Automatic Generation of Efficient Parallel Streaming Structures for Hardware Implementation

Thaddeus Egon Koehn

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Electrical Engineering

Peter M. Athanas, Chair
Robert W. McGwier
Carl B. Dietrich
Richard M. Buehrer
Joseph M. Ernst

November 11, 2016
Blacksburg, Virginia

Keywords:
Streaming, FPGA, Hardware, Digital Signal Processing, Permutation.

Copyright 2016, Thaddeus E. Koehn

Automatic Generation of Efficient Parallel Streaming Hardware

Thaddeus Egon Koehn

(ABSTRACT)

Digital signal processing systems demand higher computational performance and more operations per second than ever before, and this trend is not expected to end any time soon. Processing architectures must adapt in order to meet these demands. The two techniques most prevalent for achieving throughput constraints are parallel processing and stream processing. By combining these techniques, significant throughput improvements have been achieved. These preliminary results apply to specific applications, and general tools for automation are in their infancy. In this dissertation techniques are developed to automatically generate efficient parallel streaming hardware architectures.

Automatic Generation of Efficient Parallel Streaming Hardware

Thaddeus Egon Koehn

(GENERAL AUDIENCE ABSTRACT)

The algorithms that process data have been getting more complicated requiring more operations in less time. This trend has been going on for many years with no end in sight. Techniques must be developed to allow the processing system to meet these requirements. Assembly line techniques, or stream processing allows multiple stages in which each stage is working on a different piece of data. Increasing the number of assembly lines can further increase the number of operations, but results in large overheads. This dissertation develops automation techniques to reduce these overheads resulting in efficient hardware.

Dedication

To my beautiful wife, Laura.

Acknowledgments

I am immensely grateful to my Aunt, Ingrid Burbey, who passed on to me the job opportunity that would eventually allow me to complete my PhD. She also guided me to my Advisor, Dr. Peter Athanas, and showed me how to navigate the PhD process.

I also must thank the Hume Center for providing this opportunity and for supporting conference travel and the publication process. I especially appreciate the advice and direction of Bob, Chris, and Joey.

It has been fun collaborating with all the members of the CCM team, those past and present. A little mental break goes along way. Thanks, Ali and Ryan.

Dr. Peter Athanas always made himself available when I needed help, both academically and professionally. I owe him a huge debt of gratitude.

Thanks also go to the other members of my Committee, Drs. Buehrer, Dietrich, Ernst, and McGwier. Their insights and critiques were extremely helpful when I was unsure where to go next with my research.

Finally, the most important member of my team, my editor and wife, Laura, who took care of the children while I spent time on my research, and who read every rough draft. Without her support, I could not have completed my degree.

Contents

1 Introduction **1**

1.1 Motivation . . . 1

1.2 Summary of Contributions . . . 6

1.3 Summary . . . 7

2 Literature Survey **9**

2.1 Parallel Streaming Basics . . . 9

2.2 Permutations . . . 10

2.3 Scheduling . . . 11

2.4 Applications . . . 12

2.4.1 Fast Fourier Transform . . . 13

2.4.2 Matrix Multiplication . . . 14

2.4.3 Neural Networks . . . 15

2.5 EDA Tools for Streaming . . . 16

2.5.1 Commercial Tools . . . 16

2.5.2 Research Projects . . . 17

2.5.3 Summary . . . 18

3 Streaming Permutations **19**

3.1 Background . . . 20

3.2 Parameterized Datapath . . . 22

3.3 Minimum Number of Memory Banks . . . 24

3.4 Streaming Linear Permutations 26
3.4.1 Spatial Permutations 26
3.4.2 Temporal Permutations 29
3.4.3 General Linear Permutations 30
3.4.3.1 Jointly Minimizing Control Cost and Switches 30
3.4.3.2 Minimizing Switches 32
3.5 General Streaming Permutations 33
3.5.1 Spatial Component 34
3.5.2 Temporal Component 38
3.6 Results 44
3.6.1 Evaluation 45
3.7 Conclusion 46

4 Data Reuse **51**
4.1 Background 52
4.2 Scheduling for Sparsity 56
4.3 Hardware Implementation 59
4.3.1 Mapping the Schedule to Hardware 61
4.4 Results 65
4.5 Conclusion 68

5 Permutation Chaining **69**
5.1 Architecture 70
5.2 Single Computation Allocation 71
5.3 Chained Computations 75
5.4 Results 78
5.5 Related Work 80
5.6 Conclusion 80

6 Multi-Stage Streaming **82**

6.1 Related Work 83
6.2 Background and Problem Formulation 84
6.2.1 Neural Networks 84
6.2.2 Synchronous Data Flow 85
6.2.3 Problem Formulation 86
6.3 Architecture 87
6.4 Operation Scheduling 88
6.4.1 Forward Propagation Schedule 89
6.4.2 Backward Propagation Schedule 90
6.4.3 Combined Schedule 90
6.5 Results 92
6.6 Conclusion 93

7 The Next Generation of EDA Tools **96**
7.1 Framework 97
7.2 Design Flow 100
7.3 Conclusion 102

8 Conclusion **103**
8.1 Summary of Contributions 104
8.2 Recommendations for Future Work 104

Bibliography **106**

List of Figures

1.1 Processor frequency has scaled over time from 1985 to 2005, at which point frequencies have stalled, data compiled by CPU DB [3]. 2

1.2 Loss of data occurs when the processing is not at least as fast as the input data. 4

1.3 Processor frequency has scaled over time from 1985 to 2005, at which point frequencies have stalled. 5

1.4 The streaming FFT architecture works like an assembly line where each stage of the FFT works on one piece of data each cycle. 5

1.5 Combining replication with streaming enables more efficient processing. . . . 6

3.1 Illustration of perfect shuffle (stride) permutation over eight elements with three different implementations with different streaming widths. 20

3.2 Permutation architecture. 22

3.3 Bit-reversal permutation streamed 3 per cycle. 24

3.4 Map of input/output data vectors to a graphical representation. 25

3.5 One stage of Omega Network. It consists of a perfect shuffle of p input ports followed by $p/2$ 2×2 switches. All the switches in the stage are controlled by one common bit. 28

3.6 Initial mapping of data elements to memory element and address locations. . 34

3.7 AS-Waksman networks are constructed recursively from the outside layer toward the center. 37

3.8 The inner AS-Waksman network terminates at either a 2×2 switch or the 3×3 AS-Waksman Network. 38

3.9 Routing the outer switches in a 9×9 AS-Waksman network by edge coloring. 39

3.10 The number of data elements in the system increases until reaching a maximum after the system latency has elapsed. 40

3.11 Eight FIFOs with inputs applied in a circular shift pattern convert a stream size of size to eight. . . . 45

3.12 BRAM consumption as function of streaming width p and permutation size n compared for linear and general implementations. . . . 48

3.13 Area in slices as function of streaming width p and permutation size n compared for linear and general implementations. . . . 49

3.14 Latency in clock cycles as function of streaming width p and permutation size n compared for linear and general implementations. . . . 50

4.1 A simple Markov chain where each state connects only to one other state. . . 53

4.2 A Markov chain where states can only transition to all states that are larger than it. . . . 54

4.3 Architecture for streaming pipeline. . . . 54

4.4 Example streaming data rearrangement for partial products. . . . 55

4.5 The permutation architecture is customized to perform the replication and reordering required of the data fetch unit. . . . 60

4.6 Map of input/output data vectors to a graphical representation. . . . 61

4.7 Assigning memory banks to each data element. . . . 62

4.8 Routing the outer switches in a modified 8×8 AS-Benes network by edge coloring. . . . 63

4.9 Using structured matrix implementations allows significant improvements in throughput while using the same number of DSP multipliers. . . . 66

4.10 By fixing the number of DSPs for all cases, smaller form factor matrix multiplication implementations are possible with higher throughput the sparser the matrix. . . . 67

4.11 The processing delay is related to the sparsity of the matrices being multiplied. The higher the sparsity the fewer computations required. . . . 67

5.1 Example datapath for implementing an algorithm in a streaming architecture. 70

5.2 Algorithm 6 mapping with conflicts. . . . 72

5.3 Algorithm 6 mapping with conflicts resolved. . . . 73

5.4 Chained computation using new indexing. . . . 75

5.5 Reduced complexity chained computation. . . . 76

6.1 A fully connected multi-later network with k layers. 85
6.2 Steps describing a stage of computation. 87
6.3 Average latency over 1000 runs as a function of number of samples n from various schedulers. The number of stages $k = 10$, samples per clock $p = 2$. . 93
6.4 Average latency over 1000 runs as a function of sample per clock p from various schedulers. The number of stages $k = 10$, and number of samples $n = 1024$. . 95
6.5 Average latency over 1000 runs as a function of number of stages k from various schedulers. The number of sample per clock $p = 2$, and number of samples $n = 1024$. 95

7.1 Streaming system architecture. 97
7.2 Several deserializers showing mapping samples into space and time. 98
7.3 Model for optimizing data architecture with parallelism. 99
7.4 Synphony Model Compiler automates path from high-level algorithm to digital logic, image borrowed from [59]. 100

List of Tables

3.1 Summary of cost of the memory components 43
3.2 Comparison of required RAM and switches 44

4.1 Simple schedule for 2×2 matrix multiplication 57

5.1 Comparison of 4096 point FFT with $p = 2$ 79
5.2 Comparison of 4096 point FFT with $p = 8$ 79

6.1 Latency reduction using the different schedules 94

Chapter 1

Introduction

1.1 Motivation

Digital signal processing systems demand higher computational performance and more operations per second than ever before, and this trend is not expected to end any time soon [1]. Processing architectures must adapt in order to meet these demands. The two techniques most prevalent for achieving throughput constraints are parallel processing and stream processing. By combining these techniques, results have shown significant throughput improvements over other techniques [2]. These preliminary results apply to specific applications, and general tools for automation are in their infancy. The goal of this dissertation is to develop a framework for generating efficient parallel streaming hardware architectures.

Rather than dive into the digital design domain, consider an analogous problem. Suppose the imaginary car company T. Motors receives an order to produce 300,000 new cars, due in one year. Their current old-fashion Henry Ford assembly lines produces 50,000 cars in a year. Increasing the throughput of the plant requires building five additional lines.

Each stage of the assembly line requires a certain time period to complete equivalent to a clock cycle in digital design. The latency is the time from when the process starts at the first stage to when the first car rolls off the line. After the initial latency period one car is produced each time epoch or clock cycle.

To complicate matters, each stage of the line produces numbered parts and different model variations require the parts in a different order. The time limitations of the order require that the latency be minimized as well, so cars are produced as soon as possible. Storage limitations on the factory floor require that the assembly lines be optimized to reduce footprint and cost in sorting components for the different models. In other words,

can the individual lines share operations to minimize storage and latency?

This is the problem faced in digital signal processing, throughput requirements have risen, and the traditional streaming model (like an assembly line) must be upgraded to deal with the constraints of the problem while optimizing for latency, and area footprint.

The throughput of digital circuit is controlled by the clock frequency and the number of parallel pipelines. For many years, the clock frequency provided an easy and simple knob to handle higher throughput increases, but this is no longer possible. As with the T. Motor Company, the streaming assembly line must be expanded and optimized.

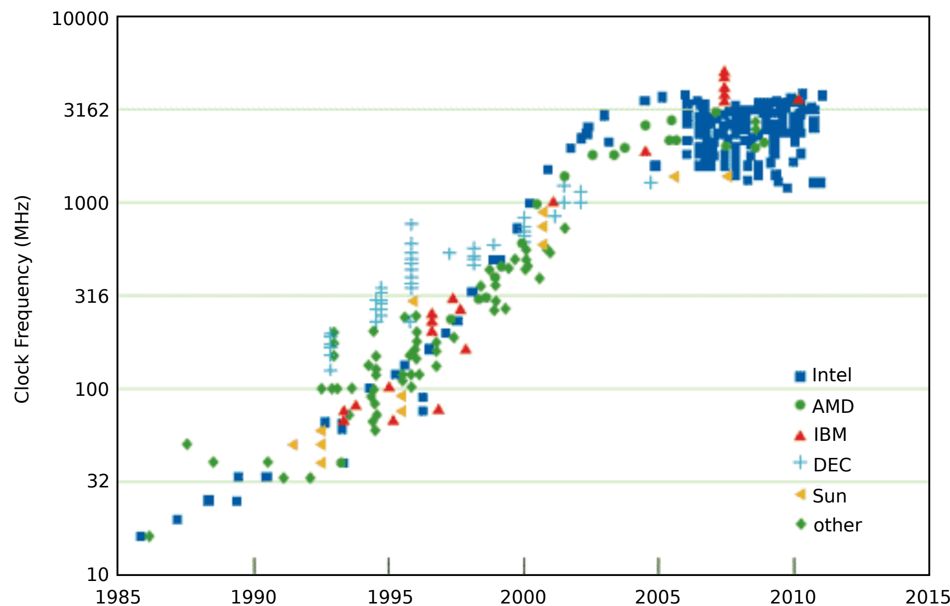


Figure 1.1: Processor frequency has scaled over time from 1985 to 2005, at which point frequencies have stalled, data compiled by CPU DB [3].

Figure 1.1 shows the processor clock frequencies of several major manufacturers over the time period of 1985 through 2012. Before approximately 2005, the processing frequency increased exponentially each year. After 2005, the processor frequencies of standard desktop processors plateaued between 3 and 5 GHz. Before 2005, computational performance was guaranteed to increase just by increasing the clock frequency; the same number of operations could be done in a smaller time period. This meant that if an algorithm required more computation, the requirements would be met within in a few years as the clock frequency rose. Since the plateau after 2005, alternative methods for meeting computational demand have become more mainstream.

While clock frequency trends no longer deliver the increased performance they once did, Moore's law continues to hold, increasing the density of transistors on a regular basis. As a result, performance gains since 2005 have focused on parallelism and architecture enhance-

ments. Platforms have typically used either parallel processing or stream processing. Stream processing traditionally has been limited to one operation per stage per clock cycle, but by extending it with parallel processing higher data rates may be achieved.

Parallel processing or replication describes instantiating multiple instances of the same component. Multi-core architectures and GPUs frequently employ this type of parallelism. Each of the cores is identical, but the capability of the component can vary significantly. In multi-core CPUs, each core can be running completely different kernels, but synchronization is difficult. In GPUs, the same set of instructions run on each core, and speedup is achieved with large data sets to offset overhead.

The problem with replication is that the data sets for each component must be independent. A processing system receives digital signals sequentially, either through data sampling or a communication medium. Algorithms typically work on data arranged sequentially, so components must store data for a long period of time before processing can occur. This represents a significant memory or resource cost, especially for large data sets. Additionally, if the results are required in real-time, the time spent buffering may delay output beyond the acceptable time limit.

An alternative parallel processing architecture is stream processing. This architecture works like an assembly line. Each step in the process performs a different operation, and at each clock cycle the data shifts to the next operation. In this case, the components form a long chain or datapath. While a limited form of this appears in most modern processors, the flexibility of custom hardware, such as ASICs and FPGAs allows for extremely long pipelines.

A computer system is said to be real-time if the total correctness of an operation depends not only upon its logical correctness but also upon the time in which it is performed [4]. This time constraint will vary by application and by level of importance. A critical process may fail if the operation is not performed in time, while a less critical process may gracefully fail with a partial result being better than none.

Stream processing applies operations to the sequence of data as they enter the system rather than buffering, which represents cost in terms of both memory storage and idle processing time. Traditionally, stream pipelines have been optimized for throughputs of up to one data element per clock period [5]. If the throughput requirements are larger than one sample per clock, and the clock rate cannot increase due to timing constraints, then alternative architectures are needed.

Solving the throughput requirements with constrained clock rates requires stream processing be extended for input with multiple data elements each clock, just as in parallel architectures based on data-parallel processing, where a single instruction causes an operation to be performed on multiple pieces of data (SIMD - single instruction multiple data). Solving the aforementioned problems requires that each stage of the processing be partitioned for parallel streams and that each operation receive the appropriate data elements.

Neither parallel processing nor stream processing is appropriate for all applications, but many data-intensive applications fit the paradigms quite well. Good candidates are applications in which there are large amounts of data that require processing in a series of repeated steps. Domains that match these requirements include digital signal processing, communications, and machine learning.

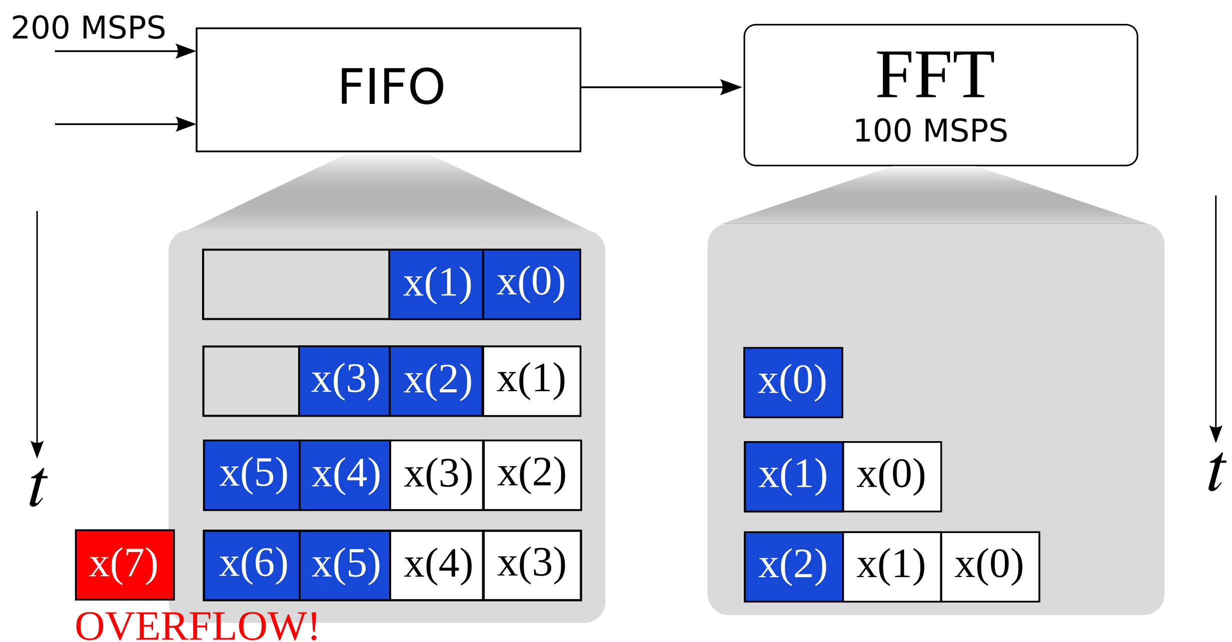


Figure 1.2: Loss of data occurs when the processing is not at least as fast as the input data.

As an example, consider the signal processing task of the Fast Fourier Transform (FFT). The required throughput is 200 Msps (million samples per second), but the FFT component runs at only 100 Msps. Figure 1.2 illustrates the problem without parallelism. At each clock period, two samples (in blue) enter the FIFO (first in, first out buffer), but the FFT only takes one sample (also in blue) out of the FIFO. After the fourth clock period, the FIFO overflows because it is not large enough to hold the input data. As long as the FFT has a lower throughput than the input, an overflow will eventually occur.

To solve this problem using replication, two FFTs are needed (see Figure 1.3). This leads to a different problem, each FFT must work on independent data sets. For an FFT of size 8, the first FFT needs samples 0 to 7, while the second FFT needs samples 8 to 15. To handle this, each FIFO must be able to receive data at the 200 Msps rate and store the full FFT size. In this case, the first eight samples would be written to FIFO 1, and the next eight to FIFO 2. For larger FFT sizes and input rates, the number and sizes of the FIFOs becomes large. This significantly increases the latency and the memory.

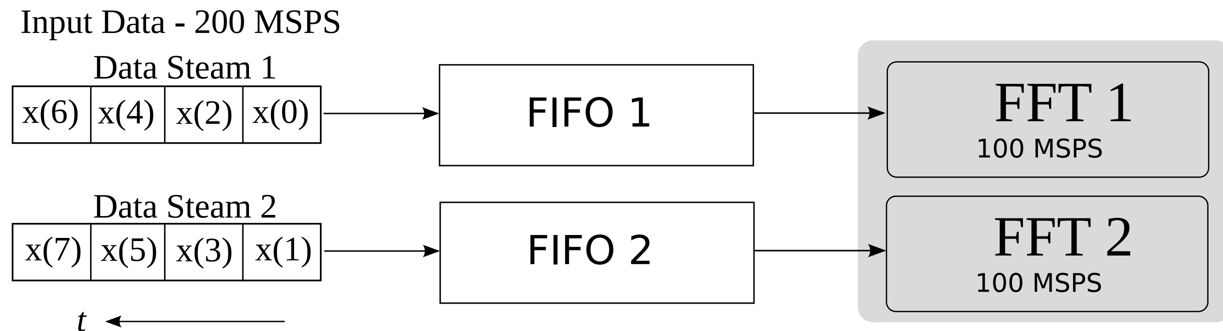


Figure 1.3: Processor frequency has scaled over time from 1985 to 2005, at which point frequencies have stalled.

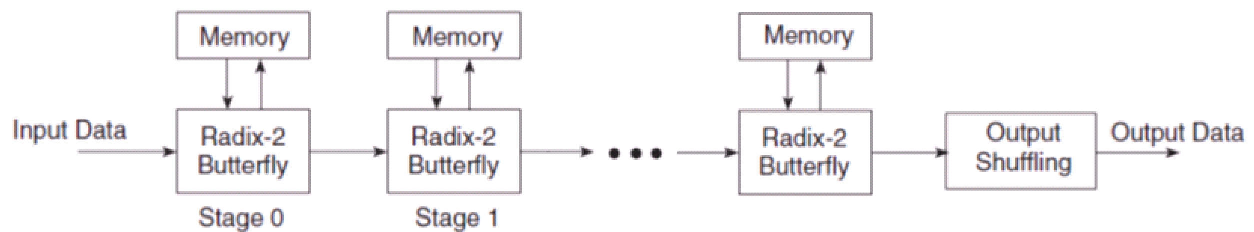


Figure 1.4: The streaming FFT architecture works like an assembly line where each stage of the FFT works on one piece of data each cycle.

Figure 1.4 shows an example of a streaming architecture for the FFT. A single data sample is provided each clock. After each clock cycle data are passed to the next stage in the process. The streaming architecture requires that samples be provided in sequential order. After a transient, one sample is provided to the output each clock cycle. If the processing rate is limited to one sample per clock cycle, then the maximum processing rate is limited by the clock frequency.

To get beyond the one sample per clock limit of conventional stream processing architectures, some of the ideas of parallel processing must be incorporated. Figure 1.5 shows the streaming chain for a 200 Msps FFT with a 100 MHz clock. Before each arithmetic operation ($f(x)$) the two data streams are permuted so that each operation receives data elements in the required order. The highlighted samples are $x(0)$ and $x(4)$. The first operation in the FFT both adds and subtracts these two elements. They are originally located in Data Stream 1 in the first and third positions. They must be co-located in the same clock cycle for $f(x)$ to perform the operation. This permutation in space (data stream) and time is a non-trivial operation that is essential to enabling processing rates above that of one sample per clock cycle.

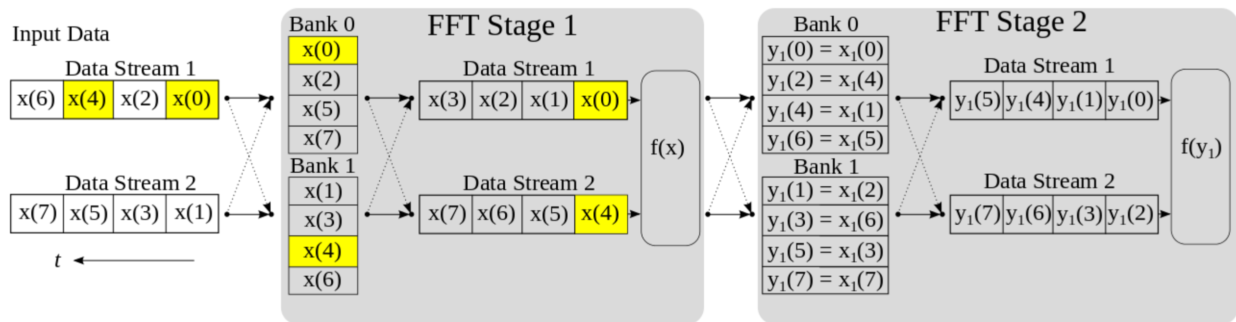


Figure 1.5: Combining replication with streaming enables more efficient processing.

There are multiple measures of efficiency for hardware architectures; these include throughput, area/resource utilization, latency, and energy efficiency. Throughput refers to the number of data elements generated at the output each second. Area or resource utilization varies depending on the target device. For FPGAs, this typically refers to the number of used slices or block memories. Latency is the amount of time it takes for the first sample to enter and exit the processing system. Energy efficiency refers to the amount of energy required to perform the processing per operation.

The framework developed in this dissertation targets a particular throughput parametrized by the clock rate, i.e., p samples per clock cycle. This framework maximizes efficiency by targeting latency. The architecture latency is strongly correlated with many of these other measures. The larger the latency the more memory is required. Similarly, the more resources used, the more energy is required to power those resources, causing lower energy efficiency. So while latency is the targeted measure, decreasing latency has a favorable effect on the other measures.

1.2 Summary of Contributions

This section provides a summary of the contributions that were made to the field of stream processing.

1. A technique to determine the theoretical lower bounds for the number of required independent RAM blocks for streaming permutations [6, 7].
2. A method to determine and achieve the theoretical lower bound for latency and memory depth [6, 7].
3. The realization of the minimum latency and memory achievable for a given permutation, improving upon the state of the art by a factor of approximately two [6, 7].

4. An approach to building streaming components with data reuse [8].
5. A matrix-matrix multiplication hardware generator for structured matrices [8].
6. A method to simplify the control logic for chains of permutations and the return of data to sequential order and then apply a new reordering for multiple computations. This method resulted in lower latency and cost when applied to the Fast Fourier Transform [9].
7. A set of algorithms for scheduling elementary operations for streaming applications to reduce latency [10].

1.3 Summary

This dissertation seeks to enable parallel pipeline processing through architectural improvements and automation. In doing so, the requirement of higher throughput can be achieved at a reduced cost. The contributions of this research include: creating a more efficient and robust streaming permutation architecture; extending this to enable data reuse; and using scheduling to reduce memory cost and latency over many stages of a pipeline.

The discussion begins in Chapter 2 with an overview of the state of current research in parallel processing and streaming architectures. Applications in areas that may benefit from improvements in streaming are analyzed, as are a few of the better known electronic design automation (EDA) tools. Their deficiencies are noted as a motivation for this research, which aims to augment their abilities, specifically in automating design for high data rates.

When assembling components for a streaming application, it is often necessary to build translation blocks between them to match the ordering of the data elements required for the subsequent processing. Chapter 3 develops a technique for realizing streaming permutations with parallel data. It can implement arbitrary permutations of different streaming widths and sizes. This technique is applied to an architecture that receives continuous input at a rate of p elements per clock cycle, and after an initial start-up latency, outputs continuously at the same rate. In addition, the memory usage and latency through the memory array are minimized. The resulting design is evaluated for permutations parametrized by size and stream width in terms of the memory elements and depths required. The class of stride permutation is considered for specific experimental evaluation. On average, architectures designed with this technique require only half the latency and requires half the memory of other techniques.

While permutations are very useful, especially in cases such as transforms and multi-rate signal processing, many algorithms require data to be reused. For instance, many algorithms can be described using matrix operations. Matrix multiplication requires each element in a row of the first matrix to be multiplied by an element in each column of the second matrix.

A permutation only considers reordering of a data sequence, not reuse. Chapter 4 develops extensions to the permutation architecture that allow data reuse.

Chapter 5 begins extending optimization beyond that of a single permutation. Many linear transforms, such as the Fast Fourier Transform, consist of multiple permutations and computation blocks. Looking at the problem as whole allows multiple permutations to be aggregated and the control costs shared. This results in designs that require less memory and fewer logic slices while performing the same function.

Chapter 6 considers the problem of reducing latency and memory across multiple stages of computation for nonlinear or random data accesses. An example use case is a partially-connected neural network, whose sparsity introduces freedom for optimization. The layers of the network translate into multiple stages of computation separated by data reordering and reuse transformations.

Chapter 7 pulls these ideas together and suggests future directions for making these ideas practical and accessible to design engineers. Rather than adding yet another tool to a large suite of tools with relatively small scope, the goal would be to augment existing tools. The new capability would be largely invisible to the user, as the backend algorithms would run behind the scenes, but it would allow higher data rates with streaming parallel processing to be realized more efficiently.

Chapter 2

Literature Survey

The field of parallel streaming architecture generation is still in its infancy. While many researchers have utilized parallel stream processing in high speed implementations, relatively few have focused specifically on automation for high-throughput processing. The customizable digital logic in FPGAs and ASICs, as well as limitations in other hardware platforms, such as CPUs and GPUs, has made FPGA and ASIC platforms an attractive option for achieving throughput requirements out of the range of most other platforms.

This chapter surveys related works at several levels, from very low level design details to high-level applications. First, research is presented on the basic requirements for a stream processing system to obtain high throughput. Then, specific techniques are discussed for implementing streaming permutations spanning both specific types as well as more general results. Many applications require streaming architectures to improve throughput, but often do so without formally utilizing streaming techniques. To demonstrate this dissertation's importance and utility, a survey of such applications is conducted. Finally, a short survey of electronic design automation tools is presented.

2.1 Parallel Streaming Basics

Many of the foundations for these concepts were developed several decades ago. There was significant research into the concept of parallel architectures. Maximizing the data to the parallel processors required memory architectures that would reduce or eliminate memory conflicts, which can cause stalls. Shapiro [11] defines the basic building blocks of conflict-free memory access as memory partitioning and a switching network.

Switching networks are a field in of themselves, but the class of non-blocking networks is particularly useful. These networks were developed for telephone switching systems. Clos networks [12], developed in 1952, define a broad class of multi-stage circuit switching networks that are well suited for pipelining. These switch networks represent a theoretical idealization. The Benes network [13] is a sub-class in which the switching elements are two-by-two switches, where each switch can either keep the order of the two inputs unchanged or switch them. Originally developed for networks of powers of two, the Benes network has been adapted for other network sizes as well [14]. Waksman [15] identified redundancies in the Benes network, so using the same topology, fewer switches are required for the same size network. This reduction can also be applied to arbitrarily-sized Benes networks, as shown by [16].

Memory partitioning involves splitting a set of data into different locations. It has long been used in dynamic memory systems, RAID controllers, and many others. The concept is well known, but the details of how the partitioning occurs determine the performance of the system. Specific memory access types have been well studied. Harper [17] considers different vector length accesses from memory systems. Matrix accesses are considered by Lee [18], while Tsai [19] describes a solution of the partitioning problem sizes with partitions of powers of two. Specific applications have also been studied; in particular Gong [20] solves the partitioning problem for a particular class of turbo decoders.

2.2 Permutations

Applying these concepts to streaming parallel architectures started with common data access patterns, the stride permutation. The stride permutation accesses data from two different locations in a sequence. The stride of the stride permutation is how far apart the locations are from each other. For example, a stride of eight means that the first and ninth samples are accessed, followed by the second and tenth. Jarvinen [21] developed a method using small memories to implement these accesses for permutation and streaming widths that are powers of two. Gorman and Wills [22] also considered a limited set of stride permutations in the context of one of the more common applications, the Fast Fourier Transform.

One of the first general-purpose approaches was a design for data format converters by Parhi [23]. This technique used registers to completely customize the data path with multiplexers and state machines producing the correct outputs at each clock cycle. It is optimal in terms of storage and can be used for all streaming widths and permutations. A similar technique is used later in this dissertation to minimize the latency and memory depth. Its downfall is that the storage distribution among many independent registers causes the control and routing to be very complicated. Especially for large permutations, the complexity ultimately leads to slow clock requirements to meet timing.

