STREAM COMPUTING ON FPGAs

by

Franjo Plavec

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Electrical and Computer Engineering
University of Toronto

© Copyright by Franjo Plavec 2010

Abstract

Stream Computing on FPGAs

Franjo Plavec

Doctor of Philosophy

Graduate Department of Electrical and Computer Engineering

University of Toronto

2010

Field Programmable Gate Arrays (FPGAs) are programmable logic devices used for the implementation of a wide range of digital systems. In recent years, there has been an increasing interest in design methodologies that allow high-level design descriptions to be automatically implemented in FPGAs. This thesis describes the design and implementation of a novel compilation flow that implements circuits in FPGAs from a streaming programming language. The streaming language supported is called FPGA Brook, and is based on the existing Brook and GPU Brook languages, which target streaming multiprocessors and graphics processing units (GPUs), respectively. A streaming language is suitable for targeting FPGAs because it allows system designers to express applications in a way that exposes parallelism, which can then be exploited through parallel hardware implementation. FPGA Brook supports replication, which allows the system designer to trade-off area for performance, by specifying the parts of an application that should be implemented as multiple hardware units operating in parallel, to achieve desired application throughput. Hardware units are interconnected through FIFO buffers, which effectively utilize the small memory modules available in FPGAs.

The FPGA Brook design flow uses a source-to-source compiler, and combines it with a commercial behavioural synthesis tool to generate hardware. The source-to-source compiler was developed as a part of this thesis and includes novel algorithms for implementation of complex reductions in FPGAs. The design flow is fully automated and presents a user-interface similar to traditional software compilers. A

suite of benchmark applications was developed in FPGA Brook and implemented using our design flow. Experimental results show that applications implemented using our flow achieve much higher throughput than the Nios II soft processor implemented in the same FPGA device. Comparison to the commercial C2H compiler from Altera shows that while simple applications can be effectively implemented using the C2H compiler, complex applications achieve significantly better throughput when implemented by our system. Performance of many applications implemented using our design flow would scale further if a larger FPGA device were used. The thesis demonstrates that using an automated design flow to implement streaming applications in FPGAs is a promising methodology.

Acknowledgments

First, I would like to thank my supervisors Professor Zvonko Vranesic and Professor Stephen Brown for their advice and support throughout the course of this thesis. I particularly appreciated the freedom I enjoyed throughout my studies, which allowed me to engage in various activities outside the thesis work that made the journey worthwhile. I am greatly indebted to you both. A special thanks to Professor Vranesic for his prompt review of the thesis manuscript, even at the time when he was very busy with his other obligations.

Next, I would like to thank Deshanand Singh for his suggestion of the thesis topic. I would also like to thank Pranit Patel for evaluating my compiler and developing the Mandelbrot benchmarks. Tomasz Czajkowski and Andrew Ling contributed to this thesis through their constructive feedback during our joint research meetings, and were great colleagues to have overall, for which I am grateful. I would like to acknowledge the contribution of my committee members Professor Paul Chow and Professor Greg Steffan. Their feedback was valuable and useful in keeping me on the right track. A special thanks to Professor Steffan for including my compiler in his graduate course project.

Many graduate students in the Computer research group and the ECE Department enhanced my learning through their participation in various reading groups and seminars. Discussions I have had with folks in LP392, mostly unrelated to our research, helped me learn valuable life lessons that no thesis work could ever have taught me. Ivan Matosevic, Davor Capalija, Cedomir Segulja and Adam Czajkowski let me crash at their place countless times in the past year and were at the same time great friends to have, for which I am thankful. Thanks to all undergraduate students, whom I have had honour to serve as a teaching assistant, for providing me the opportunities to develop my teaching skills. Thanks to all the students, staff and the faculty of the Department and the University for making my work and life on campus a pleasant experience. A special thanks to Kelly Chan for always meeting me with a smile, and promptly resolving all administrative matters.

I am immensely grateful to my dear girlfriend Veronica Shuyin Liang for sharing most of this journey with me. If there was only one reason the PhD was worth doing, this would be it: without it we probably never would have met. Thank you for being there for me, particularly during the many late nights spent working during these last few weeks when I was particularly needy.

Many outstanding individuals have shaped my life and made me who I am today. My parents, Ana and Valentin Plavec taught me honesty, integrity, hard work, and respect for all human beings. It took immense sacrifice on their behalf to persist in these values through many trying times we have had. No scientific research is needed to show that these values lead to success in life, in one way or another. I hope my father (may his soul rest in peace) would be proud today, and I know my mother is. My siblings, their

spouses and all of my extended family have been very supportive throughout, and helped me in countless ways when circumstances so required. Special thanks in this regard go to my uncle and aunt, Franjo and Vera Plavec for their financial and moral support during my studies, as well as for encouraging me to take the extra English classes.

Countless teachers and professors have taught me everything I know. At the considerable risk of forgetting someone important, let me name just a few, in the order of appearance. Marija Pupovac taught me how to read and write, Ownalla Abdel Sakhi introduced me to the beauty of formal mathematics and also taught me my first programming lessons, Nikola Habuzin taught me my first English words, Slavko Marencic lead me through my first steps in electronics, Kresimir Kovsca unlocked the magical world of physics for me, and finally Professor Sinisa Srbljic first exposed me to the world of research and encouraged my decision to pursue studies at the University of Toronto. Countless others deserve to be named here, and I am thankful to them all. It is my opinion that teaching is one of the most valuable and at the same time most undervalued professions, and you should all be proud of being great teachers. Finally, many friends have accompanied me throughout my formal education. While you are too numerous to be all named here, I appreciate every minute we spent together and thank you for sharing the journey with me.

This work has been financially supported by NSERC, the University of Toronto Fellowship, Edward S. Rogers Sr. Graduate Scholarship, Canadian Microelectronics Corporation and Altera Corporation. I acknowledge and thank them for their support, for this work would not be possible without it.

Contents

1 Introduction 1
1.1. Thesis Organization 4
2 Background and Related Work 5
2.1. Stream Computing 5
2.2. Brook and GPU Brook 6
2.3. Field Programmable Gate Arrays 11
2.4. Nios II and C2H 12
2.5. Related Work 15
3 FPGA Brook 20
3.1. Structure of FPGA Brook Programs and Its Limitations 20
3.2. Custom Stream Creation and Consumption 21
3.3. Streams of Arrays 24
3.4. Speedup Pragma Statement 29
3.5. User Interface 30
3.6. FPGA Brook Summary 31
4 Compiling FPGA Brook programs into FPGA Logic 33
4.1. Design Flow 33
4.2. Compiling FPGA Brook Code to C2H Code 34
4.2.1. Handling One-Dimensional Streams 35
4.2.2. Handling Two-Dimensional Streams 38
4.2.3. Strength Reduction of Modulo and Division Operations 40
4.3. Generating FIFO Hardware 42
4.4. Generating SOPC System and Compilation Scripts 46
5 Exploiting Data Parallelism through Kernel Replication 48
5.1. Replicating Ordinary Kernels 49
5.1.1. Collector and Distributor FIFO Buffers 51
5.1.2. Array FIFO Collectors and Distributors 52
5.2. Replicating Reductions of One-Dimensional Streams 55
5.2.1. Building reduction trees 56
5.2.2. Double-Distributor FIFOs 61
5.2.3. Code Generation for Reduction Kernel Replicas 62
5.3. Replicating Reductions of Two-Dimensional Streams 67
5.4. Theoretical Limits to Throughput Improvement due to Replication 70

5.5. Strength Reduction of Modulo and Division Operations in the Presence of Replication 75
5.6. Putting It All Together: Building the Replicated Streaming Dataflow Graph 81
6 Experimental Evaluation 83
6.1. Development of the FPGA Brook Benchmark Set 83
6.1.1. Autocorrelation 83
6.1.2. FIR Filter 84
6.1.3. Two-Dimensional Convolution 85
6.1.4. Matrix-Vector and Matrix-Matrix Multiplication 86
6.1.5. Mandelbrot Set Generator 87
6.1.6. Zigzag Ordering in MPEG-2 89
6.1.7. Inverse Quantization in MPEG-2 89
6.1.8. Saturation in MPEG-2 90
6.1.9. Mismatch Control in MPEG-2 90
6.1.10. Inverse Discrete Cosine Transform (IDCT) 91
6.1.11. MPEG-2 Spatial Decoding 93
6.2. Experimental Methodology 94
6.3. Experimental Results 97
6.3.1. Autocorrelation 98
6.3.2. FIR Filter 99
6.3.3. Two-Dimensional Convolution 102
6.3.4. Matrix-Vector and Matrix-Matrix Multiplication 102
6.3.5. Mandelbrot Set Generator 104
6.3.6. Zigzag Ordering, Inverse Quantization and Mismatch Control 105
6.3.7. Saturation 107
6.3.8. Inverse Discrete Cosine Transform (IDCT) 109
6.3.9. MPEG-2 Spatial Decoding 110
6.4. Results Summary 112
6.4.1. Replication Scalability 113
6.4.2. Comparison of FPGA Brook to C2H and Soft Processor 114
7 Discussion 117
7.1. Ease of Use 117
7.2. Compilation Time 118
7.3. Debugging 118
7.4. Streaming Limitations and Heterogeneous System Development 120

7.5. Comparing FPGA Brook and C2H 122
7.6. Comparing FPGA Brook and Manual IDCT Implementation 123
7.7. Design Flow Limitations and Optimization Opportunities 125
8 Conclusions and Future Work 129
8.1. Contributions 130
8.2. Future Work 131
A Results of Experimental Evaluation 133
Bibliography 139

List of Tables

Table 3.1 Feature comparisons between FPGA Brook and GPU Brook....31
Table A.1 Performance and area results for the autocorrelation application....134
Table A.2 Performance and area results for the 8-tap FIR-filter application....134
Table A.3 Performance and area results for the 64-tap FIR-filter application....134
Table A.4 Performance and area results for the two-dimensional convolution application....135
Table A.5 Performance and area results for the matrix-vector multiplication application....135
Table A.6 Performance and area results for the matrix-matrix multiplication application....135
Table A.7 Performance and area results for the 16-bit Mandelbrot application....136
Table A.8 Performance and area results for the 32-bit Mandelbrot application....136
Table A.9 Performance and area results for the zigzag ordering application....136
Table A.10 Performance and area results for the inverse quantization application....137
Table A.11 Performance and area results for the mismatch control application....137
Table A.12 Performance and area results for the saturation application....137
Table A.13 Performance and area results for the IDCT application....138
Table A.14 Performance and area results for the MPEG-2 application....138

List of Figures

Figure 2.1 Streaming application represented as a directed graph 5
Figure 2.2 Different types of reductions in Brook [16] 7
Figure 2.3 Example system generated by the C2H compiler [27]....13
Figure 3.1 Graphical illustration of zigzag ordering 25
Figure 4.1 FPGA Brook design flow....34
Figure 4.2 Dataflow graph for the autocorrelation application 38
Figure 4.3 Example of two-dimensional reduction 39
Figure 4.4 Input and output ports for ordinary and array FIFO....43
Figure 4.5 Block diagram of an array FIFO 44
Figure 5.1 Replicated streaming dataflow graph for the autocorrelation application 49
Figure 5.2 Two possible implementations of the 1-to-4 distributor FIFO 52
Figure 5.3 Block diagram of an array distributor FIFO 54
Figure 5.4 Examples of a reduction operation and corresponding reduction trees....57
Figure 5.5 An example of two reduction trees operating in parallel 60
Figure 5.6 Datapath for a double-distributor FIFO 62
Figure 5.7 Keeping track of the number of inputs processed by leaf kernels for two different reduction replication scenarios....65
Figure 5.8 Round-robin distributions of a 2-D stream to be reduced....68
Figure 5.9 Decomposing a reduction into two phases....69
Figure 5.10 Distribution of modulo values in a two-dimensional stream 80
Figure 6.1 Depiction of a 4-tap FIR filter and its mapping to FPGA Brook....85
Figure 6.2 Applying the window function to a 3x4 pixel image....86
Figure 6.3 MPEG-2 spatial decoding dataflow graph [59]....94
Figure 6.4 Relative throughput and area of the autocorrelation application 98
Figure 6.5 Relative throughput and area of the FIR filter application with 8 taps 100
Figure 6.6 Relative throughput and area of the FIR filter application with 64 taps 100
Figure 6.7 Relative throughput and area of the two-dimensional convolution application....102
Figure 6.8 Relative throughput and area of the matrix-vector multiplication application 103
Figure 6.9 Relative throughput and area of the matrix-matrix multiplication application....103
Figure 6.10 Relative throughput and area of the Mandelbrot application with 16-bit data....104
Figure 6.11 Relative throughput and area of the Mandelbrot application with 32-bit data....105
Figure 6.12 Relative throughput and area of the zigzag ordering application....106
Figure 6.13 Relative throughput and area of the inverse quantization application 106

Figure 6.14 Relative throughput and area of the mismatch control application 107
Figure 6.15 Relative throughput and area of the saturation application 108
Figure 6.16 Relative throughput and area of the IDCT application 109
Figure 6.17 Wall-clock throughput of different replicated configurations of MPEG-2 spatial decoding implemented on the DE2 development board (KB/s) 111
Figure 6.18 Wall-clock speedup of replicated FPGA Brook implementations 114
Figure 6.19 Results of C2H implementations relative to the best FPGA Brook implementation 115
Figure 6.20 Results of soft processor implementations relative to the best FPGA Brook implementation 115

Chapter 1

Introduction

Field Programmable Gate Arrays (FPGAs) are a type of *programmable logic devices (PLDs)* commonly used for implementation of digital logic circuits in prototyping and small to medium production environments. They contain programmable logic blocks and programmable interconnect, which makes them suitable for implementation of arbitrary digital circuits. In the past decade FPGAs have grown in size and complexity to the point where a complete computer system can be implemented on one chip, known as a *system on a programmable chip* (SOPC). Such a system usually contains one or more processing nodes, either soft- or hard-core, and a number of peripherals. Having the whole system on one chip reduces manufacturing costs and simplifies the design process. As the complexity of SOPCs grows, tools used to build these systems need to allow rapid development to meet the increasing time-to-market demands. Such tools need to support system design at a high level, abstracting away low-level details of these chips and thus reducing design time.

There are three broad classes of design methodologies for FPGAs. At the lowest level, the design can be done using one of the *Hardware Design Languages (HDLs)*, which can provide best performance and low area and power consumption, but is error-prone and time-consuming. On the other end are *soft processors*, which are microprocessors implemented using the general-purpose programmable logic inside the FPGA. Soft processors allow the design to be implemented in software and thus simplify the design process, but may result in poor performance or high area or power consumption. Finally, the third option includes methodologies that allow the system to be specified at a higher level than HDLs, but use custom logic specifically generated for the target application. Using this approach simplifies the design process compared to using HDLs, although usually not as much as a soft-processor based methodology, because the designer has to use special directives or languages to aid the compiler. The compiler takes advantage of these directives to achieve better results than the soft processor. For example, *behavioural synthesis compilers* [1] analyze programs written in a high-level sequential language, such as C, and attempt to extract instruction-level parallelism by analyzing dependencies among program statements, and mapping independent operations to parallel hardware units. Several such compilers have been released, including *C2H* [2] from Altera, which is fully integrated into their SOPC design flow.

The main problem with the behavioural synthesis approach is that the amount of instruction-level parallelism in a typical software program is limited. To compensate for that, behavioural compilers often augment the source language with special directives. In the case of C2H, programmers often have to restructure their code and may have to explicitly manage hardware resources, such as mapping of data to

memory modules, to achieve good results. Another approach is to select an existing parallel programming model as a source language and map a program written in it onto hardware [3]. This approach allows programmers to express parallelism, but they may have to deal with issues such as synchronization, deadlocks and starvation. Ideally, the source language should allow programmers to express parallelism without having to worry about such issues. One class of languages that is attracting a lot of attention lately is based on the streaming paradigm. The terms *streaming* and *stream processing* have been used for many years to describe many different systems [4]. Generally, a stream processing system consists of a number of individual processing modules that process data in parallel, and communication channels that connect the processing modules. Terminology used in describing these systems varies in the literature. We adopt the terminology used by Buck [5]; data is organized into *streams*, which are collections of data, similar to arrays, but with elements that are guaranteed to be mutually independent. If the application data does not natively consist of independent records, it is the programmer's responsibility to transform and organize data into a collection of independent elements by using the operators provided in the streaming language used. Computation on the streams is performed by *kernels*, which are functions that implicitly operate on all elements of their input streams. Since the stream elements are independent, a kernel can operate on individual stream elements in parallel, thus exploiting data-level parallelism. If there are multiple kernels in a program, they can operate in parallel in a pipelined fashion, thus exploiting task-level parallelism. Streaming is suitable for implementation of many types of applications, including signal processing and networking [6].

In this thesis we propose implementing applications expressed in a streaming language in FPGAs. The main goal of the thesis is to investigate whether applications expressed in a streaming language can be automatically implemented in an FPGA and achieve better performance than other automated design flows that take high-level application description as their input. To this end, we built a design flow that automatically converts kernels into hardware blocks that operate on incoming streams of data. We base our work on *GPU Brook* [7], which is a version of the Brook streaming language [5] targeting graphics processing units (GPUs). The GPU Brook compiler is open source and supported by a community forum [8], which allowed us to modify it to target FPGAs instead of GPUs. Brook and GPU Brook were developed at Stanford University and have been used in a number of research projects at Stanford [9-11] and elsewhere [12,13].

We show that a program expressed in Brook can be automatically converted into parallel hardware, which, when implemented in FPGA logic performs significantly faster than other similar approaches. Our design flow includes a source-to-source compiler we built, which converts a streaming program into C code with C2H directives. This code is then passed to Altera's C2H tool [2], which implements the application in FPGA logic. To implement a streaming program in an FPGA, we map each kernel to a custom hardware processing node and communicate streams between nodes through FIFO buffers. Data

parallelism can be exploited by creating several instances of each kernel, each processing a part of the input stream. We call this process *kernel replication*, which is possible because stream elements are independent, so they can be processed in parallel. Our compiler can automatically increase throughput of a given streaming program by a desired factor. We give theoretical limits to speedup through replication for different types of kernels. Using a set of benchmarks we demonstrate the effectiveness of our methodology through experimental evaluation. The experimental evaluation has three goals. First, we demonstrate that the developed design flow can correctly implement realistic applications. Second, we compare throughput and area of the designs produced by our design flow to alternative design flows for FPGAs. Finally, we explore how kernel performance scales with replication.

The main objectives of this thesis are to investigate the suitability of streaming for automated FPGA implementations, demonstrate its feasibility through a sample implementation and propose guidelines for future implementations. More specifically, we investigate the following:

- Whether the Brook streaming language is suitable for expressing applications commonly implemented in FPGAs and whether it improves programmer efficiency. While we strive to adhere to the original Brook and GPU Brook specifications as much as possible, we also propose enhancements to the language (where appropriate) to make it easier to express the applications in a way that results in efficient FPGA implementations or to better expose parallelism. We call this enhanced version of Brook the *FPGA Brook*.
- Whether the parallelism exposed by expressing the application in a streaming language can be effectively exploited by automated tools to provide performance that scales when more resources are used to implement the application. Our primary goal is improving the streaming application throughput at the expense of area increase.
- How different structures of a streaming program, such as kernels and streams, can be mapped to FPGA logic in a way that maximizes performance, subject to limitations imposed by contemporary FPGA architectures.
- Compare Brook and its features to other streaming languages and determine the possible benefits of introducing some new features into the language. Since it is not obvious which streaming language, if any, will become popular in the programmer community, our goal is to uncover the language features critical for successful implementation of streaming programs in FPGAs. We hope that our findings will be useful either directly for a Brook-like language definition, or indirectly in the development of future languages and systems.
- Since one of the long-term goals of this project is to make it easier to design systems that target FPGAs, we also discuss the ease of use of our design flow, which has a direct impact on designer productivity.

- Discuss the challenges we encountered with our implementations, present limitations of our approach and propose directions for further research and improvements.

The main research contribution of this thesis is that it demonstrates that it is possible to automatically implement applications expressed in a high-level streaming language in an FPGA so that they achieve significantly better performance than alternative approaches not utilizing streaming. In the process, we developed a novel technique for parallel implementation of complex reduction operations in FPGAs. We also experimentally evaluated performance of twelve benchmark applications implemented using our design flow, analyzed their performance and area requirements, and discussed opportunities for further performance improvement and area reduction. A detailed list of all contributions of this thesis can be found in section 8.1.

