more careful study is needed to systematically compare the two different approaches.


Figure 5.8: This work vs. DFTgen: lookup tables comparison.


Figure 5.9: This work vs. DFTgen: flip-flops comparison.

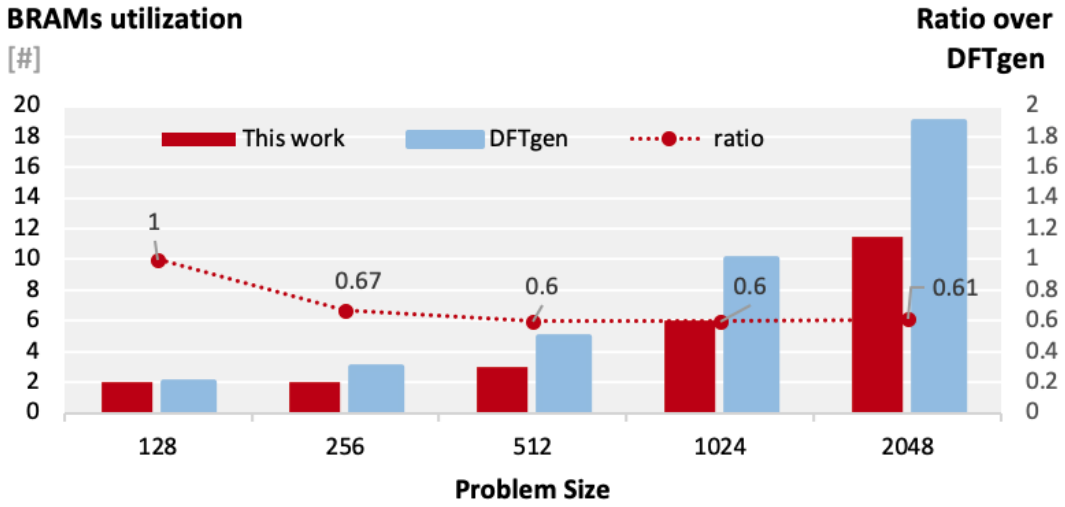



Figure 5.10: This work vs. DFTgen: block RAMs comparison.

### 5.2.4 Peak Frequency

The DFTgen designs present decent scalability in maintaining the peak frequency near 455 MHz for FFT sizes ranging from 128 to 2048. Figure 5.11 shows the peak frequency of the proposed approach, which degrades from 286 MHz to 250 MHz across the size range from 128 to 2048. The analysis of the critical path shows that the the longest path connects the hardware operators and the root compose FSM of the controller. The path travels from the perfect sub-nest FSM up to the root compose FSM because the design requires the fast response of the complete signal to start the next iteration. Thus, the peak frequency can be improved in two ways. First, the proper use of elastic buffers can truly decouple the critical path between hardware operators and reduce the length of critical path in each size. Second, The controller design can be improved by either reducing FSM levels. To achieve the peak frequency of DFTgen, a different controller structure is possibly needed.

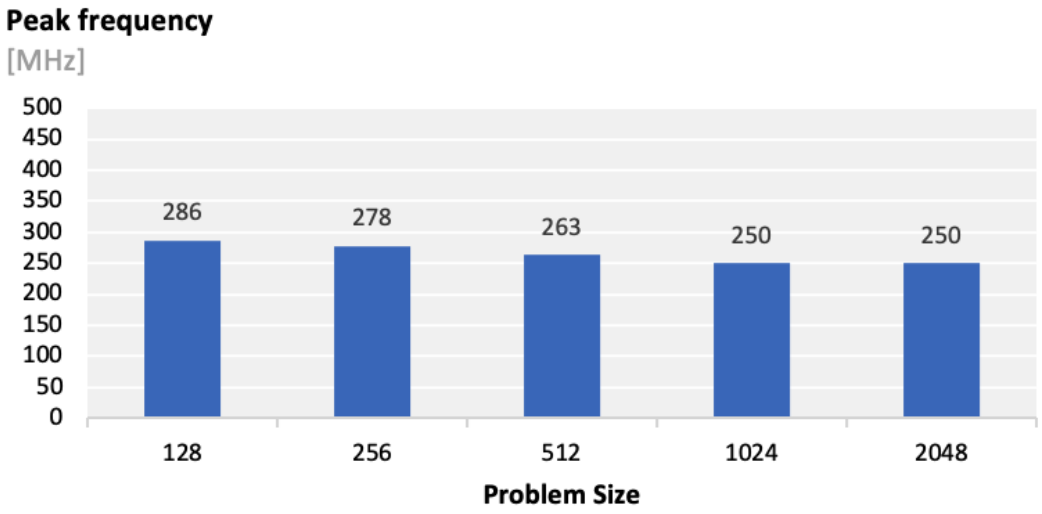



Figure 5.11: The peak frequency on FPGAs for FFTs.

### 5.2.5 Controller Cost

The peak frequency evaluation has exposed a shortcoming of the controller design. Figure 5.12 further presents the resource utilization of the controller. The Y-axis is the percentage of resources utilization of the controller over the overall design. Because the controller does not utilize BRAMs and DSPs, only the LUTs and FFs utilizations are shown. The scatter plots present the steady increase of utilizations of the two resource types while the FFT sizes increase. At FFT size of 2048, the controller utilize 19% LUTs and 6% FFs of the overall design.