Püschel [24] was the first to use a RAM-based approach to realize permutations that can be described as a permutation of the samples' binary index, which includes stride and perfect shuffle permutations. The architecture used an array of RAM elements sandwiched between two switch networks that are able to swap elements between data streams. This approach used the binary representations of the permutation to develop optimal and nearly optimal permutation control structure and switching networks. Serré et. al. [25] also developed modifications of these circuits to minimize the number of switches required. These approaches originally required dual-port RAMs, but Chen [26], while studying the energy efficiency of stride permutations, was able to use two single-port RAMs. Milder [27] proposed a memory-intensive technique that could realize any permutation and streaming size. This approach differed from others in that it required only a single switch network sandwiched between two RAM arrays. Chen [28] refined this by using a similar architecture to Püschel's to reduce the memory requirements for arbitrary permutations.

This dissertation further improves these permutation techniques by determining minimum bounds for the memory requirements for streaming permutations and develops an EDA technique to generate hardware architectures that achieve these bounds. This technique demonstrates reduced latency and memory requirements over previous solutions.

2.3 Scheduling

For the purposes of this research, the broad field of scheduling was limited to scheduling that specifically dealt with parallel processing. In particular, those studies that fit the constraints imposed by digital hardware, as opposed to CPUs and GPUs in which scheduling differs because of the operating system and memory accesses, events can occur over a time interval. For digital hardware, operations can be scheduled to occur at a particular time instance.

A scheduler for a processing unit can have the goal of minimizing resources or area, minimizing latency or delay, maximizing throughput, or some combination of these. A schedule has a series of tasks or *actors* that must be bound, that is, assigned to a resource and/or time slice to be executed. Each actor is associated with a particular cost and throughput.

A common technique to increase throughput is to target bottleneck actors (actors that limit the throughput) for increased parallelism or replication. In multicore architectures, this is very effective. Examples include [29–31] where a variety of techniques are employed to identity the bottleneck and to parallelize it by assigning it to multiple cores. Scheduling tends to be a complex process with a wide search space and no guarantee of optimality in most cases. Heuristics are commonly used to traverse the space quickly. Common heuristics are greedy algorithms, which at each step of the algorithm attempt to maximize utility. For example, Gordon sorts actors in order of decreasing computational requirements and then replicates actors, assigning them to multiple cores until the actor is no longer a bottleneck,

before considering the next element on the list [32]. Once all the cores are occupied, actors at the end of the list are scheduled to the least occupied core until all actors have been scheduled.

While FPGAs do not have cores, they do have resources such as logic slices, DSPs, and block RAMs [33], which can be parallelized and scheduled in a similar fashion, albeit on a much larger scale. Instead of scheduling a particular function to run on multiple cores, Cong [34] searches a library of modules with different area and performance constraints. The schedule must then jointly select a module from a database and perform replication to achieve a certain throughput while minimizing resources. While hardware resources take physical space and resources, many times they are only used a small fraction of the time; Sun's [35] schedulers consider potential resource sharing when selecting modules.

In addition to module replication, FPGAs can also provide parallelism through functional pipelining. Suppose one task requires the outputs from a previous task. If that task can begin before the prior task completes, executing simultaneously, then the schedule is overlapped [36]. This becomes particularly effective for pipelined architectures as actors can be fine-grained parallel operations rather than just the boundaries of larger task completions. Hwang et al. created PLS [37], which minimizes the latency, which in turn reduces resources. PLS uses a forward and backward scheduler to determine both the earliest and latest time that a task can be scheduled; by iterating over the graph the latency can be minimized. Other approaches, such as [38], formulate the problems as a set of constraints and solve using integer linear programming (ILP).

These schedules primarily consider fairly large tasks or functions, and so are not directly applicable to the very fine-grain schedule employed in this schedule. This dissertation considers basic tasks at the binary operation level that have very small constant execution times. This both simplifies parts of the problem and increases the scheduling space significantly.

2.4 Applications

While there are few authors formally utilizing the types of streaming techniques that will be discussed in this research, often a particular application requires elements of streaming, or could be improved upon by using a streaming technique. Therefore, demonstrating the utility of these techniques often requires comparison to specific application areas. This section provides a sampling of current research and implementations of specific applications that can benefit from streaming techniques.

2.4.1 Fast Fourier Transform

There are perhaps few algorithms more heavily researched than the Fast Fourier Transform (FFT), with thousands of papers already published on the topic and new ones always upcoming. Each new FPGA platform with a slightly different multiplier structure may be best served by a slightly different technique. As this dissertation is concerned primarily with throughput, a survey of this application is primarily concerned with implementations targeting high throughput and/or those with a high degree of parallelism. While some of the techniques specifically target modifications of the FFT algorithm itself, rather than the architecture making direct comparison difficult quantitatively, qualitatively insight can be gained through pieces of the implementation details. For example, increasing the radix size of the FFT will result in fewer computation stages, and fewer multipliers, but will not change the basic architecture of the data path.

One high-throughput implementation was built by Zhong [39], which runs eight parallel streams for eight samples per clock, equivalent to roughly two Gsps. This is a radix-8 parallel implementation of a 4096-pt. FFT. It occupies 52,000 logic cells, 1.5Mbits of memory and 84 multipliers with a latency of 575 clock cycles. This technique primarily focuses on improvements using the radix-8 computation structure, but still requires a streaming architecture to obtain the high throughput.

A more complete set of implementations covering multiple throughputs and streaming architectures was developed by Garrido [40]. Using pipelining and parallel streaming, a series of radix 2 FFTs was built for several different samples per clock. At eight samples per clock, a 4096-pt. FFT requires 3540 slices, 120 DSPs and a latency of 5488 clock cycles. One difficulty in comparing implementations beyond just algorithmic differences is that different targeted platforms will give different results, and often only those metrics of interest to the author are disclosed. In this case, the memory requirements of these implementations are not disclosed. In general this technique requires fewer slices, but without numbers of memory, it is unclear if slices were perhaps substituted for memory or if the design itself was much more efficient. Also at eight samples per clock is Milder's approach [41] (Milder formally considers streaming permutations in his design), which requires 148 DSPs and 11500 slices. Slightly more DSPs slices suggest that algorithmically; it is not quite as efficient as others.

Milder [42] also designed an automation tool for the Fast Fourier Transform using the streaming permutation methods in [24, 27]. This spiral hardware generation tool provides options of radices of two through eight, FFT size, as well as permutation method. Computations are performed in single precision floating point. For the 1024-pt. 8 sample per clock FFT, the Spiral generated design costs of about 1400 slices, but again memory cost was omitted.

The FFT is a particularly interesting algorithm for streaming as it is extremely parallelizable. Also, before and after each stage of butterfly computation, stride permutations of differing stride size are required to reorder the data. Finally, to bring the output data

sequence into natural data order, a bit-reversal permutation is also needed. As a result, two different permutation types are required and their frequent usage means optimizations in the streaming data path can greatly impact the efficiency of the implementation.

2.4.2 Matrix Multiplication

Matrix multiplication implemented in hardware is of two types: full and sparse. The basic structure is relatively the same among all studies, but with differing improvements in control. Matrix multiplication exercises a different type of data sequencing as data elements are used multiple times. For full matrix-matrix multiplication, the data sequence is regular (or repetitious), making data control fairly simple. Sparse matrix multiplication complicates this control as not all data elements may be needed.

Matrix multiplication most commonly employs a systolic array to perform the operations necessary to compute each output element. The rows of the first matrix are presented in parallel to an array of multipliers. The other set of inputs is the columns of the second matrix. The set of multipliers produces a result, and then the results are summed with the result from the previous element in the array. Often, the multipliers in FPGAs are coupled with a post adder to create a MACC (Multiply and Accumulate). Using these MACCs, the array elements process one partial product each and accumulate the partial products for each output row. Amira [43] introduced a scalable matrix multiplication using this systolic architecture. Numerous improvements have been made to the same basic structure, including those of Zhuo and Prassana [44], who made improvements in area and efficiency.

Sparse matrices can be described in a number of ways, with the full matrix description or by one specifying only the positions of non-zero elements. Despite this, the required operations are the same, and therefore the basic architecture is the same. The parallel processing elements consist of systolic arrays. The majority of research on sparsity has focused on matrix-vector multiplications [45, 46]. Akella et. al. [47] designed a sparse matrix-vector multiplication for a single FPGA. The primary difference comes in through the control, which determines which data elements must be presented to the multipliers and their timing. Due to the complexity and resource requirements few studies have extended these results to a matrix-matrix multiplication. A notable exception is Lin et. al. [48] who addressed sparse matrix-matrix multiplication by optimizing control for handling sparsity. Each row of operations was partitioned among a fixed number of processing elements.

Sparse matrix multiplication is another interesting application for parallel streaming architectures as the sparsity requires irregular data access, with some input begin used multiple times. Matrix operations tend to be relatively expensive, and small architectural improvements can have a significant impact, especially as matrix sizes increase.

2.4.3 Neural Networks

Neural networks represent a rather unique streaming application as multiple levels of irregular data reordering may be required between each layer of neural network. The irregularity can by introduced by a number of methods, pruning, convolution networks, and dropouts. The multiple layers of data accesses provide opportunities to rearrange operations and schedule them so as to improve latency or other architectural characteristics.

The computation requirements of neural networks include a significant number of multipliers; therefore, implementing a full neural network in hardware fabric may not be possible. Muthuramaligam [49] was able to fit a relatively small neural network of five layers, 1-6-6-6-3, where each number represented the number of nodes in each layer. He used an SVM technique to control a voltage source inverter. He studied how the bit precision affected both the performance of the neural network SVM and the effects of resource cost. The goal was to fit the design within the FPGA while achieving the highest performance and fidelity possible, and he ultimately decided on 16-bit fixed point. Even this relatively small network required most of the FPGA's resources. It was implemented and verified on the Xilinx Virtex 4 XCV400HQ240, which while small by today's standards is still significant.

Fixed point is limited in its ability to represent large dynamic ranges, but it is less expensive in terms of resources. Sahin [50] developed neural network neurons that utilize a library of fast floating point operations. This improves the fidelity of the neural network but requires significantly more resources than fixed point implementations.

As in this dissertation, Gadea [51] developed building blocks for efficient high-throughput neural networks. The structure developed includes an implementation of a systolic array for a multilayer perceptron on a Virtex XCV400. This implementation highly pipelines the backpropagation learning algorithm to improve the clock frequency and throughput. Both the forward and backward phases are performed simultaneously, allowing a high degree of parallelism. While not formally discussed by Gadea, he utilizes the FPGA fabric in combination with the embedded memories to implement the large network interconnection layers.

As very large neural networks often require more resources than available within a single FPGA, a number of techniques are used to overcome this limitation. Techniques like Gadea's [51] are designed for scalability so that multiple devices can be used in combination to implement the full network. Alternatively, memory can be used as it was by Lysaght [52] to store results after each layer and time multiplex the neuron (or nodes) between layers. The interconnects can be designed to be programmable, like the approach used in this dissertation, or if using an FPGA, rapid reconfiguration can be used to reprogram the interconnections and FPGA fabric between layer processing.

The basis for irregular interconnections between layers is found in a variety of neural network techniques. For instance, some connections may be noisy or redundant [53], so removing the connection completely can result in a more robust network. Such connections

are found by using training data with backward propagation techniques. Cun [54] tests the output deviation of removing a connection, while Hassibi [55] measures the second derivative of the deviation. Other techniques have also been proposed. The result is a set of weights with many connections pruned, or equivalently, connections of weight zero.

Connections can also be removed based on their physical meanings [56], e.g. inputs are associated and clustered. Dropouts also improve the performance of a system [57], but they do so by removing nodes within the network. Such techniques result in networks with more sparsely-connected graphs.

The nodes in these pruned networks have a varying number of input and output connections. All nodes are not equivalent. Each node will have a variable number of inputs, and each one is connected to one of a varying selection of input nodes. This makes the scheduling of these neurons challenging and worthy of study.

2.5 EDA Tools for Streaming

Streaming architectures are quite common in customized digital logic and, therefore, it is not a surprise that a significant amount of research has been devoted to increasing productivity through tools to quickly generate designs. While most tools support streaming for a single data element per clock, i.e. a maximum throughput of the clock rate, automated support for parallel elements per clock is extremely rare.

2.5.1 Commercial Tools

Single stream processing has matured to the point that algorithms have found their way into a number of commercial products. Multiple parallel data element processing is not inherently supported in any of these commercial products. In order to use these tools with higher data rates, the structures must still be manually designed and constructed within the tool. This is a time-consuming process, especially compared to the automated processes available for single element streams.

Xilinx has several tools available for use in conjunction with their FPGA product line. These include System Generator [5] and Vivado HLS [58]. These two tools support different design philosophies, but both allow streaming architectures with throughput of up to a single data element per clock. System Generator assumes a streaming model for data. The design is graphical, with each block taking an input at some rate and data precision and outputting a result at another rate and precision. Combining simple blocks allows larger algorithms to be developed quickly. One of its weaknesses is that fine-grain control of the particular data elements within the stream is difficult to implement without returning to a low-level design

supported as a custom block within the design.

Vivado HLS is a tool to convert a subset of C and C++ code into a synthesizable netlist for the Xilinx FPGAs. It requires a subset of the language suitable for hardware and again may require customization to best target hardware. It is possible to specify streaming inputs and outputs and supports directive for loop unrolling and pipelining to give a level of control to the resources and processing rate of the function. Multiple streams again require that individual streams be manually processed together to achieve data rates above the clock rate.

Altera's tools are based on converting OpenCL code into a netlist for Altera FPGAs. The advantage of OpenCL is that it was developed with the intention of targeting parallel processing. It then tends to map more readily to digital logic, but code written for a GPU still may not translates well to hardware. This requires that the generic OpenCL code be customized for the desired target. Furthermore, the translation is locked to Altera devices.

Synopsys provides a suite of tools similar to the Xilinx offerings, but they are platform-agnostic. The output can be targeted to both FPGA platforms as well as ASICs. Synphony Model Compiler [59] is similar to System Generator [5]; both are an add-on to MATLAB's Simulink with a library of basic building blocks with the tools supporting parameters for these blocks and automatically connecting the blocks and generating a netlist. Similarly, Synphony C Compiler [60] converts C code to a digital netlist. Both have the same limitations as their Xilinx counterparts, but they differ primarily in terms of the library of components and C conversion algorithms.

There are a number of tools from other commercial vendors that have a variety of differing implementation details, but in terms of scope, user interface, and process input and output, they are largely similar to those already described.

2.5.2 Research Projects

There is significant research based on projects for high-level synthesis, that is generating hardware from higher level languages such as C. As these projects are academically-based, their overall functionality is generally less impressive than commercial tools; they primarily serve as a test suite for improvements in CAD algorithms. None of these suites implicitly supports parallel data in streams; most take C input and translate into a generic HDL description. While these research projects generally consist of research into many different algorithms and related topic areas, only a single relevant paper will be cited per project. AUGH [61] is designed for generating FPGA hardware accelerators under resource constraints. PandA [62] supports hardware/software partitioning and mapping, as well as performance estimates for determining the partition placement. LegUp [63] is another project developed for hardware software co-design. GAUT [64] was specifically developed for extracting the potential parallelism in DSP algorithms and automates maximizing throughput, but

again fails to address parallel processing streams. While there are other projects, these listed are still under development and representative of the general capability of current research tools.

One of the few CAD tools that supports parallel streams is the Spiral hardware generator project [65]. Spiral is a project based on formally describing mathematical algorithms with compilers for a variety of platforms to convert these algorithms into efficient implementations on a variety of platforms, including digital logic as well as Intel processors [66]. While only a few algorithms have been optimized for digital logic platforms, these include the discrete Fourier transform [42], multiplierless multiple constant multiplication [67], streaming sorting networks [68], and LDPC codes [69]. The final synthesis of all the pieces into one combined tool will be quite powerful, but for now the individual generators output a single parametrized algorithm implementation.

2.5.3 Summary

This chapter covers a wide range of existing topics ranging from parallel processing, stream processing, scheduling, target applications and tools for generating streaming architectures. The discussion of parallel and stream processing is intended to inform the reader of the current state of these techniques to which later results may be directly compared. Applications are informative, as they demonstrate the utility of these parallel stream processing techniques, not just on a component level, but in how they affect algorithm implementations on a larger scale. Finally, the state of the electronic design tools shows the gaps in capability which this research is intended to fill. It demonstrates the importance of this body of research.

Later chapters will explore these works in more detail and show how they compare with the techniques and algorithms developed. With all results, only those details that are available can be compared directly, which can make apples-to-apples comparisons difficult. Where possible, direct comparison is provided, and where details are not available qualitative analysis is performed, with analysis those details that are provided to make a comparison as realistic as possible.

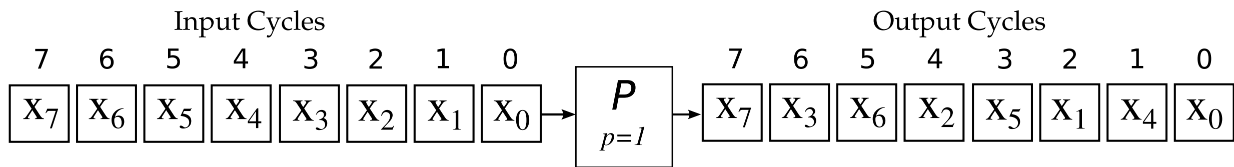
Chapter 3

Streaming Permutations

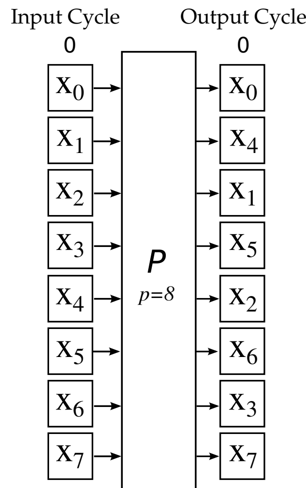
It is typical for a streaming architecture to form a long datapath composed of computation stages separated by data permutations. A data permutation is a re-ordering of the elements of a data sequence. Figure 3.1 illustrates a permutation for different streaming widths. In Figure 3.1a a streaming implementation with the minimum width of 1, is illustrated, a trivial case requiring a single RAM whose memory accesses determine the permutation sequence. This permutation is in time; that is, elements are reordered into different clock cycles. Conversely, Figure 3.1b shows the fully parallel implementation where all $n = 8$ elements of the vector are available the same clock cycle. This is a permutation in space, as the permutation consists of wires rerouting the positions of the elements. Finally, Figure 3.1c shows a case in which data arrives in subvectors of length $p = 2$ during consecutive clock cycles. In this case the permutation is in both space (wires and switches) as well as time (delays through memory).

A permutation is a reordering of a data sequence. A subclass of the general permutations is linear permutations. Linear permutations are linear transforms of the bits of the data indices. They are commonly used in a wide variety of signal processing applications. Among the most important are stride permutations (also known as matrix transpositions or corner turns), which are used in a wide variety of applications: linear transforms such as fast Fourier transform, Walsh-Hadamard transform, sine and cosine transforms, multi-rate and multi-dimensional signal processing, sorting networks and Viterbi coding. General permutations that do not fit in this subclass are much less frequently needed and are generally based on sparsity, or randomness conditions, such as sparse matrix multiple [8], or partially-connected artificial neural networks [**?**].

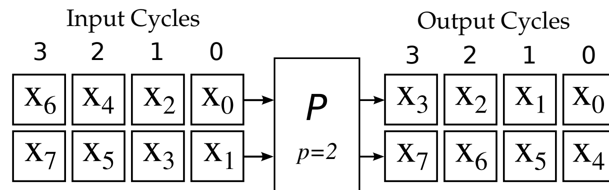
This chapter explores both linear and general streaming permutation architectures. The details of the structure and implementation requirements are described. The linear architecture is optimized for either of number of switches, or connectivity and control cost. The



(a) Fully temporal streaming permutation of eight elements with a streaming width of one.



(b) Fully spatial streaming permutation of eight elements with a streaming width of eight.



(c) Spatial and temporal streaming permutation of eight elements with a streaming width of two.

Figure 3.1: Illustration of perfect shuffle (stride) permutation over eight elements with three different implementations with different streaming widths.

general architecture focuses on minimizing the latency and memory requirements of the datapath. The advantages and disadvantages of each architecture are presented and a framework for a fair comparison of the implementation types developed.

3.1 Background

A permutation π acting on n elements, $\pi : \{0, 1, \ldots, n-1\} \mapsto \{0, 1, \ldots, n-1\}$, can be represented as an $n \times n$ matrix P_π. There are two ways to represent this matrix, and they are transposes of each other. By convention the permutation matrix is written so as to act on an n-dimensional column vector. This means that all entries are zero except that in row

i, column $\pi(i)$ equals 1.

$$P_\pi = \begin{bmatrix} \mathbf{e}_{\pi(0)} & \mathbf{e}_{\pi(1)} & \cdots & \mathbf{e}_{\pi(n-1)} \end{bmatrix}^T,$$

where $\mathbf{e}_j$ is a row vector of length n with 1 in the jth position and zero elsewhere. The set of all permutation on n points forms the symmetric group, S_n. The size of this group is $n!$.

If the number of elements in the permutation is a power of two, where $n = 2^m$, and there exists a matrix L, where L is an $m \times m$ matrix that describes how the permutation $\pi(L)$ applies to the binary representation of the elements, then $\pi(L)$ is a linear permutation [25]. If the index of an element is i, then the binary representation is a column vector of m bits. Using the notation $_b$ to represent binary notation, for $i = 4$, $i_b = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}^T$. The most significant bit is at the top. Then the bit matrix L acting on the bit index representation is equivalent to:

$$L \cdot i_b = (\pi(L)(i))_b.$$

$L \in GL_m(\mathbb{F}_2)$, where $\mathbb{F}_2$ is the Galois field with two elements. Therefore, the set of linear permutations is also a group.

The subgroup of permutations S_n that are linear is of size $\prod_{i=0}^{m-1}(2^m - 2^i)$. It is important to note the limitations of the group of linear permutations. It requires that the size of the data set it acts on be a power of two, $n = 2^m$. In addition, not every permutation on 2^m points is linear. In particular, to be linear the permutation must always leave the first element unchanged; the first element is the zero vector so it maps to $L \cdot 0_b = 0_b$.

Just as with groups, multiplications of the permutation matrices and bit matrices for linear permutation are also members of the group, and are compositions of their permutations. So, if $R = P \cdot Q$ then $\pi(R) = \pi(P) \circ \pi(Q)$. This allows more complicated permutations to be written as a product of simple permutations. This will become relevant in Section 3.4.

One permutation of note is the perfect shuffle or stride permutation. The perfect shuffle on $n = 2^m$ elements interleaves the first and second halves. Data element i maps to

$$i \mapsto \begin{cases} 2i, & \text{if } 0 \le i < 2^m - 1, \\ 2i - 2^m + 1, & \text{if } 2^{m-1} \le i < 2^m. \end{cases}$$

The equivalent bit representation is a cyclic shift: $i_b \mapsto i'_b = C_m \cdot i_b$, where

$$C_m = \begin{pmatrix} & 1 & & \\ & & \ddots & \\ & & & 1 \\ 1 & & & \end{pmatrix}.$$

3.2 Parameterized Datapath

The basic structures of both the linear and general implementations are presented in this section. The general operation and structure are described, but the control and implementation details are defined by the particular implementation.

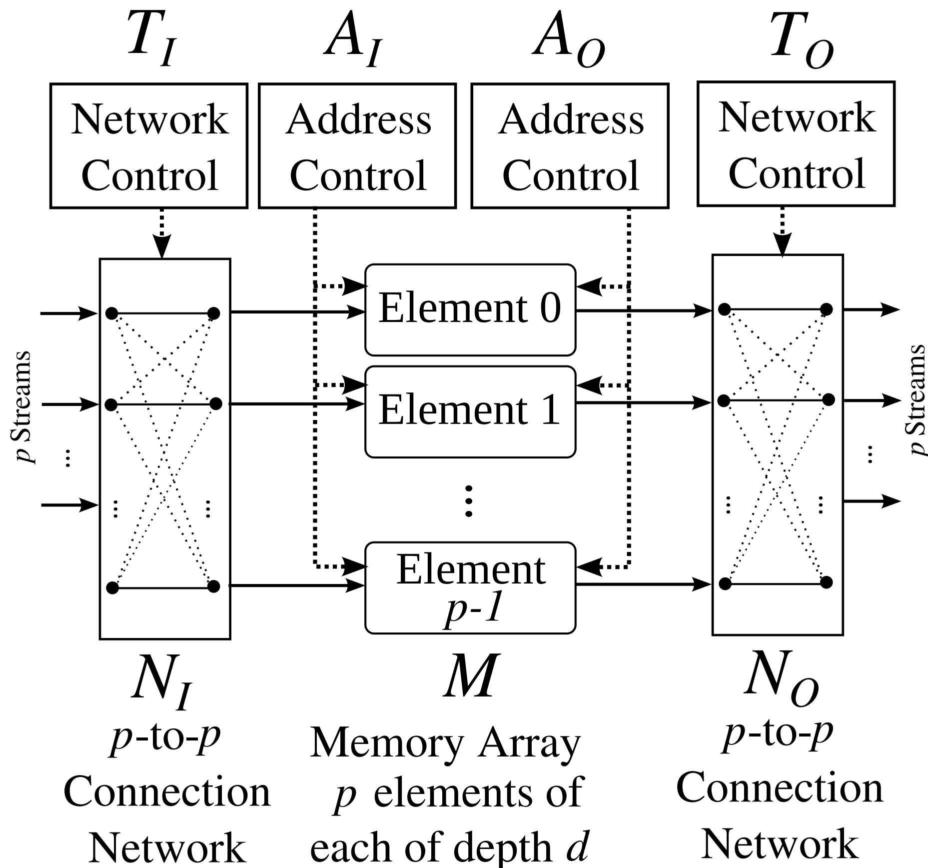


Figure 3.2: Permutation architecture.

Figure 3.2 shows the parametrized datapath architecture. It consists of a memory array

M with independent input and output addressing (i.e., dual port) for each element. Two switch networks N_I and N_O are on both the input and output ports of the memory array. Control logic for the write and read side of the RAMs and the write and read side switch networks are denoted $A_I, A_O, T_I,$ and T_O respectively.

The memory array M contains p parallel memories, each with capacity d. Each has one read port and one write port. These memories can be implemented as block RAM, or distributed memory (registers). Associated with each memory, there is a latency ℓ_M, which is the time it takes data at the write port to be produced at the read port when both addresses are the same.

One cycle of input data takes the following path through this architecture. First, p data elements are provided to N_I. The network takes the p elements and permutes them onto the p outputs. The p elements are then written into the RAMs at addresses defined by A_I. After a delay of ℓ_{data} clock cycles, the p RAMs are read according A_O, which controls the p output addresses. The data then pass through N_O, which reorders the p elements according to T_O.

One of the primary differences between the linear and general implementations are the source of the control logic for the switch network and RAM addressing. The linear implementation represents the switch control and RAM addresses as a binary matrix operations requiring a series of AND and XOR operations applied to a cycle counter. The general implementation makes no assumptions on the data permutation and thus cannot represent it as a binary transformation. Instead it uses lookup tables (ROMs) to store the control logic. The configuration of the switch network is controlled by the stored values in T_I and T_O, which are precomputed for the n/p cycles and each configuration requires $\sum_{i=1}^{p} \lceil \log_2 i \rceil$ bits. Each line of the address lookup tables A_I, and A_O require p addresses of $\lceil \log_2 d \rceil$ bits.

The implementation of the switch network is also quite different. The linear implementation minimizes the number of switches and provides only the connections necessary to implement the desired permutation. The general implementation uses a generic switch network that can perform any spatial permutation. It takes p data elements as input and outputs them in any order at the p output ports. Each switch is controlled by a single bit that either propagates the inputs to outputs in the same order or swaps the output order.

An alternative formulation [25, 27] uses two memory arrays sandwiching a single switch network. This architecture is able to realize the minimum number of switches for linear permutations, but it requires twice the number of memories and so the details of the structure are omitted.

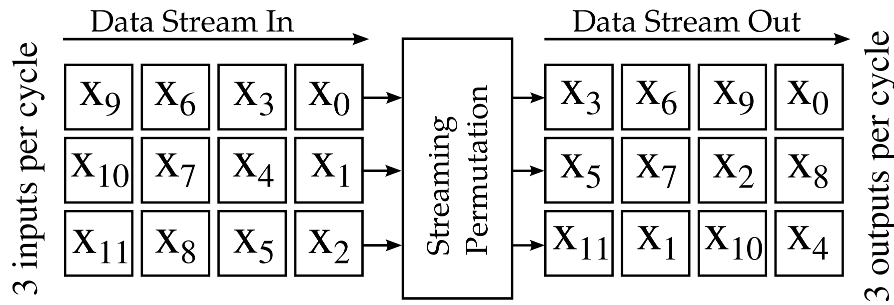


Figure 3.3: Bit-reversal permutation streamed 3 per cycle.

3.3 Minimum Number of Memory Banks

Consider an application processing a block of $n = 12$ samples. In the streaming case, Figure 3.3, $k = 3$ samples are provided each clock, and it takes four cycles to receive the entire vector. The input data stream consists of vectors of three samples, so the first input vector consists of samples x_0, x_1 and x_2, where the subscript indicates the sample number in time. The input is sequentially ordered with the first three samples in time available in the first data vector and the next three samples in time available in the second data vector. The problem is complicated by the permutation requiring a reordering in space (between the k rows of input) and in time (between the columns of input).

The minimum number of memory elements or modules to implement a computation within an algorithm is determined by the need for conflict-free memory access. One way to eliminate a conflict is to add an additional memory element and move one of the conflicting elements to it. If the extra bank avoids just one conflict, its size is quite small, but the connection networks must grow to add that bank to the network. The network complexity grows as $\mathcal{O}(p \log p)$, so it is best to minimize the number of memory elements.

The minimum number of memory elements can be determined by recasting the problem to a graphical representation. Figure 3.4 shows the translation of the example in Figure 3.3 to a graph. The data are output after a delay of ℓ cycles; this is the latency of the datapath. The input data at input Cycle 1 are contained in vertex i_1. Those data elements are found in output vertices o_0, o_1, o_3, so edges are drawn to those vertices.

The graph will be bipartite as the nodes can be divided into different classes. In this case, input and output nodes are the two classes. No input node is connected to another input node, and no output node is connected to another output node. Each edge represents a different memory element access. If there are multiple edges, each edge must correspond to a different memory element to avoid a conflict. In graph theory, the edges can be colored or labeled, with each color corresponding to a memory element. The number of different

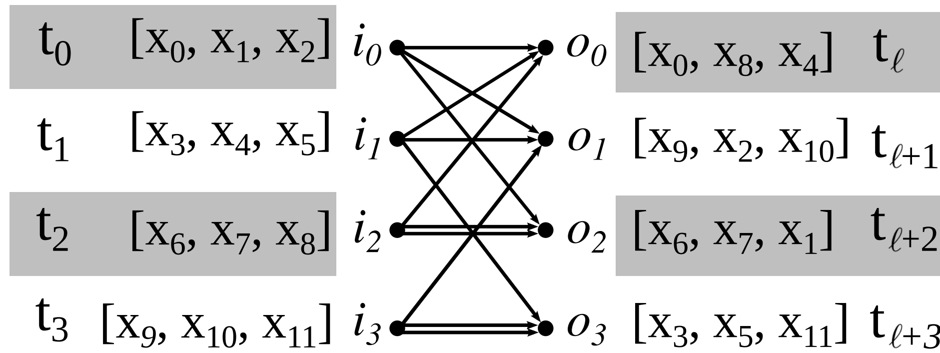


Figure 3.4: Map of input/output data vectors to a graphical representation.

colors will determine the number of memory elements needed.

Lemma 1. *Given an input of p elements each clock then the minimum number of memory elements needed for conflict-free memory access is the maximum number of edges connected to any node.*

Proof. Since the graph is bipartite, König's Line Coloring Theorem states that the minimum number of colors needed to color a bipartite graph is equal to the maximum degree of the graph [70, 71]. The degree of a node is the number of edges connected to it. Since each memory bank can be mapped to a particular edge color, the minimum number of memory banks will be the maximum degree of the graph. □

This lemma can be extended to determine the maximum number of memory banks for any size of any input and output of p elements to guarantee conflict-free memory access.

Corollary 1. *Given a permutation and a datapath with an input of p elements per clock and output of p elements, then the maximum number of memory banks needed to guarantee conflict-free memory access is p.*

Proof. Since each input node contains p data samples, each input node can have at most p edges connected to it. Likewise, since each output vector contains p data samples, at most p edges will be connected to it. The maximum degree of any node will be p. At most p colors or memory elements will be needed to color the graph. □

3.4 Streaming Linear Permutations

This section presents an overview of constructing the control structures and switching for streaming linear permutations. For more in-depth discussion and examples see [24, 25].