1.1. Thesis Organization

This thesis is organized as follows. Chapter 2 provides the necessary background on stream computing, FPGA technology and related work in these areas. Chapter 3 gives an overview of the Brook language with a focus on new features introduced in FPGA Brook. In Chapter 4 we present the design flow used to map the Brook streaming programs to FPGA logic. We describe how different structures in the language map to hardware and how our compiler interacts with Altera's tools to provide seamless, software compiler-like interface to the designer. Chapter 5 explores how data parallelism is exploited through kernel replication in our infrastructure. We also derive theoretical limits to throughput achievable through replication. In Chapter 6 we demonstrate the effectiveness of the proposed design flow through experimental evaluation. We describe the challenges and limitations of our approach, along with the ideas for overcoming these limitations in Chapter 7. Chapter 8 concludes the thesis with some final observations and ideas for future research directions.

Chapter 2

Background and Related Work

The goal of this work is to explore the suitability of implementing applications expressed in a streaming language in FPGAs. To achieve this, we use a design flow consisting of the source-to-source compiler we developed and the commercial C2H behavioural compiler [2]. In this chapter we present the background information on the components of our design flow and discuss related work in streaming and design methodologies targeting FPGAs.

2.1. Stream Computing

The term stream computing (a.k.a. stream processing, or simply streaming) has been used to describe a variety of systems. Examples of stream computing include dataflow systems, reactive systems, synchronous concurrent algorithms, and signal processing systems [4]. Streaming applications we focus on are described through a set of kernels, which define the computation, and a set of data streams containing independent elements, which define communication [5]. This organization allows the compiler to easily analyze communication patterns in the application, so that parallelism can be exploited. When a programmer specifies that certain data belongs to a stream, this provides a guarantee that the elements of the stream are independent from one another. The computation specified by the kernel can then be applied to stream elements in any order. In fact, all stream elements can be processed in parallel, limited only by the available computing resources. A streaming application can be represented as a directed graph, as shown in Figure 2.1. Kernels are represented as graph nodes, while edges represent streams. In a general case a streaming application can have more than one input and output stream.

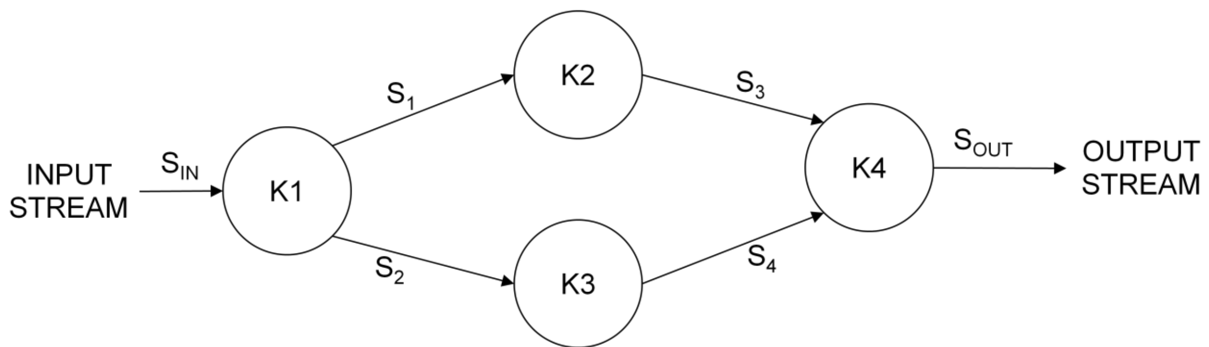


Figure 2.1 Streaming application represented as a directed graph

Streaming applications as described above expose two types of parallelism; task and data parallelism. Task parallelism can be exploited by implementing each kernel as an independent processing unit and scheduling computation in a pipelined fashion. Data parallelism can be exploited at the kernel level because stream elements are independent. In streaming the amount of data reuse is typically low. Kernels usually read input stream, process it and produce resulting output stream. The data is usually read only once, and never reused again. This is in contrast to, for example, numeric applications, which iterate over the same data set multiple times.

At the kernel level, computation is based on the Single Instruction Multiple Data (SIMD) paradigm, and is similar to vector processing [14]. The major difference between streaming kernels and vector processing is in computation granularity. While vector processing involves only simple operations on (typically) two operands, a kernel may take an arbitrary number of input streams, and produce one or more output streams. In addition, the kernel computation can be arbitrarily complex, and the execution time may vary significantly from one element to the next. Finally, elements of a stream can be complex (e.g. custom data structures), compared to vector processing, which can only operate on primitive data types. Another important difference between stream computing and vector computing is how temporary results are handled. In vector computing all intermediate results are stored in a central vector register file or memory, which is also centralized. In streaming, kernels store their temporary results locally, thus reducing the amount of global communication [15].

It is important to note that some streaming languages do not expose data parallelism in the same manner as described here. As an example, the StreamIt streaming language allows kernels to introduce dependencies between stream elements. We discuss this further in section 2.5.

2.2. Brook and GPU Brook

The recent interest in stream processing has been driven by two trends: the emergence of multi-core architectures and general-purpose computation on graphics processing units. Multi-core architectures have sparked interest in novel programming paradigms that allow easy programming of such systems. Stream processing is a paradigm that is suitable for both building and programming these systems. For example, the Merrimac supercomputer [9] consists of many stream processors, each containing a number of floating-point units and a hierarchy of register files. A program expressed in a stream programming language is mapped to this architecture in a way that captures the data locality through the register file hierarchy. Programs for Merrimac are written in the Brook streaming language [5].

Brook extends the C programming language syntax to include streams and kernels. Streams are declared similarly to arrays, except that characters “<” and “>” are used instead of square brackets.

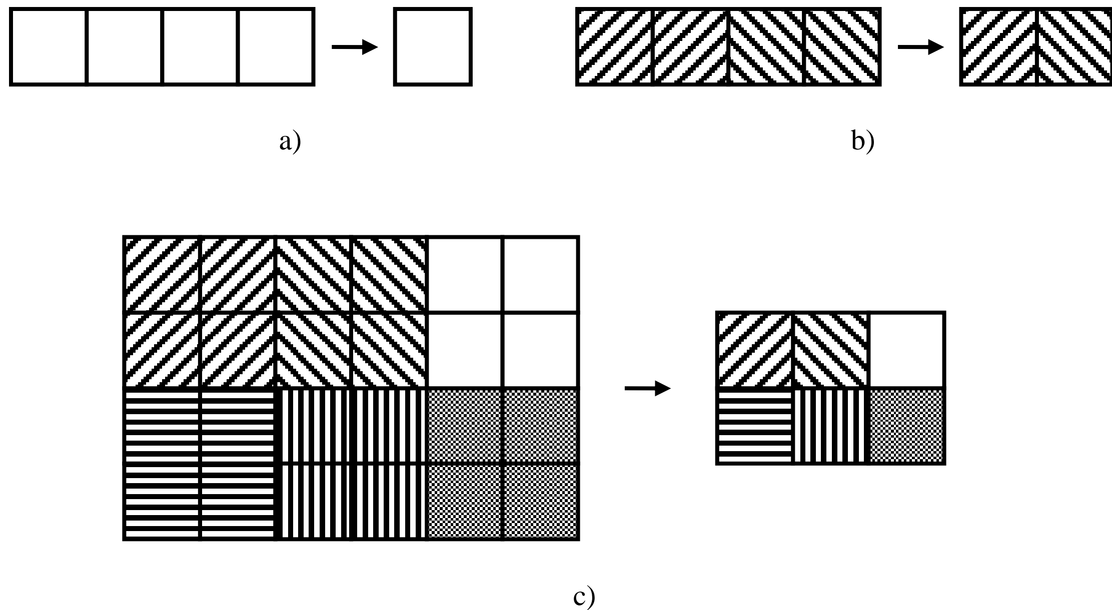


Figure 2.2 Different types of reductions in Brook [16]

Kernels are denoted using the *kernel* keyword. Kernels can operate on stream elements in parallel, but kernel code refers to streams, not stream elements. It is assumed that the operation is to be performed over all stream elements. This limitation prevents programmers from introducing data dependencies between stream elements.

A special kind of kernel, *reduction function*, can combine several elements of the input stream to produce one element of the output stream. These kernels are used to perform reduction operations, and are denoted by the *reduce* keyword. Reduction operations have to be commutative and associative so they can be performed in parallel [5]. Figure 2.2 [16] depicts three different ways reduction can be performed in Brook. In Figure 2.2a a one-dimensional (1-D) stream with 4 elements is reduced to a one-element stream. Figure 2.2b shows a 1-D four-element stream reduced to a two-element stream, where the neighbouring elements of the input are combined to produce the elements of the output. Similarly, a 2-D stream can be reduced to another 2-D stream by combining the neighbouring elements, as shown in Figure 2.2c. In the figure, the elements that get combined together are shaded with the same pattern. While reductions are most commonly used to produce operations such as summation, or finding the minimum or maximum element, Brook does not limit reductions to these operations; programmers can define complex custom reduction operations.

Another class of operations are *stream operators*, which do not involve any computation, but instead perform stream transformations. An example of a stream operator is *streamRepeat,* which creates an

output stream by repeating the elements of the input stream multiple times. Another example is the stencil operator, which selects a stream element and several of its neighbours from the input stream to create the output stream. The stencil operator is useful for the implementation of window functions commonly used in digital signal processing [17]. Other operators allow replication of stream elements, and splitting and merging of streams. Stream operators allow stream data to be reorganized to facilitate common operations on streams that cannot be performed by ordinary kernels or reductions. Throughout this thesis we use the term *ordinary kernel* to denote a kernel that is not a reduction and not a stream operator.

While Brook is based on the C programming language, it limits the usage of many features of the C language, particularly inside kernels. Kernels are stateless, meaning that values of local variables within kernels are not preserved between processing of one stream element to the next. As a result, static variable declarations are not allowed. Any non-stream parameters passed to the kernels are read-only and kernels are not allowed to access global variables. Pointers are only allowed as arguments to kernels for passing values by reference. Finally, kernel recursion is not supported. These limitations make it easier for the compiler to analyze programs and extract parallelism.

GPU Brook is a variant of the Brook streaming language specifically targeting GPUs [7]. The processing power of GPUs has led to their use for general-purpose computing. GPUs are suitable for stream processing because they typically have a large number of 4-way vector execution units, meaning that four ALUs perform computation on four independent data elements in parallel. GPUs also include a relatively large memory and utilize multithreading to hide memory latency [15]. Streaming languages can be used to program these systems, because kernel computation can be easily mapped to the processing units in a GPU. The GPU Brook compiler hides many details of the underlying GPU hardware from the programmer, thus making programming easier. Our work is based on the GPU Brook compiler, because it is open source, readily available for download [18] and supported by a community forum [8].

GPU Brook is based on the original Brook specification with many features oriented towards GPUs. Computation is still organized around kernels and streams, but GPU Brook supports only three stream operators: *streamRead*, *streamWrite*, and *stream domain*. It is assumed that the input data for the program is placed in the computer's main memory and that the results should be placed into that same memory. The *streamRead* and *streamWrite* operators are used to transfer data between the main memory and the graphics memory, which is used to store streams. The *stream domain* operator allows selecting a contiguous subset of the input stream. In addition to these operators, data can be brought from the main memory to create a stream using a *gather* operation, which allows arbitrary indexing into memory to fetch stream elements. A *gather stream* contains indices of array elements that should be fetched from the memory. *Scatter* operation works in much the same way, except that data is written to the memory array at indices specified by the *scatter stream*. This is in contrast to *streamRead* and *streamWrite*, which only support copying complete arrays to and from streams. All computation in GPU Brook is performed on

floating-point data; the integer type data is currently not supported [15]. We illustrate the features of GPU Brook by using the matrix-vector multiplication application.

Matrix-vector multiplication is a simple application that multiplies an NxN matrix and an N-element vector to produce an N-element vector. For example, a 3x3 matrix and a three-element vector are multiplied as follows:

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} * \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} a_{00} * x_0 + a_{01} * x_1 + a_{02} * x_2 \\ a_{10} * x_0 + a_{11} * x_1 + a_{12} * x_2 \\ a_{20} * x_0 + a_{21} * x_1 + a_{22} * x_2 \end{bmatrix}$$

Matrix-vector multiplication can be decomposed into two operations: element-by-element multiplication, and summation. To express this application in Brook, we have to place the matrix and the vector into streams. Since most kernels expect that all input streams are of the same size, we have to resolve the difference between the matrix (*NxN*) and vector size (*Nx1*). This can be done by replicating the vector *N* times to obtain an *NxN* matrix. For the three-element vector above, this results in:

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} \rightarrow \begin{bmatrix} x_0 & x_1 & x_2 \\ x_0 & x_1 & x_2 \\ x_0 & x_1 & x_2 \end{bmatrix}$$

To understand why the vector was replicated across rows, instead of columns, consider the multiplication operation that has to be performed next. In Brook, multiplying two 2-D streams results in elements in the same positions in the stream being multiplied. Replicating the vector across the rows results in this stream being aligned with the stream representing the matrix *A* in a way that will produce the multiplication result required to implement the matrix-vector multiplication. Finally, the summation part of the matrix-vector multiplication is implemented as a reduction of the *NxN* stream to the *Nx1* stream. As a result, the products in each row will be added together, as required by the application.

Listing 2.1 Matrix-vector multiplication implemented in GPU Brook

```
kernel void mul (float a<>, float b<>, out float c<>) {
   c = a * b;
}

reduce void sum (float a<>, reduce float r<>) {
   r += a;
}

void main () {
   float Astr<N,N>, xstr<1,N>, Tstr<N,N>, ystr<N,1>;
   float A[N][N], x[1][N], y[N][1];

   streamRead (Astr, A);
   streamRead (xstr, x);
   mul (Astr, xstr, Tstr);
   sum (Tstr, ystr);
   streamWrite (ystr, y);
}
```

A simple GPU Brook program implementing the matrix-vector multiplication is shown in Listing 2.1 [7]. The code contains two kernels (*mul* and *sum*) and the main function. Several two-dimensional (2-D) streams are declared in the main program. The input streams are first created from arrays using the *streamRead* operators, then multiplied by the *mul* kernel, and then reduced by the *sum* kernel, and finally the result is written to the main memory by the *streamWrite* operator. Normally, the GPU Brook kernels expect all of their input streams to have the same dimensionality and number of elements. If that is not the case, one of the streams will be automatically expanded by repeating some elements. In our example, vector *x* has to be replicated *N* times to obtain an *NxN* stream, so that both input streams to the mul operation have the same number of elements. The vector is replicated across rows because there is a mismatch in the number of rows.

As previously noted, kernel code does not refer to stream elements; it is assumed that the operation is to be performed over all stream elements. Although kernel code cannot select a stream element it operates on, it is possible to determine the position (within the stream) of the element currently being processed. This is done through a specialized Brook operator called *indexof*, which is useful for applications such as convolution, where the coefficient a sample should be multiplied by depends on the sample's index. A simple convolution can be represented as [17]:

$$y(n) = \sum_{k=0}^{M-1} h(k)x(n-k)$$

A kernel expressing the multiplication part of this computation is shown in Listing 2.2. In the example, we assume that M=4 and that the rest of the computation, such as shifting of x and summation, is performed in other kernels and stream operators. The values of coefficients *h* were selected arbitrarily in this example. When the kernel is executed, *indexof(x)* will return the position of the current stream element within the stream. For example, when the kernel is processing *x<2>, indexof(x)* will return 2, while on the next element, *x<3>*, it will return 3. The compiler has to ensure that this is true even when the processing is performed in parallel.

Listing 2.2 Using *indexof* operator in GPU Brook

```
kernel void mul (float x<>, out float y<>) {
   float h[4] = {1.0f, 2.0f, -3.0f, -1.0f}

   y = h[indexof(x)] * x;  // Performs h(k)*x(k), x is shifted elsewhere
}
```

2.3. Field Programmable Gate Arrays

Since the goal of our work is to implement streaming programs in FPGAs, in this section we briefly describe the FPGA technology and typical design flow for FPGAs. Terminology used to describe FPGAs varies between manufacturers and even between different device families from the same manufacturer. In this thesis, we use the terminology used for the *Cyclone II* device family from Altera [19]. Cyclone II devices consist of programmable *logic elements (LEs)* surrounded by programmable interconnection wires, which can be used to build arbitrary logic circuits consisting of many logic elements. Each logic element contains a 4-input *lookup table (4-LUT)*, a register, and some additional logic for efficient implementation of carry logic in adders. A 4-LUT is a circuit element that can implement any Boolean logic function of up to 4 inputs. Since their functionality can be programmed, LEs are referred to as *soft logic*. Contemporary FPGAs, such as Cyclone II, also contain fixed blocks of logic, such as memories and multipliers, which are referred to as *hard logic*. Memories are an important component for storage of data in SOPC systems, while multipliers are useful for digital signal processing (DSP) applications, commonly implemented in FPGAs. Cyclone II also contains *phase-locked loops (PLLs)*, which are used to produce clock signals with different frequencies and phase, based on one or more external clocks [19]. Other FPGAs have a similar structure, with varying number and complexity of soft and hard logic blocks and interconnect. Some FPGAs also contain embedded hard processors in their fabric [20].

Design for FPGAs is commonly done in one of the HDLs, such as Verilog or VHDL. A digital circuit is first described in one of these languages and is then processed by *CAD tools* and converted into the programming file for the target FPGA device. This processing is variously referred to as design compilation or *synthesis*, and is performed in several steps [21]. First, the HDL description is converted into a *netlist* of basic logic gates, which is then optimized. The optimized netlist is then mapped into LEs in the step known as *technology mapping*. After this, the mapped LEs are assigned specific locations in the target FPGA in the step called *placement*. Algorithms used for placement use metaheuristics, because the number of possible placements is large. Finally, the *routing* algorithm determines how the placed LEs should be interconnected to implement the circuit specified by the initial description.

The final result is the FPGA programming file, which is used to configure the target FPGA device. Throughout the design flow, the algorithms attempt to optimize one or more circuit properties, including maximizing design speed or minimizing area or power. *Timing analysis* can be performed at the end to determine the timing properties of the resulting circuit, such as *maximum operating frequency* (F_{max}). Since placement uses metaheuristic algorithms, the placement result is usually suboptimal. Furthermore, small changes in the initial placement can result in significant changes in the final result. Synthesis tools usually allow the user to specify a *seed*, which is a number that affects the initial placement. *Seed*

sweeping is a process of compiling the design using multiple seeds. It is usually performed to obtain small improvements in circuit timing or to get a more realistic measure of achievable timing performance [22].

2.4. Nios II and C2H

Design for FPGAs has traditionally been done in HDLs. However, growing complexity of these devices has led to other design options being explored. While on one hand the complexity of FPGAs makes design challenging, on the other hand, their increasing capabilities allow implementation of complex logic, which can be exploited to simplify the design methodology. Soft processors take advantage of this trend by using ample programmable resources available in contemporary FPGAs to present the system designer with a familiar interface of a microprocessor. Parts of a design that are not performance critical can then be implemented in software, while critical portions can still be implemented in an HDL. Major FPGA vendors provide soft processors optimized for implementation in their respective FPGAs [23,24]. *Nios II* [23] is Altera's soft processor with many configurable parameters, which can be used to build an SOPC system using the provided peripherals and other IP cores.

In the past, *reconfigurable computing systems* [25] used FPGAs as co-processors connected as peripherals to general-purpose processors. The FPGA was usually used to accelerate computation-intensive parts of the application, while the rest of the application usually executed on the general-purpose processor. With the emergence of soft processors, integration of the microprocessor and the peripheral has become even easier, because both can reside on the same FPGA chip. Furthermore, advancements in behavioural synthesis tools [1] have allowed the development of tools for automatic generation of reconfigurable systems. Such tools are commonly referred to as *high level synthesis* tools [26]. One such tool is Altera's C2H tool [2].

C2H is a high-level synthesis tool that integrates into Altera's SOPC Builder infrastructure for building soft-processor based systems. C2H allows system developers to select some functions in their C code that are suitable for hardware implementation. Altera documentation refers to these functions as accelerated functions, and to the generated hardware blocks as *hardware accelerators*. An example of a system with one accelerated function is shown in Figure 2.3 [27]. As shown in the figure, the hardware accelerator is as a coprocessor in the system, connected to the main processor through the Avalon switch fabric, which is Altera's interconnection network for SOPC systems [28]. Nios II controls the accelerator by writing input parameters to the registers in the accelerator, issuing the start command, and reading the return value (if any) from the output register in the accelerator.