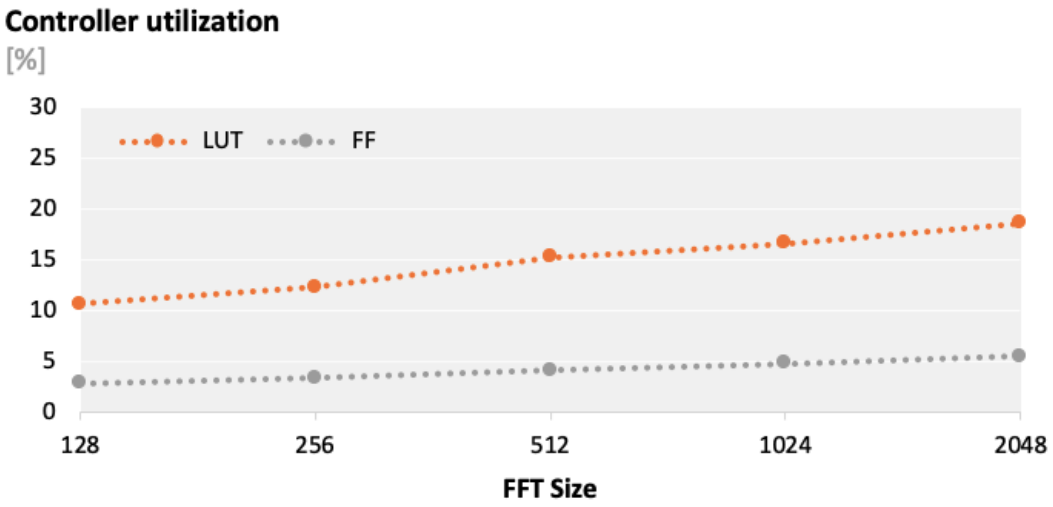



Figure 5.12: The resource cost of controllers for FFTs.

## 5.3 Summary

This chapter evaluates the effectiveness of the proposed approach for load-store architecture generation. It first reports the effectiveness of the hardware optimizations performed in the Σ-OL level, by comparing the performance and resource utilizations with and without high-level optimizations. It is presented that the pattern-based optimization improve the design quality significantly. Then, a detailed comparison between the FPGA implementations of the FFT cores generated by the proposed method and by the existing hardware backend are performed. The comparable result suggest that the flexible load-store architecture can be optimized well with enough domain-specific efforts.

# Chapter 6

# Concluding Remarks

The conventional wisdom of creating hardware accelerators is targeting a specific algorithm through deeply customizing the control logic and datapath for hardware so as to maximize efficiency. This leads to difficulties in reusing the architecture design for other algorithms. While the load-store architecture is inherently flexible to algorithms, it can incur considerable overhead in a generic design. The contribution of this dissertation is to propose an approach to reason about desirable properties in algorithms and computations before implementing the actual load-store architecture to enable hardware acceleration of a wide scope of algorithms. In this way, the flexibility is carefully realized by having a higher level view of hardware designs, and by using the properties of computations to customize the load-store architecture.

## 6.1 Lessons Learned

This work showed us that the flexible load-store architecture can be practically used for efficient hardware designs when specialization is addressed hierarchically from algorithm generation to program optimization to hardware interpretation. In this

way, the datapath can be tailored for efficient use of the given algorithm and the considerable supporting circuits for a generically programmable load-store architecture can be replaced by simpler and more specific mechanisms. Since this requires representing, analyzing and manipulating algorithms, programs, and hardware modules, multi-level DSLs are essential to drive this approach. SPIRAL provides an extensible framework to implement the proposed approach and the extensions we have made mark a significant first step to open up the full power of SPIRAL for automated hardware design generation.

We saw that, the constraint solver of SPIRAL is used for algorithm-hardware co-synthesis for load-store architecture. The computational specifications addressed by SPIRAL possess recursive or iterative nature, thus can be expanded for various algorithms. In the software generation flow of Spiral, the fixed hardware architecture directs the algorithm expansion process to assure efficient execution on hardware. In the proposed approach, the abundant architectural paradigms including but not limited to parallelism styles and memory organizations, as well as the numerous implementation options of hardware can be considered to direct the algorithm generation process. The space of valid combinations of hardware features is larger than that provided by commodity processors, thus may trigger new ideas in algorithm designs. The algorithm-hardware co-synthesis process will allow us to explore Pareto-optimal designs across a large tradeoff space between performance and resource utilization.

We saw that, by extending the Σ-OL language for capturing the desired properties of the compute pattern of imperfect loop nest programs, the latency, memory utilization and indices computation cost of the resulting hardware can be significantly optimized. In this work, we have considered the following properties:

- Fixed loop bounds
- Multi-linear indices for gather and scatter operations
- Identical parameterized kernel operations.

While the above properties constrain us to the computations of certain important kernels, relaxation is possible with domain-specific analysis or more advanced hardware compilation process.

We saw that, the icode extensions modeling the interconnected RTL modules can flexibly describe hardware implementations. The connections implemented as raw digital signals or in ready/valid protocol are both supported. Each icode operator modeling an RTL module can be treated as a code generator and can customize itself with respect to the input characteristics. This also allows external hardware generators to be incorporated into the SPIRAL framework.

## 6.2 Next Step: Parallelizing Load-store Architectures

Parallelism is crucially important to scale the compute throughput of load-store architectures. The parallelization of customized load-store architectures is highly dependent on computations. This chapter discusses how the multi-linear compute pattern of imperfectly nested loop programs can be parallelized in the form of vector parallelism, symmetric multi-processing and short-vector SIMD parallelism.

The compute pattern studied in this thesis possesses certain properties that make automated parallel hardware generation possible with SPIRAL. First, it ensures all independent operations in the perfect sub-nest of imperfect loop nest programs that can be computed in parallel. Second, the multi-linear access pattern to memory

can be transformed for various types of parallelism. Moreover, the hardware generation flow introduced in Chapter 4 is compatible with the prior parallelization work in SPIRAL for general purpose processors. Thus, high performance parallel hardware implementations using customized load-store architecture can be synthesized with the SPIRAL framework.