The streaming permutation performs the operation $j = P \cdot i$, or uses the bit representation, $j_b = L \cdot i_b$. When the streaming width p is a power of two, $p = 2^k$ then for a sequence of elements indexed 0 to $n = 2^m - 1$, an element with index $i = 2^k i_2 + i_1$ enters on the i_2th clock cycle on the i_1th input stream. This means that the bits of the index can be factored such that the cycle number i_2 is the $t = m - k$ upper bits of i_b and i_1 is the lower k bits, $i_b = \begin{pmatrix} i_2 \\ i_1 \end{pmatrix}$. This suggests the bit permutation matrix L be factored similarly as

$$L = \begin{pmatrix} L_4 & L_3 \\ L_2 & L_1 \end{pmatrix}, \text{ where } L_4 \text{ is } t \times t.$$

Then the output bit representation j_b of input index i_b can be written as

$$\begin{matrix} \textit{output cycle number} \\ \textit{output stream number} \end{matrix} \begin{pmatrix} j_2 \\ j_1 \end{pmatrix} = \begin{pmatrix} L_4 & L_3 \\ L_2 & L_1 \end{pmatrix} \begin{pmatrix} i_2 \\ i_1 \end{pmatrix}.$$

The bit matrix can decomposed into the product of multiple bit permutations that represent hardware structure capable of implementing them. These are RAMs that implement spatial permutations, and a switch network that implements temporal permutations. Using these structures in series will allow general linear bit permutations to be constructed.

3.4.1 Spatial Permutations

A spatial permutation by definition only permutes elements within the same clock cycle. For this to be true, after applying permutation $L, j_2 = i_2 = L_4 i_2 + L_3 i_1$. Then L has the form

$$L = \begin{pmatrix} I_t & \\ L_2 & L_1 \end{pmatrix}.$$

The switch network can be implemented via controlled 2×2 switches. In a single clock cycle, all 2^k indices in i_1 are present, so the switch configuration will only depend on the cycle number. This indicates that L_2 will determine the complexity of the switch network.

Theorem 3.4.1. *A full-throughput implementation for bit matrix L with a streaming width*

$p = 2^k$ that only uses 2×2-switches for routing requires at least rank$(L_2)\cdot 2^{k-1}$ many switches.

For the full derivation see [25]; but an abridged proof is presented here. Consider the set of output streams with which input stream v communicates. This set contains $2^{\mathrm{rk}(L_2)}$ elements. The set of cycles which transits from input stream v to output stream v' contains $2^t - \mathrm{rk}(L_2)$ elements. Since the size of the set is independent of v', the distribution over the possible output ports is uniform. The number of switches that an element received on cycle c and stream v passes through is denoted $\ell_{c,v}$. The number of bits on average to represent the $2^{\mathrm{rk}(L_2)}$ possible output ports for an element on input stream v is $\log_2 2^{\mathrm{rk}(L_2)} = \mathrm{rk}(L_2)$. An element received on stream v goes through at least that number of switches to reach the output port.

$$\frac{1}{2^t}\sum_{c=0}^{2^t-1}\ell_{c,v} \geq \mathrm{rk}(L_2), \text{ for every } v. \tag{3.1}$$

Considering that a switch has two input elements, 2^{t+1} elements pass through each switch. Then if there are s switches,

$$\sum_{\substack{0\leq c<2^t\\0\leq v<2^k}}\ell_{c,v} \leq s\cdot 2^{t+1}. \tag{3.2}$$

Combining (3.1) and (3.2) and solving for s yields

$$s \geq \frac{1}{2^{t+1}}\sum_{v=0}^{2^k-1}\sum_{c=0}^{2^t-1}\ell_{c,v} \geq \frac{1}{2}\sum_{v=0}^{2^k-1}\mathrm{rk}(L_2).$$

Serré and Püschel [24, 25] both use shortened Omega networks to implement these switches. One stage of an Omega network consists of a perfect shuffle followed by a column of $2^k - 1$ 2x2 switches [72]. The column of switches is controlled by a common control bit. If the bit is set the pairs of elements are swapped. Figure 3.5 illustrates a stage of an Omega network.

The desired switch configuration is controlled by the clock cycle, which can be generated using a counter of t bits. This counter value is equivalent to i_2 for all 2^k inputs in a given clock cycle. A stage of switches should toggle when $i_2 \cdot z$ equals one where z is a column vector of t bits. Since these are bit representation in GF_2, this is implemented using XOR

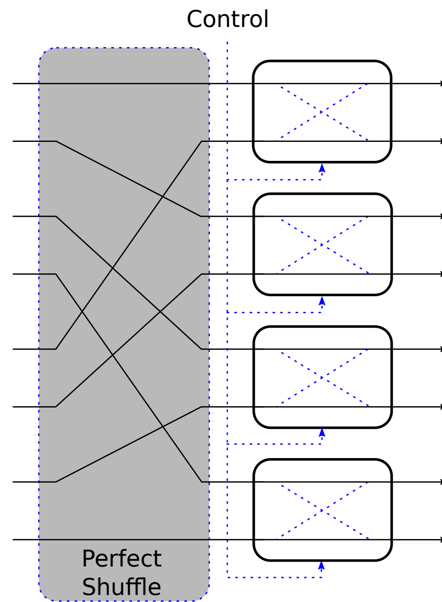


Figure 3.5: One stage of Omega Network. It consists of a perfect shuffle of p input ports followed by $p/2$ 2×2 switches. All the switches in the stage are controlled by one common bit.

gates. The matrix representation of the permutation controlled by the switches is

$$K_z = \begin{pmatrix} I_t & & & \\ & 1 & & \\ & & \ddots & \\ z^T & & & 1 \end{pmatrix}.$$

Combined with the perfect shuffle (using wires) a stage of the Omega network is described by the matrix:

$$S_z = K_z \cdot \begin{pmatrix} I_t & \\ & C_k \end{pmatrix},$$

where C_k is the bit matrix representing a cyclic shift on the bits (the perfect shuffle). Using this switch structure requires casting the bit matrix L into a formulation with $\text{rank}(L_2)$ stages of Omega network. This is done by applying Gaussian elimination on L_2 to find an invertible $k \times k$ matrix H such that HL_2 has $\text{rank}(L_2)$ non-zeros lines at the top corresponding to the

switch control vectors z_i^T:

$$HL_2 = \begin{pmatrix} z_1^T \\ \vdots \\ z_{\text{rank}(L_2)}^T \\ 0 \\ \vdots \\ 0 \end{pmatrix}.$$

HL_2 determines the control of the switching bits and computing L in terms of H yields

$$L = \begin{pmatrix} I_t & \\ & H^{-1}C_k^{k-\text{rk}(L_2)} \end{pmatrix} S_{z_{\text{rk}(L_2)}} \dots S_{z_1} \begin{pmatrix} I_t & \\ & HL_1 \end{pmatrix}.$$

The bottom right entries in the two outside matrices correspond to constant wirings. There are $\text{rk}(L_2)$ stages of 2^{k-1} switches.

3.4.2 Temporal Permutations

In temporal permutations, the input stream must equal the output stream, $j_1 = i_1 = L_2 i_2 + L_1 i_1$. This simplifies L to the form

$$L = \begin{pmatrix} L_4 & L_3 \\ & I_k \end{pmatrix}.$$

The most common method of implementing the temporal permutation uses 2^k RAM banks. There are a number of different schemes for implementing these RAM banks either single [73] or dual-port RAMs [24, 27]. The dual-port RAM option will be described in detail.

The dual-port structure allows independent read and write addressing to the RAM banks; providing two degrees of freedom. Implementing the temporal permutation L requires only one degree of freedom, so a common method is to use the cycle counter of t bits to generate the write address for each bank of RAM. The read address is computed in permuted order as $L_4^{-1} i_2 + L_4^{-1} L_3 i_1$. The first component can be computed once for all the banks, as i_2 is constant for all the data element of a particular cycle. The second component $L_4^{-1} L_3 i_1$ is added to the common term for each bank independently. This second term is a constant per stream since i_1 just depends on the stream number.

One challenge that presents itself for many permutations is that some elements of a

dataset may be written to a memory address that contains an element from the previous dataset that has not yet been read. One strategy is to use double-buffering [42]. The effective size of each RAM doubles as an additional top bit toggles after each data set. This is cost-effective in terms of logic, but less so in terms of memory. An alternative approach swaps which side of the RAM uses the counter as the address after each data cycle. This increases the control cost but not the RAM size.

3.4.3 General Linear Permutations

A general linear permutation may be implemented in one of several ways based on how the permutation is factored. Püschel chooses to factor the permutation at the read/write RAM boundary [24], while Serré factors the permutation into temporal and spatial components [25].

3.4.3.1 Jointly Minimizing Control Cost and Switches

Püschel's approach attempts to minimize the overall control cost of both the RAM addressing and switch networks. The permutation L is performed as $j = Li$ in two stages, a write stage, $w = Mi$, and a read stage, $j = N^{-1}w$.

A measure of the cost of control is the linear complexity [74], the minimum number of binary additions needed to compute $w = Mi$, and $j = N^{-1}w$. Püschel frames the linear permutation as a factorization of $L = N^{-1}M$, such that $\mathrm{rk}(M_1) = \mathrm{rk}(N_1) = k$. The goal in this factorization is to minimize the $\mathrm{rk}(M_2)$, $\mathrm{rk}(N_2)$ and the control cost of M and N. The constraint on the rank of M_1 and N_1 is a consequence of requiring that each data stream map to only a single RAM each cycle. As seen previously, $\mathrm{rk}(M_2)$ and $\mathrm{rk}(N_2)$ determine the number of switches needed for the switching network.

This problem is solved by using a form of N^{-1} that conforms to the minimum $\mathrm{rk}(N_2)$ and control cost for N. For permutations of just the bits in the matrix the solution yields the minimum $\mathrm{rk}(M_2)$ and control cost. In general, only the $\mathrm{rk}(N_2)$ and control cost of N is minimal. To determine the lower bounds, L and L^{-1} are divided into their components as

$$L = \begin{pmatrix} L_4 & L_3 \\ L_2 & L_1 \end{pmatrix} \text{ and } L^{-1} = \begin{pmatrix} L'_4 & L'_3 \\ L'_2 & L'_1 \end{pmatrix}.$$

Theorem 3.4.2. *Assume $L = N^{-1}M$ is a factorization of L, where rk(M_1) = rk(N_1) = k,*

then

$$rk(M_2) \geq k - rk(L_1') \text{ and } rk(N_2) \geq k - rk(L_1).$$

Proof. If $L = N^{-1}M$, then $NL = M$. Solving for M_1, $M_1 = N_2L_3 + N_1L_1$. Considering the ranks of these matrices,

$$\begin{aligned} \text{rk}(M_1) &= \text{rk}(N_2L_3 + N_1L_1) \\ k &\leq \text{rk}(N_2L_3) + \text{rk}(N_1L_1) \\ &\leq \min(\text{rk}(N_2), \text{rk}(L_3) + \text{rk}(L_1). \end{aligned}$$

Since N_1 has full rank, the $\text{rk}(N_1L_1)$ is just the $\text{rk}(L_1)$. Then $\text{rk}(N_2) \geq k - \text{rk}(L_1)$. The bound for M_1 is obtained similarly with $MP^{-1} = N$ as the start. □

The lower bounds on the control cost are similarly derived:

Theorem 3.4.3. *cost(M) $\geq k - rk(L_1')$ and cost(N) $\geq k - rk(L_1)$.*

Proof. In the previous proof, $\text{rk}(N_2) \geq k - \text{rk}(L_1)$. Since the field is $\mathbb{F}_2$, then N_2 contains at least $k - \text{rk}(L_1)$ non-zero elements. The $\text{rk}(N_1) = k$, so the linear complexity of the matrix $\begin{bmatrix} N_2 & N_1 \end{bmatrix}$ and therefore also of N is also at least k-$\text{rk}(L_1)$. The same discussion applies to the cost(M). □

Now that the lower bounds are known, N is designed to meet those lower bounds, and a M is found to minimize $\text{rk}(M_2)$ and cost(M) with the constrained N. N is chosen to be in the form:

$$N = \begin{pmatrix} I_{m-k} & \\ N_2 & I_k \end{pmatrix}$$

and is also its own inverse (see [24] for further detail). To satisfy $M = NL$, then $M_1 = N_2L_3 + L_1$. If $\text{rk}(M_1) = k$, then a factorization has been found. If the matrix N_2 is the zero

matrix except for location $e_{i,j}$, then M_1 is L_1 with the jth row of L_3 added to its ith row. To make M_1 full rank it is sufficient to choose $k - \text{rk}(L_1)$ rows of L_3. Since L is full rank, the submatrix $\begin{pmatrix} L_3 \\ L_1 \end{pmatrix}$ also has full rank, so the rows of L_3 can be chosen to make M_1 full rank in the case that L is a bit permutation. The column indices $i_1, \ldots, i_k - \text{rk}(L1)$ correspond to the rows of zeros in L_1, and the column indices $j_1, \ldots, j_{k-\text{rk}(L_1)}$ correspond to the non-zero basis vectors in L_3. Then N_2 has ones at locations $(i_1, j_1), \ldots, (i_{k-\text{rk}(L_1)}, j_{k-\text{rk}(L_1)})$.

In the general case, the rows of L_3 are not linearly independent. To choose the correct basis rows, the transformation $Q_1 = L_1 G$ is performed. The transformation consists of permuting the $\text{rk}(L_1)$ linear independent columns of L_1 into the first $\text{rk}(L_1)$ location and performing Gaussian elimination to zero out the last $k - \text{rk}(L_1)$ columns: essentially finding a transformation to find the linearly independent rows. Setting $Q_3 = L_3 G$ yields

$$\begin{pmatrix} L_3 \\ L_1 \end{pmatrix} = \begin{pmatrix} Q_3 \\ Q_1 \end{pmatrix} G.$$

The matrix $\begin{pmatrix} Q_3 \\ Q_1 \end{pmatrix}$ has the desired structure. The one value row indices $(i_1, \ldots, i_{k-\text{rk}(L_1)})$, of N_2 correspond to the $k - \text{rk}(L_1)$ linearly dependent rows of Q_1. The one value column indices $(j_1, \ldots, j_{k-\text{rk}(L_1)})$ correspond to the $k - \text{rk}(L_1)$ linearly independent rightmost rows of Q_3. Then, by construction $N_2 Q_3 + Q_1$ has full rank, and $M_1 = (N_2 Q_3 + Q_1) G^{-1}$ does as well.

N_2 from the construction has rank $k - \text{rk}(L_1)$, which is minimal. There are $k - \text{rk}(L_1)$ additions required in implementing N, which is also minimal. Consider $M_2 = N_2 L_4 + L_2$; if the permutation only permutes the bits of data index permutation then selecting the $k - \text{rk}(L_1)$ non-zero rows from L_3 guarantees that the corresponding row in L_4 will be zero since a bit permutation will only have one bit per row. The result is that $N_2 L_4 = 0$ and $M_2 = L_2$. The rank of $M_2 = k - \text{rk}(P_1)$ is minimal, as well as the number of additions required by M, $k - \text{rk}(P_1')$. When the permutation is not the special case of a bit permutation this optimal cost and connectivity does not hold, however, N is still minimal.

3.4.3.2 Minimizing Switches

An alternative formulation factors the bit matrix permutation at the spatial and temporal boundaries. While both a RAM-switch network-RAM and switch network-RAM-switch network formulation result in functionally correct architectures, the switch network-RAM-switch network requires half the RAM but in at least as many switches. Then L is factored

as a temporal permutation sandwiched between two spatial permutations:

$$L = \begin{pmatrix} I_t & \\ N_2 & N_1 \end{pmatrix} \begin{pmatrix} M_4 & M_3 \\ & I_k \end{pmatrix} \begin{pmatrix} I_t & \\ R_2 & R_1 \end{pmatrix}.$$

Serré [25] minimizes rk(N_2)+rk(R_2), which is equivalent to minimizing the number of switches. This does not consider the cost of control, but in many cases it reduces to solutions equivalent to Püschel's [24] as described in Section 3.4.3.1. Such a decomposition exists and achieves the bound [75]:

$$\text{rk}(N_2) + \text{rk}(R_2) \geq \max(\text{rk}(L - 2), m - \text{rk}(L_4) - \text{rk}(L_1).$$

If rk(L_4)-rk(L_1) $< m$, then this solution does not achieve the minimum number of switches. The alternative architecture of RAM-switch network-RAM does, but it costs twice as much in RAM.

This section presented the current methods in handling linear permutations. The limitations of this method are that the permutation must be streamed with a power of two width, and the permutation itself must be linear. However, this formulation has inexpensive control and switching cost, and in some cases it achieves the minimum number of switches.

3.5 General Streaming Permutations

General permutations are a broad class including both non-power of two streaming widths and datasets as well as permutations that cannot be described by a bit representation matrix. In these cases, the size of the permutation matrix P is $n \times n$, which can be quite large. Since it cannot be represented compactly with the bit representation, it is also unclear how to generate the control compactly.

The first working implementation by Milder [27] used a RAM intensive RAM-switch network-RAM formulation. When implementing permutations equivalent to Püschel's [24], it required twice the RAM in the datapath and also used ROM lookup tables for the control. Chen [28] improved on this by formulating the problem as a Benes network [13] and was able to reduce the RAM cost by half. The approach described in this chapter uses techniques from both and was originally presented in [6]. This technique further reduces the memory and thus is best suited of those cases mentioned for comparison against the linear case.

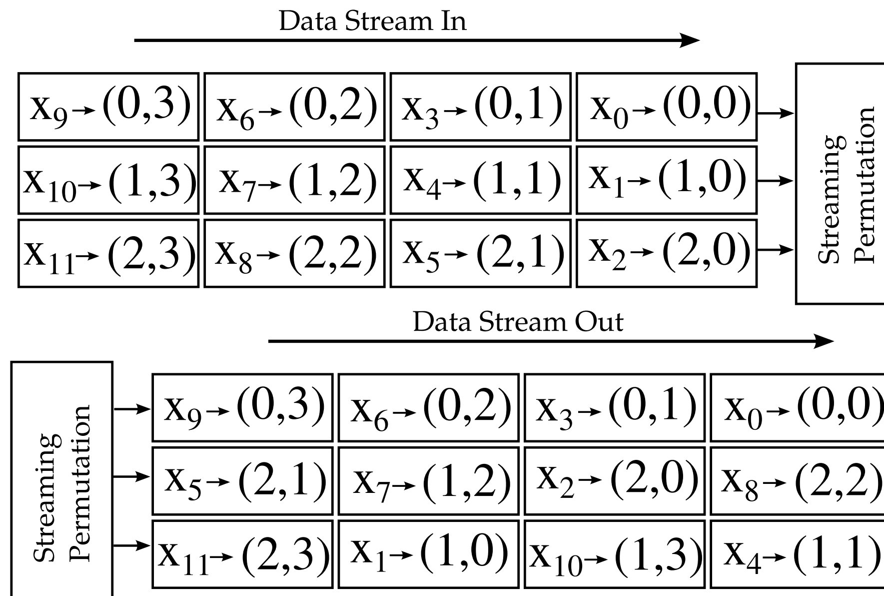


Figure 3.6: Initial mapping of data elements to memory element and address locations.

3.5.1 Spatial Component

Figure 3.6 shows the mapping of the permutation elements into the tuple (memory element, address). The memory element is initially assigned by the data element index i modulus k. The top corresponds to the input and the bottom to the output. If in any cycle more than one element is mapped to the same memory element, then a conflict will occur. In the bottom output of Figure 3.6, at Cycle 2, x_7 and x_1 are both mapped to spatial location 1 causing a conflict. Guaranteeing full throughput is equivalent to making sure that no conflicts occur.

The spatial permutation is resolved independently of the temporal by removing all time dependencies in the data, by considering only the spatial aspect of each data element. In Figure 3.4 each of the data elements in each input and output cycle are assigned a node. The edges connecting the nodes correspond to different RAM banks. Conflict-free memory access can occur if the graph is properly edge-colored. This means that at any node no more than one edge is assigned a particular color or label; no input or output cycle accesses the same RAM more than once. There are a number of techniques for proper edge coloring graphs with various run times [76–79].

The below implementation uses the initial memory element mapping as a starting point to reduce the number of algorithm steps. Each conflict is resolved using a technique similar to Gabow's recoloring algorithm [79]. This is described by Algorithm 1.

Algorithm 1 Generate Switch Configurations

```
conflict ← True
while conflict do
    [conflict, t, s, newColor] = FindConflict(outputTable)
    if conflict then ResolveConflict(t, s, false, newColor)
    end if
end while
function ResolveConflict(t, s, input, newColor)
if input then
    inputTable(t, s) ← newColor
    [i, j] = GetOutputAddressAtInput(t, s)
    outputTable(i, j)← newColor
else
    outputTable(t, s) ← newColor
    [i, j] = GetInputAddressAtOutput(t, s)
    inputTable(i, j) ← newColor
end if
[conflict, t, s, newColor] ← checkConflict(not input, i, j)
if conflict then
    ResolveConflict(t, s,not(input),newColor)
end if
```

Algorithm 1 finds each conflict in the graph and recolors or assigns a new spatial location to one of the conflicting elements. The variable `InputTable` is the spatial index of the input in Figure 3.6. It is a two-dimensional table by cycle number t and spatial index s containing the spatial assignment or memory element for each data element. Likewise, outputTable corresponds to the spatial index of the output in Figure 3.6. For example, in Figure 3.6, x_8 has an initial memory element assignment of 2, which would be located at $(2, 2)$ in the input table and at $(0, 1)$ in the output table.

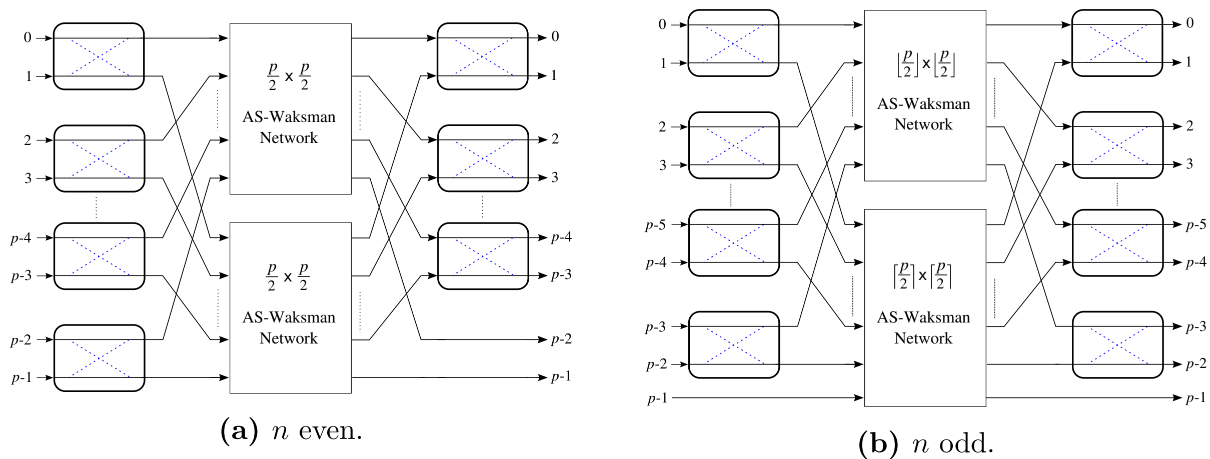
The function `findConflict` iterates through each cycle time t in the `outputTable` and looks for conflicts in the spatial assignment. If a conflict is found, it sets the variable `conflict` to true and returns the cycle time t and space index s of the data element that is to be modified to remove the conflict. It also returns `newColor`, a new spatial assignment that was previously not used at that cycle time. If no conflict was found, then `conflict` is false and the algorithm completes.

If a conflict is found, `ResolveConflict` is called with the location (t, s) of the data element to be modified, whether this conflict is in the `inputTable` or `outputTable`, and the new spatial assignment, `newColor`. The function `ResolveConflict` makes the change and propagates it to the corresponding table so that both tables know the correct location of each data element. It does this in the following way. If the conflict is in the input table, then the location (t, s) in the input table is changed to the new spatial assignment, `newColor`. Then, the function `GetOutputAddressAtInput` executes returning the `outputTable` indices that correspond to the data element in the `inputTable` at location (t, s). The data element in the `outputTable` is also assigned to the new spatial assignment. If the conflict was in `outputTable` instead, the same procedure is followed, but `GetInputAddressAtOutput` finds the data element location in the `inputTable`, which corresponds to location (t, s) in the `outputTable`.

After a change is propagated between tables, the function `checkConflict` is called. `checkConflict` is similar to `findConflict`, but it only looks at cycle i in the table to which the change is propagated. If a conflict is found, then the data element to be changed must not be on the one that was just changed at location (i, j). This prevents getting stuck in an endless loop. A different element and unused spatial assignment is selected and returned. `ResolveConflict` is then called recursively with the new data element to be reassigned until no conflicts in the path are found.

The process completes once a proper edge coloring is obtained, that is, until no nodes have edges of the same color connected to them. Each color represents the memory element or position in space to or from which the network must move the input sample. Both the input and output maps now contain mapping with no conflicts.

There are many options in switch networks to implement the two mappings. To guarantee full throughput the network must be non-blocking and capable of generating the output configuration for each of the clock cycles. For objective comparison 2×2 switches are again used as the basis for this network. The arbitrarily-sized Waksman (AS-Waksman)



(a) n even.

(b) n odd.

Figure 3.7: AS-Waksman networks are constructed recursively from the outside layer toward the center.

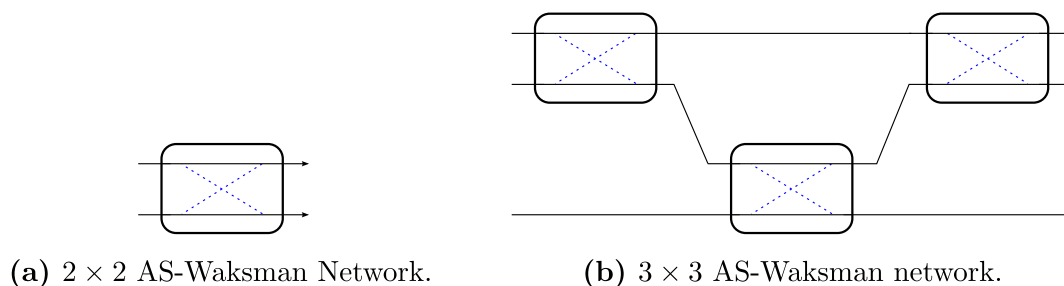
network [16] allows non-power of two inputs and outputs while providing a minimal switching structure that can implement any spatial permutation.

The construction of an AS-Waksman network is shown in Figure 3.7. The construction is from the outside in. A stage of $\lfloor p/2 \rfloor$ 2×2 switches takes in the inputs and allows them to be swapped. If p is odd, the last input does not enter a switch and is connected directly to the last input of the bottom inner network. The outputs of the switches are connected such that the even switch outputs are connected to the top inner network and the odd to the bottom, in the order of the switches. The output side of the inner network is connected to another stage of $\lceil p/2 \rceil - 1$ 2×2 switches. The outputs of inner networks are connected as the inverse of the input side, where the first output of the top inner network goes to the top of the first switch, the second output to the top of the second switch. Then the first output of the bottom inner network to the bottom of the first switch, and so forth.

The inner networks are built in the same manner with the recursion terminating at a size of two or three. The terminating networks design is shown in Figure 3.8. The inner network blocks eventually reach a terminating recursion point when they reach size 2×2 (Figure 3.8a) or 3×3 (Figure 3.8b). In this manner any spatial permutation can be realized.

The lookup table T_I is obtained for each clock cycle t by determining the switch settings that take the input sequence $1, 2, \ldots, p$ and produce the sequence corresponding to column t in `inputTable`. Conversely, lookup table T_O is obtained for each clock cycle t by determining the switch settings that take the input sequence at column t in the `outputTable` and produce the sequence $1, 2, \ldots, p$.

The correct switch settings to implement each permutation are found in a similar way.



(a) 2×2 AS-Waksman Network.

(b) 3×3 AS-Waksman network.

Figure 3.8: The inner AS-Waksman network terminates at either a 2×2 switch or the 3×3 AS-Waksman Network.

The settings of the outer stages of switches are found first, and then the inner. Just like the overall permutation problem, the problem is an edge-coloring problem where one color goes to the top inner network and one color to the bottom. Each input pair forms a node, and each output pair forms a node. If the input data element in a node is found in the output node, then an edge is drawn. The final graph is bipartite with maximum degree two, so only two colors are needed. Starting at the color corresponding to the bottom network, color the edge of the last output and then alternate colors as each node is reached. If the first element of a node starts with color 1, the switch needs to be controlled to swap the inputs.

Consider the mapping:

$$\pi : [1\,2\,3\,4\,5\,6\,7\,8\,9] \mapsto [9\,6\,7\,8\,1\,3\,5\,2\,4].$$

The inputs and outputs are grouped in adjacent pairs and are mapped to nodes as shown in Figure 3.9. Edges are drawn to connect inputs and outputs. The edge of each node is properly colored, beginning with the last output node connected to Color 1.

3.5.2 Temporal Component

The temporal aspect of the permutation requires that the write and read addressing of the RAM consistently access the proper data elements; the RAM access conflicts have already been resolved through the switch networks. The goal is to find the minimum RAM size for implementing the temporal aspect of the permutation. The latency of a system is proportional to the memory in the processing pipeline [80]. The memory depth is minimized by minimizing the latency.

The simplest way to determine the minimum memory of the datapath is to consider the

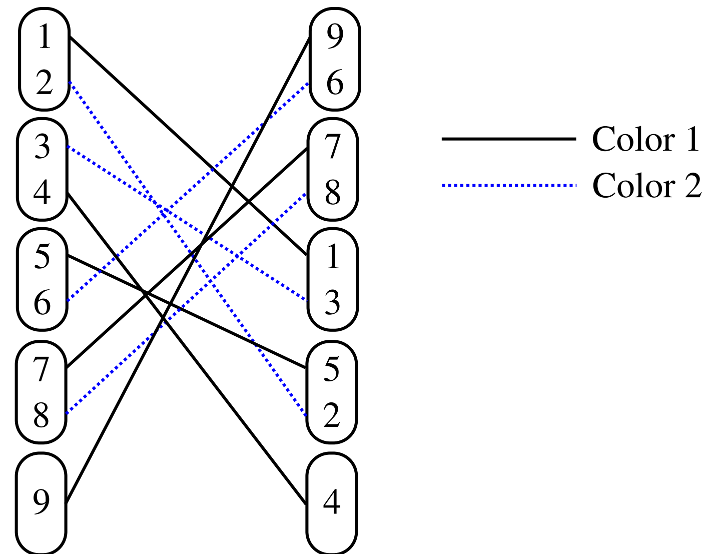


Figure 3.9: Routing the outer switches in a 9×9 AS-Waksman network by edge coloring.

input/output flow of data. Figure 3.10 describes the number of data elements in the system as a function of time. Every clock cycle, p samples enter the system. Once the system starts producing outputs after an initial latency ℓ, p data elements exit each cycle. Graphically, this appears as an increase in data samples in the system with slope p until $t = \ell$, at which point the net change is zero.

The maximum number of data elements in the system occurs at time $t = \ell$ and corresponds to $p\ell$ data elements. Each switch network has a latency of ℓ_N indicating that there are $p\ell_N$ elements in the network at steady state. As there are p spatial paths for data, each with one memory element, each path contains ℓ data elements. Then, the memory size d of one RAM is $d \geq \ell - 2\ell_N$. Minimizing the memory depth requires finding the minimum latency ℓ.

An upper bound on the memory size can be found by considering the permutation output that requires the largest latency. In the worst case, the first output cycle requires a data element that is input on the last cycle of the vector. All the other output cycle constraints will already be satisfied, so that constraint will determine the latency. The first output node can start after $N/p + \ell_M$ cycles; therefore, the required memory size d for any permutation is bounded by $d \leq N/p + \ell_M$.

In highly-pipelined stream processing, the input data rate must be less than or equal to the output rate (assuming no data reductions occur). Assuming input and output data vectors are the same length, if all of the input data enters the system in n consecutive time instances, then all of the output data must leave the system in n consecutive times instances. This fact will be significant in determining minimum latency.