C2H maps C code of accelerated functions into hardware using simple mapping rules. Mathematical operations are mapped into their equivalent combinational circuits, while assignments to variables are

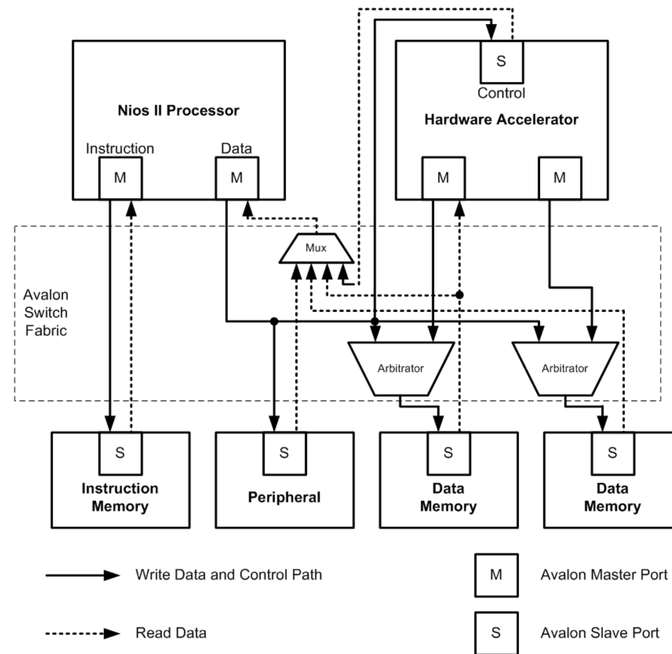


Figure 2.3 Example system generated by the C2H compiler [27]

usually mapped into registers holding outputs from the combinational circuits. Loops are mapped into state machines controlling the operation of statements inside the loop. The C2H compiler exploits instruction-level parallelism by mapping independent program statements to parallel hardware units. Logic sharing is only performed for operations that "consume significant amounts of logic" [27], such as multiplication, division or memory access. The main optimization goal is performance, so logic is shared between two operations only if sharing does not affect performance. C2H does not perform any automatic transformations that exploit data parallelism, such as loop unrolling or vectorization.

Depending on the functionality, the accelerator can have one or more master ports for accessing other (memory) modules in the system. The master ports are generated when the code uses pointers or array references. By default, the C2H compiler automatically connects all master ports on hardware accelerators to all memories in the system the main processor connects to. The C2H compiler also supports an optional *connection pragma* directive that can explicitly specify the memory modules the accelerator should connect to. If such pragmas are used, the master port denoted by the pragma directive will be connected only to the ports specified by the directive, which saves logic resources and thus reduces the area. Listing 2.3 illustrates the usage of the connection pragma. Pointers *in_ptr* and *out_ptr* of the accelerated function *dma* are specified to connect to memories *mem_1* and *mem_2*, respectively. In addition, if the memory has multiple ports, the pragma statement can also specify the memory port to connect to.

Listing 2.3 also demonstrates the use of the *interrupt pragma* directive. By default, the Nios II processor waits for the hardware accelerator to complete processing before continuing with the rest of the

Listing 2.3 Using connect pragma statements in C2H

```
#pragma altera_accelerate connect_variable dma/in_ptr to mem_1/read_port
#pragma altera_accelerate connect_variable dma/out_ptr to mem_2/write_port

#pragma altera_accelerate enable_interrupt_for_function dma

int dma (int *in_ptr, int *out_ptr, int start, int length) {
  int index;

  for (index=start; index<(start+length); index++) {
    out_ptr[index] = in_ptr[index];
  }
}
```

program. This behaviour can be changed by using the interrupt pragma, which specifies that the program can continue without waiting for the accelerator to finish. In such a case, the accelerator can produce an interrupt once the processing is complete, or the calling function can check whether the accelerator is finished by polling a special status register within the accelerator. This feature is useful when multiple hardware accelerators need to run in parallel. C2H also supports other pragma directives, which can be used to further optimize the results [27].

C2H currently has some limitations when implementing C functions in hardware [27]:

- C2H does not support floating point arithmetic.
- Recursive function calls are not supported in accelerated functions.
- The unary *&* operator used as an address operator is not supported.
- Short-circuit evaluation is not supported; all expressions in complex logical operations are fully evaluated.
- *struct* or *union* arguments cannot be passed to an accelerated function by value.
- Function pointers are supported only if used to point to functions that exist inside the hardware accelerator.
- The *goto* keyword is not supported.
- Loop unrolling or vectorization is not supported.
- Initialized arrays are stored in a global memory accessible by the Nios II processor.

Additionally, individual expressions within the accelerated function are often directly mapped to hardware without any optimization. As an example, a long expression that specifies an addition of six variables will result in six adders being produced, without any registers between them, which may lead to a long critical path, thus negatively affecting F_{max}. This means that the performance, area and power characteristics depend on the coding style of the source function. Altera provides guidelines on coding styles that provide good performance, which should be followed for optimal results [29]. The C2H

compiler also has other limitations that are less important for this work. An exhaustive list can be found in C2H documentation [27].

In addition to C2H, there are numerous other commercially available high-level synthesis tools. Examples include Catapult C from Mentor Graphics [26], Impulse C from Impulse Accelerated Technologies [30], Cynthesizer from Forte Design Systems [31], Handel-C from Agility (now Mentor Graphics) [32], and PICO from Synfora [33]. Each of these tools supports some unique features and provides some benefits relative to others. For example, Impulse C supports loop unrolling, which takes advantage of parallelism available in the application. However, the programmer has to ensure that there are no dependencies among the statements inside the loop before specifying that a loop should be unrolled. Impulse C also supports FIFO-based communication channels, referred to as streams [34]. However, it is important to note that Impulse C streams are not collections of independent data, as is the case in Brook and GPU Brook. We use C2H in our design flow because of its close integration with the rest of Altera's flow. We should also note that many of the other tools were not available at the time our work was started.

2.5. Related Work

In recent years, stream computing has been applied in many different contexts. Gummaraju and Rosenblum [10] explored compilation of streaming programs for ordinary, non-streaming microprocessors. They showed that kernel computation can be effectively mapped to a hyperthreaded CPU, while streams map well to processor caches. The performance obtained using this methodology is comparable to the performance of the same application programmed in a traditional programming language. This finding is significant because it demonstrates portability of programs written in a streaming language.

Liao et al. [12] investigated mapping of Brook streaming programs to uniprocessor and multiprocessor Intel Xeon-based machines using a parallelizing compiler with aggressive data and computation transformations. The data transformations focus on the stream operators. Since the set of stream operators supported in the Brook language is limited, several operators may be necessary to achieve a desired operation. Complex transformations may thus require several temporary streams, which require large amounts of memory for temporary storage. Instead, a set of optimizations can be performed to combine operators and reduce the number of temporary streams needed. Liao et al. also investigate computation transformations. While kernel code is implicitly parallel because of data parallelism exposed by streams, there are also opportunities for inter-kernel optimizations. By modelling kernels as implicit loop nests over all input stream elements, various compiler transformations, such as loop fusion or fission, can be performed. Care must be taken to preserve dependencies introduced by inter-kernel optimizations. These

transformations resulted in a significant performance improvement for both uniprocessor and multiprocessor machines [12].

Several research projects have investigated stream processing on FPGAs. Neuendorffer and Vissers [6] provide a good overview of the suitability of streaming languages for FPGA implementations. They also provide an overview of high-level issues with such implementations, including I/O integration and memory interfaces. Howes et al. [35] compared performance of GPUs, PlayStation 2 (PS2) vector processors, and FPGAs used as coprocessors. The applications were written in ASC (A Stream Compiler) for FPGAs, which has been extended to target GPUs and PS2. ASC is a C++ library that can be used to target hardware. The authors found that for most applications GPUs outperformed both FPGAs and PS2 vector units. However, they used the FPGA as a coprocessor card attached to a PC, which resulted in a large communication overhead between the host processor and the FPGA. They showed that removing this overhead improves the performance of the FPGA significantly. Their approach does not clearly define kernels and streams of data, so the burden is on the programmer to explicitly define which parts of the application will be implemented in hardware.

Bellas et al. [36] developed a system based on "streaming accelerators". The computation in an application is expressed as a streaming data flow graph (sDFG) and data streams are specified through stream descriptors. The sDFG and stream descriptors are then mapped to customizable units performing computation and communication. The disadvantage of this approach is that the framework does not support a high-level language for easy application development. Instead, users have to manually describe the application in the form of the sDFG. Another work based on sDFGs and stream descriptors explores an optimal mapping of a streaming application to hardware under an area constraint [37].

Mukherjee et al. [38] proposed a design flow which converts streaming applications expressed in C into hardware IP blocks interconnected in a pipelined fashion. The C programs are augmented with directives that allow the programmer to specify how an application maps to hardware. While this approach simplifies the design, the programmer still has to have an understanding of the underlying hardware.

StreamIt [39] is an architecture-independent programming language for streaming applications. The language primarily targets multi-core architectures that expose communication to software, such as Raw [40]. StreamIt programs are composed of filters, which are similar to kernels in Brook, but filters are limited to only one input and one output stream. StreamIt filters can be either stateful or stateless, but can access only locally declared variables in either case. Conceptually, StreamIt filters process infinite streams, while Brook kernels process finite streams [41]. While Brook does not specify the nature of memory buffers holding streams, StreamIt assumes that streams are passed between filters through FIFO buffers. An example of a StreamIt filter implementing convolution is shown in Listing 2.4. This code is

Listing 2.4 Sample StreamIt code

```
float->float filter FIR() {
  float[4] weights = {1.0f, 2.0f, -3.0f, -1.0f};

  work push 1 pop 1 peek 4 {
    float sum = 0;
    for (int i=0; i<4; i++) {
      sum += weights[i] * peek(i);
    }
    push(sum);
    pop();
  }
}
```

equivalent to the Brook code in Listing 2.2, except that Brook code requires samples to be shifted elsewhere in the program. StreamIt code uses operators *push*, *pop* and *peek* to explicitly access streams. The *pop* operator consumes an element from the input stream FIFO, while the *peek* operator reads an element from the input stream FIFO without consuming the element. *Peek* takes a parameter that determines the position of the element to be read from the buffer. *Push* operator writes a stream element to the output stream FIFO. The example in Listing 2.2 highlights an important difference between StreamIt filters and Brook kernels. While Brook kernels are explicitly prohibited from accessing individual stream elements because they are independent, StreamIt allows filters to address a specific element using the *peek* operator. As a result, the StreamIt filter code can introduce dependencies between the input stream elements, so the input stream elements do not necessarily have to be independent.

Filters can be composed into larger structures using three topologies: *pipeline*, *splitjoin*, or a *feedback loop*. Pipeline and splitjoin are used to express pipeline and task parallelism, respectively. Limitations on the number of filter inputs and outputs, and the choice of only three basic topologies simplify the compiler analysis and make it easier to exploit parallelism. It is important to note that in StreamIt terminology different types of parallelism are classified using different terms than in this thesis. What we call task parallelism is in StreamIt referred to as pipeline parallelism. StreamIt also uses the term task parallelism to denote the kind of parallelism that we normally refer to as data parallelism. We use the term data parallelism because stream elements in Brook are always independent, so their processing can always be performed in parallel. In StreamIt, filters can introduce dependencies between stream elements, so the programmer has to explicitly use the splitjoin structure to designate when multiple tasks can be used to perform processing on the input stream in parallel. Therefore, StreamIt refers to this kind of parallelism as task parallelism. Our approach to exploiting data parallelism in Brook programs is similar to the splitjoin structure in the StreamIt language. However, Brook reduction kernels, which do not exist in StreamIt, have to be handled differently.

Our project is most closely related to the Optimus project [42]. Optimus is a compiler for implementation of streaming programs written in the StreamIt language in FPGAs. It implements a complete design flow, from the StreamIt language to Verilog HDL, and includes several types of optimizations, including instruction level parallelism (ILP) optimizations and optimizations based on profiling data [42]. The focus of the Optimus project is on improving communication latency and reducing memory storage requirements, while our goal is throughput optimization. Furthermore, StreamIt and Brook each offer some unique challenges for FPGA implementations because of the differences between the two languages as discussed above.

Hormati et al. [42] experimentally measured performance and energy consumption of the circuits produced by the Optimus compiler and compared them to the results achieved by the embedded PowerPC 405 processor using a set of eight benchmark programs. They also implemented two of the programs using the Handel-C behavioural compiler, and performance and area of resulting circuits were compared to those of the circuits generated by the Optimus compiler. A direct comparison between our experimental results and the Optimus results is not possible because the absolute results of their work have not been published. In addition, their benchmark set has only two programs in common with ours. Qualitatively, their experiments indicate that the Optimus compiler produces circuits that perform significantly faster and use significantly more area than the circuits produced by the Handel-C compiler [42]. This is similar to the trend we observed when comparing our results to the C2H compiler. Their results also show that the circuits generated by the Optimus compiler significantly outperform the embedded PowerPC processor, which is similar to our results relative to the Nios II soft processor. None of these results are directly comparable because of the different behavioural synthesis compilers and processors used for comparison.

Several other parallel languages have been proposed in recent years. Accelerator is a .NET library for general-purpose programming of GPUs [43]. The library provides an abstract data type called data-parallel arrays, which are similar to streams. Most operations on data-parallel arrays apply to all of the array elements. The compiler then translates the library operations to sequences of GPU commands. Another approach in programming GPUs using a high-level programming language is a shading language called Sh [44]. Sh is an API library for the C++ programming language, which adds support for GPU programming. Data is organized into streams and computation is organized into shaders, which are similar to kernels. Finally, C for CUDA [45] is a popular language for general-purpose programming of NVIDIA's GPUs. C for CUDA requires programming at a lower abstraction level than other languages discussed above, with explicit data buffer and synchronization management. We also note that ATI (now AMD), which is the other major graphics card vendor, has adopted a version of GPU Brook under the name Brook+ [13] for general-purpose processing on their graphics processors.

Other research projects have explored many different ways of simplifying the FPGA design flow and exploiting application parallelism. Parallelism can be exploited by building a multiprocessor system involving one or more FPGAs [46,47]. Shannon et al. [48] describe a system integration methodology that simplifies IP module reuse by allowing their integration using programmable interconnection elements containing FIFOs. Multithreaded soft processors [49,50] offer a good trade-off between performance and area for some application classes. Another option is implementing soft vector processors, which can take advantage of parallelism in loops, in the same manner as traditional vector computers [51,52]. Soft-processor resembling a GPU, which can be programmed in a high-level language, was described in [53]. Finally, automatic generation of *custom instructions* for soft processors can be performed to optimize performance [54]. A custom instruction is a logic block added to the processor's datapath that implements some operation specific for a given application, and is closely integrated with the processor's ALU [55]. In all of the approaches for simplifying FPGA designs, reconfigurability of FPGAs plays an important role by allowing the implementation to be specifically optimized for the target application.

In this chapter we presented the necessary background for the work presented in this thesis. In the next chapter we describe FPGA Brook, which is our modified version of GPU Brook.

Chapter 3

FPGA Brook

FPGA Brook is our adaptation of the Brook streaming language. It is similar to GPU Brook because our compiler is based on the GPU Brook compiler. Therefore, we strived to keep the language similar to GPU Brook, except where GPU Brook implemented features that were GPU-oriented or where it was appropriate to improve programmability of FPGAs. As an example, GPU Brook supports a *float4* data type, which is a vector of four floating-point numbers. This data type is easily implemented in a GPU because GPUs typically contain 4-way vector units. We do not support such a data type because it is not commonly used by FPGA designers. Instead, designers can use streams of arrays (see section 3.3) to implement equivalent functionality.

In this chapter we describe the FPGA Brook language, with particular emphasis on features that are different from those of Brook or GPU Brook. We present several examples of Brook programs to demonstrate how these features are used, and we discuss limitations of FPGA Brook. The focus of this chapter is on the programming language as it appears to the programmer. We defer the discussion of implementation details to subsequent chapters.

3.1. Structure of FPGA Brook Programs and Its Limitations

The basic structure of FPGA Brook programs is similar to that of GPU Brook programs. For example, the GPU Brook code in Listing 2.1 would look the same in FPGA Brook, except that the integer data type would be used instead of floating-point. Our current design flow does not support the floating-point data type, because we rely on C2H, which presently does not support floating-point operations [27]. It is important to note that the rest of our infrastructure supports the floating-point data type. Therefore, if C2H introduces floating-point support in a future version, the FPGA Brook design flow will support it as well.

As can be seen from the code in Listing 2.1, kernels appear to the programmer similar to ordinary C functions, except that they operate on streams. This similarity makes it easy for C programmers to adapt to the new language. Furthermore, FPGA system designers often start their design by creating a system block diagram. The advantage of using a streaming language, such as FPGA Brook, is that there is usually close resemblance between the system block diagram and the streaming code [56].

FPGA Brook places several limitations on the streams, kernels and the streaming dataflow graph structure. Unlike the original Brook specification, which does not limit the number of dimensions a stream can have, the current FPGA Brook compiler supports only one-dimensional (1-D) and two-

dimensional (2-D) streams, just like GPU Brook. Syntax for declaring and using 2-D streams is the same as in GPU Brook, as previously shown in Listing 2.1. All streams have to be declared with static sizes known at compile time. In the current version of our design flow it is assumed that each stream is written by exactly one kernel and read from by exactly one kernel. This assumption greatly simplifies dataflow analysis. While none of the programs we developed required a stream to be read or written by more than one kernel, such functionality can be achieved if necessary. If a stream needs to be read by two (or more) kernels, the kernel that produces this stream should be modified to produce two (or more) streams, one of each would then be forwarded to the appropriate kernel. If a stream needs to be written to by two kernels, the destination kernel should instead take two streams and programmatically (e.g. based on the value of a parameter passed to the kernel) choose the stream to be read. Future versions of our compiler should perform this transformation automatically.

FPGA Brook does not support cycles in the streaming dataflow graph. Cycles are not supported because stream communication is implemented through FIFO buffers. As a result, a cycle in the dataflow graph would require a FIFO buffer that can fit complete streams for correct implementation. Since the amount of memory in contemporary FPGAs is limited, this may not be feasible. Lack of support for cycles in the streaming dataflow graph is not a severe limitation because many applications that benefit from streaming do not require cycles in dataflow graphs. If an application cannot be implemented without a cycle in the dataflow graph, the programmer can perform data transfer through the main memory by explicitly writing the stream to the main memory at the source of the backwards edge in the stream graph, and explicitly reading it from the main memory at the destination of the backwards edge. While the compiler could detect backwards edges and implement this data transfer automatically, performing many memory transfers can negatively impact program performance. Therefore, we believe that exposing this operation to the programmer is beneficial to ensure that the programmer uses this option only when necessary. While we believe that this is currently the best way to handle cycles in dataflow graphs, this issue should be revisited if the amount of memory available in FPGAs increases significantly in the future. It is important to emphasize that the lack of support for cycles in the dataflow graph does not prevent usage of loops within the main function, as long as such a loop does not create a cycle in the dataflow graph. This is illustrated in the next section through the autocorrelation application.

3.2. Custom Stream Creation and Consumption

As discussed in section 2.2, GPU Brook includes the stream operators *streamRead* and *streamWrite* for transfer of data between memory and streams. If there is a mismatch between the sizes of input streams to a kernel, GPU Brook automatically resizes one of the streams to resolve the mismatch. In FPGA Brook, we took a different approach. Similar to GPU Brook, we only support two types of

operators for communication with the memory. However, we do not provide an implementation for these operators, and instead let the programmer define their functionality. As a result, the programmer can ensure that streams have matching sizes. The added benefit of this is that the programmer can specify that data should be read from a custom I/O interface instead of memory. Furthermore, the programmer can specify additional operations to be performed within these operators. We illustrate this through an example of the autocorrelation application whose code is shown in Listing 3.1.

Autocorrelation is the correlation of a signal with a time-shifted version of itself. The input signal is represented as a sequence of discrete sample values and the computation is usually performed for many different time-shift values. The number of samples the signal is shifted by is known as *lag*. The autocorrelation algorithm is commonly used in digital signal processing, for example, to uncover a periodic signal corrupted by noise. The algorithm works by multiplying the time-shifted signal by the original signal on a sample-by-sample basis. The results of these multiplications are then added to produce one number and the procedure is repeated for many different lag values. If the signal is periodic or contains a periodic component, the output for the lag value that is equal to the signal's period will be significantly larger than the output for other lag values.