### 6.2.1 Vector Parallelism

The vector parallelism in processor design means essentially pipelined execution of many data operations [49]. This behavior is not attainable in typical pipelined instruction set processors because the data movement operations and computations are encoded in the same instruction stream such that data movements never coincide with computations. The vector processors, represented by the Cray processor [36], achieve simultaneous data movement and computation by exploiting the fact that in vector computations the same operation is applied to every elements of a vector.

Though counter-intuitively by its name, the customized scalar load-store architecture introduced in Section 4.4 has inherently achieved the same effect as vector processors by architecting for the multi-linear compute pattern. The scalar algorithm implemented on the architecture can already provide the compute throughput of one word per clock period, when connecting to a dual-ported fast on-chip memory. When interfacing with more complicated memory systems, the necessary algorithms can be generated with the SPIRAL framework for high performance.

#### Inherent Vector Processing

The customized scalar load-store architecture can inherently achieve the processing rate at one word per clock period as typical vector processors. Both architectures

exploits the homogeneity of computations though with varying degrees. Specialization, however, enables more complicated operations on vector elements than in vector processors.

In Cray-like vector processors, the high throughput of vector computation is obtained through vector instructions and the chaining mechanism in execution. A vector load/store instruction moves entire elements of a vector between a vector register and the memory. A vector arithmetic/logical instruction performs operations on all elements in vector registers. In hardware design, both vector movement units and functional units are fully pipelined and allow one element to be processed per clock period. The chaining technique allows a vector instruction to proceed as soon as an element in the source register is produced by a previous vector instruction. In this way, the vector registers act essentially like FIFOs and allow the continuous streaming between pipelined hardware units.

In contrast, the customized scalar load-store architecture achieves the same throughput easier thanks to the compute pattern it is designed for. In the compute pattern of imperfect loop nest programs, the basic block operation is always of a gather-compute-scatter shape. It means that, for every iteration, a vector must be loaded from memory, processed through an arbitrary kernel operation, then the resultant vector is stored back to memory. As a result, designing a pipeline physically chaining the load, compute, and store units is a natural decision. The pipeline is driven by an FSM for the base and stride of load/store indices and other necessary parameters for computation. As long as the parameters are supplied continuously from the FSM, the pipeline is capable of process one element per clock period. Hence, the customized scalar load-store architecture is reminiscent to the Cray-like vector processors.

However, specialization enables more complexity in vector processing than vector processors. In Cray-like vector processors, the feasible vector computation is constrained by the limited number of vector registers and vector functional units. Consequently, complicated computations, going beyond the intermediate storage capacities of vector registers, have to be decomposed to multiple data passes between memory. Moreover, the vector functional units always perform identical operation to each element, prohibiting the computation between elements. In contrast, a customized functional unit can enable arbitrary operation for each vector element and can buffer a few elements to compute results with multiple vector elements.

### Vectorized Algorithm Generation

While the customized scalar load-store architecture provides inherent vector processing throughput, the actually attainable performance highly relies on the memory system it connects with. Modern main memory, such as DRAMs, incurs long access latency that can be partially hidden through accessing a continuous data block. Deep memory hierarchy employing multi-level memories with different latency is common to modern computers, which requires data locality in programs for memory performance improvements. The SPIRAL framework handles these scenarios and has provided useful rules in the rewrite system for program transformation toward blocking access.

In SPIRAL, the blocking access pattern is captured by OL formula $A_m \otimes I_n$, where $n$ continuous data items are fetched in computation. In hardware execution of the customized scalar load-store architecture, it means $n$ consecutive operations to be executed in the pipeline. By using $\vec{\otimes}$ operator to denote this special pipelined operations, and introducing a vectorization tag 'vec', the necessary rules for trans-

forming fundamental OL operators and formulas for vectorization are listed (6.1) - (6.3).

$$\underbrace{A_n \otimes \mathrm{I}_m}_{\text{vec}} \rightarrow A_n \vec{\otimes} \mathrm{I}_m \tag{6.1}$$

$$\underbrace{\mathrm{I}_m \otimes A_n}_{\text{vec}} \rightarrow \underbrace{\mathrm{L}^{mn}_m}_{\text{vec}} \left( A_n \vec{\otimes} \mathrm{I}_m \right) \underbrace{\mathrm{L}^{mn}_n}_{\text{vec}} \tag{6.2}$$

$$\underbrace{\mathrm{L}^{mn}_m}_{\text{vec}} \rightarrow \left( \mathrm{L}^{mn/\nu}_m \vec{\otimes} \mathrm{I}_\nu \right) \left( \mathrm{I}_{mn/\nu^2} \otimes \underbrace{\mathrm{L}^{\nu^2}_\nu}_{\text{vec}} \right) \left( \left( \mathrm{I}_{n/\nu} \otimes \mathrm{L}^m_{m/\nu} \right) \vec{\otimes} \mathrm{I}_\nu \right), \quad \nu \mid m, n \tag{6.3}$$

In (6.1), the tagged OL formula with the exact shape is converted to the identical form by assigning the $\vec{\otimes}$ operator and dropping the tag. In (6.2), the block parallel OL formula is converted to vectorized shape with additional stride permutations. In (6.3), the stride permutation is converted to block stride permutations of vector length $\nu$, producing an irreducible permutation $L^{\nu^2}_v$ which can be processed by a customized permutation functional unit.