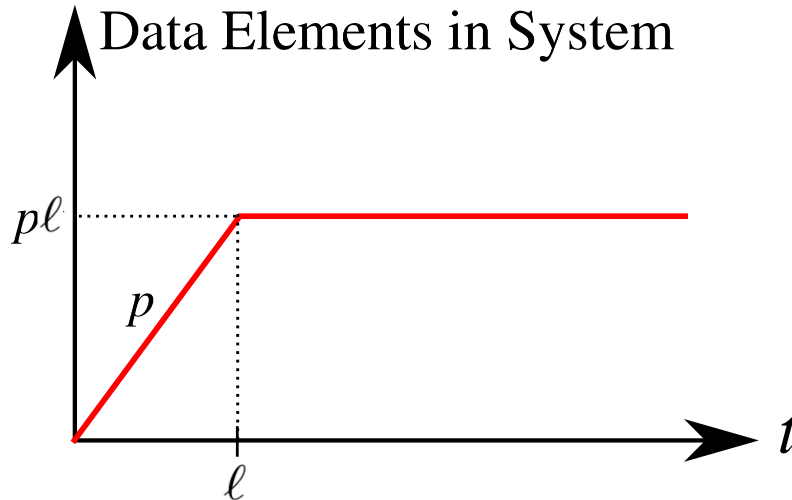


Figure 3.10: The number of data elements in the system increases until reaching a maximum after the system latency has elapsed.

The execution time $\tau(o_i)$ of output vertex o_i can occur once all its adjacent vertices have fired, or in this case once they have been provided to the system. In other words,

$$\tau(o_i) \geq \max_{v \in N(o_i)} (\tau(v)). \tag{3.3}$$

The output data must leave the system in a chunk of n time instances. Therefore, each connected input vertex implies a minimum fire time constraint for each adjacent output vertex. If output nodes are scheduled as soon as possible, then each output node can be scheduled at a time greater than a time satisfying all connected input nodes plus a constant latency corresponding to the delay through the connection networks and RAM, $\ell_M + 2\ell_N$.

The latency of the datapath is then lower bounded by

$$\ell \geq \max_{i \in \{0,\ldots,n/k\}} \left(\max_{v \in N(o_i)} (\tau(v)) - i \right) + \ell_M + 2\ell_N. \tag{3.4}$$

Equation 3.4 finds the vertex that has the strongest constraint on delaying the output sequence and uses that to find the time of the first output cycle.

Determining the minimum latency of the graph can then be found by applying a greedy schedule over the output nodes. In order to fulfill consecutive outputs, any time a gap occurs between outputs all of the nodes that were previously scheduled must be delayed to close the gap. Algorithm 2 details this process.

Algorithm 2 Find Minimum Latency

for $j \leftarrow 1$ **to** n/p **do**
  $\tau(o_j) \leftarrow \max_{v \in N(o_j)} (\tau(v)) + \ell_M + 2\ell_N$
end for
for $j \leftarrow 2$ **to** $n/p - 1$ **do**
  if $\tau(o_j) \leq \tau(o_{j-1})$ **then**
    $\tau(o_j) \leftarrow \tau(o_{j-1}) + 1$
  end if
end for
for $j \leftarrow n/p - 2$ **downto** 0 **do**
  $\tau(o_j) = \tau(o_{j+1}) - 1$
end for

There is a forward and backward pass. The forward pass prevents nodes from being scheduled at the same time slot, and the backward pass starts at the last node and removes discontinuities or gaps in the output stream so that each computation is performed immediately following the last. The result is a set of execution times for the output vertices that satisfies all the constraints.

Lemma 2 proves that the resulting fire times are the minimum latency schedule.

Lemma 2. *The firing times* $\mathbf{T} = \{T_0 = \tau(o_1), ..., T_{n/p} = \tau(o_{n/p-1})\}$ *are the minimum latency fire times.*

Proof. The proof is by contradiction. Suppose there exists a schedule of fire times $\mathbf{B} = \{B_0, \ldots, B_{n/p}\}$ in which $B_0 < T_0$. For this to be the case, then due to the consecutive execution constraint, $B_i < T_i \quad \forall i \in \{0, \ldots n/p - 1\}$, but for at least one T_i the constraint (Equation 3.3) was met with equality causing other vertices to no longer be met with equality. Therefore, $B_i = T_i$ for some i, and since the order is fixed and consecutive execution is required, $\mathbf{B} = \mathbf{T}$. So $\mathbf{T}$ is optimal. □

Minimizing the amount of required memory for the data samples is equivalent to minimizing the latency ℓ. The minimum RAM size is $d = \tau(o_0) - 2\ell_N$.

Algorithm 3 generates the addresses and reuses addresses as data elements are used such that the number of memory locations used in each memory element equals d.

The first step determines the lifetimes of each data element. For data element $t \cdot p + j$, the lifetime Δ is the length of time the data element is in the memory element. The lifetime

Algorithm 3 Generate Address Tables

```
for t ← 1 to n/p do
    for j ← 1 to p do
        Find o_y containing i_{t·p+j}
        Δ(j,t) ← τ(o_y) − τ(i_{t·p+j}) − 2ℓ_N
    end for
end for
for t ← 1 to n/p do
    for j ← 1 to p do
        [Addr, ExpireTime] = UsedAddrHeap[j].top()
        while ExpireTime = t do
            [Addr, ExpireTime] = UsedAddrHeap[j].pop()
            AddrStack[j].push(Addr)
            [Addr, ExpireTime] = UsedAddrHeap[j].top()
        end while
        A_I(j,t) = AddrStack[j].pop()
        [y,z] ← GetOutputAddressAtInput(j,t)
        A_O(y,z) = A_I(j,t)
        UsedAddrHeap[j].push([A_I(j,t), Δ(j,t) + t])
    end for
end for
```

is the time the data element leaves the system minus the time in entered minus the delay through the networks. Parhi [23] uses a similar analysis to determine the minimum number of registers for a register-based approach.

The algorithm maintains p `AddrStack` stacks corresponding to a RAM, each containing the addresses 0 to $d-1$. The input addresses are processed cycle by cycle. `UsedAddrHeap` is checked for any addresses that were used but whose lifetimes have expired. This is a heap structure in which elements at the top of the stack always have the smallest `ExpireTime`. If the expire time has elapsed, then the address is returned to its respective `AddrStack`s for reuse. Then, an address is popped off each stack and written to an element of table A_I. The function `GetOutputAddressAtInput` returns the indices in the output table that map to the data element at (j, t) in the input table. The output table A_O at those indices is then assigned to the same address. This proceeds cycle by cycle until all address tables have been generated.

In Section 3.4.2 it was mentioned that double buffering prevents data from a new dataset from overwriting the previous dataset before it can be used. As the latency of the RAM is taken into account with this scheme, double buffering is not needed. Instead, the read addresses are swapped with the write addresses on alternating datasets. This guarantees that data is not overwritten before it can be used.

The total amount of memory in the datapath will be the size of the memory array M plus the size of configuration lookup tables (A_I, A_O, T_I, T_O). In certain cases, such as the stride permutation, the lookup tables can be described using an online algorithm, greatly simplifying the control cost.

Table 3.1: Summary of cost of the memory components

Name	Type	Needed	Depth	Widths
M	RAM	p	$d \le n/p$	bits per data element
A_I, A_O	ROM	2	n/p	$p\lceil \log_2 d \rceil$
T_I, T_O	ROM	2	n/p	$\sum_{i=1}^{p} \lceil \log_2(i) \rceil$

Table 3.1 summarizes the size and number of the various ROM and RAM elements required by the datapath. In addition to these RAMs, the datapath also requires $\sum_{i=1}^{p} \lceil \log_2(i) \rceil$ two-input switches evenly distributed between the two switch networks N_I and N_O. A particular permutation may reduce the number of switches required, as some may reduce to wires during synthesis.

The throughput of this design yields p data elements per cycle, and a latency $\ell = d + 2\ell_N + \ell_M = d + 4\log_2 p - 2$ cycles.

3.6 Results

When possible the linear permutation implementation provides significantly smaller control cost and minimal switching cost compared to the general implementation. In addition, many common permutations are linear, such as the perfect shuffle or stride. In what cases is the general preferable over the linear implementation?

Table 3.2: Comparison of required RAM and switches

Designs	# RAM elements	Memory Depth	#2-to-2 Switches
Koehn [6]	p	$d \leq n/k$	$\sum_{i=1}^{p} \log_2 i$
Chen [28]	p	n/k	$2p \log_2 k$
Milder [27]	$2p$	$2n/p$	$2p' \log_2 p' - 2p' + 2$
Püschel [24]	p	n/p	$2p \log_2 p$
Chen [26]	k	n/p	$2log_2p$

where $p' = 2^{\lceil \log_2 p \rceil}$.

Table 3.2 summarizes the theoretical results of various designs for realizing fixed permutations on streaming data. Milder [27] can also utilize any fixed arbitrary permutation, but is not very memory efficient. Chen [28] improves upon this by reducing it by a factor of two. Püschel [24] has similar memory requirements, but can only implement permutations that realize

Clearly, if the permutation itself is nonlinear then the linear implementation cannot be used. The alternate scenario is when the streaming width is not a power of two, but permutation itself is still linear. In this case the linear implementation can still be used. This is done by using a set of FIFOs to change the streaming width to a power of two.

Figure 3.11 demonstrates such a conversion. The data stream of width six is written to six of the eight FIFOs with the particular FIFOs circularly shifted each cycle. Once all eight FIFOs contain data then all eight are read in parallel creating the desired stream width for the linear permutation architecture.

The general implementation also requires glue logic; the stream width of six means that the dataset size of $n = 2^m$ is never divisible by six. There is no factor of three in any power of two. In order to use this stream width FIFOs are again used, and the permutation must run at a slightly higher clock rate. The FIFOs introduces invalid data elements that are simply ignored downstream at the end of each data set. Suppose the permutation size $n = 1024$. The general permutation architecture will actual be set for a permutation size of $p\lceil n/p \rceil = 1026$. This means the two invalid data elements will appear at the end of the

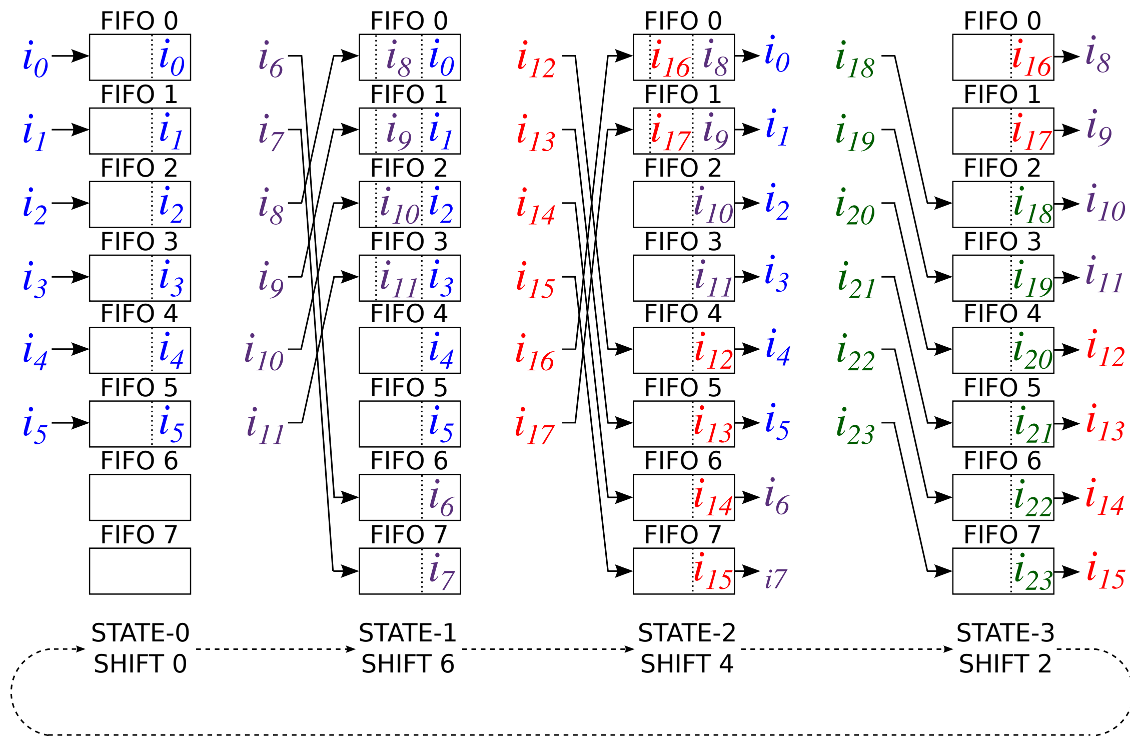


Figure 3.11: Eight FIFOs with inputs applied in a circular shift pattern convert a stream size of size to eight.

dataset. The clock rate of the general permutation must be $1026/1024 = 1.00195$ times faster than the data rate.

As a long datapath may consist of multiple permutations, for comparison purposes only the costs of the permutation implementations themselves will be compared.

3.6.1 Evaluation

Designs using the linear permutation were obtained from [65] which use the method described in [24] and compared with the general permutation implementation [6]. Both designs were implemented on a Virtex-7 FPGA (XC7VX690T) using Xilinx ISE 14.7. All data elements are assigned a width of 16 bits.

Since stride permutations are a common benchmark among streaming datapaths, this standard permutation is used for comparison purposes.

Figure 3.12 compares BRAM usage of the linear implementation to the general. This cost

is based on the number of 18K BRAMs, if a 36K BRAM was used it was converted to two 18K BRAMs for comparison purposes. For smaller permutation sizes, for instance $n = 2048$ shown in Figure 3.12a, streaming widths other than ten require more BRAM than the linear with $p = 16$. The cost in BRAM does not warrant using the general implementation. At the larger permutation sizes, $n = 32768$ in Figure 3.12b, the BRAM usage for the general is less the linear even in direct comparison for most streaming widths of eight and above. The general implementation was designed to reduce the RAM sizes so this is not completely unexpected. At the smaller permutation sizes, the fixed size of the BRAMs result in large unused areas of the BRAM while at larger sizes the savings add up to full BRAMs that are not needed. This effect is more strongly shown in Figure 3.12c.

Figure 3.13 shows the same analysis performed on the number of occupied slices. The slices contain the aggregate of both the distributed memory for control of the switch networks and addressing, as well as the switch themselves. At the smaller permutation size, $n = 2048$ in Figure 3.13a, less slices are used by the linear implementation for all streaming widths. As n increases to 32768 in Figure 3.13b, the linear implementation still uses less slices, but the trend suggests that as the streaming width increases beyond sixteen the linear's slice costs will increase beyond the general's. Figure 3.13c suggests that the permutation size has little significance on choosing one implementation over another. Except for a hump around $n = 512, 1024$ the linear is less expensive. Depending on the exact permutation size and streaming widths, the data suggests that for small permutation sizes the best choice in silicon area is the linear implementation. If BRAM is the primary cost constraint the general implementation starts becoming advantageous after streaming widths of eight. If slices and BRAM are weighed equally, the cost in slices shifts the design choice in favor of linear implementation for streaming widths of sixteen and below.

Figure 3.14 conducts the same comparison for the latency of the designs. With the exception of permutation sizes of $n \leq 128$, the general implementation is able to achieve lower latency. While the general implementation is designed to minimize the latency through the RAM, the switch network complexity is higher than the minimum connectivity switch network developed by [24]. As the permutation size increases the savings in the general implementation's RAM datapath latency outweighs the cost in latency through the switch network.

3.7 Conclusion

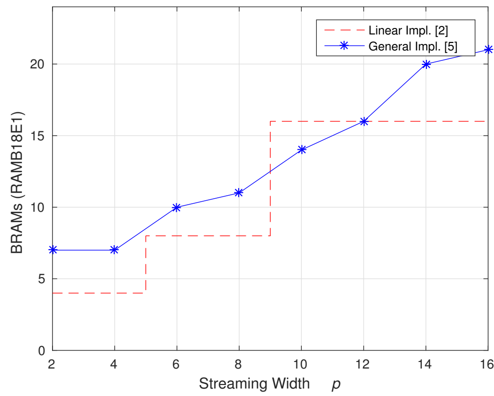
Linear permutation implementations have been demonstrated to achieve optimal conditions in some cases, and near optimal otherwise, resulting in cost efficient architectures. The constraints on these permutations require that the permutation be linear, that is the bit representation of data indices are generated from linear operations, and that the streaming width be a power of two. The first constraint is rarely a problem, but power of two stream

width becoming increasingly sparse as stream size increases, making general implementations increasingly attractive.

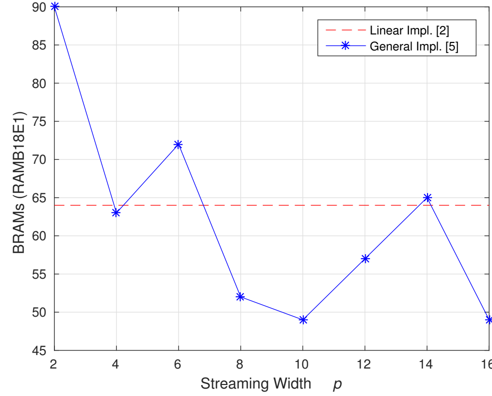
In this chapter, the designs of the latest and most area efficient linear and general implementation were reviewed. Two slightly different techniques for the linear optimize either the number of switches or jointly optimize the number of switches and control cost. The general approach instead approaches the problem as an edge-colored graph and minimizes the latency of the datapath through the RAM. Since the two types of implementation are approached in two different ways, an open research question is whether the general techniques can be applied to the linear or vice-versa.

Both implementations were applied to the perfect shuffle (stride) permutation with glue logic added to allow stream widths to be brought to the next power of two for the linear implementation, Glue logic also extended the permutation size to the next size that is divisible by the stream width for the general implementation. As stream width increased beyond eight the general implementation was less expensive in terms of RAM, but trends in the area cost in slices showed that the general implementation is not overall more cost effective until stream widths above 16.

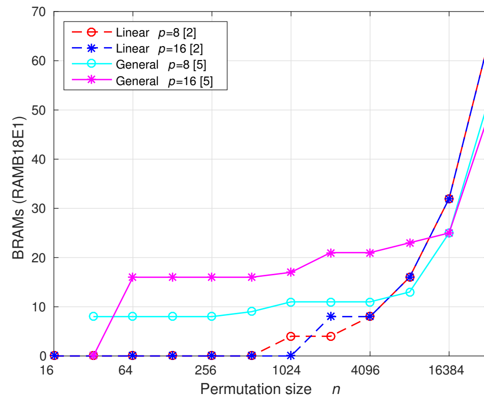
Both the linear and general permutation methods utilized a combination of multistage switch networks and dual-port RAMs. These components are common among major FPGA vendors, including Altera and Xilinx. In addition, Chen [26] demonstrated a technique to replace the dual-port RAM with two single-port RAMs, further generalizing these techniques to any customizable digital logic hardware. It is not applicable to fixed architectures such as GPUs or multicore CPUs, although aspects of the design could prove useful as preprocessing for these devices.



(a) Permutation size $n = 2048$.

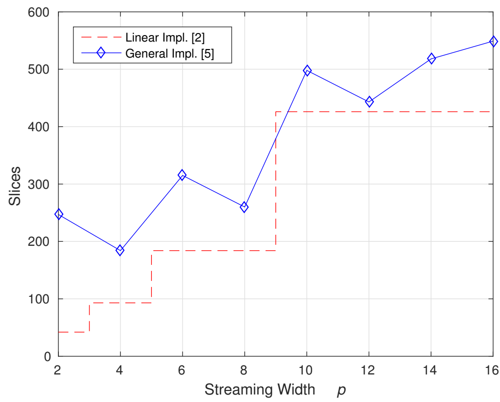


(b) Permutation size $n = 32768$.

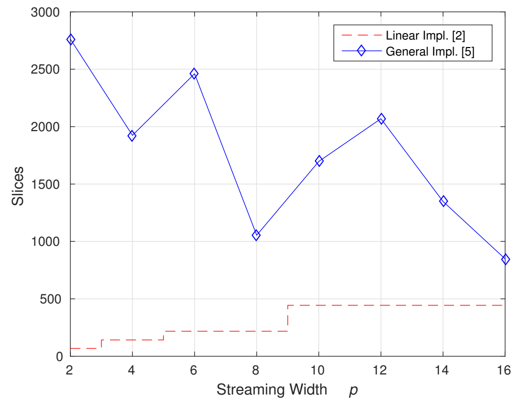


(c) Varying permutation sizes.

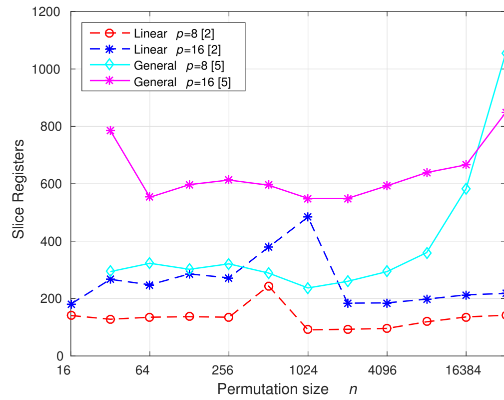
Figure 3.12: BRAM consumption as function of streaming width p and permutation size n compared for linear and general implementations.



(a) Permutation size $n = 2048$.

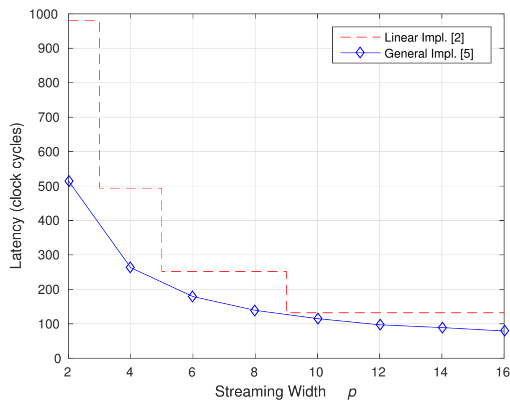


(b) Permutation size $n = 32768$.

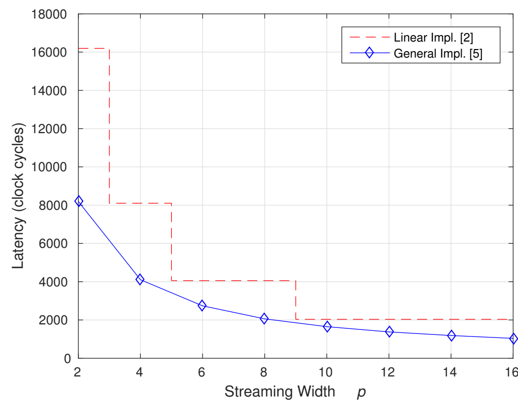


(c) Varying permutation sizes.

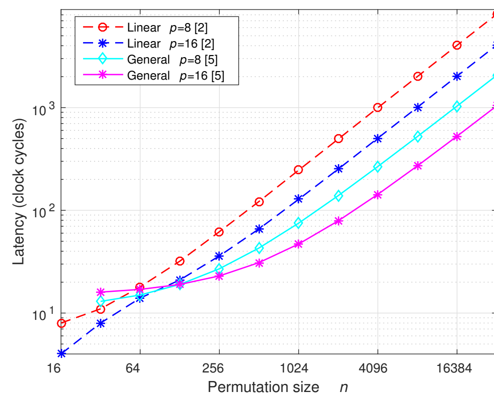
Figure 3.13: Area in slices as function of streaming width p and permutation size n compared for linear and general implementations.



(a) Permutation size $n = 2048$.



(b) Permutation size $n = 32768$.



(c) Varying permutation sizes.

Figure 3.14: Latency in clock cycles as function of streaming width p and permutation size n compared for linear and general implementations.

Chapter 4

Data Reuse

Chapter 3 considered permutations or reordering of data elements. More complicated schemes include dropping some data elements all-together, as well as replicating some data elements. The input-to-output data rate may change and depending on the extent of replication full-throughput may no longer be feasible. This chapter extends the general permutation architecture to handle these cases. The problem is framed as that of a structured matrix.

Many applications use matrices that have some structure to the data, such as known zeros located at particular positions in one or both matrices. Multiplication involving these matrices have multiple partial products terms that are known to be zero, and so do not require any computation. Hardware architectures have considered the general matrix multiply [43, 81, 82] or multiplier for sparse matrix representations [48, 83], but not for the more modest number of zeros in the matrices.

Structured matrices are often found in machine learning and decision making problems, where the state space is often described as a Markov process. Markov processes consist of a set of states with some probability of transitioning to a different state; these probabilities are encapsulated in a state transition matrix. Since many states are only reachable from certain other states, many of the entries are zero. In fact, any such system involving a fixed topology involving an incidence matrix fits into this category. Rather than use a general hardware matrix multiplication, the generator in this chapter can take advantage of these zeros to reduce the resource cost or increase the throughput when implementing these multiplications.

Many endeavors involved in efficient matrix multiplication implementation consider input fully available to the architecture at the start of the operation [81, 84]; however, for high-throughput applications with real time constraints, stream processing is often required. The system receives a stream of data as input, with only a small subset of elements received each

clock cycle. Stream processing operates on the data in an assembly-line manner. Operations are performed on a small number of samples in stages, and the process is repeated on the next set of samples at the next cycle. The Xilinx Linear Algebra Toolkit [85] is a toolkit for creating solution for streaming applications.

This chapter is organized as follows: Section 4.1 provides background and related work in the areas of matrix-matrix multiplication and stream processing. This is followed by a discussion in Section 4.2 of the process of scheduling the operations onto a streaming architecture. In Section 4.3 the hardware architecture is described and the process of mapping the matrix multiplication schedule to the control of the architecture. Section 4.4 demonstrates the improvements made through building implementations to take advantage of structured matrices. Finally, Section 4.5 provides concluding remarks.

4.1 Background

This section introduces matrix multiplication and some common examples of structured matrices with sparsity. It proceeds by structuring the individual binary operations such that they can be partition into stages for stream processing.

Matrix Multiplication. In a standard matrix multiplication $\mathbf{T} = \mathbf{R} \times \mathbf{S}$, where $\mathbf{R}$ is an $M \times L$ matrix, and $\mathbf{S}$ is an $L \times N$ matrix. The ij^{th} element of matrix $\mathbf{T}$ is

$$t_{ij} = \sum_{k=1}^{L} r_{ik} s_{kj}.$$

Each element of $\mathbf{T}$ requires L multiplications to compute the partial products, followed by an L element sum. As with the sizes of the matrices, the resource cost becomes quite high, as the number of computations is $\mathcal{O}(MLN)$.

Hardware designs of matrix-matrix multiplication can be grouped into two main categories: dense matrices, and sparse matrices.

Systolic arrays are a common method of performing the matrix multiplication. The array consists of processing elements such as MACCs (Multiply and Accumulate) which process one partial product each and accumulate the partial products for each output row. Amira [43] introduced a scalable matrix multiplication using this systolic architecture. Numerous improvements have been made to the same basic structure, including Zhuo and Prassana [44] who made improvements in area and efficiency.

The majority of research on sparsity has focused on matrix-vector multiplications [45, 46]. Systolic arrays of processing elements form the basic architecture. Akella et. al. [47]

designed a sparse matrix-vector multiplication for a single FPGA. Lin et. al. [48] addressed sparse matrix-matrix multiplication by optimizing control for handling sparsity. Each row of operations was partitioned among a fixed number of processing elements.

For illustrative purposes, consider a problem involving a hidden Markov model in the computation stream. The probability state transition matrix used in Markov chains is an example of a structured matrix. A Markov chain consists of a number of states (only a finite number are considered here) where the probability of transitioning to another state depends only on the current state. A state i has a probability of transitioning to state j of p_{ij}. Many states do not directly connect to another state, so the probability of transition is zero.

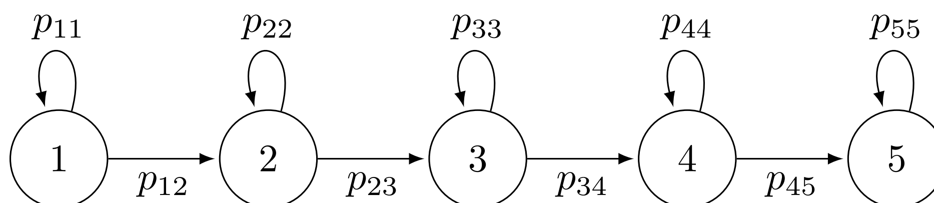


Figure 4.1: A simple Markov chain where each state connects only to one other state.

In Figure 4.1 each state i has only two possibilities it can either stay in the current state with probability p_{ii} or transition to the next state with probability $p_{i,i+1}$. Once the last state is reached there is zero probability of leaving that state. The probability transition matrix (or topology matrix) $\mathbf{P}$ for this chain looks like this:

$$\mathbf{P} = \begin{bmatrix} pb_{11} & pb_{12} & 0 & 0 & 0 \\ 0 & pb_{22} & pb_{23} & 0 & 0 \\ 0 & 0 & pb_{33} & pb_{34} & 0 \\ 0 & 0 & 0 & pb_{44} & pb_{45} \\ 0 & 0 & 0 & 0 & pb_{55} \end{bmatrix}.$$

This matrix is a band matrix that has zero elements outside a diagonally bordered band. In an $N \times N$ band matrix $A = [a_i j]$, an element $a_{ij} = 0$ if $j < i - k_1$ or $j > i + k_2$. For this example $k_1 = 0$, $k_2 = 1$.

Another common Markov chain is shown in Figure 4.2. Each state can transition to any larger state, but it can never go backwards. The state transition matrix looks like this:

$$\mathbf{P} = \begin{bmatrix} pb_{11} & pb_{12} & p_{13} & pb_{14} & pb_{15} \\ 0 & p_{22} & pb_{23} & pb_{24} & pb_{25} \\ 0 & 0 & pb_{33} & pb_{34} & pb35 \\ 0 & 0 & 0 & pb_{44} & pb_{45} \\ 0 & 0 & 0 & 0 & pb_{55} \end{bmatrix}.$$

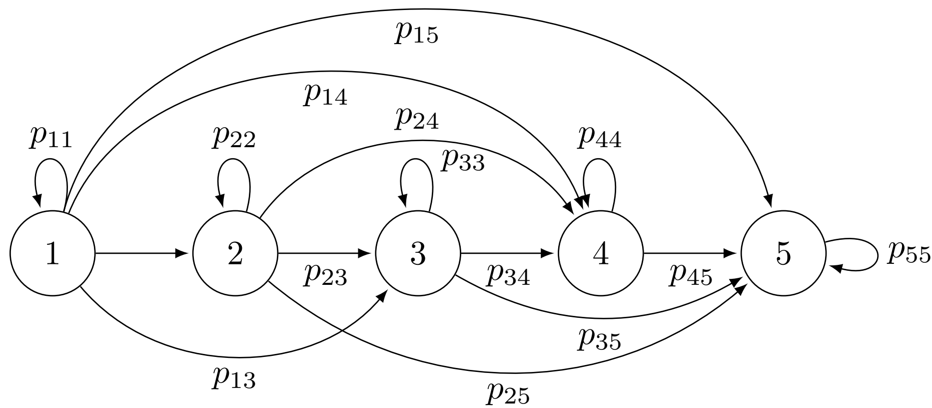


Figure 4.2: A Markov chain where states can only transition to all states that are larger than it.

P is an upper triangular matrix.

Streaming datapaths. The streaming datapath in this chapter has as input two matrices; the first is an $N \times L$ matrix which has been partitioned into k_1 data samples per cycle over $\lceil NL/k_1 \rceil$ consecutive cycles and an $L \times M$ matrix partitioned into k_2 data samples per cycle. The inputs can be combined to describe a general case for an input vector of $NL+LM$ data elements that are partitioned into $k = k_1 + k_2$ data elements each cycle. In Figure 4.3 one element from each matrix is provided each cycle, $k_1 = k_2 = 1$.