Listing 3.1 demonstrates how autocorrelation can be implemented in FPGA Brook. The program assumes that the input signal is stored inside the memory array called *input*. Two operators, *create1* and *create2*, are used to create two streams; one stream with the original signal and another with a time-shifted version of the signal. The signals are then multiplied by the *mul* kernel and multiplication results are summed by the reduction kernel *sum*. Finally, the result is written to memory by the *write* operator. The operation is repeated for many different lag values.

Operators *create1* and *create2* in Listing 3.1 are examples of *streamRead-type operators* in FPGA Brook. They are defined as kernels with no input streams and a special kind of output stream designated by the *vout* keyword. In ordinary Brook, *vout* streams are used in kernels that can produce more than one output stream element per one input stream element. We chose to use *vout* streams for streamRead-type operators because of the *push* operation associated with these streams, which determines when a new stream element should be "pushed" to the output stream. This is convenient to allow the programmer to perform preparatory operations on the data before sending it to the output stream. In the autocorrelation example, this is used to ensure that the output stream has the same number of elements, regardless of the lag value. To achieve this, kernel *create2* fills the stream elements with zero values when necessary.

Operator *write* in Listing 3.1 is an example of the *streamWrite-type operator* in FPGA Brook, defined as a kernel with no output stream. These operators are usually simpler than streamRead-type operators and only perform writing of stream data to memory. Every read of the stream variable within the streamWrite-

Listing 3.1 Autocorrelation application expressed in FPGA Brook

```
kernel void create1 (int array[], vout[] int stream<>) {
  int i;

  for (i=0; i<INPUTS; i++) {
    stream = array[i];
    push (stream);
  }
}

kernel void create2 (int array[], int instance, vout[] int stream<>){
  int i;

  for (i=0; i<INPUTS; i++) {
    if (i < (INPUTS-instance))
      stream = array[i+instance];
    else
      stream = 0;
    push (stream);
  }
}

kernel void mul (int a<>, int b<>, out int c<>) {
  c = a * b;
}

reduce void sum (int a<>, reduce int r<>) {
  r += a;
}

kernel void write (int stream<>, int *array) {
  *array = stream;
}

void main () {
  int input[INPUTS], output[LAGS];
  int input1<INPUTS>;
  int input2<INPUTS>;
  int mul_result<INPUTS>;
  int reduce_result<1>;
  int instance;

  for (instance=0; instance<LAGS; instance++) {
    create1 (input, input1);
    create2 (input, instance, input2);
    mul (input1, input2, mul_result);
    sum (mul_result, reduce_result);
    write (reduce_result, output+instance);
  }
}
```

type kernel is assumed to be a destructive read, which means that it consumes one stream element from the input stream FIFO. The streamWrite-type operators can perform additional processing if desired, just like the streamRead-type operators. However, it is recommended that as much processing as possible

should be performed by kernels, because the compiler can better take advantage of parallelism in kernels than in operators. This is because the operators can introduce dependencies among the stream elements.

While ordinary kernels are implicitly executed as many times as necessary to process all stream elements (as declared by the stream size), streamRead- and streamWrite-type operators are executed only once, exactly as specified by the programmer. The custom nature of streamRead- and streamWrite-type operators implies that the programmer is responsible for writing code that produces and consumes a sufficient number of stream elements within these operators. As a result, our design flow cannot take advantage of any potential data parallelism within the operator code. This is not a severe limitation because these operators typically access off-chip memory, so their performance is usually limited by the memory bandwidth, not by computation within the operators. In case one of these operators produces (or consumes) a 2-D stream, the stream should be produced (or consumed) in a row-major order because other kernels assume that stream elements are passed between kernels in row-major order.

User defined streamRead- and streamWrite-type operators allow programmers to express complex behaviour. For example, the convolution operation discussed in section 2.2 requires a shift register to create a stream of samples. Such an operation could not be implemented in GPU Brook, or more precisely it would have to be implemented as a software function, which would not allow its acceleration, but is easily implemented using the streamRead-type operator. While the original Brook specification [5] provides several stream operators, including the stencil operator, which can be used to fill this gap, such operators are not supported in GPU Brook, and consequently we do not support them in FPGA Brook either. While streamRead-type operators work well to prepare streams for processing when the operation such as convolution is the first kernel in the application, stream operators are necessary to perform such preparation when the convolution kernel is a part of a larger application. For example, if convolution needs to process an output stream of another kernel, the data is stored in a stream and is not available in the main memory for the streamRead-type operator to process it, so the stencil operator becomes necessary. Therefore, stream operators should be implemented in future versions of FPGA Brook to support such functionality. In such a case, a custom streamRead-type operator would not be necessary for this application. Instead, it would be implemented as an ordinary streamRead operator, which simply copies the whole array into a stream, followed by the stencil operator. An interesting observation is that optimizations discussed by Liao et al. [12], which combine stream operators, may combine these two operations, and produce a result similar to our streamRead-type operator.

3.3. Streams of Arrays

One of the advantages of stream computing over vector computing is the capability of streams to hold complex data, such as structures or arrays. In this section we demonstrate how this capability can be used

to implement image and video processing applications through an example of the zigzag kernel, which is a part of the MPEG-2 video compression standard [57].

MPEG-2 is a video compression standard used for high-definition television (HDTV) and various video storage and transmission applications [57]. While the complete MPEG-2 decoding process is complex, its components are relatively simple and can be used to illustrate concepts in stream processing. MPEG-2 uses several compression techniques, including spatial encoding, which consists of 2-D discrete cosine transform (2-D DCT), quantization and zigzag ordering algorithms. The video sequence is first represented as a sequence of still pictures, which are broken down into blocks of 8x8 picture elements. The picture elements are commonly referred to as pixels. Both encoder and decoder use this block of 64 pixels as the basic unit of data they operate on. Spatial encoding and decoding can be performed independently on different blocks of data, making it suitable for streaming implementation. The encoder first performs *2-D DCT* on the block of pixels, which transforms the pixel values from the time domain into the frequency domain. The quantization process then divides the frequency components by predefined factors, based on characteristics of the human visual system, which usually results in a significant reduction in the amount of information that has to be stored, while resulting in only a small loss of perceived video quality. Finally, the output values are ordered by increasing frequency, known as *zigzag* ordering, which is useful for other compression techniques performed as a part of MPEG-2 encoding. In the decoder, inverse operations are applied in reverse order. First, zigzag order is reversed, followed by inverse quantization, and finally inverse DCT (IDCT).

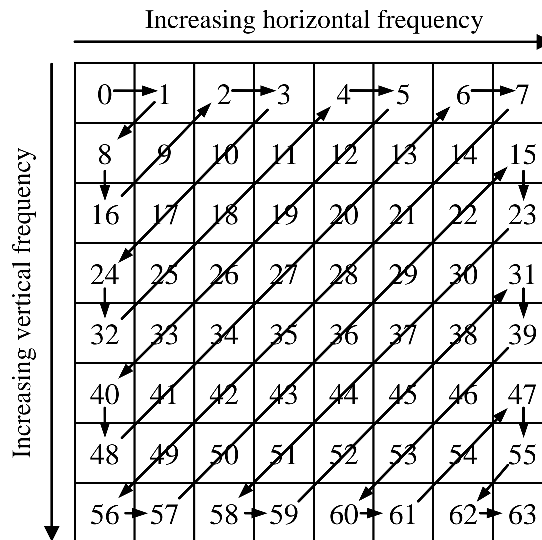


Figure 3.1 Graphical illustration of zigzag ordering

Figure 3.1 depicts the zigzag ordering performed by the MPEG-2 encoder. The data block is represented as a two-dimensional array because it is a part of a picture in a video sequence. Numbers indicate the position of each data element when the block is stored in memory in row-major order. Zigzag ordering is performed on data blocks output by the 2-D DCT transformation. One of the properties of the transformation is that its output is ordered by increasing horizontal frequency within rows and increasing vertical frequency within columns. Consequently, the value corresponding to the lowest frequency is in the top-left corner, while the value corresponding to the highest frequency is in the bottom-right corner. The arrows in Figure 3.1 indicate one possible ordering of samples by increasing frequency, taking both horizontal and vertical frequencies into account.

Zigzag ordering can be expressed as a function in the C programming language as shown in Listing 3.2. While the code has only one simple loop, it is challenging for a parallelizing compiler to extract parallelism, because the output array is accessed irregularly, so dependencies cannot be reliably eliminated. It is also not immediately obvious how this ordering can be performed in Brook or GPU Brook. If the block elements are put into a stream, the zigzag ordering cannot be performed inside a kernel because kernels cannot reorder stream elements. Stream operators in the original Brook specification [5] also do not support arbitrary reordering of stream elements. The key to implementing zigzag ordering using streaming is to observe that although individual data elements do not exhibit data parallelism, blocks of data are completely independent and zigzag ordering can be performed on them in parallel. In such a situation, one should use a stream of arrays to organize data. In this configuration, each stream element is an array consisting of 64 elements, and each array can be processed independently. FPGA Brook code implementing the zigzag operation using streams of arrays is shown in Listing 3.3.

Listing 3.2 Zigzag ordering expressed in the C programming language

```
void zigzag (short in_array[], short out_array[]) {
  int i;
  const char zig_zag_scan[64] = {
      0,  1,  5,  6,  14, 15, 27, 28,
      2,  4,  7,  13, 16, 26, 29, 42,
      3,  8,  12, 17, 25, 30, 41, 43,
      9,  11, 18, 24, 31, 40, 44, 53,
      10, 19, 23, 32, 39, 45, 52, 54,
      20, 22, 33, 38, 46, 51, 55, 60,
      21, 34, 37, 47, 50, 56, 59, 61,
      35, 36, 48, 49, 57, 58, 62, 63};

  for (i=0; i<64; i++) {
    out_array[zig_zag_scan[i]] = in_array[i];
  }
}
```

Listing 3.3 FPGA Brook implementation of zigzag ordering

```
kernel void create (vout[] short stream<>[64], short *array) {
  int i, j;

  for (i=0; i<1000; i++) {
    for (j=0; j<64; j++) {
      stream[j] = array[i*64+j];
    }
    push (stream);
  }
}

kernel void zigzag (short a<>[64], out short c<>[64]) {
  int i;
  const char zig_zag_scan[64] = {
      0,  1,  5,  6,  14, 15, 27, 28,
      2,  4,  7,  13, 16, 26, 29, 42,
      3,  8,  12, 17, 25, 30, 41, 43,
      9,  11, 18, 24, 31, 40, 44, 53,
      10, 19, 23, 32, 39, 45, 52, 54,
      20, 22, 33, 38, 46, 51, 55, 60,
      21, 34, 37, 47, 50, 56, 59, 61,
      35, 36, 48, 49, 57, 58, 62, 63};

  for (i=0; i<64; i++) {
    c[zig_zag_scan[i]] = a[i];
  }
}

kernel void write (short stream<>[64], short *array) {
  int i,j;

  for (i=0; i<1000; i++) {
    for (j=0; j<64; j++) {
      array[i*64+j] = stream[j];
    }
    pop (stream);
  }
}

void main () {
  short input[64000], output[64000];
  short input_stream<1000>[64];
  short output_stream<1000>[64];

  create (input_stream, input);
  zigzag (input_stream, output_stream);
  write (output_stream, output);
}
```

The main function in the code demonstrates how streams of arrays are declared. The streamRead-type operator *create* in the code first initializes all the elements of an array and then “pushes” the array to the output stream. Stream elements are then processed by the *zigzag* kernel. As with ordinary streams, kernel code does not refer to individual stream elements. However, since stream elements are arrays, the code

can perform all operations that are normally allowed on C arrays. As a result, the kernel code looks almost identical to the C function implementing the same functionality (Listing 3.2), with the exception of declarations. This is just one example demonstrating how Brook's syntax and semantics make transition from programming in C easy for the programmers. Finally, the results are written back to memory by the streamWrite-type operator *write*. Once all the elements of an array have been stored into memory, the array is removed from the input stream using the *pop* operation, which is unique to FPGA Brook. It was necessary to introduce this operation to distinguish between reading array elements and consuming a stream element. Since the pop operation is in its nature similar to the push operation, which already exists in both Brook and GPU Brook, it should be easy for a programmer used to one of these languages to adapt. Such an operation was not necessary in GPU Brook, because GPU Brook currently does not support streams of arrays [58]. The original Brook specification [5] does not pay special attention to streams of arrays. It is reasonable to assume that streams of arrays are handled by streamRead and streamWrite operators, although their precise behaviour in terms of required data layout is not defined in the specification.

It is interesting to compare our approach to implementing the zigzag ordering to the implementation of the same application in StreamIt, shown in Listing 3.4 [59]. For simplicity we only show the filter implementing the actual operation. This filter uses a different ordering matrix than the one used by the FPGA Brook code. This is because the FPGA Brook code implements the zigzag ordering performed by the MPEG-2 encoder, whereas the StreamIt code implements the reverse operation performed by the

Listing 3.4 StreamIt implementation of zigzag ordering [59]

```
int->int filter ZigZagOrdering {
  int[64] Ordering = {
      00, 01, 08, 16, 09, 02, 03, 10,
      17, 24, 32, 25, 18, 11, 04, 05,
      12, 19, 26, 33, 40, 48, 41, 34,
      27, 20, 13, 06, 07, 14, 21, 28,
      35, 42, 49, 56, 57, 50, 43, 36,
      29, 22, 15, 23, 30, 37, 44, 51,
      58, 59, 52, 45, 38, 31, 39, 46,
      53, 60, 61, 54, 47, 55, 62, 63};

  work pop 64 push 64 {
    for (int i = 0; i < 64; i++) {
      push(peek(Ordering[i]));
    }
    for (int i = 0; i < 64; i++) {
      pop();
    }
  }
}
```

decoder. Other than the specific ordering of array elements, the operations performed by these two code segments are equivalent. An important observation is that the StreamIt code looks nothing like the C code implementing the same functionality. One of the advantages of the stream programming model is that it separates computation and communication and simplifies buffer management [56]. While StreamIt and FPGA Brook achieve this goal in different ways, we believe that our approach is easier for programmers unfamiliar with stream programming to understand. This is one of the reasons we based our work on the Brook streaming language.

3.4. Speedup Pragma Statement

When an FPGA Brook program is implemented in an FPGA, by default the compiler exploits only task parallelism by mapping each kernel into a separate hardware unit, and connecting hardware units by FIFO buffers, so that they can operate in parallel in a pipelined fashion. In the current implementation of our design flow, the programmer can take advantage of data parallelism through the speedup pragma statement. The pragma statement has the following format:

```
#pragma br2c_speed_up <kernel_name> <desired_speedup>
```

For instance, if the zigzag kernel is found to have throughput that is four times lower than needed, the programmer can insert a pragma statement that specifies that the kernel should be accelerated four times. In an application with multiple kernels, each kernel's throughput can be increased by an arbitrary factor, and the speedup factors for different kernels do not have to match. The speedup pragma cannot be used with streamRead- and streamWrite-type operators. This is because these operators may contain dependencies that the compiler cannot resolve, as discussed in section 3.2. Finally, there are theoretical limits to achievable speedup from data parallelism. For example, the maximum theoretically achievable speedup in an ordinary kernel is equal to the number of stream elements, at which point all elements are processed in parallel. If a desired speedup is not theoretically achievable, our compiler will report an error. We discuss theoretical speedup limits due to data parallelism in more detail in Chapter 5.

Our approach to exploiting data parallelism requires the programmer to determine throughput of each kernel and specify that the kernels that do not meet the desired throughput be sped up. This can be done in one of two ways. As the application is being developed, it is common to develop and test each kernel individually. In such a case, it is relatively easy to measure throughput of the kernel while it is being tested. If the application is not developed in such a modular fashion, programmers can estimate which kernels may be bottlenecks based on computational complexity of the kernels, and apply speedup pragmas accordingly. This process could be further automated by performing analysis of each kernel code and estimating its throughput. In such a case, the compiler could automatically speed up critical kernels to meet a target throughput. Since our design flow relies on Altera's C2H compiler, such an automated flow

could rely on C2H's performance estimates. C2H provides scheduling information for loops in the code, which specify the number of cycles required to execute each iteration of the loop [27]. Since kernels are implemented as loops in C2H code, as described in the next chapter, this information could be used to estimate kernel performance and further automate our design flow.

3.5. User Interface

Programmers write their FPGA Brook code using a text editor of their choice and save it as a *.br* file (e.g. *zigzag.br*), based on the same file extension used by GPU Brook. In the current version of our compiler it is assumed that all source code is placed in a single file. The code is first compiled by our source-to-source compiler by issuing a command on the command-line (e.g. *br2c zigzag.br*). As a result, the compiler creates a project folder with a name based on the source file name (e.g. *zigzag_sopc_system*) and populates it with a number of files needed for further processing by Altera's tools. One of the files is a script called *create-this-app*, which is based on a script provided by Altera's Software Build Tools [60]. The script is customized for the application being compiled and placed into the *software\app* subfolder of the project (e.g. *zigzag_sopc_system\software\app*). The programmer only has to run this script, which runs the C2H tool that produces custom hardware accelerators, followed by Quartus II, which produces the FPGA programming file that implements all the hardware for the application. The script also invokes the Nios II compiler, which compiles the code for the Nios II processor that controls all the hardware accelerators. After this, the system is ready to be downloaded into the FPGA and run. This can be achieved through two more commands issued on the command line; *nios2-configure-sof* and "*make download-elf*", which download the FPGA programming file and Nios II binary executable to the FPGA board, respectively. Both files are needed because C2H requires a Nios II processor in the system, as was shown in Figure 2.3.

It is important to emphasize that the above procedure does not require any pin assignments, constraint settings, or any other FPGA-specific settings or development. While in real production environment there will eventually be a need for FPGA expertise to integrate the design into the overall system, most of the development can be performed by programmers with limited understanding of the FPGA technology. Our compiler automatically generates an SOPC system and all the files necessary to target a specific FPGA board, which in our case is the Altera's DE2 board [61]. Other boards could be supported in a similar fashion. The compilation procedure outlined above could be further streamlined by integrating it into a graphical user interface (GUI).

Feature	FPGA Brook	GPU Brook
Floating-point support	No	Yes
Integer Support	Yes	No
Custom streamRead- and streamWrite-type operators	Yes	No
Built-in stream operators	No	Three operators supported
Gather and scatter streams	No	Yes
Loops in the streaming dataflow graph	No	Yes
Streams of arrays	Yes	No
Custom data structures (*struct*)	No	Yes
Data parallelism exploited	Explicit pragma	Implicitly, based on available hardware
Automatic stream resizing	No	Yes
Iterator streams	No	Yes

Table 3.1 Feature comparisons between FPGA Brook and GPU Brook

3.6. FPGA Brook Summary

In summary, FPGA Brook adds three features to the GPU Brook streaming language. First, FPGA Brook supports custom streamRead- and streamWrite-type operators, which provide more flexibility for stream creation than built-in operators. Second, FPGA Brook supports streams of arrays, which are currently not supported by GPU Brook [58]. To support streams of arrays within custom streamRead- and streamWrite-type operators, FPGA Brook introduces the *pop* operator and extends the semantics of the *push* operator for the custom operators. Finally, FPGA Brook supports the speedup pragma statement, which allows explicit control over exploitation of data parallelism in the application. At the same time, FPGA Brook does not support some of the features that GPU Brook supports. Differences between the FPGA Brook and GPU Brook compilers are summarized in Table 3.1.

The entries in Table 3.1 specify whether the respective compiler supports a given feature or not. The table shows only the features that are different between the two compilers. With the exception of iterator streams, all other features in the table were discussed earlier in this thesis. Iterator streams allow automatic creation of streams initialized with an arithmetic sequence of numbers. Given a size of the iterator stream and lower and upper bounds of the sequence, the GPU Brook compiler generates code that automatically fills the stream with the appropriate values. We did not implement iterator streams because none of our benchmarks required them. Their implementation in FPGA Brook would be straightforward; it only requires our source-to-source compiler to generate a simple *for* loop with appropriately adjusted loop limits.

Since FPGA Brook relies on the C2H compiler for behavioural synthesis, it also inherits the limitations of the C2H compiler, which were described in section 2.4. Two of these (lack of support for

recursion and lack of support for the *goto* keyword) are also limited in GPU Brook, while the remaining ones are unique to FPGA Brook because of its reliance on C2H.

In this chapter we presented FPGA Brook from the programmer's perspective, avoiding specific implementation details. In the next chapter we focus on implementation details and techniques used to implement FPGA Brook programs in FPGA logic.