### 6.2.2 Symmetric Multi-processing Parallelism

In symmetric multi-processing (SMP), the scalar core is replicated multiple times and share a larger memory in between. In general SMP processors, cache memory is typically employed to move data automatically between the private memory of cores. Ideally, the compute task is evenly distributed in each core, accessing data resided in the local cache. When inter-core communication is necessary, the data is better to be transferred in cache-line granularity to avoid false sharing.

The multi-linear compute paradigm is SMP parallelizable by having abun-

dant independent iterations in the perfect sub-nest. For obtaining high performance from an SMP architecture, existing rewrite rules in SPIRAL can handle the code transformation issues. When building SMP parallelism for customized load-store architecture, certain hardware blocks are required but the complicated ones can be potentially generated with external tools.

### Interpreting Algorithms for SMP Parallelism

The compute pattern studied in this thesis possesses rich independent iterations in the perfect sub-nest. As seen in classic loop parallelization tricks, the perfect sub-nest can be re-organized to have an outermost loop with the loop count equivalent to the prescribed number of cores. Then each iteration of the outermost loop is mapped to unique execution core. At each core, data required by the corresponding outermost loop iteration is loaded from the shared memory, computed locally, then the result is stored back to the shared memory.

However, the simple code transformation at loop level does not necessarily bring higher performance. Though already load-balanced, the different cores can possibly access different data stored in the same cache line. This causes the false-sharing problem and can trigger substantial cache coherence traffic and thus incurs serious performance penalty. The next subsubsection explains the existing rewrite rules that can be used in SPIRAL for solving the problem.

### SMP Algorithm Generation

The existing SMP rewrite rules in SPIRAL handle the load-balancing problem and avoids false sharing when parallelizing OL operators. The SMP platform is captured in two parameters: $p$ the number of cores, and $\mu$ the cache line length. An $\text{smp}(p, \mu)$ tag denotes OL formulas to be parallelized. After the transformation, the load-

balanced parallel formula is denoted by the $\otimes_{\parallel}$ operator, and the data permutation avoiding false-sharing is denoted by the $\bar{\otimes}$ operator. The relevant rules are shown (6.4) - (6.7).

$$\underbrace{\mathrm{I}_p \otimes A_n}_{\mathrm{smp}(p,\mu)} \rightarrow \mathrm{I}_p \otimes_{\parallel} A_n. \tag{6.4}$$

$$\underbrace{P \otimes \mathrm{I}_\mu}_{\mathrm{smp}(p,\mu)} \rightarrow P \bar{\otimes} \mathrm{I}_\mu. \tag{6.5}$$

$$\underbrace{A_m \otimes \mathrm{I}_n}_{\mathrm{smp}(p,\mu)} \rightarrow \Big( \left(\mathrm{L}^{mp}_m \otimes \mathrm{I}_{n/p\mu}\right) \bar{\otimes} \mathrm{I}_\mu \Big) \left(\mathrm{I}_p \otimes_{\parallel} \left(A_m \otimes \mathrm{I}_{n/p}\right)\right) \Big( \left(\mathrm{L}^{mp}_p \otimes \mathrm{I}_{n/p\mu}\right) \bar{\otimes} \mathrm{I}_\mu \Big), \quad \mu \mid n/p \tag{6.6}$$

$$\underbrace{L^{mk}_k}_{\mathrm{smp}(p,\mu)} \rightarrow \Big(I_p \otimes_{\parallel} L^{mk/p}_{k/p}\Big)\Big(\left(L^{pm}_p \otimes I_{k/pu}\right) \bar{\otimes} I_u\Big) \tag{6.7}$$

In (6.4), the tagged OL formula with the exact shape is converted to the identical form by assigning the $\otimes_{\parallel}$ operator and dropping the tag. In (6.5), block stride permutation is converted to the identical form by assigning the $\bar{\otimes}$ operator and dropping the tag. In (6.6), the vectorizable OL formula is converted to parallelizable shape with additional stride permutations. In (6.7), the stride permutation is converted to block stride permutations of cache line length $\mu$ and the parallelized localized stride permutation.

### Hardware Building Blocks

The SMP architecture requires cache memory and the interconnect network between the cores. These are complicated hardware building blocks that require high

development cost if by hand. The Rocket Chip generator [33] includes components for generating a shared memory hierarchy of coherent caches interconnected with on-chip networks. Configuration options include the number of tiles, the coherence policy, the presence of a shared L2 cache, the number of memory channels, the number of cache banks per memory channel, and the implementation of the underlying physical networks. These generators are all based around TileLink, a protocol framework for describing a set of cache coherence transactions that implement a particular cache coherence policy.

### 6.2.3 Short-vector SIMD Parallelism

A short vector architecture is an economical way to scale the performance of scalar architecture when multiple banks of fast memory such as SRAMs are available. In a short vector architecture, the vector core, with a vector load/store unit and a vector functional unit, is connected to multiple memory banks, each of which supplies a scalar data word to the core. The vector lanes parameter specifies the number of data items that can be transferred and processed simultaneously. Similar to the customized scalar architecture, a short vector architecture can be made a memory-memory architecture thanks to functional unit specialization. Figure 3.5a shows a two-lane short-vector design of specialized load-store architecture. The vector functional unit includes a vector arithmetic / logic unit and a permutation unit. The permutation unit handles data movements between vector lanes and can further permute a larger data set by having an internal storage buffer.