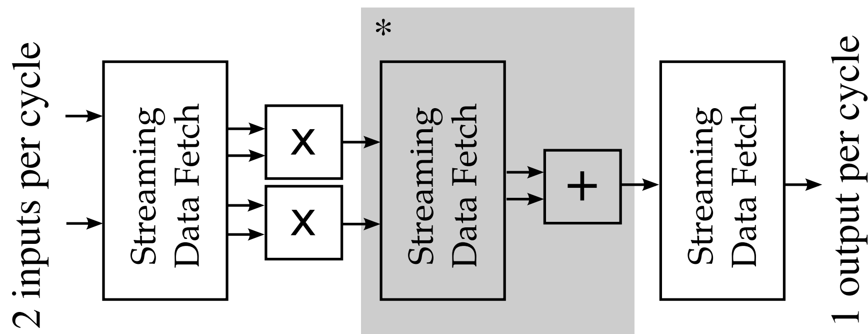


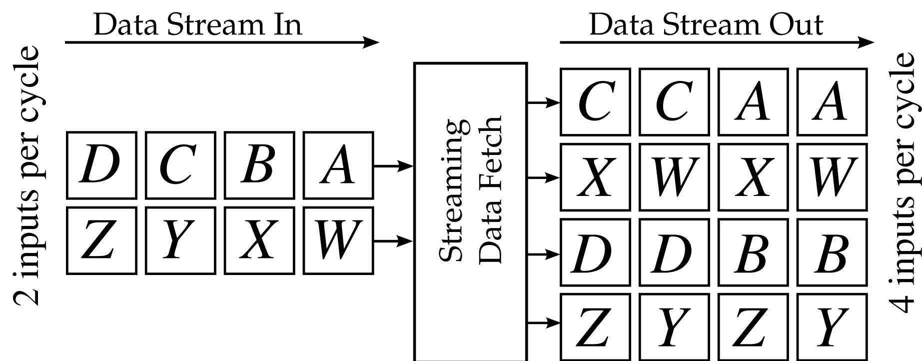
Figure 4.3: Architecture for streaming pipeline.

Figure 4.3 illustrates the streaming architecture with resources constrained to two multipliers and followed by one adders for a matrix multiple with $N = L = M = 2$. Each clock cycle, two data elements are presented; they are replicated and permuted by the data

fetch unit to provide the appropriate inputs for the multipliers. This basic stage repeats, but with a different data sequence from the data fetch unit and different number and/or type of resource. The partial products are summed and then go through a final fetch unit to arrange the output in row-major order. For this size matrix, only one stage of adds is needed, but larger matrices will require multiple stages that can be generated by duplicating the highlighted stage with different resources allocated to each stage.

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} W & X \\ Y & Z \end{bmatrix} = \begin{bmatrix} AW + BY & AX + BZ \\ CW + DY & CX + DZ \end{bmatrix}$$

(a) Two by two matrix multiply



(b) Example streaming data rearrangement for partial products.

Figure 4.4: Example streaming data rearrangement for partial products.

Figure 4.4b illustrates the first stage of the streaming architecture for the matrix multiplication for Figure 4.4a. Each cycle, one data element from each matrix enters the system in row-major order. The data fetch unit produces four data elements each clock after some initial latency and organizes the data elements such that each multiplier has the data needed to compute a partial product. For example, the output of the matrix multiplication at location (1,1) is $AW + BY$. The second partial product BY requires that B and Y be adjacent to each other in the same clock cycle. Referring to Figure 4.4b, in the first output cycle the third and fourth outputs are B and Y, which would be multiplied together by the second multiplier in Figure 4.3. If data element Y were known to always be zero, then BY and DY would not be computed, and all the partial products could be computed in three cycles with two multipliers.

The *initiation interval* (II) is the number of clock cycles between when the first element of the first matrix enters the datapath and when the first element of the next set of matrices enter. For the two-by-two matrices in the example, $II = 4$. This is full throughput; the number of multipliers is sized to process all of the data elements that are received each clock

cycle. Full throughput is obtained when the initiation interval equals the time required for the full input data set or matrix to be received. As the size of the matrices grows, $\mathcal{O}(NLM)$ multiplications are required, making it extremely costly to design for full throughput. If the sizes $N = M = L = 4$ were applied to the same architecture, the initiation interval would increase to 32, because 64 partial products must be computed requiring 32 clock cycles with 2 pipelined multipliers.

The sparsity α is the ratio of zero elements to total number of elements in the matrix. If the number of processing elements, or in this case multipliers, is constrained to $k/2$ (each multiplier can process two inputs), then the initiation interval is lower bounded by the number of multiplications that must take place divided by the number of multipliers available.

$$\text{Initiation Interval } \geq \frac{2(1-\alpha)^2 N^3}{k},$$

where two $N \times N$ matrices of the same structure are to be multiplied and k processing elements, or multipliers, are run in parallel. The total processing delay is the time elapsed from when the first element of the input set is received to when the full result is available. This is equal to the initial interval plus the delay or length of the processing pipeline and the length of the output sequence.

4.2 Scheduling for Sparsity

For sparse matrices, many partial products are zero and do not need to be computed. This breaks the regular structure of general purpose or dense matrix multiplication. In this section, the binary operations of the matrix multiplication are scheduled with the goal of reducing latency.

To structure this matrix multiplication, each binary operation in the matrix multiplication is mapped to a node in the graph. If an operation will yield a constant zero, then it can be dropped. Nodes are grouped by pipeline stage according to their inputs. All of the partial products have inputs that are the inputs to the system, so their nodes will be mapped to one of the multiplier resources at a particular clock cycle. Those partial products are then summed. The sum is composed as a tree. In the first stage of adds, each takes two partial products. If there are an odd number, a zero is added to the list of potential inputs to this stage. During the next state, each adder sums two of the outputs of the previous stage, and the process continues until an output element has been computed.

Table 4.1 shows the nodes from the example in Figure 4.4b and their corresponding relative clock cycles within the stage. The input has two inputs each cycle, so two nodes are generated. The first stage has two multipliers available, $p = 2$, so two nodes (consisting of

Table 4.1: Simple schedule for 2×2 matrix multiplication

Cycle	Input	Stage 1 Multiply	Stage 2 Sum
1	A, W	AW, BY	AW+BY
2	B, X	AX, BZ	AX+BZ
3	C, Y	CW, DY	CW+DY
4	D, Z	CX, DZ	CX+DZ

the partial products) are created per cycle. The second stage has only a single adder and so has only one node per cycle.

Scheduler. Figure 4.4b does not consider sparsity and so has a straightforward schedule. If A were known to be zero, then all zero nodes could be removed and the results shifted to decrease the number of processing cycles.

The proposed scheduler finds a resource-constrained schedule for all operations with the goal of minimizing latency. The latency is related to the amount of memory required in the datapath. A minimum latency schedule will provide data sooner and also require a smaller memory footprint.

For full throughput to be achieved in stream processing, the input data throughput to the system must be equal to the throughput leaving the system times a factor related to the data expansion or reduction of the system. For a matrix multiply, this factor is the number of output elements in $\mathbf{C}$ divided by the number of data elements in $\mathbf{A}$ and $\mathbf{B}$. All of the input data enters the system in II clock cycles. Then after some latency, the output must also exit the system within II clocks; otherwise the amount of data within the system is constantly increasing. As time goes to infinity, the required amount of memory in the system also goes to infinity.

Consider each stage of operations independently. Let each output node of the stage be referenced by o_i, the ith output node in set O. The execution time $\tau(o_i)$ of output vertex o_i can occur once all its adjacent vertices have fired, or in this case been input to a stage,

$$\tau(o_i) \geq \max_{v \in N(o_i)} (\tau(v)). \tag{4.1}$$

The output data must leave the stage in II time instances; therefore, each connected input vertex creates a minimum fire time constraint for each adjacent output vertex. If output nodes are scheduled as soon as possible, then each output node can be scheduled at a time greater than a time satisfying all connected input nodes plus the delay through data fetch unit ℓ_F.

Algorithm 4 Schedule for Minimum Latency

```
function SCHEDULE FOR MINIMUM LATENCY(V(:,:))
    for all o ∈ O do
        τ(o) ← max_{v∈N(o)} τ(v) + ℓ_F
    end for
    [value, idx] ← ascendSort(O, τ)
    resourcesUsed ← 1
    for j ← 2 to  length(O) do
        if τ(o_{idx(j)}) ≤ τ(o_{idx(j−1)}) then
            if resourcesUsed < maxResources then
                τ(o_{idx(j)}) ← τ(o_{idx(j−1)})
                resourcesUsed++
            else
                resourcesUsed = 1
            end if
        end if
    end for
    resourcesUsed = 1
    for j ←length(0) − 1 downto 1 do
        if resourcesUsed < maxResources then
            τ(o_{idx(j)}) = τ(o_{idx(j+1)})
        else
            τ(o_{idx(j)}) = τ(o_{idx(j+1)}) − 1
        end if
    end for
end function
```

Algorithm 4 first sets the array of scheduled execution times τ to the earliest possible time that satisfies Equation 4.1. The function `ascendSort` returns the values and indices of the outputs arranged by τ in ascending order. The next loop prevents more operations than there are resources from being scheduled in the same time slot, and the final loop starts at the last node and removes discontinuities or gaps in the output stream so that each computation is performed consecutively. The result is a set of execution times for the output vertices that satisfies all the constraints.

This process repeats from the initial stage until, but not including, the final stage. In the final stage, the order of the nodes is constrained to maintain a particular output order, so the ascending sort is skipped. But the execution times for those nodes are still propagated to the final stage from the previous stage.

4.3 Hardware Implementation

This section introduces the hardware architecture for the data fetch unit. This is the heart of the overall architecture. The multipliers and adders are just discrete components, but the data fetch unit provides the data routing with the correct timing to each component.

Several streaming architectures have been proposed for arbitrary permutations [6,27,28] which take n data elements and reorder or permute the elements, while still providing n output elements. In a matrix multiplication, elements of the matrix will be reused multiple times based on the size and structure of the matrix. For instance, if $\mathbf{R}$ is of size $N \times L$ and $\mathbf{S}$ is of size $L \times M$ and each have all non-zero elements, then each element of $\mathbf{R}$ will be used M times, and each element of $\mathbf{S}$ will be used N times. Even though this is not a permutation, the theory of operation and developed structures still applies.

Figure 4.5 shows the parametrized datapath architecture for the data fetch unit. Its structure is similar to the permutation architecture in the previous chapter. It consists of a memory array M with independent input and output addressing (i.e. dual port) for each element. Two connection networks N_I and N_O on both the input and output ports of the memory array, and lookup tables (ROMs) contain the address and control data (A_I, A_O, T_I, T_O).

The data fetch unit requires several modifications to the permutation architecture. The choice for the sizing of m and the memory depth d will depend on the structure of the matrices as well as the input and output streaming widths p and w.

The memory array M contains m parallel memories, each with capacity d. Each has one read port and one write port. These memories can be implemented as block RAM, or distributed memory (registers). Each memory has latency ℓ_M, which is the time it takes data at the write port to be output at the read port when both addresses are the same.

The primary difference between the data fetch and permutation architectures is in the

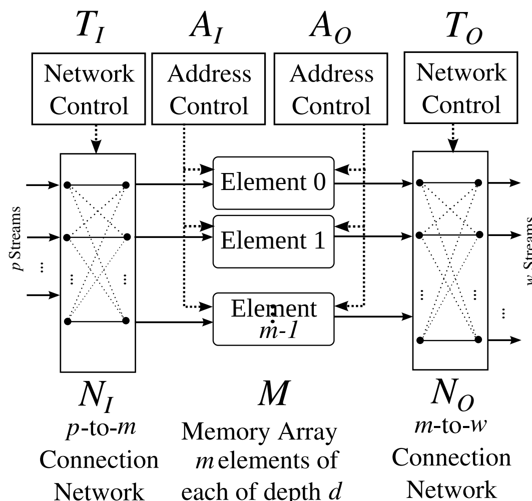


Figure 4.5: The permutation architecture is customized to perform the replication and reordering required of the data fetch unit.

switch networks and their control. The connection network N_I is capable of taking p points as input and outputting them in any order at the m output ports. As will be shown later, $m = \max(p, w)$, and the network is constructed as an m-to-m network with $m - p$ input ports left disconnected. The network is built as explained in [15,16,86], utilizing $\sum_{i=1}^{m} \lceil \log_2 i \rceil$ 2-by-2 switches and has a latency $\ell_N = 2\log_2 m - 1$, as registers are placed after each stage of switches. As one data element may be used multiple times, each 2-by-2 switch is controlled by two bits rather than one. The second bit allows both outputs of the switch to take the first input value or second input value based on the first bit. This requires double the control cost in general, but in practice this replication is used infrequently for many of the switches, so during synthesis unused logic will be removed. The switch network N_0 is built in the same manner.

The data takes the following path through the datapath. First, p data elements are input to N_I. The network takes the p elements and permutes them onto the m outputs. The configuration of the network is controlled by the stored value in T_I, which is precomputed for n/p cycles. Each configuration requires $2\sum_{i=1}^{m} \lceil \log_2 i \rceil$ bits. The m samples are then written into the memory modules at addresses defined by table A_I, with each line consisting of m addresses of $\lceil \log_2 d \rceil$ bits. After a delay of ℓ_{data} clock cycles, the m memories are read according to each line A_O, the p output addresses. A_O is the same size as A_I. The data then passes through N_O which reorders and duplicates the m elements according to the table T_O, which is also the same size as T_I.

In practice, only one of the switch networks needs to duplicate data elements. If $p \geq w$ then only the N_0 needs to duplicate elements, and the control for N_I can be reduced by one half. If the control bits are constant, synthesis optimizations will automatically perform this

reduction. If $p \leq w$, both networks may need to retain the full control with data duplication.

4.3.1 Mapping the Schedule to Hardware

The scheduler requires that the data fetch unit be sized appropriately. The minimum number of memory elements or modules to implement a computation within an algorithm is determined by the need for conflict-free memory access. One way to eliminate a conflict is to add an additional memory element and move one of the conflicting elements to it.

The minimum number of memory elements can be determined using another graph. From Algorithm 4, the execution times of all nodes have been determined. A new graph is generated by merging all nodes of the same stage with the same execution time, and edges are preserved. Figure 4.6 shows an example translation.

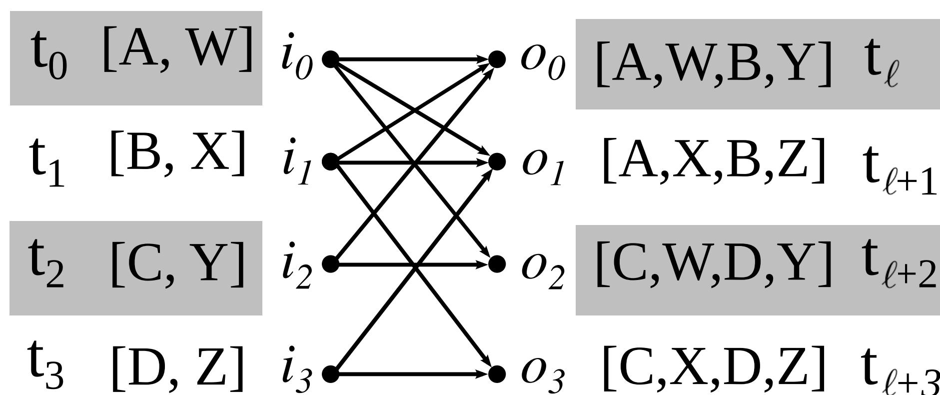


Figure 4.6: Map of input/output data vectors to a graphical representation.

Consider a subgraph composed of just the input and output nodes of each stage. Each subgraph will be bipartite as the nodes can be divided into different classes. In this case, input and output nodes are the two classes. Each edge represents a different memory access by the multiplication or addition resource. If there are multiple edges, each edge must correspond to a different memory element to avoid a conflict. A conflict occurs when two different addresses are accessed from the same memory element. In graph theory, the edges can be colored or labeled, with each color corresponding to a memory element. The number of different colors will determine the number of memory elements required.

Lemma 3. *The minimum number of memory elements needed is equal to the maximum number of edges connected to any node in the graph.*

Proof. Since the graph is bipartite, König's Line Coloring Theorem states that the minimum

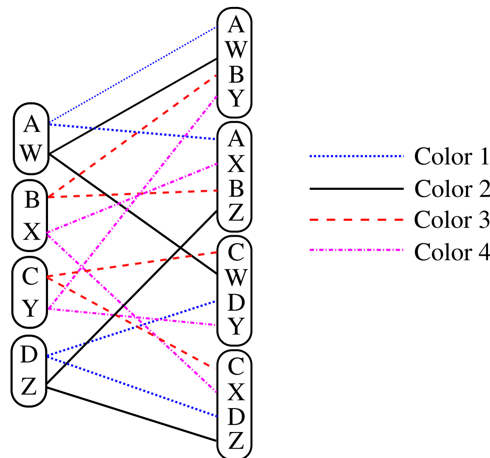


Figure 4.7: Assigning memory banks to each data element.

number of colors needed to color a bipartite graph is equal to the maximum degree of the graph [70, 71]. The degree of a node is the number of edges connected to it. Since each memory bank can be mapped to a particular edge color, the minimum number of memory element will be the maximum degree of the graph. □

This determines the number of memory elements $m = \max(p, w)$ needed in each stage.

Spatial Mapping. As memory element mapping configurations are equivalent to edge coloring, mapping data to a memory element is equivalent to properly edge coloring a graph, that is coloring the edges such that no color is used twice at the same node. There are a number of techniques for properly edge coloring a graph, with varying run times [76–79].

In this case, there is a slight variation on proper edge coloring. Consider Figure 3.4. The maximum degree of the graph is four, so four memory banks are needed. If both elements from a node go to multiple memory banks, then a conflict will occur. However, this does not mean that they cannot go to multiple output node with the same edge color. It is a requirement that they do if the element is used more than once. To handle this the 'proper' constraint is relaxed in edge coloring. Instead the constraint is that edges are drawn to the subnodes, i.e. the data element and those edges must all be the same color.

In Figure 4.7 the edges are colored to observe this constraint. Note that each internal data element within a node has only one edge color connected to it. Edge coloring assigns a particular memory bank to each data element.

Obtaining the lookup table values for the switch networks is done by considering each cycle independently. At time t_1 B and X are assigned to Banks 3 and 4. The switch settings must be configured for N_I to take the input on ports 1 and 2 and connect them to output

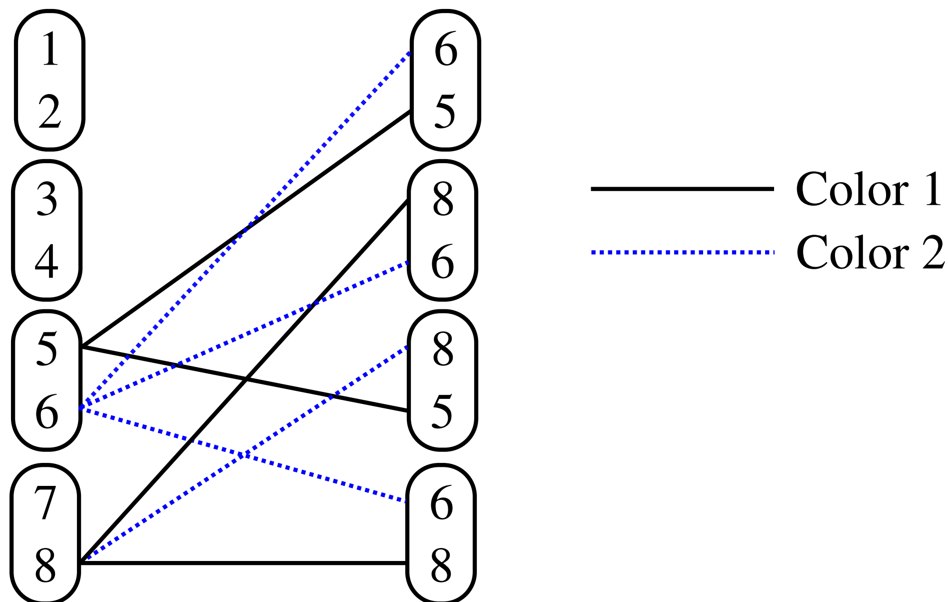


Figure 4.8: Routing the outer switches in a modified 8×8 AS-Benes network by edge coloring.

ports 3 and 4. The approach is similar to that taken for AS-Benes networks [14]. The switch settings are found recursively by determining whether to connect an input to the top inner network or the bottom. The inputs and outputs are grouped in adjacent pairs and are mapped to nodes as shown in Figure 4.8 as in the case for standard AS-Waksman networks. Edges are drawn to connect inputs and outputs. In this case, however the nodes are not properly edged colored. The last output element of the last node (8) is mapped to Color 1. If that node contains only one element i.e. (8,8), then the switch is set to replicate the either the first or second element. In the case of only one element of an input node being used. The edges can be colored in either color, but must guarantee that the connected output node maintains proper edge coloring, that is only one color per element. The switch settings then coincide to the order of the colors. A switch swap is indicated by Color 2 connected to the 2nd element.

Temporal Mapping. In this section the input data elements to a stage are mapped at each cycle to a particular location in the memory element. This step populates the address tables for each memory element and determines the size of each memory element.

This approach seeks to minimize the amount of memory used by fully utilizing the memory space. After input data has been accessed by all its dependents its lifetime has expired, and the slot in the memory element can be reused by incoming data. Algorithm 3 more fully describes the mapping procedure.

First, a lifetime analysis of each data element is performed. Its lifetime is how long it is

Algorithm 5 Find Minimum Latency

for $t \leftarrow 1$ **to** $\lceil n/p \rceil$ **do**
  for $j \leftarrow 1$ **to** p **do**
    Find o_y, containing $i_{t \cdot p+j}$
    $\Delta(j,t) \leftarrow \tau(o_y) - \tau(i_{t \cdot p+j}) - 2\ell_N$
    AddrHeap[j].push($t-1$)
  end for
end for
for $t \leftarrow 1$ **to** n/k **do**
  for $j \leftarrow 1$ **to** k **do**
    [Addr, ExpireTime] = UsedAddrHeap[j].top()
    while ExpireTime = t **do**
      [Addr, ExpireTime] = UsedAddrHeap[j].pop()
      AddrHeap[j].push(Addr)
      [Addr, ExpireTime] = UsedAddrHeap[j].top()
    end while
    $A_I(j,t)$ = AddrHeapStack[j].pop()
    $[y,z] \leftarrow$ GetOutputAddressAtInput(j,t)
    $A_O(y,z) = A_I(j,t)$
    UsedAddrHeap[i].push($[A_I(j,t), \Delta(j,t)+t]$)
  end for
end for

in the memory, from when it is written to memory to when it is last read. This is assigned to an array Δ for each data element. The addresses in A_I and A_O only care about the lifetime in the memory array, which is just the full lifetime minus the latency of both switch networks $2\ell_N$. Once an address is used, the address will expire and be available for reuse at the current time plus its lifetime.

The algorithm maintains an array of k heaps, `AddrHeap` corresponding to a memory element. Each heap contains the addresses 0 to n/k. A heap is a data structure in which the data are always kept in sequence. In this case the data in the heaps are arranged in ascending order such that the lowest address in the heap is always at the top. Each memory element's heap has been initialized with the addresses one to n, guaranteeing enough memory for all the inputs.

The input addresses are processed cycle by cycle. `UsedAddrHeap` is checked for any addresses that were used but whose lifetimes have expired. In a heap, the elements at the top of the stack always have the smallest `ExpireTime`. If the expire time has elapsed, then the address is returned to its respective `AddrHeap` for reuse. Then, an address is popped off each stack and written to an element of the table A_I. The function `GetOutputAddressAtInput` returns the indices in the output table that map to the data element at (j, t) in the input table. The output table A_O at those indices is then assigned to the same address. This proceeds cycle by cycle until all address tables have been generated.

4.4 Results

This approach on a Virtex 7 VX690T was compared with two different techniques. The first is a commercial tool from Xilinx for streaming dense matrix-matrix multiplication. The Linear Algebra Toolkit [85] (LAT) generates streaming matrix multiplication with the number of multipliers parametrized to the size of the rows of the first matrix.

For simplicity, both input matrices **A** and **B** are square matrices of the same size, $N = L = M$. The Xilinx Linear Algebra Toolkit (LAT) automatically increases the number of DSP multipliers used as the matrix sizes increases, so the number of multipliers is equal to N^2 for all the designs. The **A** input is a full matrix with all non-zero elements. Figure 4.9 compares the throughput, the average number of samples per clock cycle out, for a variety of matrix sizes. The full matrix-matrix multiplication tool matches the throughput of the Xilinx tool identically for all matrix sizes. Two structured matrix types from the original examples, the banded matrix with $(k_1 = 0, k_2 = 1)$ and the upper triangular matrix, shows increased throughput of up to two orders of magnitude. The banded matrix shows a constant throughput, unlike all the other cases had enough resources that the input sample rate had to be increased to attain that throughput. In particular, for the full matrix multiply with $N = 16, k = 16, k_1 = k_2 = 8$, on the Virtex 7 VX690T, the LAT required 154 BRAMs and

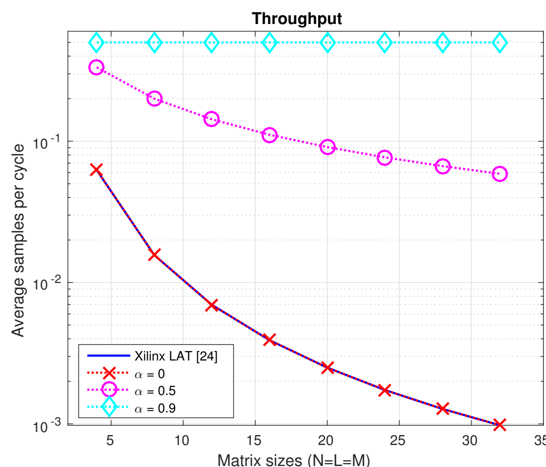


Figure 4.9: Using structured matrix implementations allows significant improvements in throughput while using the same number of DSP multipliers.

15440 slices at 272 MHz, as compared to 22 BRAMs and 1922 slices at a clock rate of 264 MHz for this approach. When the sparsity increased with the banded matrix, this approach reduces resources to 10 BRAMS and 903 slices at 279 Mhz.

The Xilinx LAT tool automatically increases the amount of multiplier resources with the matrix sizes. So in Figure 4.9 the number of multipliers is equal to N. In many cases, the number of multiplier resources available cannot scale. Figure 4.10 shows the throughput when the number of multipliers is constrained to eight. Xilinx LAT does not allow this flexibility so results for it are not available. As expected with constrained resources, as matrix size increases the throughput decreases, as there are more partial products per multiplier output.

The second technique developed by Lin et. al. [48] targets sparse matrices with no defined structure. Figure 4.11 compares the processing delay of this system to the theoretical lower bound as well as the results obtained by [48]. The processing delay was measured in simulation for a range of sparsity on matrices of size 256×256. The results from this paper were averaged over 100 simulated trials to obtain the average processing delay. The stream width is $k = 32, k_1 = k_2 = 16$, and 16 multipliers resources was used. This streaming implementation performed very close to the lower bound for processing delay. It does not achieve the bound because of the delay introduced by the data fetch unit as well as the processing delay through the pipeline. However, these delays are on the order of N^2 clock cycles as compared to the lower bound which is on the order of N^3 clock cycles. For high sparsity (90%), the gap does increase, but still significantly outperforms [48].

For these matrix sizes, modern FPGAs like the Virtex 7 VX690T did not contain enough on-chip memory to store the necessary partial products in BRAMs. For larger matrices, memory requirements continue to grow, requiring off chip memory. This was not formally

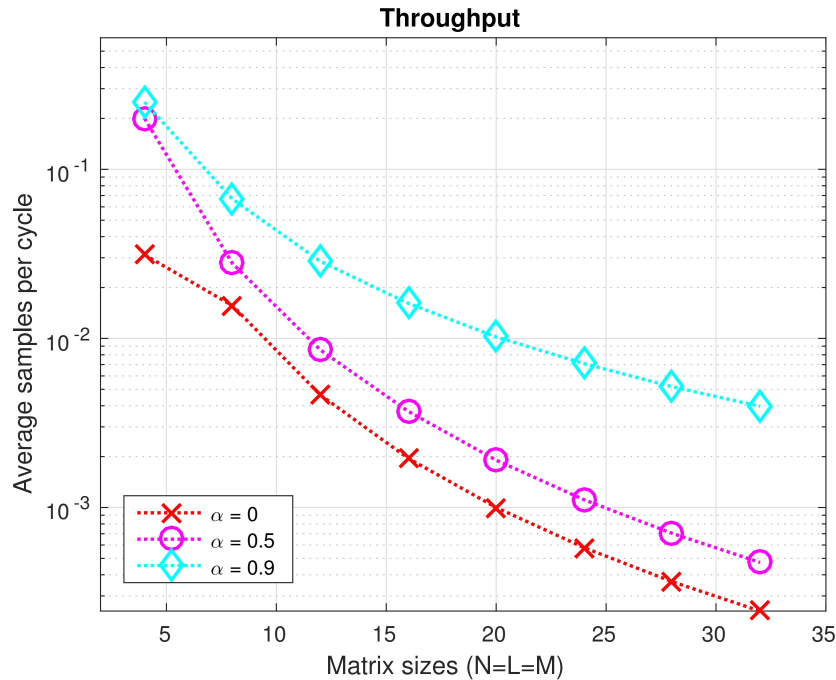


Figure 4.10: By fixing the number of DSPs for all cases, smaller form factor matrix multiplication implementations are possible with higher throughput the sparser the matrix.

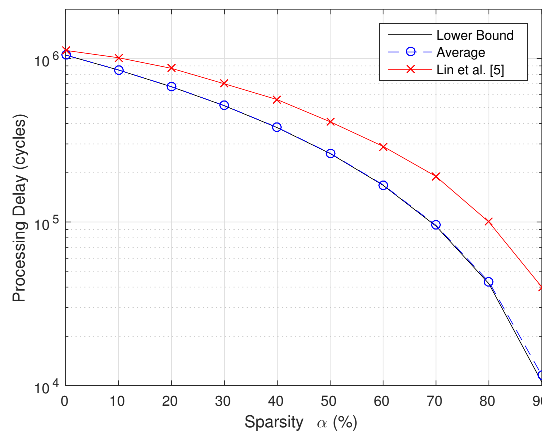


Figure 4.11: The processing delay is related to the sparsity of the matrices being multiplied. The higher the sparsity the fewer computations required.

addressed in this design, but the latency of the off-chip memory needs to be considered in the address allocations. This increases the potential amount of memory required, but does not fundamentally alter the design. Furthermore, the number of parallel streams is constrained

to the bandwidth of the off-chip memory. Memory such as the Hybrid memory cube, is capable of 320 GB/s more than enough for 32 samples per clock, with on the order of 75ns in latency [87].

4.5 Conclusion

Matrix multiplication requires that input data be both replicated and permuted. Using structured or sparse matrices there is more freedom in the data transformation and is therefore better able to showcase the data fetch unit.

The data fetch unit uses the same basic architecture and algorithms developed for permutations, but extend the switch networks and sizes of the structures to accommodate changes in stream width and data replication. This allows the final architecture to both reorder data and to either replicate or drop data elements as needed.

The presented generation tool is able to partition structured matrix multiplication into binary operations and apply scheduling techniques to assign the exact dataflow in time and space. The data elements are mapped to the hardware by the control signals of the data fetch units.

The final results show that the realized streaming structured matrix multiplication enables higher throughputs for the same number of constrained resource. It also tracks with the commercial techniques when implementing dense (all non-zero element) multiplications. In comparison with research for sparse matrices it also improves upon the processing delay and achieves close to the lower bound.

Like the permutation architecture, the extensions of this Chapter are also applicable to customized digital logic hardware, including FPGAs and ASICs. None of the extensions require any hardware structures beyond those discussed in Chapter 3.

Chapter 5

Permutation Chaining

The previous chapters consider the single use case of a data permutation or sequence transformation, however in most cases this transformation is not isolated. For instance, if a set of computation requires a permutation for stream processing often a permutation is required after the computation to return the output sequence to a new sequence order. If the datapath consists of a long datapath this might be another permutation or chain of permutation. If the data is output to another system, that system expects the data in a particular order, generally sequential.

This raises the question of whether it possible develop the permutations in such a way that the chains of permutation can

1. share control logic, or
2. be partitioned such that the overall set of permutation requires less area.

The primary motivator for this work are linear transforms like the fast Fourier Transform (FFT). An FFT of size N can be computed using $\log_2 N$ stages. Each stage permutes the data and performs a set of computations (a butterfly for radix-2 algorithms). When the data is received by the next set of computations, not only is it no longer in the sequential order described by algorithm descriptions, but it is also not in the correct order for the next computation.