Chapter 4

Compiling FPGA Brook programs into FPGA Logic

In this chapter we focus on the design flow we developed to implement FPGA Brook programs in FPGAs. We start with a high-level overview of the design flow, describing different tools used in the flow. We then focus on our source-to-source compiler, which maps Brook programs into C2H code. As in previous chapters, we describe our methodology through examples of applications implemented in FPGA Brook.

4.1. Design Flow

Our design flow is shown in Figure 4.1. As mentioned in section 3.5, the FPGA Brook code is first compiled by our source-to-source compiler. The result is C2H code in which kernels have been converted into C2H functions to be implemented as hardware accelerators, and SOPC system description, which instantiates the Nios II processor, peripherals in the system (e.g. memory interfaces) and sufficient number and types of FIFO buffers to interconnect the kernels. The C2H code also contains C2H pragma statements that specify how hardware accelerators connect to the FIFOs in the system. The C2H compiler then processes this code and produces Verilog HDL code for each of the hardware accelerators, and also updates the SOPC system to include these accelerators and connect them to the FIFOs and the rest of the system. The Verilog code is then synthesized into FPGA logic by the Quartus II CAD tool to produce the FPGA programming file, which can then be used to program the FPGA device. Early ideas on the proposed design flow were described in our previous publications [16,62-64].

C2H also produces C wrapper functions that activate the accelerators and, depending on the type of the accelerator wait until the accelerator completes its operation. While all the code outside of kernels, including the main function, is executed on the Nios II processor, calls to kernels are replaced by calls to the wrapper functions during the linking phase. This code is compiled by the Nios II *gcc* compiler producing the Nios II binary executable that is downloaded into the FPGA device running the SOPC system.

In this section we provided a high-level overview of the steps involved in our design flow. We describe the flow in more detail in the following sections.

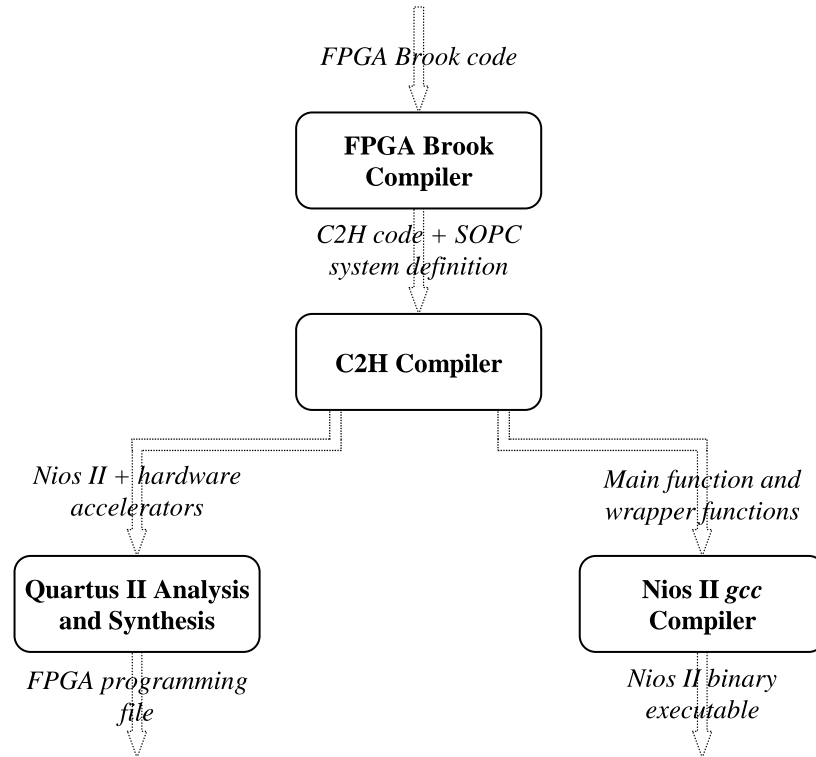


Figure 4.1 FPGA Brook design flow

4.2. Compiling FPGA Brook Code to C2H Code

The first step in our design flow converts the program written in FPGA Brook into C2H code by means of a source-to-source compiler we built. This compiler was developed by modifying the GPU Brook compiler [18]. GPU Brook was based on the CTool open-source C parser, which was extended to support GPU Brook keywords and other constructs. CTool parses the source code and produces an *abstract syntax tree*, which is a graph representing the structure of the original program. CTool allows the information in the syntax tree to be modified and can also output the code represented by the syntax tree [65]. This makes it possible to make modifications to the original code and perform source-to-source compilation. We use this capability to modify the Brook code where needed and output the resulting code suitable for C2H compilation.

The design space for FPGA implementation of streaming applications is large. For instance, a kernel could be implemented as custom hardware, a soft-core processor, or a streaming processor. In either case, several parallel instances of hardware implementing the kernel may be necessary to meet the throughput requirement. The choice of types and numbers of hardware units will affect the topology of the interconnection network. Finally, stream elements can be communicated through on-chip or off-chip

memories, organized as regular memories or FIFO buffers. This is because the Brook specification does not specify the nature of memory buffers holding streams, unlike StreamIt, which assumes FIFO communication [41].

We generate custom hardware for each kernel in the application. An ordinary soft processor would be a poor choice for implementing kernels, because it can only receive and send data through its data bus, which may quickly become a bottleneck. Custom hardware units can have as many I/O ports as needed by an application and are likely to provide the best performance. However, if a kernel is complex, the amount of circuitry needed for its implementation as custom hardware may be excessive, in which case a streaming processor may be a better choice. While our current work focuses on implementing kernels as hardware units, streaming processors remain a viable option and should be explored in the future.

The code generator in our source-to-source compiler emits a C2H function code for each kernel and instructs the C2H compiler to implement it as a hardware accelerator. We use C2H connection pragma directives to define how streams are passed between kernels through FIFOs. We use FIFO buffers because of their small size, which allows us to implement them in an on-chip memory in the FPGA. Since contemporary FPGAs still have a relatively small amount of on-chip memory, it is not possible to store large amounts of data, such as complete streams, on-chip. Off-chip memory is a poor choice because of the bandwidth limitations. FIFOs are a good choice because they also naturally fit into the streaming paradigm, because they act as registers in the pipeline. FIFOs are used instead of simple registers because they provide buffering for cases when execution time of a kernel varies between the elements. For example, if a kernel takes a long time to process a stream element, the next kernel downstream could become idle if there was just one register between the two kernels. Using FIFOs, kernels can process data from FIFOs as long as upstream kernels deliver stream elements at a sufficient average rate.

4.2.1. Handling One-Dimensional Streams

We illustrate the work done by our source-to-source compiler using the autocorrelation application shown in Listing 3.1. Since kernels implicitly operate on all stream elements, our compiler generates an explicit *for* loop around the statements inside the kernel function to specify that the kernel operation should be performed over all stream elements. For the Brook code in Listing 3.1, our compiler produces code similar to the one shown in Listing 4.1. For brevity, we only show code generated for the kernels. We do not show the code generated for the streamRead- and streamWrite-type operators and the main program, because they undergo only small changes compared to the original FPGA Brook code. In the code shown in Listing 4.1, all compiler-generated variable names start with the "_" character. For the *mul* kernel, the code is a straightforward *for* loop that reads elements from the input stream FIFOs associated with pointers *a* and *b*, multiplies them and writes the result to the output stream FIFO associated with pointer *c*. The boundary for the *for* loop (*INPUTS*) was automatically inserted by the compiler, based on

the sizes of the streams passed to the *mul* kernel from the main program. The value of the loop iterator *_iter* can be directly used to implement the *indexof* operator in the original FPGA Brook code. The *indexof* operator is useful to determine the position of the current stream element within the stream, as described in section 2.2.

Listing 4.1 C2H code generated for the autocorrelation application

```
void mul () {                                                               (1)
  volatile int *a, *b, *c;  int _iter;                                      (2)
  int _temp_a, _temp_b, _temp_c;                                            (3)

  for (_iter=0; _iter<INPUTS; _iter++) {                                    (4)
    _temp_a = *a;                                                           (5)
    _temp_b = *b;                                                           (6)
    _temp_c = _temp_a * _temp_b;                                            (7)
    *c = _temp_c;                                                           (8)
  }
}

void sum() {                                                                (9)
  volatile int *a, *r;                                                     (10)
  int _temp_a, _temp_r, _iter;                                             (11)
  int _reduction_ratio = INPUTS/1;                                         (12)

  for (_iter=0; _iter<INPUTS; _iter++) {                                   (13)
    if ((_iter%_reduction_ratio == 0) && (_iter != 0))                     (14)
      *r = _temp_r;                                                        (15)
    _temp_a = *a;                                                          (16)
    if (_iter%_reduction_ratio == 0)                                       (17)
      _temp_r = _temp_a;                                                   (18)
    else                                                                   (19)
      _temp_r += _temp_a;                                                  (20)
  }
  *r = _temp_r;                                                            (21)
}

#pragma altera_accelerate connect_variable create1/stream to input1_fifo/in
#pragma altera_accelerate connect_variable mul/a to input1_fifo/out

#pragma altera_accelerate connect_variable create2/stream to input2_fifo/in
#pragma altera_accelerate connect_variable mul/b to input2_fifo/out

#pragma altera_accelerate connect_variable mul/c to mul_result_fifo/in
#pragma altera_accelerate connect_variable sum/a to mul_result_fifo/out

#pragma altera_accelerate connect_variable sum/r to reduce_result_fifo/in
#pragma altera_accelerate connect_variable write/stream to \
        reduce_result_fifo/out

#pragma altera_accelerate enable_interrupt_for_function create1
#pragma altera_accelerate enable_interrupt_for_function create2
#pragma altera_accelerate enable_interrupt_for_function mul
#pragma altera_accelerate enable_interrupt_for_function sum
```

Pointers *a*, *b* and *c* are connected to FIFOs, which is specified by C2H pragma statements generated by our compiler as shown at the bottom of the listing. FIFOs are implemented in hardware, so kernel code does not have to manage FIFO read and write pointers. Temporary variables *_temp_a*, *_temp_b* and *_temp_c* are used to preserve semantics of the original Brook program. Consider the statement *c = a + a;* in FPGA Brook. According to Brook semantics, this statement is equivalent to *c = 2*a*; which is true if temporary variables are used. However, if the temporary variables were not used and stream references were directly converted to pointers, the original statement would get translated into **c = *a + *a*; which would produce an incorrect result. This is because each pointer dereference performs a read from the FIFO, so two consecutive stream elements would be read, instead of the same element being read twice. Temporary variables ensure that only one FIFO read operation is performed per input stream element, and that only one FIFO write operation is performed per output stream element.

The code generated for the *sum* kernel is more complicated because *sum* is a reduction kernel. Although in our example the input stream with *INPUTS* elements is reduced to a stream with only one element, in a general case the reduction operation can result in more than one element in the output stream. As was previously shown in Figure 2.2, this means that several consecutive elements of the input are combined to produce one element of the output, in a general case. We will refer to a one-dimensional reduction that reduces an input stream with *X* elements to an output stream with *A* elements as an *X-to-A* reduction. We implicitly assume that $A<X$ and that *A* divides *X* evenly, otherwise the result is undefined. We refer to the ratio *X/A* as the *reduction ratio*. For example, consider a summation that takes a twenty-element stream on the input and produces a four-element stream on the output (*20-to-4* reduction). The reduction ratio is 5, which means that 5 consecutive elements of the input stream are added to produce one element of the output stream. This means that after every 5 input stream elements are processed, an output stream element should be produced and a new reduction should start. In a general case of an *X-to-A* reduction, an output should be produced and a new reduction started after every *X/A* (i.e. reduction ratio) input elements.

The code shown in Listing 4.1 for the *sum* reduction kernel implements an *INPUTS-to-1* reduction, but is general enough to handle a general case as well. Every time the number of input elements processed is equal to the reduction ratio (line 14), a new output is produced (line 15). The second part of the condition in line 14 (*_iter != 0*) ensures that no output is produced at the very beginning, when no result has been calculated yet. At the same time (line 17), a new addition is started by reading the current element (line 18), instead of adding it to the running sum (line 20), as is done in subsequent steps. The final element of the output has to be produced separately (line 21), because lines 14 and 15 do not get executed for the last output element.

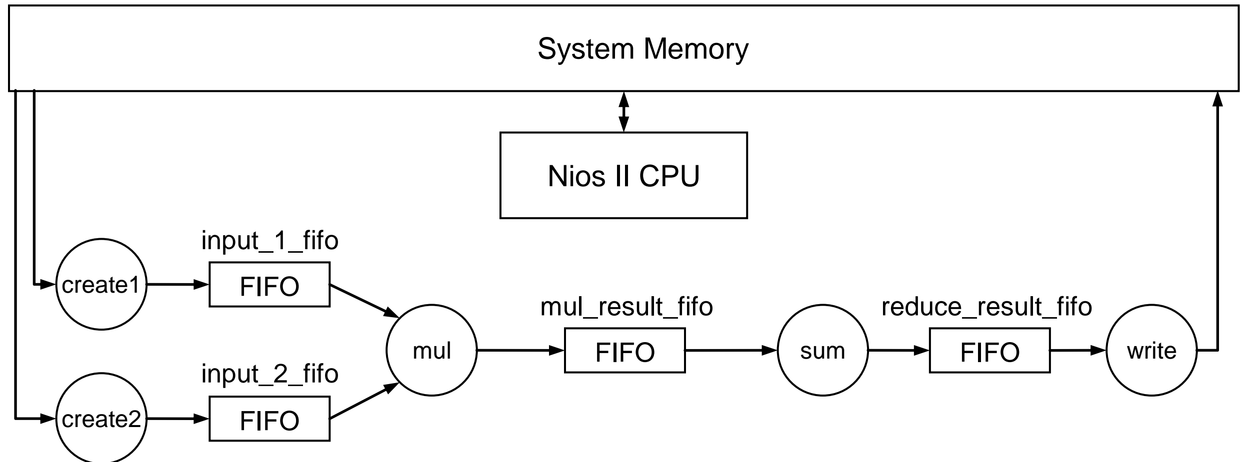


Figure 4.2 Dataflow graph for the autocorrelation application

The code in Listing 4.1 also contains pragma statements, which specify how kernels connect to FIFOs in the system, and consequently to one another. For example, the fifth pragma statement specifies that the pointer *c* defined in the kernel *mul* connects to the *in* port of the FIFO named *mul_result_fifo*. The pragma statement below it specifies that the pointer *a* defined in the kernel *sum* connects to the *out* port of the same FIFO. Together, these pragmas define a connection between the *mul* and *sum* kernels through a FIFO, as shown in Figure 4.2, based on dataflow analysis of the main program. The figure also shows that the operators *create1*, *create2* and *write* connect to the main system memory. As previously discussed in section 2.4, if connection pragmas are not used for a pointer, C2H compiler will connect the pointer to all the memories Nios II connects to.

The final four pragmas in Listing 4.1 specify that all kernels and operators except the *write* operator should run in the interrupt mode. This means that the main function will not wait on their completion, so all the kernels and operators can run in parallel. The write operator does not run in this mode, so that the main program can detect when the hardware accelerators have finished processing (i.e. when the last element of the *reduce_result* stream is written to the memory). As previously discussed, the main program, and any other code outside the kernels and custom stream operators, runs on the Nios II processor.

4.2.2. Handling Two-Dimensional Streams

Two dimensional streams can be processed by any kernels or reduction kernels. Kernel code in the FPGA Brook source is the same for both 1-D and 2-D streams. Our source-to-source compiler takes care of all the necessary details and generates different code depending on dimensionality of the streams being passed to it. Ordinary kernels processing 2-D streams are handled similarly to their 1-D counterparts, with

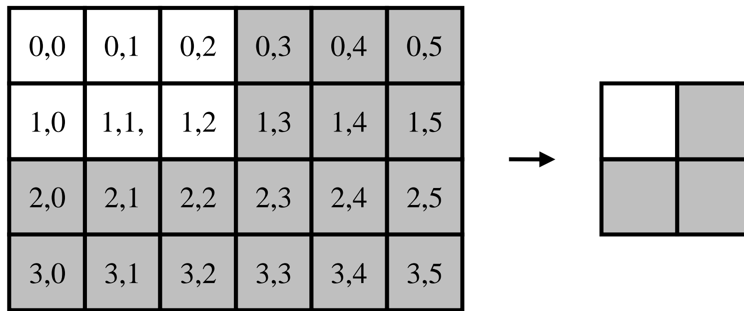


Figure 4.3 Example of two-dimensional reduction

the exception that the kernel code is enclosed inside two *for* loops. While one loop with appropriately adjusted loop boundaries may suffice for some kernels, two loops are necessary to support the *indexof* operator. This operator returns a two-dimensional value when applied to a 2-D stream, which can be used to address a two-dimensional array depending on the current stream element's index. For example, such functionality is useful to implement 2-D convolution, commonly used in image processing applications.

Two-dimensional reductions are more challenging to implement than one-dimensional reductions. This is because stream elements arrive in a row-major order through the FIFO, but reductions have to be performed over neighbouring elements, which span different rows, as shown in Figure 4.3. The numbers denote row and column indices of the stream elements the way they would be returned by the *indexof* operator. Since stream elements arrive in a row-major order, the reduction kernel does not receive all the elements of the input needed to produce one element of the output in a sequence. As a result, the reduction kernel has to allocate a memory buffer that can store one row of the output stream, which means that the size of the buffer is equal to the number of columns in the output stream. The example in the figure reduces a 4x6 stream to a 2x2 stream. C2H code implementing such a reduction is shown in Listing 4.2.

We refer to Figure 4.3 to illustrate how the code works. We refer to a two-dimensional reduction that takes an input stream with X rows and Y columns, and produces an output stream with A rows and B columns as *<X,Y>-to-<A,B>* reduction. We refer to the ratio *X/A* as the *row reduction ratio* and *Y/B* as the *column reduction ratio*. Finally, we refer to an area of the input stream that includes all elements of the input stream that contribute to one element of the output stream as a *reduction window*. For example, Figure 4.3 shows a <4,6>-to-<2,2> reduction with row reduction ratio of 2 and column reduction ratio of 3, which means that a reduction window has 2 rows and 3 columns.

The code in Listing 4.2 uses two iterators to keep track of indices of the input elements: row iterator *_iter_0* and column iterator *_iter_1*. A compiler generated array *_temp_r* is used as a memory buffer

Listing 4.2 C2H code generated for two-dimensional reduction

```
void  sum() {                                                          (1)
  volatile int  *a;                                                    (2)
  int  _temp_a;                                                        (3)
  volatile int  *r;                                                    (4)
  int  _temp_r[2];                                                     (5)
  int _iter_0;                                                         (6)
  int _iter_1;                                                         (7)
  int row_ratio = 2;                                                   (8)
  int col_ratio = 3;                                                   (9)

  for (_iter_0=0; _iter_0<4; _iter_0++) {                             (10)
    for (_iter_1=0; _iter_1<6; _iter_1++) {                           (11)
      _temp_a = *a;                                                   (12)
      if ((_iter_0%row_ratio == 0) && (_iter_1%col_ratio == 0))       (13)
        _temp_r[_iter_1/col_ratio] = _temp_a;                         (14)
      else                                                            (15)
        _temp_r[_iter_1/col_ratio] += _temp_a;                        (16)
      if ((_iter_0%row_ratio == (row_ratio-1)) &&
          (_iter_1%col_ratio == (col_ratio-1)))                       (17)
        *r = _temp_r[_iter_1/col_ratio];                              (18)
    }
  }
}
```

storing partial reduction results. A new reduction always starts in the top-left corner of the reduction window, which is when both row and column iterators are divisible by their respective reduction ratios (code line 13). The index of the memory buffer where the partial result can be stored is determined by dividing the column iterator by the column reduction ratio (lines 14, 16 and 18). Finally, processing of a reduction window is complete, and a new output element can be produced, when its bottom-right corner has been processed. This is the case when the remainder of the division of the row and column iterator with their respective reduction ratios is at its maximum possible value (line 17). The code shown in the previous two sections used modulo and division operations for illustrative purposes. Our compiler actually simplifies these operations for efficient hardware implementation. We discuss details of this implementation in the next section.

4.2.3. Strength Reduction of Modulo and Division Operations

As shown in the previous two sections, implementing reduction operations requires modulo and division operations to determine when an appropriate number of elements have been processed and a new reduction should be started. Unfortunately, these operations are not efficiently implemented in FPGAs because FPGAs do not contain hard logic blocks implementing them, and their implementations in soft logic are inefficient. For example, a division could be implemented as a combination of a state machine and a simple datapath that performs serial division, which may take many clock cycles to complete. On the other hand, a circuit that performs division in fewer clock cycles would have a long propagation delay

and thus may negatively impact F_{max} of the rest of the circuit. Therefore, division and modulo operations should be avoided where possible [29].