### SIMD Algorithm Generation

The existing SIMD rewrite rules in SPIRAL handle the SIMD vectorization problem. The SIMD platform is captured in parameter $\nu$ for the vector length. An simd($\nu$) tag denotes OL formulas to be SIMD vectorized. After the transformation, the SIMD vectorized formula is denoted by the $\otimes_{\|}$ operator, and the data permutation avoiding false-sharing is denoted by the $\vec{\otimes}$ operator. The relevant rules are shown in (6.8) - (6.10).

$$\underbrace{A_n \otimes \mathrm{I}_\nu}_{\mathrm{vec}(\nu)} \rightarrow A_n \vec{\otimes} \mathrm{I}_\nu \tag{6.8}$$

$$\underbrace{\mathrm{I}_\nu \otimes A_n}_{\mathrm{vec}(\nu)} \rightarrow \left(L^n_v \vec{\otimes} I_v\right) \left(I_{n/v} \otimes \underbrace{L^{v^2}_v}_{\mathrm{vec}(\nu)}\right) \left(\mathrm{A}_n \vec{\otimes} I_v\right) \left(\mathrm{I}_{n/\nu} \otimes \underbrace{\mathrm{L}^{\nu^2}_\nu}_{\mathrm{vec}(\nu)}\right) \left(\mathrm{L}^n_{n/\nu} \vec{\otimes} \mathrm{I}_\nu\right) \tag{6.9}$$

$$\underbrace{\mathrm{L}^{mn}_m}_{\mathrm{vec}(\nu)} \rightarrow \left(\mathrm{L}^{mn/\nu}_m \vec{\otimes} \mathrm{I}_\nu\right) \left(\mathrm{I}_{mn/\nu^2} \otimes \underbrace{\mathrm{L}^{\nu^2}_\nu}_{\mathrm{vec}(\nu)}\right) \left(\left(\mathrm{I}_{n/\nu} \otimes \mathrm{L}^m_{m/\nu}\right) \vec{\otimes} \mathrm{I}_\nu\right), \quad \nu \mid m, n \tag{6.10}$$

In (6.8), the tagged OL formula with the exact shape is converted to the identical form by assigning the $\vec{\otimes}$ operator and dropping the tag. In (6.9), the block parallel OL formula is converted to vectorized shape with additional vectorized stride permutations. In (6.10), the stride permutation is converted to block stride permutations of vector length $\nu$. Note that (6.9) and (6.10) produces irreducible permutation $\mathrm{L}^{\nu^2}_\nu$ that must be handle as the base case.

### Hardware Building Blocks

The vectorization rules produce irreducible permutations of the size of $v^2$ where $v$ is the vector length. When generating software implementations for SIMD microprocessors, they are implemented as in-register shuffle instructions that are supported by the pre-built networks between vector registers. When it comes to specialized load-store architectures, specific networks between vector registers dedicated to certain algorithms are preferred.

The permutation operations for data size at multiples of vector length can be implemented as streamed permutations. In this way, the input as a size-$v$ data chunk is streamed from memory through vector loads, then flowed along the streamed permutation unit, and finally streamed back to memory through vector stores. Since data streamed in earlier may not be allowed to streamed out in time, a memory buffer is required in the streamed permutation unit. Such a permutation unit can be generated with the bit matrix method described in [50] when the data size is power of two, and the satisfiability approach in [16] for arbitrary data size.

In this section, we looked at the parallelization problems of the customized load-store architecture for the multi-linear compute pattern. The properties of the compute pattern makes it easy to be interpreted to the vector parallel architecture and the SMP architecture, while the SIMD parallel architecture requires substantial effort for assuring SIMD vectorizable algorithms. The existing parallelization rules are compatible to the hardware generation flow and thus can be applied for parallel customized load-store architecture synthesis. For all parallel paradigms discussed in this chapter, SPIRAL rewrite rules can assure high performance in implementations by shaping the algorithm at high level. New hardware building blocks are

required in SMP parallelism and SIMD parallelism, while the complicated ones can be automatically generated with existing tools.

## 6.3 Future Work

This thesis concludes with a brief discussion of some possible future extensions of this work. First, by embedding a new specialized architecture to the SPIRAL framework, extra extensions could be required to realize the full framework capability. In addition, the compute pattern discussed in this thesis could be extended to support more complicated computations. Moreover, diversified hardware implementation options could be incorporated to this framework to explore a larger design space. Finally, the combination of domain-knowledge and hardware designs may allow some difficult hardware compilation problems to be addressed in a specific context.

**Resource modeling.** To explore the design space regarding the cost/performance tradeoff, SPIRAL must obtain the quantitive metrics of the hardware implementation cost and the execution performance. However, the time required for measuring hardware implementations is typically in units of hours if not days, thus imposes great challenges in searching for the Pareto-optimal designs. To reduce the metric retrieval time, estimations through modeling have been employed in the previous streaming architecture generation effort in Spiral. The resource estimation turns out to be more difficult than performance estimation, because in highly optimized hardware the full utilization of resources allows the performance to be calculated with simple functions of design configurations.

In the previous effort in generating streaming hardware implementations with Spiral, several solutions addressing the long synthesis time obstacles for design space exploration have been proposed. However, porting them to the more complicated

load-store architecture designs is non-trivial. In [51], an exact model for the DFT streaming cores is developed to estimate the resource utilization in FPGA platforms, including the slice and hard macro utilizations. Compared to the streaming core, the design configurations for load-store architectures involve more dimensions including the controller implementations, parallelism styles, and memory system configurations. Thus, building an exact resource model for load-store architectures requires substantial effort in characterizing each component. In [52], a machine learning-based approach for predicting Pareto-optimal solutions is proposed, aiming at capturing high-level features of the design, allowing the statistical models to capture the particular patterns of the target application. Though this work is more general than the previous one, it is an open question how to capture the high-level features of load-store architecture designs for the machine learning-based approach.

**Compute pattern extensions.** In this work, the compute pattern is limited to static loop bounds, multi-linear access patterns and identical kernel operations. Though we have shown several application examples that fits the pattern, further flexibility requires extending the pattern.