This chapter describes a method to use a single permutation architecture between computations and reuse control of the previous permutation block to create simple reusable control structures. It also breaks the dependence of the permutation on the permutations from earlier in the chain. The resulting streaming architecture has significantly lower latency and area cost than commercial tools from Xilinx as well as other FFT streaming architectures.

This chapter is organized as follows: Section 5.1 describes the streaming architecture components for building a complex algorithm. The case of a single computation is considered in Section 5.2. This is extended to a chain of multiple components in Section 5.3. The results of this method are compared with available streaming FFT architectures in Section 5.4 and in Section 5.5 other methods that are not quite one-to-one comparisons are detailed. Section 5.6 concludes.

5.1 Architecture

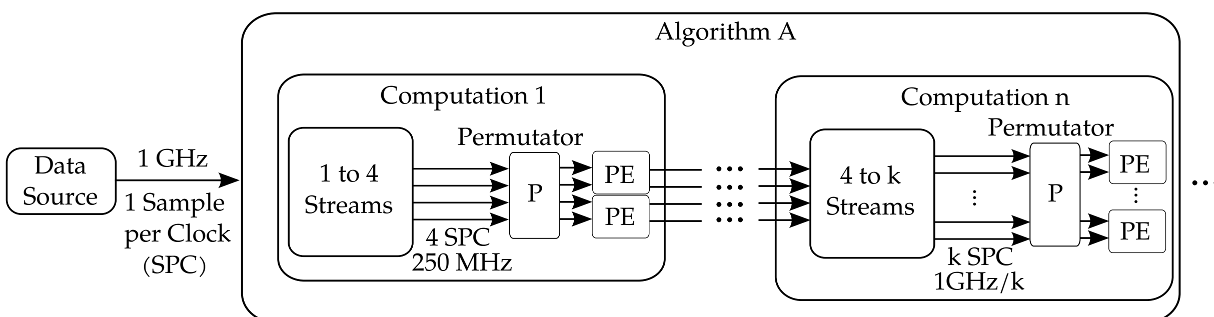


Figure 5.1: Example datapath for implementing an algorithm in a streaming architecture.

A typical streaming architecture to implement a complex algorithm consists of the following basic structure (Figure 5.1). Data is received via a high-speed serial connection. In many cases, high-speed analog to digital converters (ADCs) sample analog data and using a limited number of wire connection transmit data serially at speeds several times larger than the digital logic's clock frequency [88]. The digital logic uses deserializers that demultiplex the serial signal into a vector of samples at the lower digital logic clock rate. The algorithm is partitioned into multiple computation steps each with an optional serializer/deserializer to change the streaming width and clock rate between computation blocks. This is followed by an optional permutation block and finally a set of processing elements (PEs) that perform the actual computation.

The translation in streaming widths is not specifically considered in this chapter, but Chapter 3 described how a series of parallel FIFOs could be utilized to change the streaming width from a non-power of two width to a power-of-two. The same concept applies to any stream width conversion. This step is optional and if there are no data reductions in the datapath (consistent with linear transforms), once the initial streaming width is established it is not necessary to change the width. An open question is whether changing the stream width can result in lower area cost or other performance metric.

After the optional stream width translation, a permutation or series of permutation may be necessary to place samples in the order required by the computation. That is the operands must be located appropriately for the computation. If the computation requires a multiplication, then using a Xilinx DSP slice, the two operands must be received by the DSP slice in the same clock period.

The final component are the processing elements. These consist of small kernels which are able to process the vector of p elements without further permutation. All operations within the kernel use only wire connections. Typical operations include additions, subtractions, multiplication as well as logical operations.

5.2 Single Computation Allocation

In cases where an algorithm requires a single data allocation to provide the correct sample sequence for an entire algorithm, input data is typically provided in ascending sample order. This is also applicable for the first computation of any algorithm. If p streams are input every clock, then samples 0 to $p-1$ are available at the first time instance, followed by p to $2p-1$ at the second time instance. At a time instance t, samples tp to $(t+1)p-1$ are available. This sample ordering is common for any kind of standard communication process. The procedure here is applicable to the first step of algorithm.

A common example of a series of computations separated by permutations is the fast Fourier Transform (FFT). The description of the FFT reference samples in every stage of its iteration by samples indexed by their sequential order. The FFT algorithm is a useful example in that it is well known and computations are done on samples which are not adjacent in time index. While a Finite Response Filter (FIR) is a useful algorithm, because the computations do not require that the sample sequence be modified, the data allocation architecture reduces to connecting the input to the output. The FFT provides a more interesting test case.

Consider the first stage of computations in the radix-2 decimation-in-frequency FFT algorithm. The below pseudo-code illustrates the required data orderings:

The output sequence from the PE remains in the same order as the data sequence is received. The PE is strictly responsible for performing some arithmetic function and outputting the result. If the output sequence requires a different ordering, then later logic is responsible for implementing that logic. This topic is revisited in the next section.

A first attempt at mapping this computation to the data allocation unit might be to write the input data samples directly to the memory bank corresponding to their position in the input vector. Then starting at time 0 let the write address equal the to the time that the input data is received. For simplicity let $p = 2$. Figure 5.2 shows this mapping. The

Algorithm 6 FFT Stage N

1: **procedure** $FFTStageN(x, N)$
2: **for** $i \leftarrow 0, N/2 - 1$ **do**
3: $x_1(2i) \leftarrow x(i) + x(i + N/2)$
4: $x_1(2i + 1) \leftarrow x(i) + W(i, N)x(i + N/2)$
5: $y_1(i) \leftarrow x_1(2i)$
6: $y_1(i + N/2) \leftarrow x_1(2i + 1)$
7: **end for**
8: **return** y_1
9: **end procedure**

blue solid line indicates a Bank 1 access, while the red dotted line indicates a Bank 2 access. While this is an easy mapping it fails to attain the conflict-free property. The output at each time index requires samples from the same memory bank.

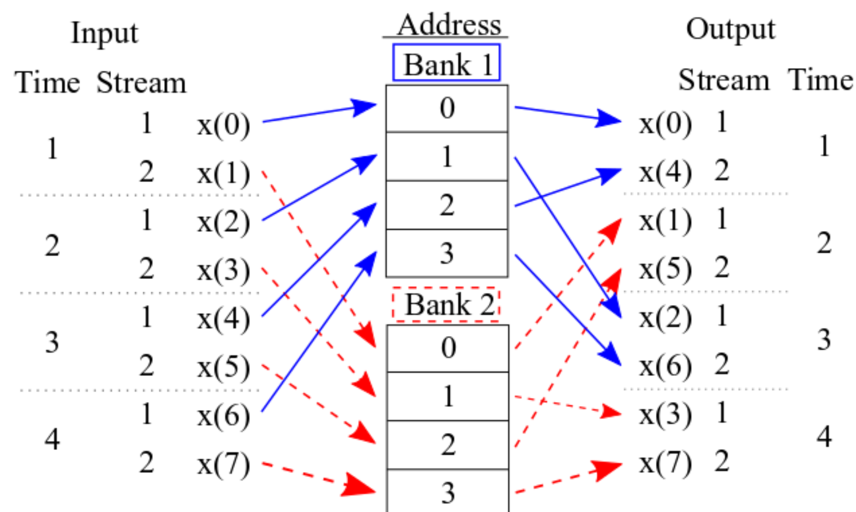


Figure 5.2: Algorithm 6 mapping with conflicts.

A useful property of the first computation case is that desired samples' indices are a simple function of the time index, which means that the input addresses can remain simple and the output addresses can handle the function of the time index. The conflict can be resolved using the permutation unit. At the first conflict at time 1, Bank 1 is accessed for both streams. To resolve the conflict, the second stream must instead access Bank 2. Following the arrow backward for output Stream 2 at Time 1, this corresponds to input Stream 1 at Time 3. If that sample is written into Bank 2 instead of Bank 1, then the output at Time 1 can be read from Banks 1 and 2 without conflict. Then Stream 1 at input Time 3 must be written to Bank 2 and Stream 2 to Bank 1. Update all the connections to

reflect this change.

This process can be repeated at each remaining conflict to obtain a conflict-free mapping. Figure 5.3 shows the mapping after all conflicts have been resolved. At each time instance each bank has only one sample read. In the case of the FFT, each data sample is only used once, so only two memory banks are needed to obtain conflict-free memory access. If the conflicts cannot be resolved through permutations, an additional memory bank can be added.

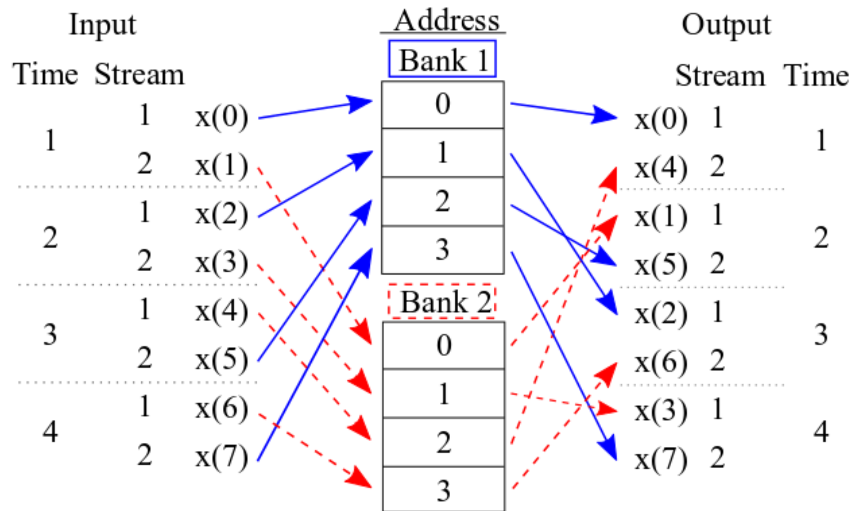


Figure 5.3: Algorithm 6 mapping with conflicts resolved.

The next step is to translate the algorithm in pseudo-code directly into the data allocation architecture without the intermediate steps. Since the sample sequence is the important part of the code as far as the data allocator is concerned, the sample indices provide the relevant information. First, the variable i iterates from 0 to $N/2-1$; this can be described using a counter. Since p samples are being processed each clock, the hardware counter effectively counts by p rather than 1. Let the counter output $c = i/p$ This means the width of the counter requires $\lceil \log_2((N/2)/p) \rceil$ bits. The counter c resets after $N/2/p-1$ counts. In this case the number of samples per clock evenly divides the number of clocks. Things become slightly more complicated when this is not the case. As discussed in previous chapters, the length of the permutation may be extended. Alternatively, this approach can still be used through the judicious use of asymmetric FIFOs to convert the number of samples per clock to a different clock rate where this property holds.

As mentioned previously, for input data samples that are in time index order the write address for each bank is simply the counter value. The read address generation then depends on the specific indices called out by the algorithm. Suppose the algorithm is broken into binary operations. Each PE receives two data streams. Even-numbered streams consist of

the data samples that are the first operand in the computation. Odd-numbered streams consist of data samples that are the second operand. The streams are further divided by the number of PEs. In this case the first operands are values of the counter c, while the second operand $i + N/2$.

While for simplicity only one PE was used, suppose two PEs were used instead, then the first output stream consists of the first operand samples decimated by the number of PEs, samples with indices by $i/(p/2)$. For the second output stream, which consists of the second operand samples to be processed by PE one, the samples contain indices $(i + N/2)/(p/2)$. Putting these in terms of k will result in necessary read addresses. For the even streams, stream $2d$, starting at zero, the read address is $\lfloor (k + d)/(p/2) \rfloor$. The odd streams, $2d + 1$ stream has addresses $\lfloor (k + d + N/2)/(p/2) \rfloor$. For a general algorithm, the first operand indices are a function $o_1(i)$, and the second operand indices a function $o_2(i)$. Translating that function into k and d results in the appropriate read addresses.

The next step is setting up the permutations such that memory accesses are conflict-free. In the previous step, the samples were divided into first and second operand streams for each PE. To avoid conflicts, each streams' samples must be available in different memory banks. Clearly, the first and second operand samples must be in different banks. A loop that accesses an array with a constant offset between each element used has a constant stride. In algorithms with constant stride the second operand will always occur some constant s samples after the first operand. This means that all samples after that s (if s is even) will need to swapped from between the pairs of even and odd streams. If s is odd, then the second operands will naturally fall in differing banks. In Algorithm 6, $s = N/2$, and is even, requiring permutation. The permutation for an algorithm with constant stride is the stream index plus the number of streams dividing by two modules the total number of streams. Now output samples at the same output time instance are all in different memory banks.

The final step is the permutation of the output data. While the data is all available in different banks, the order of the output vector does not necessarily result in a sample in the vector going to the proper stream and PE. The output permutation is needed to correctly order the vector of samples at a particular time instance. Because of the way the input permutation was constructed, for constant stride, every other output time instance a permutation equivalent to the input permutation type will occur. Since the first p samples for $p < s$ are read each time instance and all go the first operand, every other time instance the permutation needs to move the sample back to the first operand stream. This puts the first operand data samples at the output of the first $p/2$ output streams and the second operand data on the last $p/2$ output streams. For $p > 2$ the streams must then be wired to the appropriate PE. This results in the output of permutation matching the desired sequence of samples for each data stream.

So far the case of input data ordered in sequential time, presented in blocks of p samples have been mapped to the memory architecture. This is the first step for any algorithm. In addition, for algorithms with a constant stride, the control units for the memory addresses

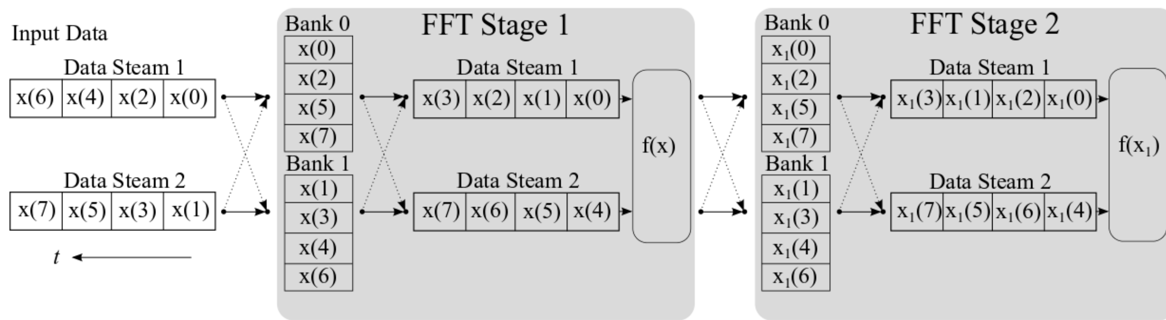


Figure 5.4: Chained computation using new indexing.

and permutation units are generated algorithmically allowing a software tool to procedurally generate and customize the architecture for a particular algorithm.

5.3 Chained Computations

The procedure for mapping the first stage of an algorithm with input in sequential order is relatively straightforward, but the next stage of computations requires a completely different data order. The second stage of the FFT essentially executes the same function (Algorithm 6), but with y_1 and $N/2$ as the input parameters. The function is executed twice, once on the first $N/2$ samples of y_1 and then on the last $N/2$ samples. Unfortunately, after the computation is executed in hardware the output sequence is x_1, not y_1. This section discusses two existing methods for re-sequencing and proposes a novel third approach that combines the benefits of the existing methods without their drawbacks.

One method is to use a data allocation unit to reverse the sequence used in the computation and return to index ascending order. This maps the computation output x_1 to y_1. The reverse structure of the allocation unit from the previous stage of computations can be used. The second stage of computations uses y_1, but not in time sequence order; a second data allocation unit must be used to put y_1 in the necessary order. Then the method used in Section 5.2 can be repeated for this stage of computations.

A second technique is to change the algorithm references from sequential order to the output order of the first computation. Rather than use y_1 as an input to the algorithm, determine which indices of y_1 are used and then find the indices of x_1 that correspond to those samples. Depending on the particular orders this can become quite complex. Suppose this method is performed on the second stage of the FFT of size 8. Figure 5.4 shows an example of the processing chain for the first two stages of the FFT. The output of the data allocation unit described in Section 5.2 is computed, and the necessary sequence of x_1 for

the stage two computation is determined. The sequence x_1 is in sequential order so the same procedure is performed.

The drawback to this approach is the complexity of determining the necessary sequence in terms of x_1. For the eight samples used in this example finding the needed logic sequence in terms of x_1 is not particularly difficult, and it is still possible to use fairly simple logic to generate the addresses based on counter values. This becomes significantly more difficult when the number of samples increases and multiple stages of computation have been performed. Each stage of computation adds a layer of index mapping, as output order of the computation does not match the order specified by the algorithm. Computers are capable of following the logic, but for many samples the address and permutation control soon becomes unwieldy. Complicated control schemes cannot be generated at run time. Instead, they require RAM modules to store the control logic for each time instance.

The third method is like the first in terms of simple address and permutation control, and like the second in that it only uses a single data allocation unit. The key to reducing the complexity of the data allocation between successive computations is to use both ports on the RAM banks. The input side of the memory can be used to allocate the previous output samples into a time-sequential order. The read address generation then works like the single computation case. The main difference comes into play with the input bank mapping. Using the output bank mapping from the previous computation may result in the unavailability of needed samples at a particular time at different banks. This is solved by first performing the previous output bank mapping generation, and then applying the bank mapping of the next allocation (ignoring the previous computation) to the result.

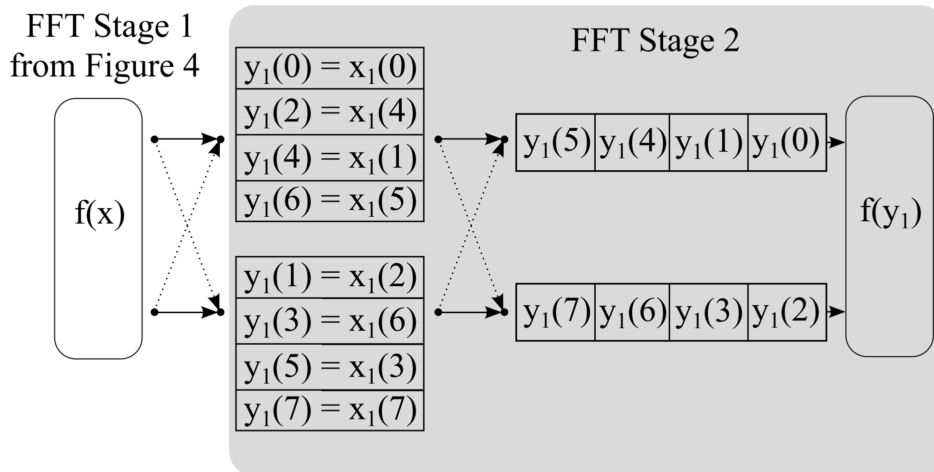


Figure 5.5: Reduced complexity chained computation.

Figure 5.5 shows the computation chain for this method after leaving the computation of the first stage of the FFT as in Figure 5.4, $f(x)$ refers to actual butterfly computation, an addition, and a multiple by a constant and accumulate operation. Using the previous

output address and permutation almost results in the desired sequence y_1 stored in memory sequentially, just as x was stored in the first stage of the FFT. The addition of a permutation results in the correct memory bank assignments.

The control units to generate this mapping are fairly simple to derive, and because the previous computation's sample order is separated from the needed sample order of the next computation, both the read and write addresses of the memory banks are independent. The output address map from the first data allocation unit is used as the input address map of the current element. Repeating the standard process, for the dth even stream, starting at zero, the read address will be $\lfloor (k + d)/(D/2) \rfloor$. The dth odd stream will have addresses $\lfloor (k+d+N/4)/(D/2) \rfloor$. The input bank map also carries over the previous memory partition, but it will be further modified based on the required output. The input bank map is the remainder of the counter k divided by the difference of the indices of x. This yields $k \bmod N/4$. Appending this to the output bank mapping from Algorithm 6 results in $(k \bmod N/4) + (k \bmod N/2)$. The output bank mapping is based on the remainder of the counter k divided by the number of memory banks D.

With the FFT algorithm, the output will always contain the same number of samples as the input, so the computation takes two operands and computes two different outputs. In many cases the computation will take two operands and have a single output, effectively reducing the data rate by two. If enough computations do this then the data rate will eventually drop below the clock rate of the computation, and other methods that don't require data to be processed every clock can be employed.

Reduced complexity chaining can still be employed when the number of input samples does not equal the number of output samples. In such cases an additional mapping is needed. If the algorithm re-bases the indices of the result based on the computation order, then no extra steps are required because the output is already in the order specified by the algorithm. In cases where this is not the case, the previous computation's read address and permutations may still be partially utilized. This may be done by removing the particular output address and permutation controls from the previous computation. If every second operand is consumed in the computation, then all odd samples no longer exist and the shifting the counter and addresses right by one will effectively remove all connections for the odd input samples. The result will then have to be checked to verify that the data is mapped to the appropriate memory banks. Depending on the regularity of the puncturing, the permutation may no longer work appropriately.

In summary, the procedure for mapping a general algorithm into a series of data allocation units followed by an array of processing elements is as follows. First, convert the algorithm into stages of binary operations. For each computation determine the required input sequence. If the computation requires samples that are not in the input order, create a data allocation unit. Use the read addresses and permutation control logic from the last data allocation unit as the write controls for the new data allocation unit. Then generate the addresses and permutations for this stage of the computation assuming its input samples are

in sequential order. Remove controls for consumed input samples as needed and verify that the data is written into the memory modules in a sequential order relative to the algorithm specification. If not modify the permutation appropriately. Apply the read side permutation matrix on top of the permutation matrix used from the previous computation. Repeat this process until every stage of the computation has been generated. If the last computation stage outputs need to be in sequential order, add a final data allocation unit using the same procedure. The details of the address and permutation generation were described in detail for constant stride algorithm, but the using reduced complexity computation chaining is independent of the algorithm data access patterns. This general algorithm can be applied to any arbitrary series of computations, not just the FFT.

5.4 Results

To demonstrate the efficiency of the chained computation approach, this approach was tested on an algorithm against two other cases. The second case was generated with components from vendor specific tools. The third case was generated with two allocation units between computations; while this is not optimal, it is relatively easy to build based on current literature [17]. Comparing the footprints and resource costs among these three cases will demonstrate the efficiency of the presented reduced complexity chained computation approach.

The algorithm that was tested was a 4096-pt. complex FFT of radix 2. The design was implemented on a Xilinx Virtex 7 VX690T FPGA. Block scaling was used with convergent rounding and a 3-multiplier structure for each complex multiply. The input and output are 16-bits, with 18-bits used for coefficients.

While Xilinx does not provide an equivalent core [89] to handle sample rates greater than the clock rate, it is possible to generate designs by generating multiple FFT cores and using multiple first in, first out buffers (FIFOs) to alternate which FFT is provided data. This design allows the core to be used, but it is resource inefficient. Resources are not shared between the FFTs, and the FIFOs are extra but necessary to work around the constraints imposed by the core generator.

For a relevant comparison with approaches based on current literature, after each stage of computation in the FFT a data allocation is used to return the output samples to sequential order. Then a second data allocation unit is used to change the sequential order into the order required by that stage of computation.

Table 5.1 shows a comparison between the chained reduced complexity computation allocation presented in this chapter and the designs based on the Xilinx core generator and per-computation allocation for a two input data stream, i.e. the sample rate is twice the clock rate. The chained allocation technique provides a greater than 50% reduction in BRAM and

Table 5.1: Comparison of 4096 point FFT with $p = 2$

	BRAM (36kbit)	Slice	DSP	Latency (clocks)
Xilinx Coregen	62	2942	40	12442
2 Per-Computation	30	2703	29	6319
Chained Reduced	28	1984	29	5264
% Reduction				
vs. Coregen	54.8%	32.6%	27.5%	57.7%
vs. 2 Per-Comp.	6.7%	26.6%	0%	16.7%

Table 5.2: Comparison of 4096 point FFT with $p = 8$

	BRAM (36kbit)	Slice	DSP	Latency (clocks)
Xilinx Coregen	280	12902	160	12449
2 Per Computation	65	8294	108	1699
Chained Reduced	46	6717	108	1424
% Reduction				
vs. Coregen	83.6%	47.9%	32.5%	88.6%
vs. 2 Per-Comp.	29.2%	19.0%	0%	16.2%

latency, with a 32% reduction in logic slices and 27% in DSP slices. Compared to the two data allocation units per-computation, the chained allocation posts more modest reduction of around 7% and 17% in BRAM and latency, with a 27% reduction in logic slices.

In the case of the eight input data stream FFT, the sample rate is eight times the clock rate, as shown in Table 5.2. The cost in BRAM is even more significant. Each data stream gets a Xilinx core with a buffer large enough to hold eight times the FFT size. As a result, the reduced complexity chained allocation case shows a reduction of over 80% in BRAM and latency. It also compares more favorably against the two units per-computation method showing 30% reduction in BRAM and 16% in latency. This illustrates the importance of efficiency as the number of samples per clock increases. While the effect on resources was noticeable between the per-computation and chained computation allocation at $p = 2$, when $p = 8$ the effect became much more pronounced.

This comparison gives an idea of the costs of using a fast development tool that is currently available. The core generator would allow a designer to generate a design in a fairly short amount of time, but would still require some development and debugging. A good designer would be capable of developing a comparable design to the optimal memory

usage presented in this article, but the cost is in development time rather than resources. It also shows how utilizing previous allocations can reduce memory resources and latency.

5.5 Related Work

There are a number of related works attempting to optimize FFT in a variety of techniques. As this work is concerned primarily with resolving data dependencies of computations in an efficient mapping of algorithm to hardware, rather than optimizing the computations themselves direct quantitative comparison is not possible, however, qualitative comparison can be made in terms of aspects of the structure. For example, an increasing the radix of the FFT will result in fewer computation stages, and less multipliers.

A radix-8 parallel implementation of a 4096-pt. FFT running at 8R, 8 samples per clock, was presented in [39]. It occupies 52,000 logic cells, 1.5Mbits in memory and 84 multipliers can has a latency of 575 clock cycles. While the amount of memory is approximately equal to the approach presented here, the radix-8 algorithm requires only 4 stages of computation compared to the 12 of the radix-2, a factor of three difference, suggesting this method would be show similar latency, but better memory usage.

Presented in [40] are a series of radix 2 FFTs for several different samples per clock. At 8R, a 4096-pt. FFT requires 3540 slices, 120 DSPs and a latency of 5488 clock cycles. While it requires less slices without numbers of memory it is unclear of the full cost and the significantly higher latency (~4x larger) suggests high memory usage. Also compared is [41] requiring 148 DSPs and 11500 slices. Latency is unknown.

The spiral hardware generation framework [42] has results for size 256-pt radix-2,4, and 8 FFTs as well as 1024-pt. radix-2 FFT. Computations were performed in single precision floating point. For the 1024-pt. 2Gsps FFT the Spiral generated design costs about 1400 slices, but BRAM and latency is not provided. Qualitatively, this is about twice the chained reduced cost, but it is unclear how much the floating point affects this number, and how the BRAM and DSP slices compare.

While exact comparison is difficult with only partial results and or significant algorithm differences, the proposed method results in reductions in BRAM and latency cost.

5.6 Conclusion

A general streaming architecture was proposed for building large algorithms out of smaller components. Rather optimizing an individual permutation, by considering chains of permutations the area cost and latency can be reduced. This reduction occurs through sharing

simple control structure from the previous permutation in the chain to simplify control cost, and by reducing multiple permutations between computations into a single permutation block.

It was demonstrated that with short development times, Xilinx Core Generator could be leveraged by adding latency to handle high data rates. This results in significantly larger resource footprints and latency, but orders of magnitude faster development times than a custom design. This exemplifies the large gap between currently available software tools and the usage of proper memory partition techniques that reduce latency and resources.

In comparison with other streaming techniques this method also performed significantly better. Although direct one-to-one comparison examples were unavailable, extrapolated results showed a considerable qualitative difference in cost.

The techniques demonstrated in this chapter build on the previous architectures and are equally applicable to the same broad classes of digital hardware. Dual-port RAM was specified for the memory banks, which are common in most FPGA vendors. They are also a common library component for ASICs.

Chapter 6

Multi-Stage Streaming

This chapter expands upon chains of computations separated by fixed permutations by adding a degree of freedom to chain. The correctness of a computation depends on having their operands available at the same time period. Therefore, a large set of computations will be functionally correct if all operand constraints are fulfilled. The input and output data sets are expected to be in a particular order, but internally the order of computations is not fixed. This chapter considers such a scenario and proposes several scheduling algorithms to minimize the latency, and reduce memory in the datapath, while still preserving functional correctness.

Deep learning neural networks are one of the many application that exhibit multiples stages of computation with permutations required between layers. In addition, the latency of these networks is important, as a large amount of data must be processed and a decision made before the environment changes. Platforms with large amounts of parallelism, such as many GPUs work well in performing the training task for the neural network offline, but they do not translate well to processing high data rates in a streaming pipeline for fast decision making. Reconfigurable logic allows a low-latency streaming architecture. In addition, the small form factor and power efficiency of FPGAs also make them an attractive platform for implementing the forward neural network data processing.

For fully-connected neural networks, different schedules exhibit little to no difference in latency, as each node in a layer is connected to all nodes in the previous layer. Each node is functionally equivalent. As will be shown, the correct choice of schedule reduces the latency for networks which exhibit a more sparsely-connected graph. Examples of such networks include partially-connected neural networks as well as networks utilizing *dropouts*.

Sparse neural network techniques improve system performance (correct classification) by removing redundancies that primarily contribute noise. Partially-connected neural networks

reduce the number of redundant connections. The simplified network should perform at least as well as the full network. A variety of techniques have been developed [53–55, 90, 91] that reduce the connections during training, which are based on metrics related to the sensitivity of the system to a particular connection. Connections can also be removed based on their physical meanings [56], e.g. inputs are associated and clustered. Dropouts also improve the performance of a system [57], but they do so by removing nodes within the network. These two techniques result in networks with more sparsely connected graphs.

In this chapter, actors are used to describe the binary operations of each node and layer in a neural network. Each operation in the same layer is mapped to a corresponding pipeline stage of the streaming datapath. Using techniques from synchronous dataflow graphs (SDFGs), schedulers can determine the dependencies and constraints on the execution or firing of any particular node. The schedulers proposed in this chapter use greedy heuristics to reduce the latency of the neural network. This work presents four schedulers of increasing complexity for use in multiple sample streaming datapaths. Results are evaluated using a generic neural network structure over Monte Carlo simulations with different neuron connections to evaluate schedulers on representative sets of neural networks with varying levels of connectivity.

This chapter is organized as follows: Section 6.1 discusses other works related to scheduling. This is followed by background and problem formation in Section 6.2 and a proposed architecture in Section 6.3. Section 6.4 describes a set of algorithms for reducing latency, and Section 6.5 applies these schedules to a variety of partially-connected neural networks. Section 6.6 concludes with a summary and some future directions.

6.1 Related Work

A scheduler for the consumption of data by a processing unit can have the goal of minimizing resources or area, minimizing latency or delay, maximizing throughput, or some combination of these. The focus of this work is on techniques that meet a throughput constraint and minimize latency and area. A schedule has a series of tasks or *actors* that must be assigned a resource and/or time slice to be executed. Each actor has a particular cost and throughput associated with it.

A common technique to increase throughput is to target bottleneck actors (actors limiting the throughput) for increased parallelism or replication. This effort is effective in multi-core architecture [29–31] in which there are resources, in this case processors available for tasking. A common heuristic in scheduling is the use of greedy schedulers. For example, Gordon sorts actors in order of decreasing computational requirements and then replicates actors, assigning them to multiple cores until the actor is no longer a bottleneck, before considering the next element of the list [32]. Once all the cores are occupied, actors at the end of the list are

scheduled to the least occupied core until all actors have been scheduled.

The same concept also applies to FPGAs, where resources such as logic slices, DSPs, and block RAMs are substituted for cores [33]. Expanding on actor replication, Cong [34] searches a library of modules with different area and performance constraints to expand the scheduling task to both select a module from a database and to perform replication to arrive at solutions that achieve a throughput constraint. In addition to module selection, Sun [35] considers potential resource sharing when selecting modules to schedule.