While implementing FPGA Brook applications we observed three distinct opportunities for implementing division and modulo operations more efficiently. First, expressions whose both operands are compile-time constants can be evaluated at compile time. For example, reduction ratios can be evaluated by our compiler because all stream sizes are known at compile time. Second, modulo and division operators needed to implement reductions can be implemented as counters because they depend on the loop iterator values. Similarly, modulo and division expressions involving the *indexof* operator can be implemented as counters because the *indexof* operator is implemented through loop iterators as well. Our techniques for strength reduction are similar to those described by Sheldon et al. [66]

Listing 4.1 contained the *sum* reduction kernel which implemented reduction of a one-dimensional stream into another one-dimensional stream. A modulo operator was used to determine when an output stream element should be produced and a new reduction started. Since the modulo operation involves the loop iterator, it can be replaced by a counter that can implement the modulo operation at a significantly lower cost. This is shown in Listing 4.3. The *_mod_iter* variable emulates the modulo operation by counting up to the value of reduction ratio and then being reset to zero (lines 6 and 7). The rest of the code is the same as in Listing 4.1, except that the modulo iterator is used instead of performing the modulo operation.

Modulo operations inside of reductions handling two-dimensional streams can be implemented as counters in the same manner, as shown in Listing 4.4. Two modulo iterators (*_mod_iter_0* and *_mod_iter_1*) are used to implement the two required modulo operations. In addition, *_mod_iter_1* is used

Listing 4.3 The *sum* reduction kernel with modulo counter

```
void sum() {                                                        (1)
  volatile int *a, *r;                                              (2)
  int _temp_a, _temp_r, _iter, _mod_iter=0;                         (3)
  int _reduction_ratio = INPUTS;                                    (4)

  for (_iter=0; _iter<INPUTS; _iter++, mod_iter++) {                (5)
    if (_mod_iter == _reduction_ratio)                              (6)
      _mod_iter = 0;                                                (7)

    if ((_mod_iter == 0) && (_iter != 0))                           (8)
      *r =_temp_r;                                                  (9)
    _temp_a = *a;                                                  (10)
    if (_mod_iter == 0)                                            (11)
      _temp_r = _temp_a;                                           (12)
    else                                                           (13)
      _temp_r += _temp_a;                                          (14)
  }
  *r = _temp_r;                                                    (15)
}
```

Listing 4.4 Reduction of two-dimensional streams with modulo counters

```
void  sum() {
  volatile int  *a;
  int  _temp_a;
  volatile int  *r;
  int  _temp_r[2];
  int _iter_0, _iter_1;
  int _mod_iter_0=-1, _mod_iter_1=-1, _div_result=-1;
  int row_ratio = 2;
  int col_ratio = 3;

  for (_iter_0=0; _iter_0<4; _iter_0++) {
    _mod_iter_0++;
    if (_mod_iter_0 == row_ratio)
      _mod_iter_0 = 0;
    _mod_iter_1=-1;
    _div_result=-1;
    for (_iter_1=0; _iter_1<6; _iter_1++) {
      _mod_iter_1++;
      if (_mod_iter_1 == col_ratio)
        _mod_iter_1 = 0;
      if (_mod_iter_1 == 0)
        _div_result++
      _temp_a = *a;
      if ((_mod_iter_0 == 0) && (_mod_iter_1 == 0))
        _temp_r[_div_result] = _temp_a;
      else
        _temp_r[_div_result] += _temp_a;
      if ((_mod_iter_0 == (row_ratio-1)) && (_mod_iter_1 == (col_ratio-1)))
        *r = _temp_r[_div_result];
    }
  }
}
```

to implement the division operation using an additional iterator *_div_result*. This iterator is incremented every time the modulo iterator overflows, thus implementing the division operation. All the iterators are initialized to the value of -1 to compensate for the first increment operation and *_mod_iter_1* and *_div_result* are initialized to this value in the outer loop for the same reason.

While examples in the code listings in this section accurately represent the work our compiler does to implement modulo and division operations, the actual generated code is more complex than the code presented here, which has been simplified for illustrative purposes. In reality, the compiler-generated code is more cryptic because it is generated automatically.

4.3. Generating FIFO Hardware

In addition to generating the C2H code, our compiler also generates some HDL code, such as the code for the FIFO buffers. While SOPC Builder provides a simple FIFO implementation, which could be used to pass streams, we do not use this implementation because it introduces a latency of one extra clock

cycle. This approach maximizes FIFO throughput in systems that utilize very high clock frequencies. Since the systems we generate do not operate at very high clock frequencies, we generate our own FIFO module, based on the *scfifo* parameterizable module provided with the Quartus II CAD tool [67]. The *scfifo* module implements a single-clock FIFO module whose data width and FIFO depth can be parameterized, and it also provides the option to minimize latency. We use the *scfifo* module to implement FIFO buffers of appropriate data widths, depending on the stream types declared in the Brook code. For example, a stream of *char* data type requires data width of one byte, while a stream of *int* data type requires data width of four bytes. The *scfifo* module is instantiated inside a wrapper module which provides an interface compatible with Altera's Avalon switch fabric.

Streams of arrays require a special kind of FIFO buffer for their implementation. Since each stream element is an array, kernel code can refer to individual elements of the array in an arbitrary way, as discussed in section 3.3. To support such functionality, we build a special kind of FIFO buffer, which we call *array FIFO*. While ordinary FIFOs contain a number of data words, array FIFOs contain a number of arrays, with an address port that allows access to elements of the array currently at the head of the FIFO. In an ordinary FIFO, elements are added to the tail of the FIFO and consumed from the head of the FIFO, and elements can be read and written only one at a time. The same is true for array FIFOs: arrays are always added to the tail of the FIFO and consumed from the head of the FIFO, and arrays can be read and written only one at a time. Since individual array elements can be written in an arbitrary order, array FIFOs contain special control ports that are used to "push" an array into the FIFO. Writing to this port designates that all elements of the array have been written and that this array can now be stored inside the FIFO buffer, to be read at a later time. A similar control port is used to "pop" an array from the array FIFO; writing to this port designates that all elements of the array have been read and that this array can now be discarded from the FIFO buffer to create room for newly arriving arrays. Figure 4.4 shows

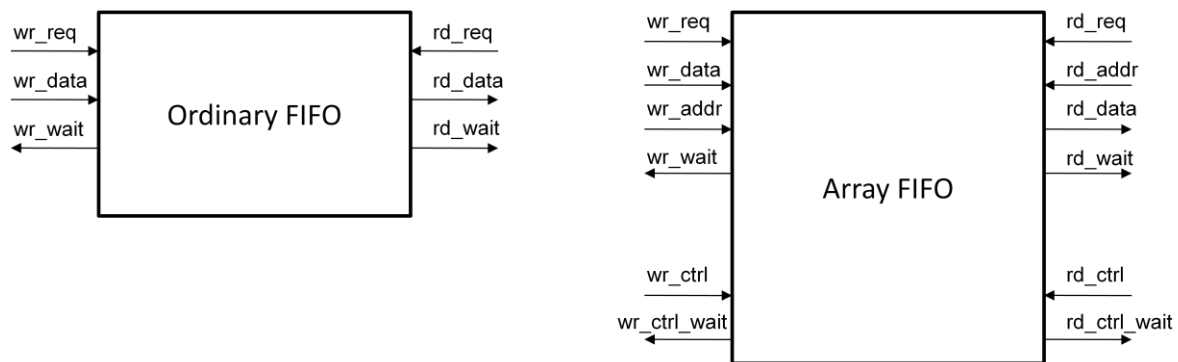


Figure 4.4 Input and output ports for ordinary and array FIFO

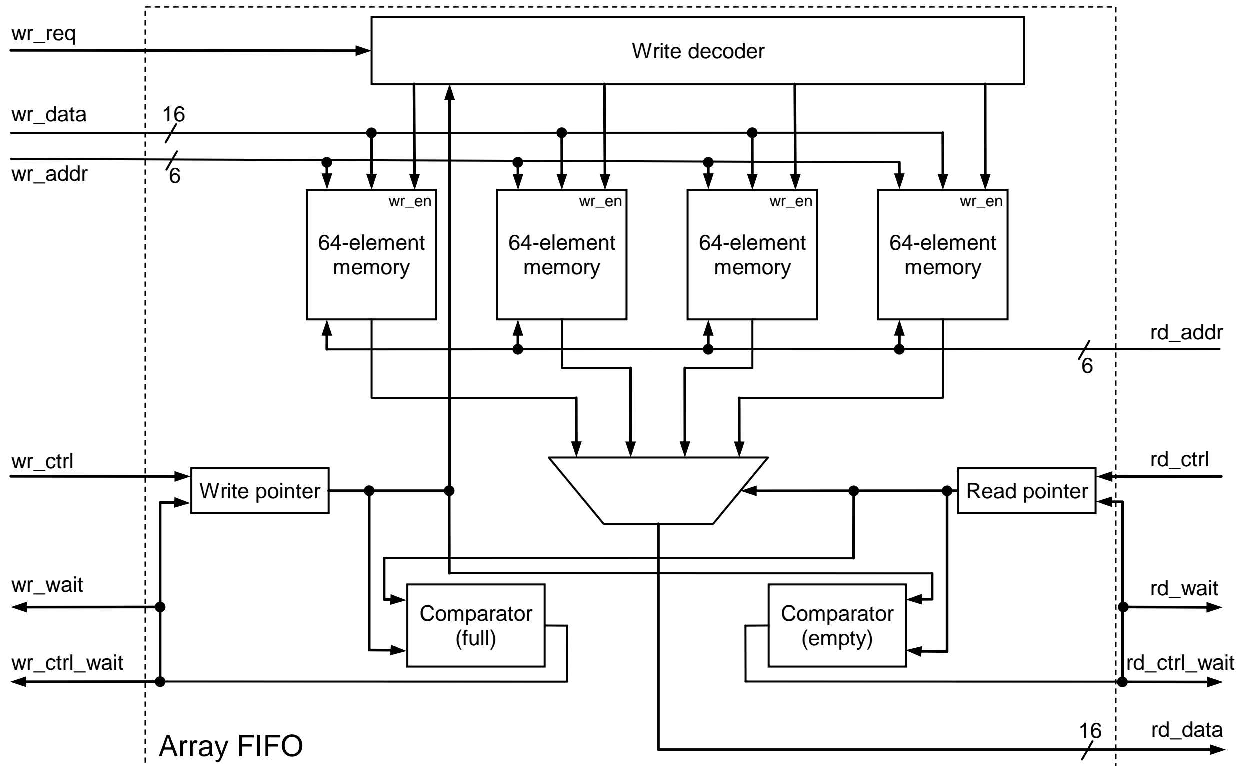


Figure 4.5 Block diagram of an array FIFO

ports for an ordinary FIFO and an array FIFO. While the ordinary FIFO has only the data port and read or write request on each side, the array FIFO also has address ports to designate the array element currently being accessed through the data port and control ports to "push" or "pop" an array. Signals with prefix *wr_* refer to signals on the side of the FIFO where elements are written (i.e. FIFO tail), while signals with prefix *rd_* refer to signals on the side of the FIFO where elements are read (i.e. FIFO head). Signals ending with *_wait* in Figure 4.4 are wait-request signals, as per the Avalon switch fabric specification [28], used by slave devices to denote that they are not ready to process the current request. For example, attempting to read data from an empty array FIFO would result in the *rd_wait* signal to be asserted until at least one array is "pushed" into the FIFO and ready for reading. Similarly, attempting to "pop" an array from an empty array FIFO would result in the *rd_ctrl_wait* signal to be asserted until at least one array is "pushed" into the FIFO.

Quartus II does not provide an existing implementation of array FIFOs, so we have to build a custom hardware module. The internal structure of an array FIFO that is four slots deep, each slot holding an array containing 64 elements, with each element 16 bits in width, is depicted in Figure 4.5. For each slot there is one memory module that can hold a complete array. The array currently being read is selected by the read pointer, and the array to be written to is selected by the write pointer. As in an ordinary FIFO, the two pointers are compared to detect when the FIFO is empty or full, which is useful to generate wait-

request signals. As can be seen in the figure, reads from the array or writes to the array have no influence on the read or write pointers. The pointers are only advanced once the appropriate control port is written to. As a result, each array behaves as an independent FIFO slot.

Our compiler generates special C2H code for reading and writing array FIFOs, as shown in Listing 4.5. This code implements the zigzag kernel first introduced in section 3.3. As with other kernels, all kernel code is enclosed in a *for* loop, which ensures that all stream elements are processed. Inside the loop, the kernel code is unchanged, with two additions. First, the pointers to FIFOs' data ports are initialized to point to the first location in an array. For example, the pointer *a* is assigned the address *INPUT_STREAM_FIFO_RD_ARR_BASE*. This is the base address of the *rd_arr* port of the *input_stream_fifo* buffer that is defined in the *system.h* file. This file is automatically generated by the SOPC Builder during C2H compilation and contains the base address of every Avalon slave port in the system. By pointing to the base address of the Avalon slave, the pointer points to the first element (i.e. element 0) of an array, which is the expected behaviour. As shown in Figure 4.5, the address port of an array FIFO is connected to all memory modules in the FIFO. Thus the base address of the FIFO's data

Listing 4.5 C2H code generated for the zigzag kernel

```
#include "system.h"

void  zigzag() {
  volatile short *a;
  volatile char *_a_ctrl;
  volatile short *c;
  volatile char *_c_ctrl;
  int _iter;
  int i;
  const char zig_zag_scan[64] = {/* omitted for brevity */};

  for (_iter=0; _iter < 1000; _iter++) {
    a = INPUT_STREAM_FIFO_RD_ARR_BASE;
    c = OUTPUT_STREAM_FIFO_WR_ARR_BASE;
    for (i = 0; i < 64; i++) {
      c[zig_zag_scan[i]]=a[i];
    }
    *_a_ctrl = 0;
    *_c_ctrl = 0;
  }
}

#pragma altera_accelerate connect_variable zigzag/a to \
        input_stream_fifo/rd_arr
#pragma altera_accelerate connect_variable zigzag/_a_ctrl to \
        input_stream_fifo/rd_ctrl
#pragma altera_accelerate connect_variable zigzag/c to \
        output_stream_fifo/wr_arr
#pragma altera_accelerate connect_variable zigzag_1d_replica_0/_c_ctrl to \
        output_stream_fifo/wr_ctrl
```

port always points to the first element of the currently selected array. The pointer initialization is performed in the outer loop to ensure that the pointers have correct values, in case the kernel code uses pointer arithmetic and modifies them. FPGA Brook allows pointer arithmetic to be used with a pointer referring to an array within a stream of arrays, as long as the changes to pointers are local to the kernel.

For each stream of arrays declared within the kernel, an additional pointer is generated, which points to the control port of the appropriate FIFO. Once all the kernel code has been executed, which implies that an input stream element (i.e. an array) has been processed and an output stream element (i.e. another array) has been produced, input and output control ports are written to advance the pointers in their respective FIFOs and thus advance processing to the next stream element. The value written to the control port is irrelevant because it is not used by the FIFO module. Associations between pointers and FIFO ports are specified by the pragma statements at the bottom of Listing 4.5.

In this section we described the hardware modules generated by our compiler that implement FIFOs used to interconnect kernels. Our compiler also generates other types of FIFOs, which will be described in Chapter 5, after describing kernel replication in more detail.

4.4. Generating SOPC System and Compilation Scripts

The C2H compiler requires a functional Nios II system before C2H code can be compiled. The compiler then integrates hardware accelerators into the system. We created a template Nios II system with components that are always used in every system, which includes the Nios II processor, UART, timer, off-chip memory controller, and a PLL producing the correct clock for the off-chip memory. While most of this template system can be used to implement FPGA Brook applications without any modifications, some files have to be customized for each application. One such file is the *.sopc* file describing the system components and their connections. Since each FPGA Brook program has different numbers and kinds of streams, and thus requires different numbers and types of FIFO buffers, a custom SOPC file containing these FIFOs has to be generated for each program. This is because C2H requires all modules other than hardware accelerators to be present in the system before compilation can begin. Generating the custom *.sopc* file involves instantiating the correct number and types of FIFOs, setting their parameters, setting valid base addresses for each Avalon slave port of each FIFO, and connecting each FIFO slave port to the Nios II data master port. While these connections are not necessary to implement applications, the slave ports cannot be left unconnected, because the C2H compiler verifies the integrity of the existing system before the actual compilation. Since unconnected slave ports are considered as errors that cause compilation to fail, we connect all FIFO slave ports to the Nios II data master, which has an added benefit for debugging because it allows the software running on the Nios II processor to inspect FIFO contents and inject data into the FIFOs. Debugging of FPGA Brook programs will be discussed in more detail in

Chapter 7. Our compiler also generates wrapper Verilog files for each of the FIFOs, which is necessary for the system to pass the integrity tests performed by the C2H compiler.

As discussed in section 3.5, a script called *create-this-app*, which allows the programmer to run the C2H compilation flow, is generated by our compiler. This script contains commands that invoke the C2H compiler and other scripts necessary to compile the system, including calls to Quartus II synthesis. Our compiler customizes this script by specifying the names of all kernels that should be implemented as hardware accelerators. These names are passed as switches to the script that invokes the C2H compiler.

In this chapter we described the detailed steps involved in compiling FPGA Brook programs into FPGA logic, with particular focus on our source-to-source compiler. In the next chapter we describe how our compiler takes advantage of data parallelism in FPGA Brook programs through kernel replication.

Chapter 5

Exploiting Data Parallelism through Kernel Replication

In the previous chapter we showed how our compiler implements streaming applications in FPGA logic. The kernels, which are implemented as hardware accelerators, operate in parallel in a pipelined fashion, thus exploiting task-level parallelism. However, each accelerator processes stream elements one at a time, meaning that data parallelism is not exploited. Data parallelism exists because stream elements are independent, so they can be processed in parallel. One way to exploit data parallelism is to replicate the functionality of a kernel one or more times, so that each replica processes a part of the input stream. We wish to emphasize that programmers do not need to understand the replication details presented in this chapter, because the process is fully automated by our compiler.

Let us consider the example of the autocorrelation application, whose dataflow graph is shown in Figure 4.2. The programmer may implement the application in FPGA Brook and realize that its throughput is two times lower than desired. The programmer may then instruct the compiler to speed up kernels *mul* and *sum* two times using the speedup pragma statements. Both kernels have to be sped up because they have comparable complexity and thus comparable throughput. In such a case, our compiler will produce a system whose streaming dataflow graph is shown in Figure 5.1. The *mul* kernel has been replaced by two kernels, each of them processing stream elements independently, thus effectively doubling the throughput. To support this functionality, we use a special FIFO that we call the *distributor FIFO*, which distributes stream elements from its input to its outputs in round-robin order. While the distributor in Figure 5.1 has only two outputs, in a general case the distributor can have an arbitrary number of outputs, depending on the replication factor. *Replication factor* is the number of replicas of a kernel, and is equal to the desired speedup for ordinary kernels. The *sum* kernel is a reduction kernel and requires special handling. First, the kernel is replicated two times, just like an ordinary kernel, but an additional replica of the kernel is necessary to combine the partial results from the first two replicas. A *collector FIFO* performs the inverse functionality of the distributor FIFO, gathering elements from its inputs in round-robin order and forwarding them to its output. A collector can have an arbitrary number of inputs. We refer to the graph in Figure 5.1 as the *replicated streaming dataflow graph*. We describe implementation details of components in the graph in the following sections.