To maximize the benefit of compute pattern extensions, the algorithm generation process needs to be extended as well so that Spiral's constraint solver produces algorithms fitting the provided hardware parameters. Extending Spiral for new algorithms turns out to be a difficult task. Spiral has provided extensions of the constraint solvers for non-power-of-2 DFTs, some linear algebra kernels, and computations in software-defined radio. The ongoing effort is addressing graph algorithms with Spiral through linear algebra operations.

In the code generation stage, depending on the exact change, the modifications require different extents of effort. To support diversified kernels in basic

blocks, the hardware interpretation backend must handle resource sharing between basic blocks performing different operations. Typically, compute-intensive basic blocks can be shared by multiplexing the primitive arithmetic units in DFGs. The techniques have been explained in literatures [29][30]. Other access patterns can be natively supported in this framework by having a new index generation module. The inductive calculation method used in optimizing multi-linear indices are applicable to some other calculations such as power functions and modulo functions. Optimizations for other access patterns are to be explored. The data-dependent control flow requires extra communications between the datapath and the controller. If the communication overwhelms the decoupled design between the controller and the datapath, a major modification of hardware paradigm may be required.

**Incorporating implementation options.** This thesis has only addressed a small set of hardware implementation options that have been developed in the computer architecture and digital design community. Various options can be incorporated to this framework and are compatible with the constraint solver of Spiral for exploring the design space.

In this work, the controller is implemented as cooperative FSMs. Other controller implementations include ROM-based controllers and instruction-based controllers. The ROM-based controllers store the control sequence onto a read-only memory. It is preferred when storage resources are cheaper than logic resources. The instruction-based controllers are specified by the supported instruction format and produce the control sequence by fetching, decoding and executing an instruction sequence provided by users. It incurs instruction handling overhead, but is flexible to perform arbitrary control. These cost and benefit of these options have not been precisely modeled, and can be addressed as commensurate implementation options in

algorithm generation process, which will be evaluated in the final implementations.

This work handles only a single level memory, which is limited in either problem sizes when interfacing with on-chip memory, or compute throughput when interfacing with off-chip memory. It is beneficial to support multi-level memories with off-chip memory because a lot of computations exhibits data locality that can benefit from fast buffers. In the past, the SPIRAL framework has been extended for the scratchpad memory of the Cell platform [53], which implements a multi-buffering mechanism. It is also possible to interface with the memory interface infrastructures such as CoRAM [54] and Fluid [55]. The memory system impacts algorithm generation, thus must be considered formally in the constraint solver of Spiral.

Finally, by developing a systematic method to synthesizing hardware accelerators of important computational kernels, it provides a unique opportunity to investigate emerging hardware implementation approaches. One such example is using partial reconfiguration in FPGAs to swap datapath implementations in the fabric. This technique has not been widely use because of the runtime overhead in partial reconfiguration. Because the proposed framework will gradually support a wide range of algorithms, it is possible to have a compute specification that execute long enough time at each of the multiple stages to amortize the partial reconfiguration cost.

**Fusing static and dynamic hardware compilation methods.** In this work, the hardware designs is synthesized from high-level specifications represented by multi-level domain-specific languages. The hierarchitecal representations can provide useful information to fusing static and dynamic hardware compilation methods in a single design.

Since the inception of the hardware generation work of SPIRAL in 2008, the hardware compilation methods have achieved substantial advancements. To date, the most influential methods are the so-called static methods and the dynamic methods. The static methods take in the control-data flow graph (CDFG) representation, and then calculate a legal execution schedule, which is interpreted to FSM-controlled datapth, under the user-specified timing and resource constraints. The most recent static compilation method is based on a system of difference constraints (SDC) [41] and have been successfuly used in commercial hardware compilers. However, the static method requires precise latency of operations in the CDFG, and falls short in the scenarios with data-dependent latency and control flows. In contrast, the dynamic compilation method [22] maps spatially the CDFG into the elastic circuit [28] that uses data tokens to enable distributed control. This method fits data-dependent control but incurs higher resource utilization. The fusion of these two methods for a single design in a general context remains an open question.

A simpler context for hardware compilation setup up in this work can ease the fusion of the two compilation methods. First, since the designers are supposed to specify the specification breakdown rule, the base case operators do not overlap in execution can be tagged with the desired compilation method. Second, the load-store architecture has decoupled most control flows and memory accesses from computations, thus the selection of compilation methods will be decided purly based on the computation itself, without worrying about the complicated loop nest control and memory interface.

**General pattern-based synthesis flow.** This thesis exploited the regularity in basic blocks to enable execution overlapping for the sake of latency improvement. Such a single unified pattern is unusual when more complicated computations

are to be accelerated. A general pattern-based synthesis flow has been developed in static hardware compilation for reducing multiplexor usage [29]. The flow takes in a set of DFGs, then patterns of DFGs are searched, selected, scheduled and bound to hardware resource. It will be benefitial to develop a similar flow that is directed by different assumptions in our special context.

Compared to the input assumptions made in [29], this framework utilizes multi-level abstractions for computational specifications and allows dynamic operations in basic blcoks. The high level abstractions can allow high value patterns to be manually identified as shown in Section 4.2.3. Hence, a pattern-based synthesis flow should simultaneously support manual and automatic pattern recognition across multiple abstraction levels. The dynamic operations introduce disconnected DFG fragments within a basic block, breaking the assumption in [29] that each fragment is mapped to a basic block and never overlap in execution. Hence, the input as a flat set of DFGs is no longer applicable. The set must contain information of which basic block a fragment belong to. When scheduling patterns, only one of the multiple instances belong to the same basic block can be considered to allow concurrent execution within the same basic block, which is required by the proposed framework in this thesis.