In addition to module replication, FPGAs can also provide parallelism through functional pipelining. Suppose one task requires the outputs from a previous task. If that task can begin before the prior task completes, executing simultaneously, then the schedule is overlapped [36]. This becomes particularly effective for pipelined architectures as actors can be fine grained parallel operations rather than just the boundaries of larger task completions. Hwang et al. created PLS [37], which minimizes the latency, which in turn reduces resources. PLS uses a forward and backward scheduler to determine both the earliest and latest time that a task can be scheduled; by iterating over the graph the latency can be minimized. Other approaches, such as [38], formulate the problems as a set of constraints and solve using integer linear programming (ILP).

6.2 Background and Problem Formulation

This section describes the structure of the neural network data processing, specific assumptions, and requirements of the stream processing architecture for multiple samples per clock.

6.2.1 Neural Networks

A neural network is composed of k layers where the first layer is the input layer, and the last layer is the output layer with $k-2$ hidden layers. Each node implements a function $h_\theta(x)$ where x is a vector representation of the inputs to the node and θ is a vector of weights. A common function to use is $h_\theta(x) = \text{sigmoid}(\theta^T x)$.

Figure 6.1 is an example of a neural network. Before training, each node in a particular layer is connected to all the nodes in the previous layer; this is a fully connected neural network. The network may be simplified by pruning to remove connections. These connections may be noisy or redundant [53], so the pruned network is more robust. The weights θ are determined using training data with backward propagation techniques. Cun [54] tests the output deviation of removing a connection, while Hassibi [55] measures the second derivative of the deviation. Other techniques have also been proposed. The result is a set of weights with many connections pruned, or equivalently, connections of weight zero.

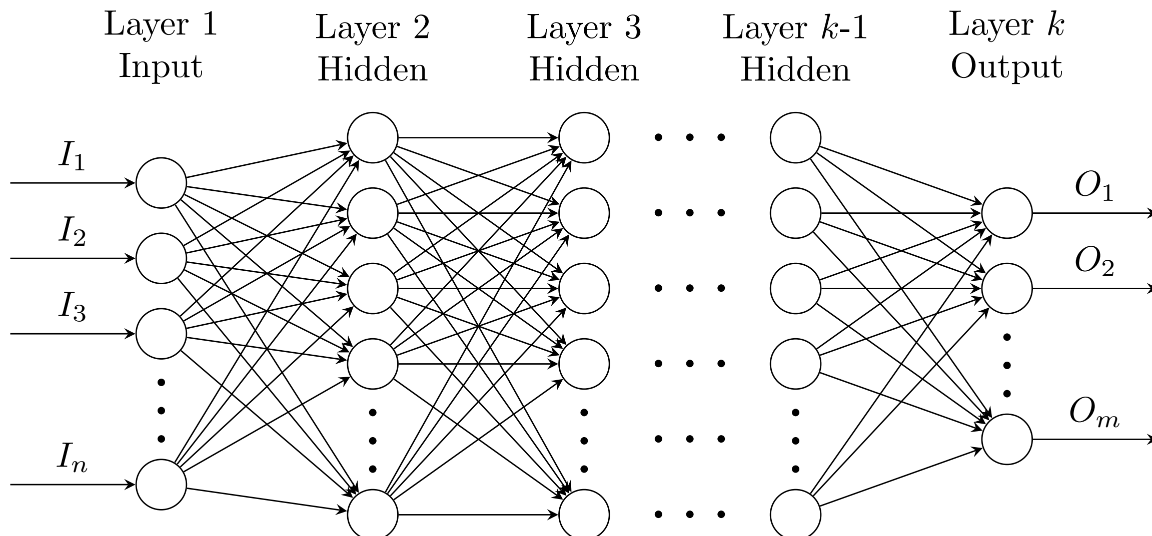


Figure 6.1: A fully connected multi-later network with k layers.

The nodes in the final pruned network have a varying number of input and output connections. Scheduling the nodes of this network no longer consists of scheduling equivalent nodes that are differentiated just by their weights; the nodes now are not-equivalent. Changes in the schedule has a significant impact on the latency.

6.2.2 Synchronous Data Flow

Synchronous data flow graphs (SDFG) are a natural model for streaming architectures. The concurrency is evident in the graphical structure, and mapping the node execution to specific clock times allows for efficient hardware.

Mapping a neural network to an SDFG involves parsing each node into its basic computations, describing these as actors, and assigning these actors to a particular stage within the pipeline.

Some assumptions are necessary to put the problem in the proper context:

- The SDFG is non-terminating, meaning that it can run indefinitely without deadlock.
- The SDFG is connected. If not, then each unconnected graph can be independently considered.

Given a series of computations on a block of data, the computation dependencies can

be described as a SDFG, denoted $G(V, E)$. V is the set of nodes that correspond to a particular computation, and E is the set of edges showing the dependencies on the nodes. In a standard scheduling problem, the delay of the node d would also be part of the problem. In this formulation, the computations will be fully pipelined so the delay for all nodes in a same stage is constant and therefore does not impact the actor order. The set of nodes V is indexed by the stage number s, and node ID i within the stage.

Internal nodes start with undetermined execution time τ and order o within a stage, but they are constrained by the edges representing a computation's data dependences. Suppose a computation occurring in the first stage requires samples $2p$ and $4p$. Edges are drawn from $V(1, 2)$ and $V(1, 4)$ to the dependent node. The execution or fire time must be greater than equal to its dependent nodes, therefore $\tau \geq 4$.

The goal is to minimize the latency of the algorithm and to reduce the memory of the system. The longer it takes an algorithm to complete, the more memory must be used to hold the samples that are constantly streaming into the system. Since a computation cannot begin execution until its data dependencies have arrived, each node must be scheduled to run at a time τ that is greater than or equal to its dependent nodes.

6.2.3 Problem Formulation

In this section the neural network is formulated as a problem of determining operations and scheduling them for parallel stream processing. Data are processed in sets of n samples. This set enters the system in a format of p samples each clock cycle. After all n samples have been received, a new data set of n samples enters the system, and the neural network begins processing this new data. Without loss of generality, p divides n evenly. If p did not divide n evenly, then n could be extended by considering a new data set composed of multiple data sets.

While the neural network has a graphical representation, for fine-grain scheduling each neural network node is divided into elementary operations. Each node performs the operation sigmoid $(\theta^T x)$. This vector operation consists of a number of multiply operations equal to the length of the vector θ with only non-zero elements. Each multiply operation is mapped to an actor, and edges are drawn from it to each x as well as θ. This discussion will focus on x, but the same procedure can be followed for θ. The only difference is that θ is known at compile time and can be shuffled appropriately based on the final schedule. The output of each multiply is then summed, resulting in another node and edges to the multiply actors. Finally, the sigmoid function is mapped to an actor connected to the sum actor.

The operations are mapped by stage such that all multiply actors in the first later of the neural network are mapped to the first stage of the schedule. All sum actors in the first layer are mapped to the second stage, followed by the sigmoid actors of the first layer. This proceeds layer-by-layer until all the layers have been mapped. The following constraints

are then applied to the problem: first, the source nodes consisting of the sequential input x is connected to the layer one multiply actors; second, sink nodes in sequential order are connected to the sigmoid actors of the last layer of the neural network.

The final SDFG representing the neural network must be scheduled onto the hardware. The number of resources allocated to each stage matches the number required for a full throughput pipeline. As p samples are available each clock number of resources available must be capable of processing p samples each stage. If the operation is an addition, then the operation has two operands so $p/2$ addition resources are allocated for that layer.

The problem then consists of scheduling the p connections per clock cycle for each actor in a stage over all the stages in the graph. The goal of the schedulers presented is to reduce the latency of the overall processing. Reduced latency will also impact memory and power. If the latency is reduced, then the amount of data in the system is reduced and less memory is needed. Likewise, if less memory is used, then less power will be used.

6.3 Architecture

The block of n samples is the data input to a processing system that performs a set of k stages of pipelined computations on the streaming data. Each stage of the computation is divided into two steps; these steps are shown in Figure 6.2.

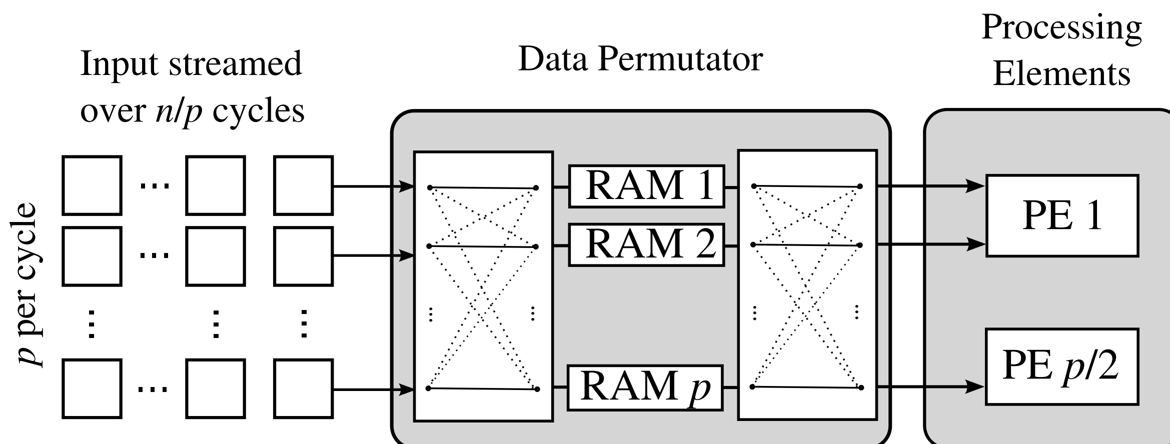


Figure 6.2: Steps describing a stage of computation.

Each clock period, p samples are supplied to the permutation block. This block implements a permutation of data such that the output of this step is arranged appropriately for the desired computations. Typically, the permutation consists of at least p memory banks

and one or more interconnection networks around the banks enabling the input samples to switch to a different stream, or a different clock cycle. Milder [27] has an architecture to implement any arbitrary permutation using $2p$ memory banks and one interconnection network, while Chen [28] reduces the number of memory banks to p, but requires two interconnection networks. For particular classes of permutations Püschel [24] demonstrates optimal control costs for an architecture similar to [28]. This approach is limited to power of two streaming widths and permutation lengths, and it only works on specific permutations that can be described using linear transformations on the bits of the data addresses. This architecture is unsuitable for permutations with a richer degree of randomness.

The output of the permutation step is then directed to the processing step, which performs the desired operation on the data stream. In Figure 6.2, each operation or actor requires two inputs; if the actor has only a single operand then p processing elements (PE) would be required.

6.4 Operation Scheduling

Given a set of operations and data dependencies, an SDFG is constructed. The source nodes have a fire time τ specified in relation to the clock cycle they are supplied to the system. The nodes also have an order o specified that corresponds to their relative order, so $o \in [0, n/p-1]$. The sink nodes also are ordered 0 to $n/p-1$, but their fire time is not fixed. The order o of each stage can be set to any permutation independent of the order of previous stages; however, the fire time of the node will depend on order and the fire time of all the nodes in the current stage and the previous stages.

This is because of constraints inherited by the pipeline. If all actors do not fire in a continuous sequence, then a bubble is introduced in the pipeline. This means that there is a resource that is not being utilized for a clock cycle. A pipeline state must be allocated to handle this. If the input arrived in n/p clocks, then it must be processed in n/p clocks or a delay will be added before the next iteration can run. After many iterations, a large amount of memory becomes dedicated to holding these delayed samples. As a consequence, the stage cannot have gaps in execution. This is resolved by requiring that all actors fire in a continuous sequence, delaying the fire times of some actors.

A brute force approach to this problem must consider all possible orders of nodes. If there are k stages and p samples connected to each node, then the number of possible combinations that must be tried is $(p!)^k$. If stage i has p_i samples, then the number of possible schedules is $\prod_{i=1}^{k} p_i!$, which becomes unreasonable for any moderate problem size. For even modest sizes of p, above 25 the amount of time needed to fully explore the schedule space even for a single stage is enormous.

In [92], a single stage of actors was examined with the goal of minimizing the latency and

memory of the required permutation. A natural approach is to extend this greedy approach stage-by-stage, beginning with the first stage.

6.4.1 Forward Propagation Schedule

The first proposed scheduler propagates constraints from the first stage to the second. If then schedules all actors of the second stage to minimize the latency while respecting the constraints based on the data dependencies and continuous outputs. That schedule then implies constraints on the next stage. This process then repeats. This is detailed in Algorithm 7. In all methods described, the schedules are statically computed in advance during hardware generation.

Algorithm 7 Forward Propagation

function FWDSCHEDULE(V(:,:))
  for $i \leftarrow 1$ **to** $k-1$ **do**
    for all $V \in V(i,:)$ **do**
      $V.\tau \leftarrow \max_{m \in V->V(i-1,m)} V(i-1,m).\tau$
    end for
    if $i \neq k-1$ **then**
      $[value, idx] \leftarrow$ ASCENDSORT$(V(i,:),\tau)$
    end if
    $Order \leftarrow 1$
    for $j \leftarrow 2$ **to** num_actors **do**
      $V(i, idx(j)).o = Order$
      if $V(i, idx(j)).\tau \leq V(i, idx(j-1)).\tau$ **then**
        $V(i, idx(j)).\tau \leftarrow V(i, idx(j-1)).\tau + 1$
      end if
      $Order \leftarrow Order + 1$
    end for
    for $j \leftarrow num_actors - 1$ **downto** 1 **do**
      $V(i,j).\tau = V(i,j+1).\tau - 1$
    end for
  end for
end function

Algorithm 7 runs from Stage 1 to Stage k. At each stage, the time is first set to the maximum of all the nodes that are connected to that node from the previous node. These nodes are then sorted in ascending order based on the earliest time the node can be executed, the time that has been set.

The ascending sort is the greedy part of the algorithm. The next loop removes repetitions.

Two nodes cannot be scheduled for the same time t, so running through the set of nodes from earliest start time to final time, any node with the same execution time gets bumped to the next execution time.

The final loop removes any gaps in execution time. That way all nodes in the stage are executed continuously as a block.

This process repeats from the initial stage up to the final stage. The final stage is slightly different because the order of the nodes is constrained to maintain a particular output order. As a result, the ascending sort is skipped. The execution times for those nodes are still propagated to the final stage from the previous stage.

6.4.2 Backward Propagation Schedule

The forward propagation schedule propagates constraints from the initial stage to final stage. Backward propagation considers the problem starting at the output constraints. Propagating these constraints backward will then result in a greedy schedule, but it may differ from Algorithm 7's schedule.

Algorithm 8 is similar to the forward algorithm (Algorithm 7), but starts at the last stage. Also, instead of setting the absolute execution time of the node, it sets the order of the nodes, 1 to p. Once the order of all nodes has been processed, then a variant of the forward pass is run, but with no change in order. `AscendSort()` is not run. This preserves the order of the node execution from the schedule of the backward propagation algorithm but fills in the execution time for all the nodes in the system.

6.4.3 Combined Schedule

In Figure 6.3 the results of the forward algorithm and backward algorithm show different latencies. This means that each generates a slightly different schedule, and because sometimes the backward outperforms the forward, there is still room for improvement.

A minor improvement is to simply call both schedulers and pick the one that performs the best. A slightly more expensive scheduling algorithm improves upon both the backward and forward algorithm by merging the schedules that are based on the two different constraints, input and output order. Algorithm 9 details the scheduler operations.

This schedule essentially generates both the forward and backward schedules and then declares a transition stage. The forward scheduler's actor order is applied up to the transition stage and the backward scheduler's actor order is applied after. Then fire times are computed based on that order. If there are k stages, then all possible stages are considered for the transition. This allows the scheduler to take the best case of k results.

Algorithm 8 Backward Propagation

function BWDSCHEDULE(V(:,:))
    for $i \leftarrow k-1$ **to** 1 **do**
        for all $V \in V(i,:)$ **do**
            $V.o \leftarrow \min_{m \in V->V(i+1,m)} V(i+1,m).o$
        end for
        if $i! = 1$ **then**
            $[value, idx] \leftarrow$ ASCENDSORT$(V(i,:), o)$
        end if
        for $j \leftarrow 2$ **to** num_actors **do**
            if $V(i, idx(j)) \leq V(i, idx(j-1))$ **then**
                $V(i, idx(j)).o \leftarrow V(i, idx(j-1)).o + 1$
            end if
        end for
        for $k \leftarrow L-1$ **downto** 0 **do**
            $\Gamma(k)[0] = \Gamma(k+1)[0] - 1$
        end for
    end for
    COMPUTEFIRETIMES(V(:,:))
end function

function COMPUTEFIRETIMES(V(:,:))
    for $i \leftarrow 1$ **to** $k-1$ **do**
        for all $V \in V(i,:)$ **do**
            $V.\tau \leftarrow \max_{m \in V->V(i-1,m)} V(i-1,m).\tau$
        end for
        $[value, idx] \leftarrow$ ASCENDSORT$(V(i,:), o)$
        for $j \leftarrow 2$ **to** num_actors **do**
            if $V(i, idx(j)).\tau \leq V(i, idx(j-1)).\tau$ **then**
                $V(i, idx(j)).\tau \leftarrow V(i, idx(j-1)).\tau + 1$
            end if
        end for
        for $j \leftarrow num_actors - 1$ **downto** 1 **do**
            $V(i,j).\tau = V(i, j+1).\tau - 1$
        end for
    end for
end function

Algorithm 9 Combined Algorithm

```
function CombinedSchedule(V(:,:))
    BestLatency ← +∞
    V_fwd ← FwdSchedule(V(:,:))
    V_bwd ← BwdSchedule(V(:,:))
    for i ← 1 to k do
        for j gets1 to i do
            V_comb(j, :) ← V_fwd(j, :)
        end for
        for  j ← i + 1 to k do
            V_comb(j, :) ← V_bwd(j, :)
        end for
        ComputeFireTimes(V)
        Latency ← Latency(V_comb)
        if Latency < BestLatency then
            BestLatency ← Latency
            V_best ← V_comb
        end if
    end for
    return V_best
end function
```

6.5 Results

Neural networks characteristics vary based on the number of layers, number of neurons per layer, and training set. To demonstrate the performance of the schedulers discussed previously, Monte Carlo simulations of 1000 runs for each design point were run. The neural network connections between nodes were randomly generated with $p/2$ actors per stage. The baseline schedule is constructed by scheduling actors as soon as possible based on the original order while satisfying the pipeline constraints. Design points are based on the number of inputs n, the number of stages k, and the number of connections between each stage of actors p.

Figure 6.3 is representative of the overall trends of latency as a function of the number of input samples n. The number of reduced stages in the neural network is 10 after collapsing the sum and sigmoid actors. The number of samples per clock is $p = 2$. The latency is exponential as a function of the number of samples. Using just a simple forward or backward greedy schedule shows significant improvement over the original schedule.

It is difficult to view the impact of the different schedules on a logarithmic scale. Table 6.1 shows the actual values. Both the forward and backward algorithm have similar results, within 1 clock on average with the forward algorithm tending to give slightly better perfor-

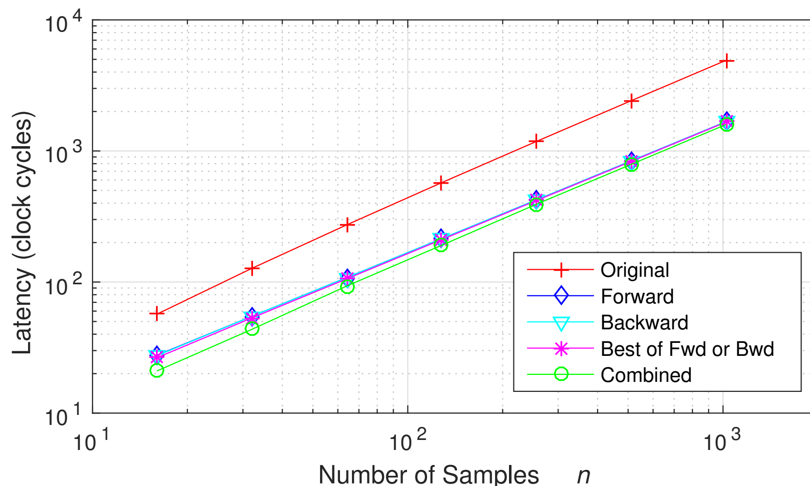


Figure 6.3: Average latency over 1000 runs as a function of number of samples n from various schedulers. The number of stages $k = 10$, samples per clock $p = 2$.

mance. Just taking the best between the forward and backward gives a slight improvement to latency, while the combined algorithm performs best of all. Also, number of clock cycles saved appears to follow an exponential trend as well. The larger n, the more clock cycles saved by the combined algorithm.

Figure 6.4 shows that latency is a function of the number of samples per clock p. All schedules appear to converge as p increases. This should make sense in that regardless of how many samples can be processed each clock, the dependencies will limit the number that actually can be processed without waiting. Again, the combined algorithm performs best.

Figure 6.5 is a plot of the average latency for the different schedules as a function of the number of stages k. The latency decreases approximately linearly with number of stages. The gap between the other schedules and the combined schedules increases slightly as the number of stages increases. With only a few stages there are fewer degrees of freedom for the combined schedule to exploit, and so the forward and backward schedules tend to perform almost as well.

6.6 Conclusion

This chapter explored exploiting the freedom in long sets of computation by using greedy schedulers to reorder computations to reduce latency. This approach was applied to partial-connected neural networks. Latency is of particular concern for many applications using these networks to interact with their environments. The situation being processed by the

Table 6.1: Latency reduction using the different schedules

	Latency (clock cycles)			Percent Reduced (%)	
n	Original	Best of Fwd and Bwd	Combined	Best of Fwd and Bwd	Combined
16	56.8	26.4	20.9	53.5	63.1
32	127.0	53.1	43.5	58.2	65.8
64	272.9	105.7	93.2	61.3	65.9
128	572.7	210.7	190.6	63.2	66.7
256	1182.8	418.7	392.4	64.6	66.8
512	2421.3	833.8	791.5	65.6	67.3
1024	4923.9	1665.1	1593.9	66.2	67.6

network can change rapidly and so it is imperative that the processing complete before the results are no longer applicable.

The approach taken to enabling low-latency neural network decision making was to map the neural network to a streaming architecture, fully leveraging the partially-connected structure of most networks. Significant improvements in the latency and throughput are achieved by increasing the width of the streaming datapath, that is, increasing the number of samples processed by each stage of the pipeline per clock cycle. This modification requires new techniques to further improve latency through scheduling.

Four variations on greedy scheduling were proposed. Constraints on both the input and output sequence order causes the scheduling algorithm to differ from traditional methods. Using these techniques the processing latency was reduced by more than 67% as compared to the original sequential schedule. This result shows the utility of using FPGA-based neural networks for fast decision making. It also demonstrates the effectiveness of considering large algorithm as a whole rather than focusing on a specific permutation or implementing the computation like on a traditional processor.

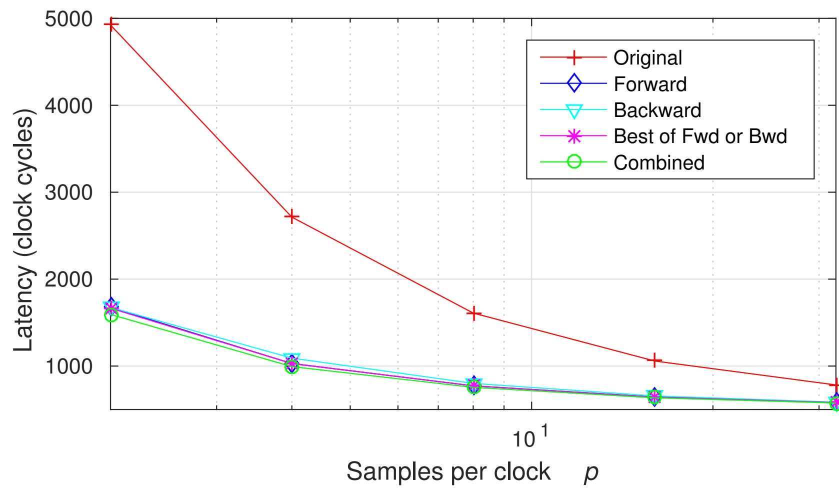


Figure 6.4: Average latency over 1000 runs as a function of sample per clock p from various schedulers. The number of stages $k = 10$, and number of samples $n = 1024$.

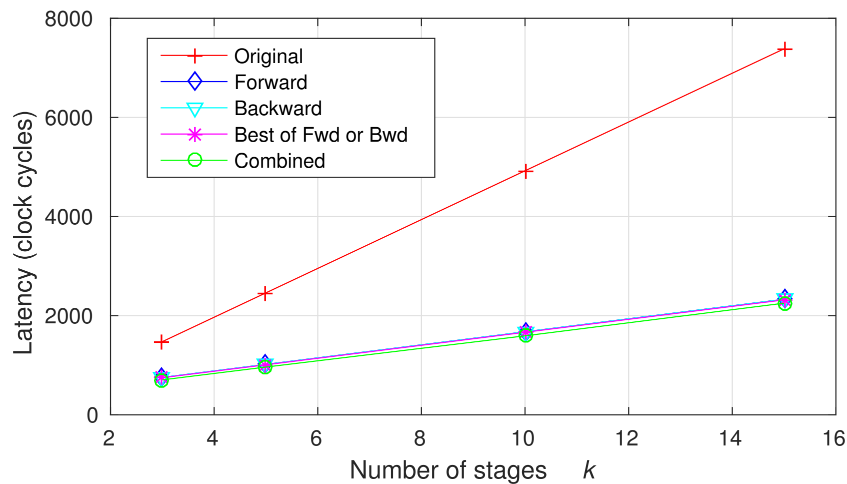


Figure 6.5: Average latency over 1000 runs as a function of number of stages k from various schedulers. The number of sample per clock $p = 2$, and number of samples $n = 1024$.

Chapter 7

The Next Generation of EDA Tools

The previous chapters have developed specific techniques for handling high-bandwidth streaming data. The algorithms and tools developed are powerful but still require that a user recognize its utility and properly apply it to a particular design. The true power of this work starts to be realized when integrated with other high-level synthesis tools. In such a setting, these algorithms can be applied in combination, resulting in more efficient hardware design while significantly improving development time.

Current tools have significant limitations, most notably for high data rate stream processing. There are two primary variations. The first type is a graphical tool consisting of set of basic component blocks that can be connected to implement a complicated signal processing algorithm. Commercial products include Xilinx System Generator, and Synopsis Symphony. The basic building blocks themselves support at maximum a single data sample per clock, resulting in a maximum data rate equal to the clock rate unless a parallel structure is manually built. In addition, connecting blocks at different clock rates and data rates is not handled well, resulting in over-built implementations requiring much more hardware than a hand-coded solution. Specifically, if the data rate is much less than the clock rate then resources can be shared, or time multiplexed. Instead, data rate changes are handled with clock enables, which simply stall operation to match the data rate. While suitable for building proof-of-concept designs to demonstrate an algorithm, systems constrained by power or resources require more efficient results.

The second type is that of high-level language translation. Altera supports an OpenCL compilation, translating parallel C-like code into hardware components. A significant limitation is that not all OpenCL code translates well to hardware, requiring that the generic OpenCL code be customized for the desired target, and the translation is locked to Altera devices. Similarly, Xilinx presents VivadoHLS which translates C code to hardware. It requires a subset of the language suitable for hardware and again may require customization

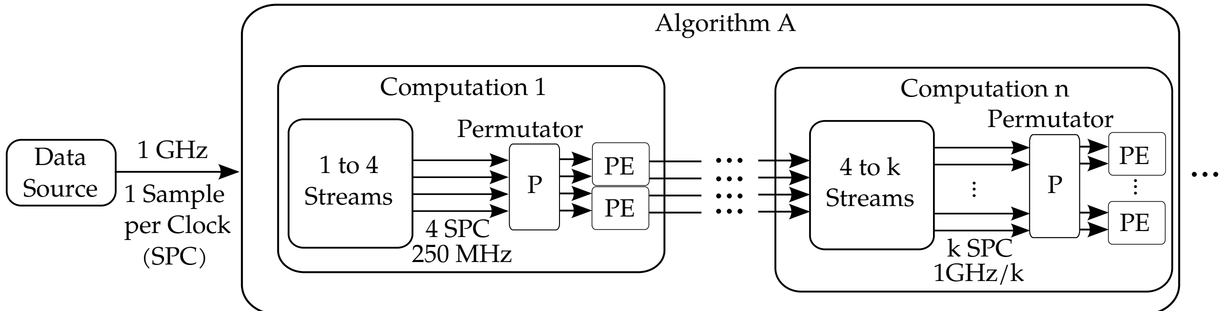


Figure 7.1: Streaming system architecture.

to best target hardware. Like the graphical tools, in order to process data at data rates above the clock rate, the parallel nature of the data stream must be manually developed.

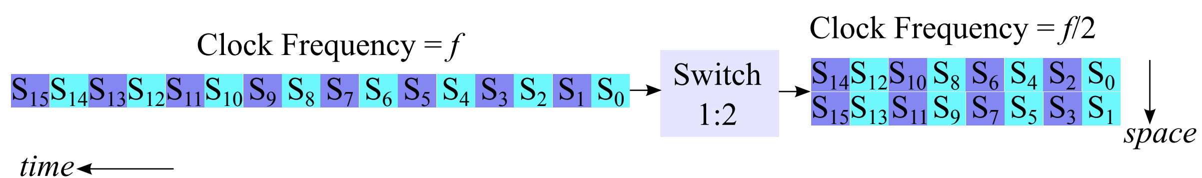
The automation algorithm developed in this work can be integrated with these existing tools to enable high data rate signal processing without manual parallelization and create efficient connections between components. This would make possible a user experience devoid of many of the manual processes currently required. It also creates hardware implementations optimized to reduce latency, power or resources.

7.1 Framework

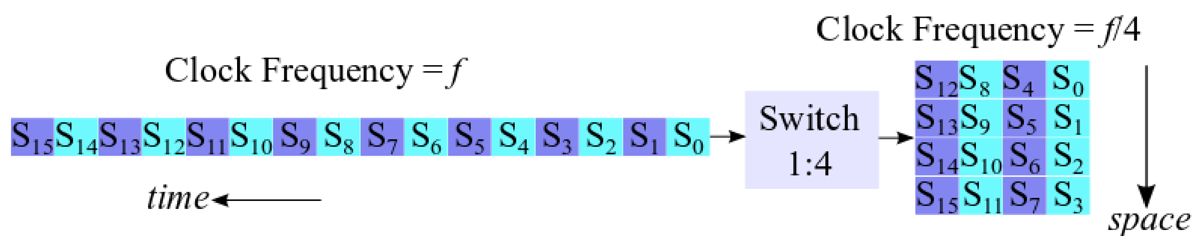
A data stream consists of a data source providing some number of samples in a given time interval. Figure 7.1 gives an example of a streaming architecture for processing such a stream. In the example, there is a single data source providing data at one trillion samples per second. This could also be expanded to handle multiple data sources each at a different data rate.

To locally optimize, an algorithm is broken down into a set of basic computation stages. Each computation stage consists first of a serializer/deserializer to change the number of parallel streams. In Figure 7.1, the first computation deserializes a single stream into four streams at one quarter of the clock rate. Then, a permutation unit P permutes the samples in time and between streams to reflect the data dependencies of the computation. The permuted streams go to the processing elements (PEs) that perform the basic operations of the computation. This optimization process is applied to each stage of computation making up the algorithm. Each computation stage may be assigned a different number of parallel streams to process based on some criteria to optimize such as area, power, frequency, or latency.