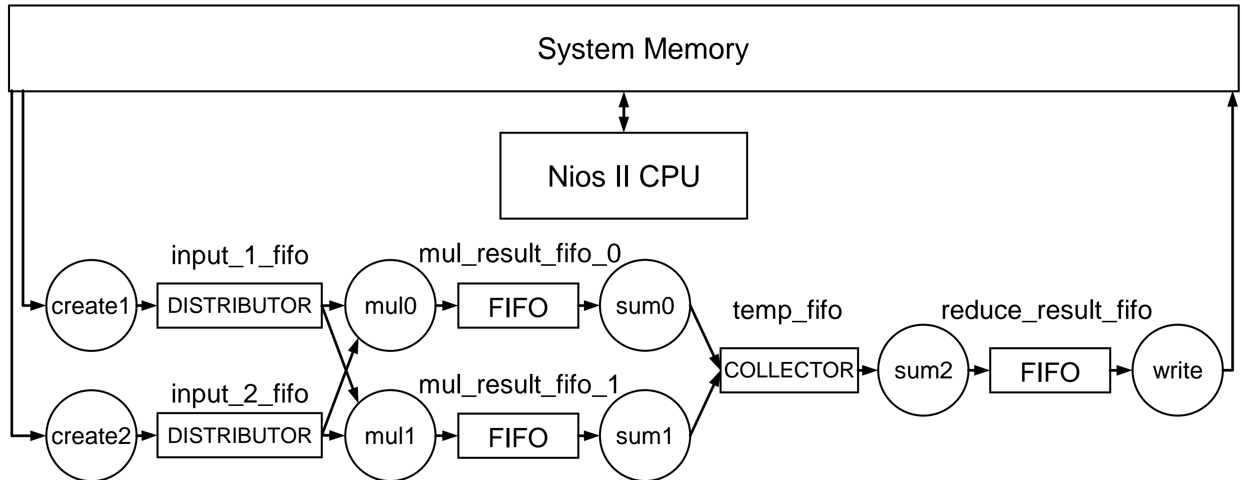


Figure 5.1 Replicated streaming dataflow graph for the autocorrelation application

5.1. Replicating Ordinary Kernels

An ordinary kernel, such as the *mul* kernel in the autocorrelation application, can be replicated by simply creating the number of kernel instances equal to the desired speedup. Ignoring potential synchronization overheads due to round-robin distribution and collection of stream elements, such replication produces the throughput improvement equal to the number of replicas, because replicas perform processing in parallel. The C2H code generated for each of the kernel replicas has to be customized. Each replica has a different initial value of the iterator and the amount the iterator gets incremented by depends on the replication factor. This is relatively straightforward for 1-D streams, where the initial iterator value is assigned sequentially according to the round-robin distribution order, and the amount iterators get incremented by is equal to the replication factor.

Kernels operating on 2-D streams require additional attention because the elements are passed between kernels in a row-major order. Due to this, if the number of columns in the stream does not divide the replication factor, iterator values in the kernel replicas exhibit complex behaviour. Consider an example of the *mul* kernel, where the kernel multiplies two 5x3 streams and the replication factor is 4. Consider the first replica, which should process every fourth stream element, starting with the first. Since stream elements are passed between kernels in a row-major order, this means that the first replica should process stream elements <0,0>, <1,1>, <2,2> and <4,0>. The C2H code generated in such a case is shown in Listing 5.1. The *replica_id* variable contains a unique ID assigned to each replica. The IDs are assigned in the same order the round-robin distribution is performed in, starting with 0 for the first replica. The loop iterators are initialized as shown in lines 9 and 10. These expressions represent basic relations converting

Listing 5.1 C2H code generated for the first replica of the *mul* kernel multiplying 2-D streams

```
void  mul_replica_0() {                                                    (1)
  volatile int  *a;                                                        (2)
  volatile int  *b;                                                        (3)
  volatile int  *c;                                                        (4)
  int  _temp_a, _temp_b, _temp_c;                                          (5)
  int _ROWS = 5, _COLS = 3;                                                (6)
  int _replica_id = 0;                                                     (7)
  int _replication_factor = 4;                                             (8)
  int _iter_0 = _replica_id/_COLS;                                         (9)
  int _iter_1 = _replica_id%_COLS;                                        (10)

  for ( ; _iter_0<_ROWS; ) {                                              (11)
    while (_iter_1>=_COLS) {                                              (12)
      _iter_0++;                                                          (13)
      _iter_1 -= _COLS;                                                   (14)
    }
    if (_iter_0<_ROWS) {                                                  (15)
      for ( ; _iter_1<_COLS; _iter_1+=_replication_factor) {              (16)
        _temp_a = *a;                                                     (17)
        _temp_b = *b;                                                     (18)
        _temp_c = _temp_a * _temp_b;                                      (19)
        *c = _temp_c;                                                     (20)
      }
    }
  }
}
```

coordinates from a one-dimensional iteration space to the two dimensional iteration space, as required to address the two-dimensional stream. This is needed because replica IDs are assigned sequentially (i.e. in one-dimensional iteration space). The code in Listing 5.1 shows that these values are computed at run-time, but this is for illustrative purposes only. Our compiler precomputes these values and substitutes their usage with precomputed values, making them run-time constants.

Initially, the loop iterators for the first replica will both be 0. The column iterator *_iter_1* is incremented by the replication factor in every iteration of the inner loop (line 16). Once the inner loop finishes, the column iterator is out of bounds (i.e. higher than the number of columns) and has to be adjusted to select the correct column index of the next element to be processed. The next column index can be calculated by computing the modulo of the index value with the number of columns. As previously discussed, our compiler avoids using the modulo operation and implements equivalent functionality using counters (lines 12 through 14). The code uses the *while* loop instead of a simple *if* condition in line 12, to handle the case when the value of the column iterator is twice the number of columns or higher. This can occur when a whole row is skipped by the current replica, such as between the elements <2,2> and <4,0> in our example. This can happen only if the replication factor exceeds the number of columns. Concurrently with adjusting the column iterator, we also adjust the row iterator (line 13), one or more times, as appropriate. This adjustment could result in the row iterator getting out of bounds of the stream

dimensions, which means that this replica has processed all the elements assigned to it. The condition in line 15 prevents further processing in such a case.

The code in Listing 5.1 could be simplified by using only a single iterator and setting the loop limit to one quarter of the total number of stream elements (because there are four replicas). However, the code presented in the listing is general enough to handle kernel code using the *indexof* operator, which is why the code maintains correct values of both iterators.

The C2H code generated by our compiler contains one C function for each kernel replica. Each function is implemented as an independent hardware accelerator, which is specified in the compilation scripts generated by our compiler. Additionally, our compiler generates C2H pragma statements that correctly connect replicas to the collector and distributor nodes. Finally, the call to the original kernel in the *main* function is replaced by calls to all replicas, to ensure that all hardware accelerators are activated when the application is implemented in hardware.

5.1.1. Collector and Distributor FIFO Buffers

Collector and distributor FIFOs are necessary to distribute stream elements to the replicas and collect stream elements from the replicas, respectively. The collection and distribution is performed in round-robin fashion. We refer to a collector with n inputs as an *n-to-1* collector, and a distributor with n outputs as a *1-to-n* distributor. Each logical input and output port contains three physical ports. An input logical port contains data input, write request and wait signals, while an output logical port contains read request, data output and wait-request signals. Figure 5.2 shows two possible ways to build a distributor FIFO; using one or multiple FIFO buffers. The distributor in Figure 5.2a contains only one FIFO buffer where the stream elements are written by the source kernel. The elements are passed to the kernel replicas connected to the output ports one at a time, in round-robin fashion, based on the current value of the *selector_counter*. Only one output port of the distributor FIFO can be read from at a time. If a kernel at the output is ready to process the next element, but it is not currently selected for reading, it has to wait until one or more other kernels read their elements, even if the stream element needed by this kernel is available in the FIFO buffer. The distributor in Figure 5.2b addresses this issue by employing multiple FIFOs, one per output port of the distributor. The stream elements are written into one of the FIFO buffers in the round-robin fashion. Each of the kernel replicas can read from its FIFO independently, as long as data is available in the FIFO.

Comparing the two approaches to building distributor FIFOs, the first approach uses less area, but may result in poor performance. This approach is suitable when execution time of the replicated kernel does not vary from one stream element to the next, and the kernel computation time (in cycles) is higher than the minimum number of cycles between two reads on the same output port. If this is not the case, the

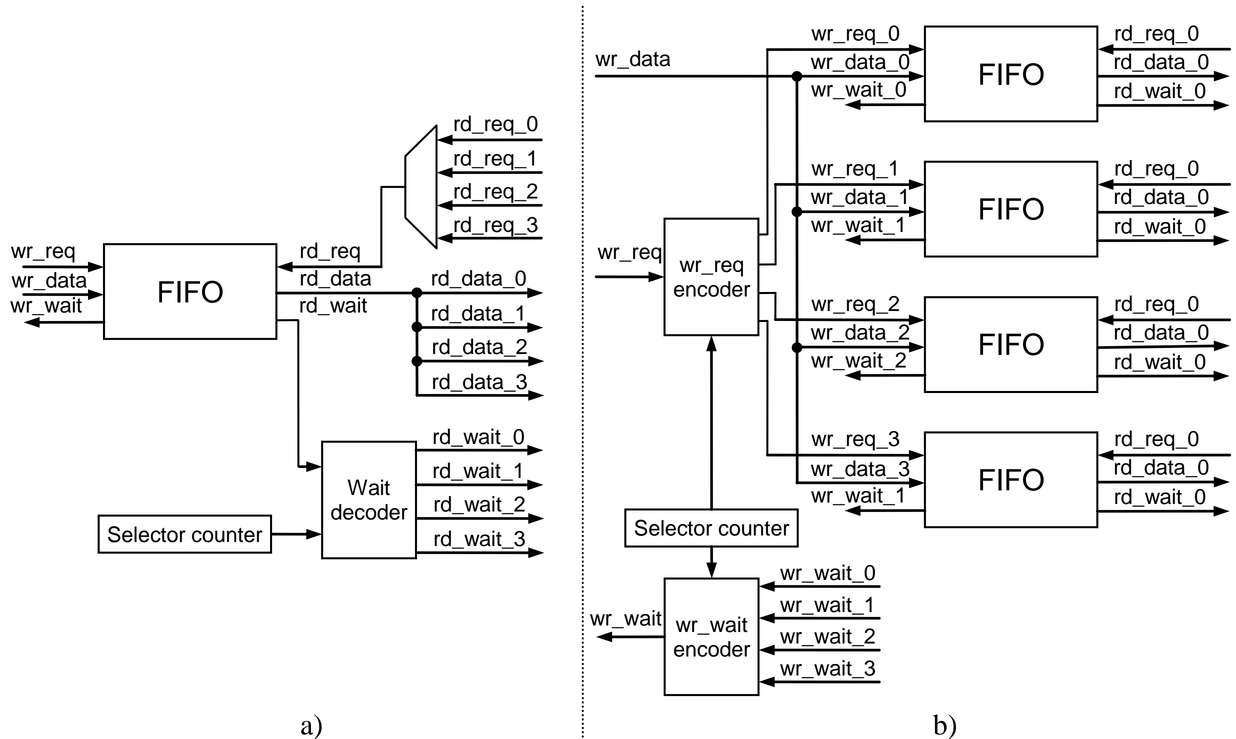


Figure 5.2 Two possible implementations of the 1-to-4 distributor FIFO

second approach should be used. The same principle can be applied when building collector FIFOs, and the same criteria can be used to decide which kind should be implemented. The current version of our compiler supports only collectors and distributors with a single FIFO, because we found that they meet the needs of most applications in our benchmark set. Adding support for the collectors and distributors with multiple FIFOs to our design flow would be straightforward, because of its modular design.

5.1.2. Array FIFO Collectors and Distributors

Collector and distributor FIFOs described in the previous section work well for distributing and collecting elements of ordinary streams. However, applying the same design principle to implement collectors and distributors that transfer streams of arrays does not provide satisfactory performance. This is because the time required to process one stream element is much higher for streams of arrays than ordinary streams. For example, the zigzag kernel, first introduced in section 3.3, operates on stream elements that are arrays of 64 elements. Even if the kernel used a local buffer to store the array, it would take at least 64 clock cycles to transfer the array elements between the FIFO and the local buffer, assuming that reads are performed one element at a time. If the distributor design presented in Figure 5.2a were used to store streams of arrays, it would force all replicas other than the one currently selected for reading to wait for at least 64 cycles until the currently selected replica finishes reading. In addition, it would increase the amount of required memory because of the need for a local buffer, and would also hurt

performance of the kernel replicas because even the selected replica cannot process data while the transfer is in progress. This is because kernel code can access array elements in an arbitrary order, so it has to wait until all the elements have been read, to ensure that all elements are available before starting processing. The distributor design shown in Figure 5.2b when applied to streams of arrays solves both of these problems, but may have unacceptably high cost for large arrays or high replication factors. Instead we consider a modification to the basic distributor design in Figure 5.2a.

The basic distributor design allows only one output to be active at any given time to ensure the correct ordering of output elements. However, since streams of arrays require access to a stream element over a longer period of time, this design choice has to be reconsidered. For example, consider a FIFO buffer that currently contains three arrays, which have been written into it previously. It is possible to design a circuit that would allow three kernel replicas connected to the distributor's outputs to perform reads simultaneously. While such a design seemingly violates the first in, first out principle of the FIFO buffer, this does not necessarily have to be the case when streams of arrays are involved. This is because reading from a stream of arrays consists of two distinct operations. First, data elements are read from the array in an arbitrary order. Then, after all the elements have been read and processing is finished, the array is popped from the FIFO. Therefore, the FIFO ordering can be enforced when the arrays are popped from the FIFO, while allowing parallel access to array elements. This approach allows array elements to be accessed by multiple replicas simultaneously and still enforces the ordering of stream elements. A possible implementation of such an array distributor FIFO is shown in Figure 5.3. The write side of the distributor (the left side in the figure), operates in the same way as an ordinary array FIFO, as described in section 4.3. The following explanation focuses on the read side of the array distributor FIFO (the right side in the figure).

The block diagram in Figure 5.3 shows a 1-to-3 distributor FIFO with four memory slots for storing arrays. To simplify the diagram we do not show all inputs and outputs individually, but instead use the suffix *_x* to denote a set of all inputs or outputs of the same type. For example, *rd_addr_x* is used to replace *rd_addr_0*, *rd_ addr_1* and *rd_ addr_2*, each of them carrying a 6-bit address. The lines carrying such signals are also shown thicker than others in the figure. The distributor uses several counters to keep track of its internal state and provide data to all outputs simultaneously when sufficient data is available in the memory slots. The *pop pointer* points to the memory slot that will be popped from the FIFO next. There is one *output distance* counter for each of the outputs, which keeps track of the distance of each output from the currently selected output. Each output performs reads from a different memory slot. A set of multiplexers is used to select the data from the appropriate memory slot for each of the outputs. The memory slot to be read can be determined by adding the pop pointer and the corresponding output distance. If the number of memory slots is not a power of two, special care has to be taken to handle

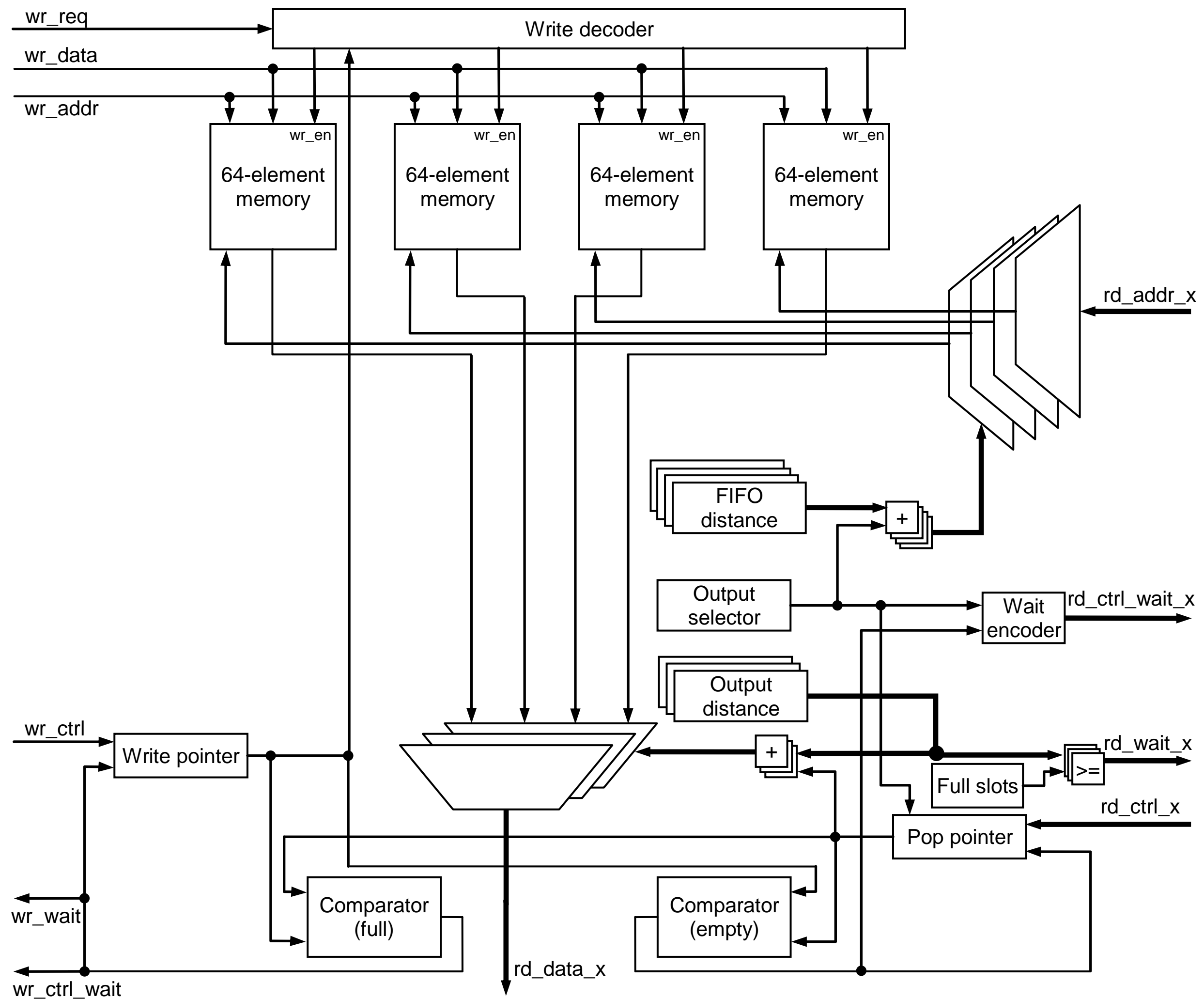


Figure 5.3 Block diagram of an array distributor FIFO

overflow from this addition correctly. The *full slots* counter keeps track of the number of memory slots that currently hold valid data. This is useful to determine which outputs are allowed to perform reads from arrays. Multiple outputs are allowed to read data simultaneously, as long as only one output at a time is allowed to pop data, in the correct order. By comparing the output distance of each output to the number of slots holding valid data, the circuit can determine whether the output can perform reads or not.

The *output selector* counter is used to keep track of the output that is the next in the sequence to pop an array from the FIFO. The pop pointer is only incremented if the output selected by the output selector issues a popping request and the distributor FIFO is not empty. The *wait encoder* block ensures that the wait signal is asserted for all other outputs, based on the value of the output selector. The wait signals are also asserted when the distributor FIFO is empty.

Another set of multiplexers is used to select an address that is forwarded to each of the memory modules. While it may appear that the address can be selected using the same control signals used to select data from memory slots, this is not the case, because of the different number of inputs to these two sets of multiplexers and the different number of multiplexers in each set. Instead, we use another set of counters that we call *FIFO distance*. This set contains as many counters as there are memory slots, and they keep track of the distance of each memory slot from the slot currently selected for popping. The address to be used for a memory slot can be determined by adding the FIFO distance to the value of the output selector, paying special attention to overflow, because the number of outputs may not be a power of two. The addition of *FIFO distance* and *output selector* is performed in modulo arithmetic, so that it can never select a non-existing output. In our example this means that the addition is performed in modulo-3 arithmetic.

In this section we described a possible design for an array distributor FIFO that allows multiple outputs to access data in the FIFO simultaneously, but enforces popping of data from the FIFO in the correct order. We note that similar functionality can be achieved by building a distributor based on the design in Figure 5.2b, with each FIFO holding only one memory slot. Such a design would be limited to containing the number of memory slots that is a multiple of the number of outputs of the distributor. Our design is more general and can contain any number of memory slots that is higher than the number of outputs. A design with fewer memory slots than the number of outputs would not be useful, because it could provide data in parallel to only as many outputs as there are memory slots in the design. Therefore, it would limit parallelism and effectively render any additional kernel replicas connected to outputs useless. Finally, a collector FIFO can be built in a similar fashion as the distributor FIFO described above.

5.2. Replicating Reductions of One-Dimensional Streams

Replicating reduction kernels is more complex than ordinary kernels, because stream elements cannot be considered completely independent, since a sequence of neighbouring elements in the input stream have to be combined to produce an output element. Reduction kernels need to be replicated if the input arrival rate is higher than the throughput of a single, non-replicated reduction kernel and a higher throughput is desired, which means that the reduction kernel is a bottleneck. This may be true if other kernels in the application are replicated, and thus many input elements arrive to the reduction tree simultaneously, or if a complex custom reduction operation is performed.