To summarize, this thesis proposed a flexible way to generating hardware accelerator designs using customized load-store architectures. Since hardware specialization requires dedicated mappings between computations and hardware implementations, various hardware design paradigms and the design synthesis methods can be incorporated into the framework. The Spiral-based framework is designed to be extensible and allows plugging in necessary features to gradually deliver the flexibility of the proposed approach.

# Bibliography

[1] Franchetti, et, al., "Spiral: automating high quality software production." http://www.spiral.net/doc/pdf/spiral-tutorial2019.pdf, 2019. Accessed: 2022-06-06. (document), 2.1

[2] F. Franchetti, Y. Voronenko, and M. Püschel, "Formal loop merging for signal transforms," in *Programming Languages Design and Implementation (PLDI)*, pp. 315–326, 2005. (document), 2.3.3, 2.8, **??**, **??**, 4.5

[3] M. Püschel and J. M. F. Moura, "The algebraic approach to the discrete cosine and sine transforms and their fast algorithms," *SIAM Journal of Computing*, vol. 32, no. 5, pp. 1280–1316, 2003. 2.2.1

[4] A. Gacic, M. Püschel, and J. M. F. Moura, "Automatically generated high-performance code for discrete wavelet transforms," in *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, vol. 5, pp. V–69, 2004. 2.2.1

[5] D. S. McFarlin, F. Franchetti, M. Püschel, and J. M. Moura, "High-performance synthetic aperture radar image formation on commodity multicore architectures," in *Algorithms for Synthetic Aperture Radar Imagery XVI*, vol. 7337, pp. 72–83, SPIE, 2009. 2.2.1, 3.4.1

[6] F. Franchetti, F. de Mesmay, D. McFarlin, and M. Püschel, "Operator language: A program generation framework for fast kernels," in *IFIP Working Conference on Domain Specific Languages (DSL WC)*, vol. 5658 of *Lecture Notes in Computer Science*, pp. 385–410, Springer, 2009. 2.2.1, 3.4.1

[7] F. d. Mesmay, S. Chellappa, F. Franchetti, and M. Püschel, "Computer generation of efficient software viterbi decoders," in *International Conference on High-Performance Embedded Architectures and Compilers*, pp. 353–368, Springer, 2010. 2.2.1, 3.4.1

[8] C. F. Van Loan, "The ubiquitous kronecker product," *Journal of computational and applied mathematics*, vol. 123, no. 1-2, pp. 85–100, 2000. 2.2.1

[9] J. R. Johnson, R. W. Johnson, D. Rodriguez, and R. Tolimieri, "A methodology for designing, modifying, and implementing fourier transform algorithms on various architectures," *Circuits, Systems and Signal Processing*, vol. 9, no. 4, pp. 449–500, 1990. 2.2.2

[10] F. Franchetti, M. Püschel, J. M. F. Moura, and C. W. Ueberhuber, "Short vector SIMD code generation for DSP algorithms," in *High Performance Extreme Computing (HPEC)*, 2002. 2.2.3

[11] J. H. adn D.A. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufman, 3rd ed., 2003. 2.3

[12] T. Popovici, T.-M. Low, and F. Franchetti, "Large bandwidth-efficient FFTs on multicore and multi-socket systems," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, 2018. 2.3

[13] J. G. Lewis and H. D. Simon, "The impact of hardware gather/scatter on sparse

gaussian elimination," *SIAM Journal on Scientific and Statistical Computing*, vol. 9, no. 2, pp. 304–311, 1988. 2.3.1

[14] e. a. Sandra Loosemore, "The gnu c library reference manual." https://www.gnu.org/software/libc/manual/pdf/libc.pdf. [Online; accessed 11-August-2021]. 2.4.1

[15] P. A. Milder, F. Franchetti, J. C. Hoe, and M. Püschel, "Computer generation of hardware for linear digital signal processing transforms," *ACM Transactions on Design Automation of Electronic Systems*, vol. 17, no. 2, 2012. 2.5

[16] P. A. Milder, J. C. Hoe, and M. Puschel, "Automatic generation of streaming datapaths for arbitrary fixed permutations," in *2009 Design, Automation & Test in Europe Conference & Exhibition*, pp. 1118–1123, IEEE, 2009. 2.5, 6.2.3

[17] P. Milder, F. Franchetti, J. C. Hoe, and M. Püschel, "Computer generation of hardware for linear digital signal processing transforms," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 17, no. 2, pp. 1–33, 2012. 2.5

[18] B. Akın, F. Franchetti, and J. C. Hoe, "Ffts with near-optimal memory access through block data layouts," in *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 3898–3902, IEEE, 2014. 2.5

[19] D. M. Ritchie, "The development of the c language," *ACM Sigplan Notices*, vol. 28, no. 3, pp. 201–208, 1993. 3.1

[20] O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured programming*. Academic Press Ltd., 1972. 3.1

[21] Z. Zhang and B. Liu, "Sdc-based modulo scheduling for pipeline synthesis," in *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 211–218, IEEE, 2013. 3.2

[22] L. Josipović, R. Ghosal, and P. Ienne, "Dynamically scheduled high-level synthesis," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 127–136, 2018. 3.2, 6.3

[23] G. Weisz and J. C. Hoe, "C-to-coram: Compiling perfect loop nests to the portable coram abstraction," in *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, pp. 221–230, 2013. 3.2

[24] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Math. of Computation*, vol. 19, pp. 297–301, 1965. 3.2, 3.4.1

[25] G. K. Wallace, "The jpeg still picture compression standard," *IEEE transactions on consumer electronics*, vol. 38, no. 1, pp. xviii–xxxiv, 1992. 3.2