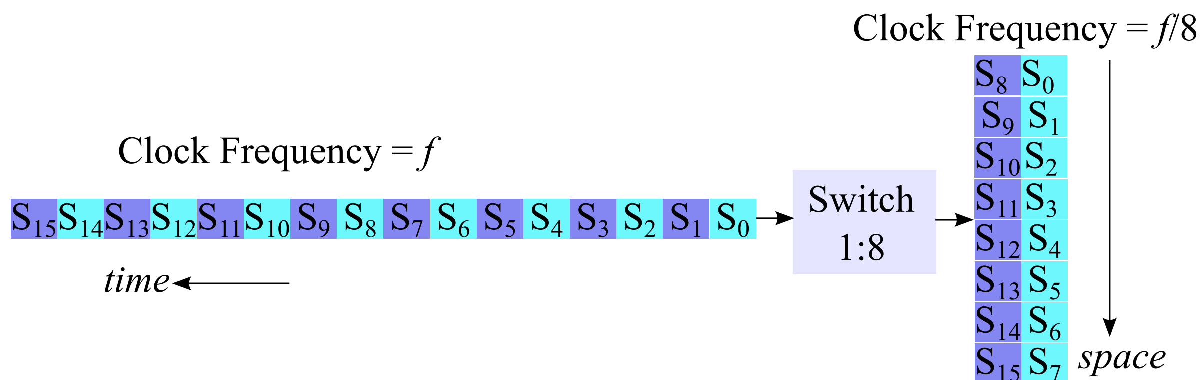
The first piece of a computation consists of a clock rate and stream changer. It translates an input clock rate and number of parallel streams into a new clock rate and/or number of parallel streams. This is similar in concept to common computer interface standards like Ethernet and PCIe, which use high-speed serial links at high clock rates to move data on a limited number of wires. The transmitter serializes data to a common communication rate, and the receiver deserializes the data stream into vectors of data whose length the receiver has been optimized to process. Figure 7.2 shows how a deserializer maps a sequential stream of data into a vectorized stream at a lower clock rate, resulting in multiple samples at each time. The input stream is multiplexed among n output streams, and the result is sampled at $1/n$ the input rate. Through this mechanism, components operating at different clock rates can still process data at the same rate.



(a) Two to one deserializer.



(b) Four to one deserializer.



(c) Eight to one deserializer.

Figure 7.2: Several deserializers showing mapping samples into space and time.

After the number of data streams is set to the desired input, the automated techniques developed can be used to translate a single computation, such as a multiplication into several parallel processing elements for the desired data rate. A permutation or data fetch unit will be added automatically to bring the necessary data elements to each processing element at the correct time. In addition, as more computations are added, the optimizations of later chapters can be used to reduce the overall cost of the permutations and share resources between later permutations. Finally, scheduling can be employed to determine if changing the order of operations can result in reduced costs.

Processing at various levels of parallelism will have different effects on the objective of optimization. An excellent example of this is the split-radix (SR) Fast Fourier Transform (FFT) [93–95]. The computation complexity of SR-FFT is significantly less than the radix-2 FFT. The number of multiplies and additions is greatly reduced; however, this comes at the cost of computation efficiency. Because of the data dependencies within the algorithm, when the parallel processing path is two samples wide, some of the multiply operations within the SR-FFT are idle half of the time. This results in a higher resource/area cost, and potentially requires more power as well.

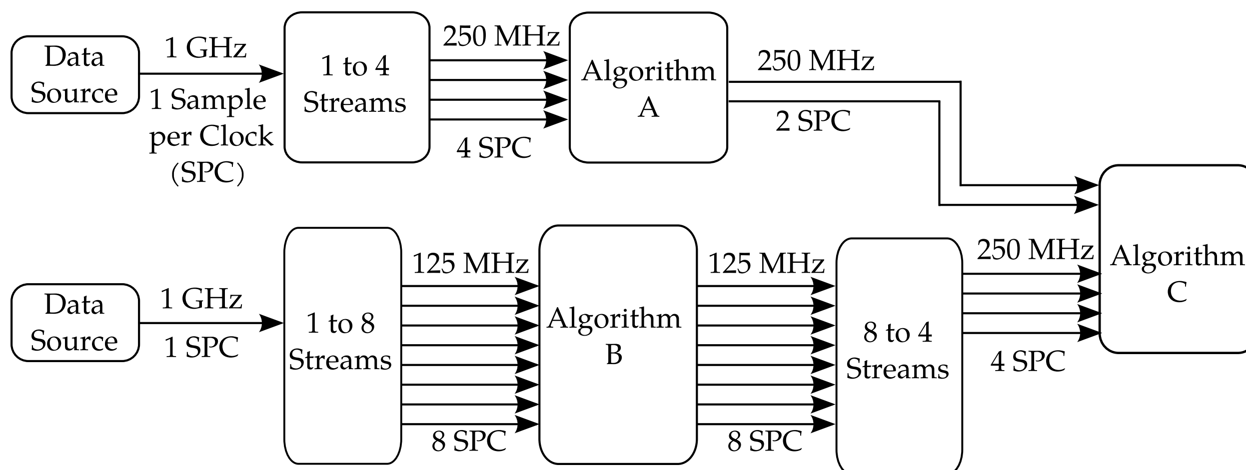


Figure 7.3: Model for optimizing data architecture with parallelism.

Figure 7.3 illustrates a possible processing architecture that has been optimized. Algorithm A may be performed on a particular device with a 250MHz clock rate. To handle the incoming data, it processes four samples at a time. In contrast, Algorithm B has a slower clock rate of 125MHz. It processes eight samples at a time. Then, Algorithm C needs the data from both A and B, but it runs at a rate of 250MHz. Data are already at that rate, but B needs to be increased. The design space is such that different devices may have different requirements for clock rate or amount of parallelism. In addition, as in the split-radix case, some algorithms can be less computationally complex but result in less computational efficiency at some levels of parallelism.

A final data implementation might appear similar to Figure 7.3 with multiple algorithms running at different clock and data rates from different sources.

The full design space of available algorithmic and parallelism options is too large to explore easily. As the two domains are mutually restricting, the choice in one domain will result in different choices in the other domain. In addition, the characteristics of the optimal algorithm and data-path structures are highly dependent on an application's performance goals and cost requirements. Zuluaga et. al. [96] developed a program to explore the pareto-optimal curve of the design space. Using such a flow combined with techniques developed in this research would further push the curve into more efficient hardware for high speed data while still significantly improving design time through automated techniques.

7.2 Design Flow

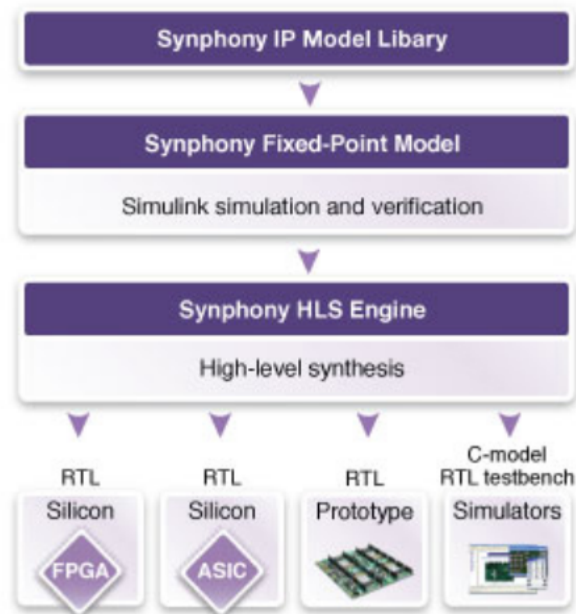


Figure 7.4: Synphony Model Compiler automates path from high-level algorithm to digital logic, image borrowed from [59].

Figure 7.4 is the design flow for the Synphony Model Compiler. In this section, the steps the tool takes to generate RTL level code from high-level descriptions will be described. In order to demonstrate how this research fits into this type of design flow, underlined steps will be added to the compilation flow.

1. **IP Model Library:** The model library provides the building blocks for building more complicated systems. Currently, the components by parametrized by input and output type, and data rate. These data rates are constrained to be less than the clock rate.

   (a) **MCode or C:** Custom components can be written in MATLAB's mcode or C. These components are later translated to RTL.

   (b) **High Data Rate Kernel Blocks:** The model library and custom components will be expanded such that a base kernel or processing element is parameterized with a maximum data rate, then the block data rate is specified. During the later compilation, the base kernel architecture will be replicated to form an array of elements sufficient to process data at the specified rate.

   (c) **Data Element Specification:** In addition to computation kernels for high data rates, a data order specification block will also be available. Parameters for this block would include data rate, set size n, and desired data element output order. An option would also allow the output order to malleable for allowing computation reordering.

2. **Validate Model:** The first actual compilations step to validate the model. This step checks that the data types and rates between blocks are consistent. This is basically equivalent to a syntax check in C compilation.

3. **Block Translation:** In this step, all high-level language blocks specified in C or as MCode is translated to representative abstract data structures for translation into RTL. These abstract types include dataflow graph descriptions easily modifiable for retiming and pipelining.

4. **Design Space Exploration:** At this stage the design space will be explored, a search of the number of parallel streams for each high data rate will be conducted in conjunction with computation reordering. The design strategy will be specified by the user as to what metrics are best to prioritize: resources, latency, or memory. After this exploration completes the level of parallelism for each block will be known and its computation order.

5. **Data Reordering:** The required architecture components for data stream width changes will be added and the desired permutations or data element selection fixed.

6. **Resource Mapping:** The abstract dataflow graphs are mapped to resources particular to the target platform, for FPGAs this includes block RAMs, DSP slices and shift registers.

7. **Translate to RTL:** The flow graphs and selected resources are then translated into a RTL netlist for synthesis.

7.3 Conclusion

In the past, designers had to work within the constraints of clock-rate fixed by the demands of the data rate, the device characteristics and available resources. By moving into a more abstract level of design the algorithm inputs may remain fixed, even as clock rates change over device or system generations. Using the proposed tool flow with the translation framework developed would automate the translation to the new domain. It would select the appropriate number of data streams for each computation, instantiate the correct number of processing elements, and implement data movers such as the permutation or data fetch unit, while performing optimization over multiple computations within the design.

This type of tool would facilitate both forward and backward compatibility without sacrificing efficiency. In addition, by integrating with existing tools, and enhancing their capability, the same development power they bring to the table would allow multiple computations to be described quickly and easily with either high level language or graphical components.

Chapter 8

Conclusion

Streaming architectures are an important method for addressing the high computation loads required to support large data sets for many algorithms. Increasing clock frequency no longer delivers the exponential increases in processing power that it did before 2005. Instead, parallelism is needed. The approach described in this dissertation uses parallel pipelines to create a large multiplier for the clock rate, resulting in a total data rate of the clock rate multiplied by the streaming width.

This dissertation focused on creating structures to handle the data movement needed for the computation engines. These structures consist of permutations, which reorder the elements in the data set, and the data fetch unit, which allows data elements to be dropped or used multiple times. These two structures are the building blocks for streaming applications, as with these and processing blocks large streaming architectures for complex algorithms can be implemented quickly.

The methods and techniques of this dissertation were developed for application on general customized digital logic platforms. None of the architecture components require any vendor-specific features. The largest restriction is on the use of dual-port memory for data storage. Most FPGAs support this component, and for those that do not, methods exist to substitute two single-port RAMs with additional switch logic.

This dissertation followed a spiral development, beginning with individual streaming permutations and expanding to answer questions of data reuse and data dropping. From there the problem space was increased to consider multiple permutations in a chain of computations. This brought up question such as, are there any optimization that can be performed to result the cost of these permutations using knowledge of the previous permutation and then what then can be done if the permutations are malleable but the result is still functionally correct.

To these questions, the answer is yes there are, and optimizations can be done in some cases. The goal is to both provide building blocks for streaming architecture and to reduce the cost of implementing these blocks, in latency, memory, and resource area.

The topics of this dissertation address these questions and offer solutions, and are summarized below. While solutions are provided, there are always future research directions, as every answer to a question brings up its own new questions.

8.1 Summary of Contributions

The contributions this dissertation provides are reiterated below:

- Techniques were developed to improve upon on RAM-based streaming permutations. This included applying theorems and algorithms from fields such as graph theory to determine lower bounds for latency and memory sizes. These algorithms were then automated with the streaming width and permutation set as input and HDL code as output.
- The permutation architecture and tools were expanded upon to allow both data reuse and skips for arbitrary data access. The techniques that were applied to permutations were applied to the new architecture with slight differences, again resulting in minimum memory and latency.
- A method was developed to reduce a complicated permutation composed of two permutations into simple control using a single permutation architecture.
- A method to improve latency was devised to break complex algorithms down into their basic computations and reorder those computations to reduce latency. Four variations on greedy schedulers were developed. They use heuristics to create schedules that have improved latency.

8.2 Recommendations for Future Work

There are still several aspects of the implementation of streaming permutations that can be explored. In Chapter 3 linear permutations were compared with the general case. Work is still needed for further reductions in area and resource cost. The latency minimization could be applied to linear permutations. This technique was used in the FPGA design in [2], but it was never formalized to a general approach. In addition, it would be significant if the offline control methods of linear permutations could be applied to non-power-of-two

streaming widths. It seems likely that such an implementation would require division, but perhaps subtraction could be substituted in some case for an overall savings in resource cost.

Data reuse is primarily an extension of the permutation architecture. Many of the same optimizations are applicable to it as well. For some classes of applications, it should be possible to describe the transformation using the bit index representations. Online control could also be applied in this case.

Both permutations and data reuse architectures would benefit from a joint optimization of latency, control and connectivity. If the tool could be adjusted to weigh one more heavily than the others, designers could better utilize their available resources. In addition, the potential impact of a small increase in latency is unknown; it could result in a significant decrease in control cost and area.

For algorithms consisting of multiple computations, only reducing the latency of the permutation was considered. Each permutation can add a significant amount of resource overhead. An alternative approach that might be more cost-effective is to consider if the reordering of operations can reduce the need for some permutations altogether, albeit likely at the cost of a larger permutation for those remaining. It seems likely that such an approach could ultimately prove more cost-effective; it almost certainly would in resource cost, but also possibly in latency, as each switch network adds a finite delay to the processing chain.

Bibliography

[1] F.-L. Luo, W. Williams, R. M. Rao, R. Narasimha, and M. J. Montpetit, "Trends in signal processing applications and industry technology [in the spotlight]," *IEEE Signal Processing Magazine*, vol. 29, no. 1, pp. 184–174, 2012.

[2] V. Adhinarayanan, T. Koehn, K. Kepa, W.-C. Feng, and P. Athanas, "On the performance and energy efficiency of FPGAs and GPUs for polyphase channelization," in *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*. IEEE, 2014, pp. 1–7.

[3] A. Danowitz, K. Kelley, J. Mao, J. P. Stevenson, and M. Horowitz, "CPU DB: recording microprocessor history," *Communications of the ACM*, vol. 55, no. 4, pp. 55–63, 2012.

[4] K. G. Shin and P. Ramanathan, "Real-time computing: A new discipline of computer science and engineering," *Proceedings of the IEEE*, vol. 82, no. 1, pp. 6–24, 1994.

[5] "System generator for DSP user guide (ug640)," Xilinx Inc., October 2012. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_4/sysgen_user.pdf

[6] T. Koehn and P. Athanas, "Arbitrary streaming permutations with minimum memory and latency," in *Proc. IEEE/ACM Intl. Conf. on Computer-Aided Design*. IEEE Press, 2016.

[7] ——, "Data staging for efficient high throughput stream processing," *IEEE Transactions on Computers*, Submitted for publication.

[8] ——, "Automating structured matrix-matrix multiplication for stream processing," in *Proc. 2016 Intl. Conf. ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, 2016, p. To Appear.

[9] ——, "Buffering strategies for ultra high-throughput stream processing," in *Proc. 2015 Intl. Conf. ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, 2015, pp. 1–7.

[10] ——, "Optimizing schedulers for data marshalling in neural network applications," 2017, unpublished.

[11] H. D. Shapiro, "Theoretical limitations on the efficient use of parallel memories," *IEEE Trans. Comput.*, vol. 100, no. 5, pp. 421–428, May 1978.

[12] C. Clos, "A study of non-blocking switching networks," *Bell System Technical Journal*, vol. 32, no. 2, pp. 406–424, 1953.

[13] V. E. Benes, *Mathematical Theory of Connecting Networks and Telephone Traffic.* Academic Press, 1965.

[14] C. Chang and R. Melhem, "Arbitrary size Benes networks," *Parallel Processing Letters*, vol. 7, no. 03, pp. 279–284, 1997.

[15] A. Waksman, "A permutation network," *Journal of the ACM (JACM)*, vol. 15, no. 1, pp. 159–163, 1968.

[16] B. Beauquier and É. Darrot, "On arbitrary size Waksman networks and their vulnerability," *Parallel Processing Letters*, vol. 12, no. 03n04, pp. 287–296, 2002.

[17] D. T. Harper, "Block, multistride vector, and FFT accesses in parallel memory systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 2, no. 1, pp. 43–51, Jan. 1991.

[18] D. Lee, "Solution to an architectural problem in parallel computing," in *Proc. 3rd Symp. Frontiers of Massively Parallel Computation.* IEEE, 1990, pp. 434–442.

[19] P.-Y. Tsai and C.-Y. Lin, "A generalized conflict-free memory addressing scheme for continuous-flow parallel-processing FFT processors with rescheduling," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst*, vol. 19, no. 12, pp. 2290–2302, Dec. 2011.

[20] C. Gong, X. Li, Z. Wu, and D. Liu, "A conflict-free access method for parallel turbo decoder," in *6th Intl. Conf. Wireless Commun. Signal Process. (WCSP).* IEEE, Oct. 2014, pp. 1–6.

[21] T. Järvinen, P. Salmela, H. Sorokin, and J. Takala, "Stride permutation networks for array processors," *The Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, vol. 49, no. 1, pp. 51–71, 2007.

[22] S. F. Gorman and J. M. Wills, "Partial column FFT pipelines," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 42, no. 6, pp. 414–423, 1995.

[23] K. K. Parhi, "Systematic synthesis of DSP data format converters using life-time analysis and forward-backward register allocation," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 39, no. 7, pp. 423–440, 1992.

[24] M. Püschel, P. A. Milder, and J. C. Hoe, "Permuting streaming data using RAMs," *Journal of the ACM (JACM)*, vol. 56, no. 2, p. 10, 2009.

[25] F. Serre, T. Holenstein, and M. Püschel, "Optimal circuits for streamed linear permutations using RAM," in *Proc. 2016 ACM/SIGDA Intl. Symp. Field-Programmable Gate Arrays*. ACM, 2016, pp. 215–223.

[26] R. Chen and V. K. Prasanna, "Energy-efficient architecture for stride permutation on streaming data," in *Reconfigurable Computing and FPGAs (ReConFig)*. IEEE, 2013, pp. 1–7.

[27] P. A. Milder, J. C. Hoe, and M. Püschel, "Automatic generation of streaming datapaths for arbitrary fixed permutations," in *Proc. Conference on Design, Automation and Test in Europe*. European Design and Automation Association, 2009, pp. 1118–1123.

[28] R. Chen and V. K. Prasanna, "Automatic generation of high throughput energy efficient streaming architectures for arbitrary fixed permutations," in *Field Programmable Logic and Applications (FPL)*. IEEE, 2015, pp. 1–8.

[29] M. I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," in *ACM SIGOPS Operating Systems Review*, vol. 40, no. 5. ACM, 2006, pp. 151–162.

[30] M. Kudlur and S. Mahlke, "Orchestrating the execution of stream programs on multicore platforms," in *ACM SIGPLAN Notices*, vol. 43, no. 6. ACM, 2008, pp. 114–124.

[31] A. H. Hormati, Y. Choi, M. Kudlur, R. Rabbah, T. Mudge, and S. Mahlke, "Flextream: Adaptive compilation of streaming applications for heterogeneous architectures," in *Proc. 18th Intl. Conf. Parallel Architectures and Compilation Techniques (PACT'09)*. IEEE, 2009, pp. 214–223.

[32] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze *et al.*, "A stream compiler for communication-exposed architectures," in *ACM SIGPLAN Notices*, vol. 37, no. 10. ACM, 2002, pp. 291–303.

[33] A. Hagiescu, W.-F. Wong, D. F. Bacon, and R. Rabbah, "A computing origami: folding streams in FPGAs," in *Proc. 46 ACM/IEEE Design Automation Conf. (DAC'09)*. IEEE, 2009, pp. 282–287.

[34] J. Cong, M. Huang, B. Liu, P. Zhang, and Y. Zou, "Combining module selection and replication for throughput-driven streaming programs," in *Proc. of the Conference on Design, Automation and Test in Europe*. EDA Consortium, 2012, pp. 1018–1023.

[35] W. Sun, M. J. Wirthlin, and S. Neuendorffer, "FPGA pipeline synthesis design exploration using module selection and resource sharing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 254–265, 2007.

[36] S. L. Sindorf and S. H. Gerez, "An integer linear programming approach to the overlapped scheduling of iterative data-flow graphs for target architectures with communication delays," in *Proc. Workshop on Embedded Systems (PROGRESS '00)*, 2000.

[37] C.-T. Hwang, Y.-C. Hsu, and Y.-L. Lin, "PLS: A scheduler for pipeline synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 12, no. 9, pp. 1279–1286, 1993.

[38] P. Sucha, Z. Pohl, and Z. Hanzalek, "Scheduling of iterative algorithms on FPGA with pipelined arithmetic unit," in *Proc. 2004 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '04)*. IEEE, 2004, pp. 404–412.

[39] K. Zhong, H. He, and G. Zhu, "An ultra high-speed FFT processor," in *Proc. 2003 Intl. Symp. Signals, Circuits and Systems (SCS 2003)*, vol. 1. IEEE, 2003, pp. 37–40.

[40] M. Garrido, J. Grajal, M. Sanchez, O. Gustafsson *et al.*, "Pipelined radix-feedforward FFT architectures," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, no. 1, pp. 23–32, 2013.

[41] P. A. Milder, F. Franchetti, J. C. Hoe, and M. Püschel, "Formal datapath representation and manipulation for implementing DSP transforms," in *Proc. of the 45th Annual Design Automation Conference*. ACM, 2008, pp. 385–390.

[42] 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, p. 15, 2012.

[43] A. Amira, A. Bouridane, P. Milligan, and P. Sage, "A high throughput FPGA implementation of a bit-level matrix product," in *2000 IEEE Workshop on Signal Processing Systems (SiPS)*. IEEE, 2000, pp. 356–364.

[44] L. Zhuo and V. K. Prasanna, "High-performance designs for linear algebra operations on reconfigurable hardware," *IEEE Transactions on Computers*, vol. 57, no. 8, pp. 1057–1071, 2008.

[45] M. DeLorimier and A. DeHon, "Floating-point sparse matrix-vector multiply for FPGAs," in *Proc. 2005 ACM/SIGDA 13th Intl. Symp. Field-Programmable Gate Arrays*. ACM, 2005, pp. 75–85.

[46] L. Zhuo and V. K. Prasanna, "Sparse matrix-vector multiplication on FPGAs," in *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays*. ACM, 2005, pp. 63–74.

[47] S. Akella, M. C. Smith, R. T. Mills, S. R. Alam, R. F. Barrett, and J. S. Vetter, "Sparse matrix-vector multiplication kernel on a reconfigurable computer," in *Proc. 9th Ann. High-Performance Embedded Computing Workshop*, 2005.

[48] C. Y. Lin, N. Wong, and H. K.-H. So, "Design space exploration for sparse matrix-matrix multiplication on FPGAs," *International Journal of Circuit Theory and Applications*, vol. 41, no. 2, pp. 205–219, 2013.

[49] A. Muthuramalingam, S. Himavathi, and E. Srinivasan, "Neural network implementation using FPGA: Issues and application," *International Journal of Information Technology*, vol. 4, no. 2, pp. 86–92, 2008.

[50] S. Sahin, Y. Becerikli, and S. Yazici, "Neural network implementation in hardware using FPGAs," in *Proc. Intl. Conf. Neural Information Processing.* Springer, 2006, pp. 1105–1112.

[51] R. Gadea, J. Cerdá, F. Ballester, and A. Mocholí, "Artificial neural network implementation on a single FPGA of a pipelined on-line backpropagation," in *Proc. 13th Intl. Symp. System Synthesis.* IEEE Computer Society, 2000, pp. 225–230.

[52] P. Lysaght, J. Stockwood, J. Law, and D. Girma, "Artificial neural network implementation on a fine-grained FPGA," in *Proc. Intl. Workshop on Field Programmable Logic and Applications.* Springer, 1994, pp. 421–431.

[53] J. Sietsma and R. J. Dow, "Neural net pruning-why and how," in *Proc. IEEE Intl. Conf. Neural Networks.* IEEE, 1988, pp. 325–333.

[54] Y. LeCun, J. S. Denker, and S. A. Solla, "Optimal brain damage." in *Adv. Neural Inform. Process. Syst.*, vol. 2, 1990, pp. 598–605.

[55] B. Hassibi, D. G. Stork, G. Wolff, and T. Watanabe, "Optimal brain surgeon: Extensions and performance comparison." *Advances in Neural Information Processing Systems*, vol. 6, pp. 263–270, 1994.

[56] S. Kang and C. Isik, "Partially connected feedforward neural networks structured by input types," *IEEE Transactions on Neural Networks*, vol. 16, no. 1, pp. 175–184, 2005.

[57] D. B. T. Jiang Su and P. Y. K. Cheung, "Increasing network size and training throughput of FPGA restricted Boltzmann machines using dropout," in *Intl. Symp. Field-Programmable Custom Computing Machines (FCCM)*, 2016.

[58] "Vivado design suite user guide: High-level synthesis (ug902)," Xilinx Inc., July 2012. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2012_2/ug902-vivado-high-level-synthesis.pdf

[59] "Synphony model compiler high-level synthesis with synphony model compiler," Synopsys, March 2014. [Online]. Available: http://http://www.synopsys.com/cgi-bin/proto/pdfdla/pdfr1.cgi?file=synphony_model_comp_ds.pdf

[60] "Synphony C compiler high-level synthesis from C/C++ to RTL," Synopsys, March 2014. [Online]. Available: http://www.synopsys.com/Tools/Implementation/RTLSynthesis/Documents/synphonyc-compiler-ds.pdf

[61] A. Prost-Boucle, O. Muller, and F. Rousseau, "Fast and standalone design space exploration for high-level synthesis under resource constraints," *Journal of Systems Architecture*, vol. 60, no. 1, pp. 79–93, 2014.

[62] K. Bertels, *Hardware/Software Co-design for Heterogeneous Multi-core Platforms.* Springer, 2012.

[63] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, "LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 2, p. 24, 2013.

[64] E. Martin, O. Sentieys, H. Dubois, and J.-L. Philippe, "Gaut: An architectural synthesis tool for dedicated signal processors," in *Proc. European Design Automation Conference(EURO-DAC'93).* IEEE, 1993, pp. 14–19.

[65] "SPIRAL hardware generator." [Online]. Available: http://www.spiral.nel/hardware.html

[66] M. Puschel, J. M. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko *et al.*, "SPIRAL: Code generation for DSP transforms," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 232–275, 2005.

[67] Y. Voronenko and M. Püschel, "Multiplierless multiple constant multiplication," *ACM Transactions on Algorithms (TALG)*, vol. 3, no. 2, p. 11, 2007.

[68] M. Zuluaga, P. Milder, and M. Püschel, "Streaming sorting networks," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 21, no. 4, p. 55, 2016.

[69] S. C. Han, "A flexible decoder and performance evaluation for array-structured LDPC codes," Ph.D. dissertation, Carnegie Mellon University, 2007.

[70] D. König, "Graphok és alkalmazsuk a determinánsok és a halmazok elméletére," *Matematikai és Természettudományi Értesitő*, vol. 34, pp. 104–119, 1916.

[71] N. Biggs, E. Lloyd, and R. Wilson, *Graph Theory 1736-1936, 1976.* Clarendon Press, 1976.

[72] J. Lenfant and S. Tahé, "Permuting data with the omega network," *Acta Informatica*, vol. 21, no. 6, pp. 629–641, 1985.

[73] R. Chen, S. Siriyal, and V. Prasanna, "Energy and memory efficient mapping of bitonic sorting on FPGA," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays.* ACM, 2015, pp. 240–249.

[74] P. Bürgisser, M. Clausen, and A. Shokrollahi, *Algebraic complexity theory.* Springer Science & Business Media, 2013, vol. 315.

[75] F. Serre and M. Püschel, "Generalizing block LU factorization: A lower–upper–lower block triangular decomposition with minimal off-diagonal ranks," *Linear Algebra and its Applications*, vol. 509, pp. 114–142, 2016.

[76] R. Cole, K. Ost, and S. Schirra, "Edge-coloring bipartite multigraphs in O(E log D) time," *Combinatorica*, vol. 21, no. 1, pp. 5–12, 2001.

[77] R. Cole and J. Hopcroft, "On edge coloring bipartite graphs," *SIAM Journal on Computing*, vol. 11, no. 3, pp. 540–546, 1982.

[78] N. Alon, "A simple algorithm for edge-coloring bipartite multigraphs," *Information Processing Letters*, vol. 85, no. 6, pp. 301–302, Mar 2003.

[79] H. N. Gabow and O. Kariv, "Algorithms for edge coloring bipartite graphs," in *Proc. 10th Annual ACM Symp. Theory of Computing.* ACM, 1978, pp. 184–192.

[80] E. Sprangle and D. Carmean, "Increasing processor performance by implementing deeper pipelines," in *29th Annual Intl. Symp. Computer Architecture.* IEEE, 2002, pp. 25–34.

[81] J. Jiang, V. Mirian, K. P. Tang, P. Chow, and Z. Xing, "Matrix multiplication based on scalable macro-pipelined FPGA accelerator architecture," in *Proc. 2009 Int. Conf. ReConFigurable Computing and FPGAs (ReConFig).* IEEE, 2009, pp. 48–53.

[82] S. J. Campbell and S. P. Khatri, "Resource and delay efficient matrix multiplication using newer FPGA devices," in *Proc. 16th ACM Great Lakes Symposium on VLSI.* ACM, 2006, pp. 308–311.

[83] R. Dorrance, F. Ren, and D. Marković, "A scalable sparse matrix-vector multiplication kernel for energy-efficient sparse-BLAS on FPGAs," in *Proc. 2014 ACM/SIGDA Intl. Symp. on Field-Programmable Gate Arrays.* ACM, 2014, pp. 161–170.

[84] S. M. Qasim, S. A. Abbasi, and B. Almashary, "A proposed FPGA-based parallel architecture for matrix multiplication," in *Proc. IEEE Asia Pacific Conf. Circuits and Systems (APCCAS '08).* IEEE, 2008, pp. 1763–1766.

[85] Xilinx. (2013, Mar) LogiCORE IP linear algebra toolkit v2.0. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/linear_algebra_toolkit/v2_0/pg131-linear-algebra-toolkit.pdf

[86] K. Y. Lee, "On the rearrangeability of $2(\log_2 n) - 1$ stage permutation networks," *IEEE Transactions on Computers*, vol. 34, no. 5, pp. 412–425, May 1985.

[87] P. Rosenfeld, "Performance exploration of the hybrid memory cube," Ph.D. dissertation, University of Maryland,, 2014.

[88] R. Giddings, "Real-time digital signal processing for optical OFDM-based future optical access networks," *IEEE J. Lightw. Tech.*, vol. 32, no. 4, pp. 553–570, Sep. 2014.

[89] "LogiCORE IP fast Fourier transform v8.0 (ds808)," Xilinx Inc., July 2012. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/ds808_xfft.pdf

[90] D. E. Duckro, D. W. Quinn, and S. J. Gardner Iii, "Neural network pruning with Tukey-Kramer multiple comparison procedure," *Neural Computation*, vol. 14, no. 5, pp. 1149–1168, 2002.

[91] E. D. Karnin, "A simple procedure for pruning back-propagation trained neural networks," *IEEE Transactions on Neural Networks*, vol. 1, no. 2, pp. 239–242, 1990.

[92] T. Koehn and P. Athanas, "Finding space-time stream permutations for minimum memory and latency," in *Proc. Intl. Symp. Field-Programmable Custom Computing Machines (FCCM)*, 2016.

[93] J. García, J. A. Michell, G. Ruiz, and A. M. Burón, "FPGA realization of a split radix FFT processor," in *Microtechnologies for the New Millennium*. International Society for Optics and Photonics, 2007, pp. 65 900P–65 900P.

[94] A. Petrovsky, S. L. Shkredov *et al.*, "Automatic generation of split-radix 2-4 parallel-pipeline FFT processors: hardware reconfiguration and core optimizations," in *Proc. Intl. Symp. Parallel Computing in Electrical Engineering*. IEEE, 2006, pp. 181–186.

[95] C. Watanabe, C. Silva, and J. Muñoz, "Implementation of split-radix fast fourier transform on FPGA," in *Programmable Logic Conference (SPL), 2010 VI Southern*. IEEE, 2010, pp. 167–170.

[96] M. Zuluaga, A. Krause, P. Milder, and M. Püschel, "Smart design space sampling to predict pareto-optimal solutions," in *ACM SIGPLAN Notices*, vol. 47, no. 5. ACM, 2012, pp. 119–128.