[26] T. M. Low, F. D. Igual, T. M. Smith, and E. S. Quintana-Orti, "Analytical modeling is enough for high-performance blis," *ACM Transactions on Mathematical Software (TOMS)*, vol. 43, no. 2, pp. 1–18, 2016. 3.2, 3.4.1

[27] L. P. Carloni and A. L. Sangiovanni-Vincentelli, "Coping with latency in soc design," *IEEE Micro*, vol. 22, no. 5, pp. 24–35, 2002. 3.3.1

[28] J. Cortadella, M. Kishinevsky, and B. Grundmann, "Synthesis of synchronous elastic architectures," in *2006 43rd ACM/IEEE Design Automation Conference*, pp. 657–662, IEEE, 2006. 3.3.1, 6.3

[29] J. Cong and W. Jiang, "Pattern-based behavior synthesis for fpga resource reduction," in *Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, pp. 107–116, 2008. 3.3.1, 6.3

[30] L. Josipovic, "High-level synthesis of dynamically scheduled circuits," tech. rep., EPFL, 2021. 3.3.1, 6.3

[31] G. De Micheli, *Synthesis and optimization of digital circuits*. No. BOOK, McGraw Hill, 1994. 3.3.2

[32] V. Rajagopalan, V. Boppana, S. Dutta, B. Taylor, and R. Wittig, "Xilinx zynq-7000 epp: An extensible processing platform family," in *2011 IEEE Hot Chips 23 Symposium (HCS)*, pp. 1–24, 2011. 3.3.2

[33] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, *et al.*, "The rocket chip generator," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, vol. 4, 2016. 3.3.2, 6.2.2

[34] F. Sadi, J. Sweeney, T.-M. Low, J. C. Hoe, L. Pileggi, and F. Franchetti, "Efficient spmv operation for large and highly sparse matrices using scalable multi-way merge parallelization," in *MICRO*, 2019. 3.3.2

[35] S. Mashimo, T. Van Chu, and K. Kise, "High-performance hardware merge sorter," in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 1–8, IEEE, 2017. 3.3.2

[36] R. M. Russell, "The cray-1 computer system," *Communications of the ACM*, vol. 21, no. 1, pp. 63–72, 1978. 3.3.3, 6.2.1

[37] T. M. Low and F. Franchetti, "High assurance code generation for cyber-physical systems," in *2017 IEEE 18th International Symposium on High Assurance Systems Engineering (HASE)*, pp. 104–111, IEEE, 2017. 3.4.1

[38] A. Gacic, *Automatic implementation and platform adaptation of discrete filtering and wavelet algorithms*. PhD thesis, Carnegie Mellon University, 2004. 3.4.1

[39] K. D. Cooper, L. T. Simpson, and C. A. Vick, "Operator strength reduction," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 23, no. 5, pp. 603–625, 2001. 3.4.2

[40] L. Josipovic, P. Brisk, and P. Ienne, "An out-of-order load-store queue for spatial computing," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 5s, pp. 1–19, 2017. 3.4.2

[41] J. Cong and Z. Zhang, "An efficient and versatile scheduling algorithm based on sdc formulation," in *2006 43rd ACM/IEEE Design Automation Conference*, pp. 433–438, IEEE, 2006. 3.4.3, 6.3

[42] M. Andrews and J. Johnson, "Performance analysis of a family of wht algorithms," in *2007 IEEE International Parallel and Distributed Processing Symposium*, pp. 1–8, IEEE, 2007. 4.5.2

[43] 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," in *DAC Design Automation Conference 2012*, pp. 1212–1221, IEEE, 2012. 5.1.1

[44] Lawson, et al., "Chisel testers." https://github.com/freechipsproject/chisel-testers, 2021. Accessed: 2021-06-06. 5.1.1

[45] Snyder, et al., "Verilator." https://www.veripool.org/verilator/, 2021. Accessed: 2021-06-06. 5.1.1

[46] Peter A. Milder, "Dft (discrete fourier transform) verilog ip generator." https://www.spiral.net/hardware/dftgen.html, 2014. Accessed: 2022-06-06. 5.2.1

[47] F. Serre, T. Holenstein, and M. Püschel, "Optimal circuits for streamed linear permutations using ram," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 215–223, 2016. 5.2.3

[48] F. Serre and M. Püschel, "Memory-efficient fast fourier transform on streaming data by fusing permutations," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 219–228, 2018. 5.2.3

[49] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011. 6.2.1

[50] M. Püschel, P. A. Milder, and J. C. Hoe, "Permuting streaming data using rams," *Journal of the ACM (JACM)*, vol. 56, no. 2, pp. 1–34, 2009. 6.2.3

[51] P. A. Milder, M. Ahmad, J. C. Hoe, and M. Püschel, "Fast and accurate resource estimation of automatically generated custom dft ip cores," in *Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*, pp. 211–220, 2006. 6.3

[52] M. Zuluaga, A. Krause, P. Milder, and M. Püschel, "" smart" design space sampling to predict pareto-optimal solutions," in *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, pp. 119–128, 2012. 6.3

[53] S. Chellappa, F. Franchetti, and M. Püeschel, "Computer generation of fast fourier transforms for the cell broadband engine," in *Proceedings of the 23rd international conference on Supercomputing*, pp. 26–35, 2009. 6.3

[54] E. S. Chung, J. C. Hoe, and K. Mai, "Coram: an in-fabric memory architecture for fpga-based computing," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pp. 97–106, 2011. 6.3

[55] J. Melber and J. C. Hoe, "A service-oriented memory architecture for fpga computing," in *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, pp. 91–97, IEEE, 2020. 6.3