While the subject of parallelization of reduction operations has been explored before, there are some significant differences between reductions in FPGA Brook and other languages. First, most parallelizing frameworks, such as the Message Passing Interface (MPI), only support reducing an array of values into one value [68]. As previously discussed, FPGA Brook also supports reduction of a stream (1-D or 2-D)

into a smaller stream (1-D or 2-D). Second, in systems like MPI, data to be reduced is distributed across the nodes performing the reduction. In our framework, nodes that perform the reduction receive data one element at a time, through one or more FIFO buffers. Finally, in MPI the programmer decides how many nodes will participate in the reduction operation, and this choice affects performance. Our goal is to automatically select a minimal number of reduction kernel replicas necessary to achieve the desired speedup.

Consider the *sum* reduction kernel in the autocorrelation application. As shown in Figure 5.1, if round-robin distribution of stream elements is used, creating two replicas of the reduction kernel is not sufficient. A third replica is necessary to combine the partial results computed by the first two replicas. This is because the round-robin distribution assigns different stream elements contributing to one output element to different replicas. Another option would be to not use the round-robin distribution, and assign all stream elements contributing to one output element to one replica. This approach is not satisfactory when the reduction ratio is high, because the number of consecutive elements assigned to one replica would be large, which could lead to other kernel replicas being idle, thus effectively negating any benefits from replication. Considering that the round-robin distribution is also used for ordinary kernels, we chose to use the round-robin distribution for reduction kernels as well, and instantiate as many replicas as necessary to achieve the desired throughput improvement.

The sum kernel in the autocorrelation application, whose code was shown in Listing 3.1, reduces the whole stream into an output stream with only one element. When the kernel is not replicated, one hardware accelerator processes all input elements. In the replicated case shown in Figure 5.1, the two replicas, *sum0* and *sum1*, process stream elements in parallel, so the throughput is doubled compared to the non-replicated case. The last replica (*sum2*) adds the outputs of the first two replicas only once, so its throughput is much higher than that of the other two replicas. While determining the throughput improvement is easy in this case, it is more complex in general.

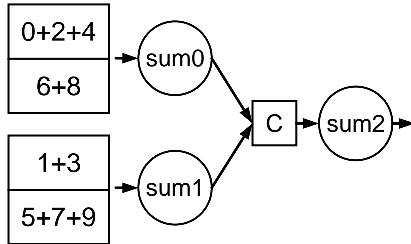
5.2.1. Building reduction trees

Consider a reduction that takes a twenty-element stream on the input and produces a four-element stream on the output (20-to-4 reduction). For simplicity, assume that the reduction operation is a summation, and assume that the programmer specified that it should be sped up 2 times, relative to the non-replicated reduction. A natural choice for parallelizing reductions in general is to build a reduction tree [69], like the simple tree with three nodes we built for the *sum* reduction kernel of the autocorrelation example. Let us start by using the same reduction tree in this example. Since the desired speedup (2) does not divide the reduction ratio (5) evenly, the round-robin distribution of input elements will assign the members of the input to the leaves of the tree (i.e. the first two replicas, *sum0* and *sum1*) unevenly. This is illustrated in Figure 5.4.

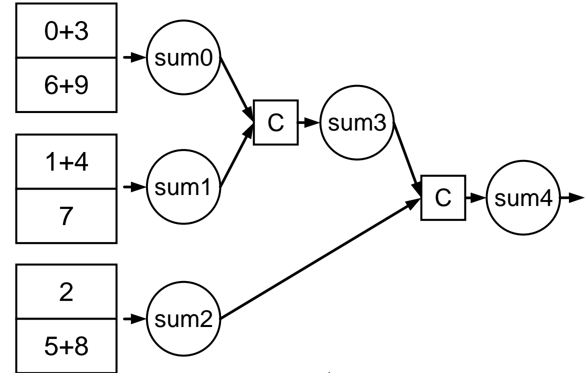
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

0+1+2+3+4	5+6+7+8+9	10+11+12+13+14	15+16+17+18+19

a)



b)



c)

Figure 5.4 Examples of a reduction operation and corresponding reduction trees

Figure 5.4a depicts the input stream of 20 elements, labelled by their index, and the resulting stream of 5 elements, labelled by the sum of indices of the appropriate input stream elements. Figure 5.4b shows how the first ten input elements are distributed between the replicas. To compute the first element of the output, the *sum0* kernel contributes by adding three input elements, while the *sum1* kernel contributes by adding two input elements. The roles are reversed when the second element of the output is being computed. Considering that the two kernels alternate between processing 3 and 2 input elements per output element, on average they process 2.5 elements. Since the original, non-replicated kernel had to process 5 input elements per output element, replication has doubled the throughput on average, assuming that there is no variation in processing times between different input elements. However, this is only true if the average throughput is of interest. If we want to ensure that every element of the output is produced at least twice as fast as in a non-replicated case, no node in the reduction tree should process more than 2.5 input elements per one output element. Since nodes can only process whole elements, this means that no node in the tree should process more than 2 elements. The appropriate reduction tree is shown in Figure 5.4c, along with the distribution of the first ten input elements. The same reduction tree would be appropriate to increase the throughput 2.5 times. When computing the first element of the output the *sum2* kernel replica processes only one input element, which means that it simply passes the value from the input to the output. While it may seem that this replica is redundant, this is not the case because it processes two input elements to compute the second element of the output. This is because replicas alternate between processing one and two elements of the input per output element. This means that, on

average, these replicas process fewer than two input elements per output element, so it may appear that the speedup of this reduction tree is greater than 2.5. However, replicas *sum3* and *sum4* still process two input elements per output element, so they limit the overall speedup to 2.5.

Since the reduction tree in Figure 5.4c has three leaves, two additional nodes are necessary to completely calculate the reduction. One node is not sufficient because it would have to process data from the three leaf nodes, so it would be processing three elements from its input per one output element, which does not meet the throughput requirement. Throughout this chapter we use the term node to denote a kernel, or its replica in the streaming dataflow graph. A node can also refer to one of the collectors or distributors, which are depicted by squares in the graphs. Collector nodes are also labelled by letter C, and distributor nodes by letter D.

In a general case, given a reduction ratio R and desired speedup S, no node in the tree should process more than $floor(R/S)$ input elements to produce one output element. The algorithm for building a reduction tree in such a case has two high-level steps. In the first step, $ceil(R/floor(R/S))$ leaf nodes are created, so that each node processes at most $floor(R/S)$ input elements per output element. The rest of the tree is built so that each node has at most $floor(R/S)$ incoming edges. A node with multiple incoming edges is preceded by a collector FIFO, which allows all hardware accelerators implementing the kernel replicas to have the same I/O structure.

While building the reduction tree we wish to minimize the number of nodes in the tree and minimize the tree depth. Both goals can be achieved by employing the combinatorial merging algorithm, due to Golumbic [70], which is a generalized version of the well known Huffman algorithm [71]. The combinatorial merging algorithm applies generally to any set of integer weighted nodes Q, with $|Q|$ elements, that are to be mapped to an *n-ary* tree, such that the set of nodes Q become leaves of the tree. An *n-ary* tree is a tree in which no node has more than n children. The optimal tree can be built by finding a number q, such that $|Q| = q + k(n - 1)$, where $2 \leq q \leq n$ and $k > 0$. Once such a number is found, the algorithm proceeds as follows. In the first iteration q nodes from Q with the lowest weights are connected to a new parent node. These nodes are then removed from the set Q and the new parent node is added to the set Q. In all subsequent iterations, n nodes from Q with the lowest weights are connected to a new parent node. These nodes are then removed from the set Q and the new parent node is added to the set Q. In each iteration, the parent node is assigned a weight that is one higher than the weight of a child with the highest weight. The algorithm terminates when only one node is left in the set Q, which becomes the root of the tree. The choice of the number q guarantees that there are exactly n nodes available to be connected to a new parent in every iteration after the first. The algorithm optimizes the number of nodes in the tree, and it minimizes the weight of the root node, where weight of a node is defined as being higher by one than the maximum weight of all of its children. Therefore, if all leaves are initially assigned equal weights, the algorithm minimizes the tree depth.

Listing 5.2 Pseudocode for the reduction tree building algorithm

```
BuildReductionTree (R, S)
    n ← floor(R/S)
    create ceil(R/n) leaf kernel replica nodes
    initialize a priority queue Q with the leaf nodes, assigning each node weight of 1
    compute q, such that |Q| = q + k(n - 1), where 2 ≤ q ≤ n and k > 0
    t ← q
    repeat
        create a new collector node C with t inputs
        remove t nodes with lowest weight from Q and connect them to the inputs of C
        create a new kernel replica node P
        connect output of C to P
        weight(P) ← max(weight of all inputs of C) + 1
        Q ← Q + P
        t ← n
    until |Q| = 1
end
```

The algorithm for building a reduction tree can be summarized by the pseudocode shown in Listing 5.2 . The algorithm takes two parameters: reduction ratio R and desired speedup S. The nodes in the tree are replicas of the reduction kernel, except for the collector nodes, which are added to connect children nodes to their parent node. For the purpose of the combinatorial merging algorithm, the collector nodes do not increase the weight of the parent node because they do not perform any computation.

The maximum possible speedup can be achieved by building a tree in which each node processes only two input stream elements per one output stream element. A node cannot process fewer than two input elements because reduction operations combine multiple input elements, so the farthest they can be broken down to is processing two elements at a time. Considering that in the non-replicated case the reduction kernel processes R input elements per one output element, where R is the reduction ratio, this means that the maximum speedup achievable by a reduction tree is $R/2$. However, if the reduction kernel is the bottleneck in the application even after building the largest possible reduction tree, a higher speedup may be required. In such a situation, the kernel preceding the reduction delivers stream elements at a higher rate than the reduction tree can withstand. While building a larger tree does not bring any benefits, we observe that since input stream elements arrive at a much higher rate than one reduction tree can process, it may be possible to compute two or more elements of the output in parallel. This can be achieved by building multiple reduction trees, each operating independently.

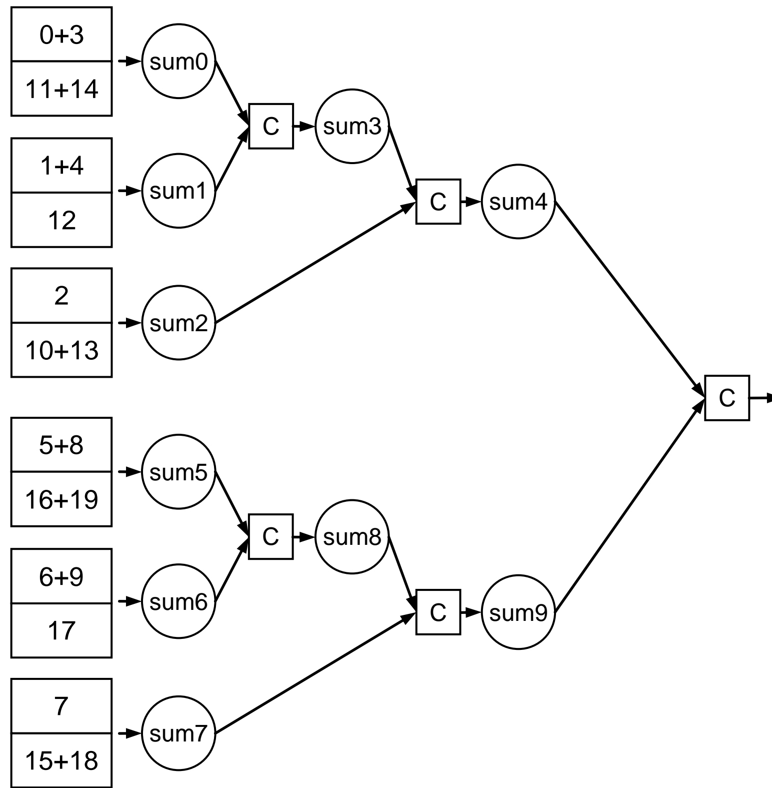


Figure 5.5 An example of two reduction trees operating in parallel

An example of two trees built for the 20-to-4 reduction we discussed earlier is shown in Figure 5.5. Since the goal is for each tree to compute output elements independently, ordinary round-robin distribution of input elements among the leaves of the two trees is not appropriate, because it would distribute the elements that contribute to one output to two different trees. Therefore, we implement "double round-robin" distribution, in which the first R elements are distributed in round-robin fashion to the first tree, next R elements in the same fashion to the next tree, and so on, with trees themselves being selected in round-robin fashion. In the example in Figure 5.5, the first 5 elements are assigned to the first tree, while the second 5 elements are assigned to the second tree. Next 5 elements are assigned again to the first tree, starting with the node where the distribution for the first tree left off. This means that the element with index 10 is assigned to the node *sum2*, because the last element assigned to this tree (index 4) was assigned to the node *sum1*. This approach enables each tree to operate in the same way in a multiple-tree configuration as in a single-tree configuration. Finally, outputs of the trees are collected in round-robin order by the collector node at the root of the tree structure. Although the method for building multiple trees we have just described effectively produces a single larger tree, we will refer to this configuration as multiple trees, to distinguish it from the previously described configuration. An easy way

to distinguish between the two cases is by observing that the root of a reduction tree is always a kernel replica, while the root of the multiple-trees configuration is always a collector node.

5.2.2. Double-Distributor FIFOs

The configuration using multiple trees, described in the previous section, requires elements to be assigned to the trees in the round robin fashion. Additionally, elements are distributed to the leaves within each tree in the round robin fashion. This necessitates a special hardware module, because the ordinary distributor FIFO does not have such a capability. We call this hardware module the *double-distributor FIFO*. The block diagram for a double-distributor FIFO is shown in Figure 5.6. The double-distributor FIFO shown in the figure is designed for the two trees in Figure 5.5. Therefore, it has six logical outputs; three for each of the two trees. The module uses three counters to determine which output port data should be sent to: tree selector counter, leaf selector counter and element counter. The tree selector counter contains the index of the currently selected tree, while the leaf selector counter keeps track of the index of the currently selected leaf node within the tree. Element counter is an auxiliary counter, which counts how many elements of the input have been processed.

To explain the functionality of the module, we use the example of the 20-to-4 reduction discussed in the previous section. Therefore, the reduction ratio (R) is 5. The element counter is a modulo counter that counts from 0 to 4, or from 0 to $(R-1)$ in the general case. This counter reaches its maximum value when all the input elements contributing to one output element have been processed and a new sequence should start. At that point, the tree counter is advanced by 1 and the leaf counter is reset. The tree counter is another modulo counter that counts from 0 to one less than the number of trees. In our example with two trees, the tree counter can only contain values 0 and 1. The leaf counter is also a modulo counter that counts from 0 to one less than the number of leaves in the tree. In our example, each tree has three leaves, so the leaf counter counts from 0 to 2. Given that the element counter counts from 0 to 4, this means that the first five values of the leaf counter will be 0, 1, 2, 0, 1. This sequence is exactly what is needed to obtain the assignment of elements as depicted in Figure 5.5.

As discussed in the previous section, once a tree is selected for the second and all subsequent times, the round robin distribution among the leaves of the tree has to continue where it left off the last time this tree was selected. To support this functionality, the double-distributor module uses the saved leaf-selector register, as shown in Figure 5.6. This register holds the next value that the leaf counter should contain when the tree is next selected. Therefore, whenever the element counter reaches its maximum value, the leaf counter is not reset to 0, but rather loaded with the value from this register. While it may appear at first that we require one such register for each tree, this is not the case. If one observes that each tree goes through an identical sequence of leaf indices, one can realize that all the trees have to start with the same

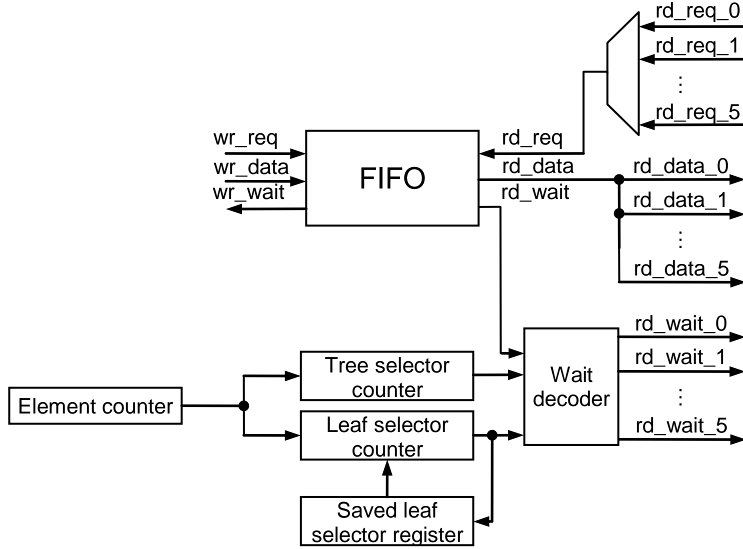


Figure 5.6 Datapath for a double-distributor FIFO

leaf index in the next traversal through the tree sequence. Thus, if the updates to this register are carefully timed, one register is sufficient and its value can be used by all the trees in the sequence. Finally, the wait decoder is a simple decoder that sets the appropriate wait signals depending on the currently selected tree and the leaf within the tree. Verilog code for all double-distributors is automatically generated by our compiler, with appropriate number of outputs and appropriate parameters, such as the reduction ratio, number of trees and number of leaves. C2H pragma statements are used in the generated C2H code to connect them to the kernel replicas and other components in the system.

5.2.3. Code Generation for Reduction Kernel Replicas

The code generated for replicated reduction kernels has to be custom-generated for each replica. Let us first consider the non-leaf replicas in the reduction tree. Each non-leaf replica performs a simple *t-to-1* reduction, where *t* is the number of the node's children, or more precisely, the number of children of the preceding collector node in the reduction tree. Leaf replicas have to be handled differently. Given a reduction whose reduction ratio is *R,* and which is implemented by a reduction tree with *L* leaves, there are two possible cases to be considered. If the reduction ratio is divisible by the number of leaves in the reduction tree, each leaf node performs a simple *(R/L)-to-1* reduction. However, if the reduction ratio is not divisible by the number of leaves, the code generated for leaf nodes has to take into account that the number of input elements processed per output element changes periodically. To produce the first output element, the first *R%L* nodes process *ceil(R/L)* elements, while the remaining nodes process *floor(R/L)* elements. To produce the next output element, the next *R%L* nodes process *ceil(R/L)*, where the next nodes are selected modulo *L*. To simplify the following discussion, let us call the process of producing

one output element a *step*. Consequently, in the example in Figure 5.4c, where the reduction ratio (*R*) is 5 and the number of leaves (*L*) is 3, in the first *step*, the first two leaves (*sum0* and *sum1*) process two input elements, while the third leaf (*sum2*) processes only one input element. In the next step, the next two leaves process two elements, where the next two leaves are the last and the first (*sum2* and *sum0*), while the leaf *sum1* processes only one element. Our compiler inserts necessary modulo counters into the generated C2H code to keep track of the number of inputs to be combined in each step, as illustrated in Listing 5.3. The structure of the code is similar to the non-replicated reduction kernel, shown in Listing 4.3, with additional counters to keep track of the changing number of elements to process. The number of elements of input that have to be processed in each step is contained in the variable *_reduction_ratio* in

Listing 5.3 C2H code generated for the sum0 reduction kernel replica

```
void  sum_replica_0 () {                                                    (0)
  volatile int  *a, *r;                                                     (1)
  int   _temp_a, _temp_r;                                                   (2)
  int _iter, _mod_iter=0;                                                   (3)
  int _reduction_ratio;                                                     (4)
  int _replica_id = 0;
  int _total_elements = 7;                                                  (5)
  int _helper = 0;                                                          (6)
  int _helper_next = 2;                                                     (7)

  for (_iter=0; _iter < _total_elements; _iter++, _mod_iter++) {            (8)
    if (_helper < _helper_next)                                             (9)
      condition = (_replica_id>=_helper) && (_replica_id<_helper_next);    (10)
    else                                                                   (11)
      condition = (_replica_id>=_helper) || (_replica_id<_helper_next);    (12)
    if (condition)                                                         (13)
      _reduction_ratio = 2;                  // i.e. = ceil(R/L)           (14)
    else                                                                   (15)
      _reduction_ratio = 1;                  // i.e. = floor(R/L)          (16)

    if (_mod_iter == _reduction_ratio)                                     (17)
      _mod_iter = 0;                                                       (18)
    if ((_mod_iter == 0) && (_iter != 0)) {                                (19)
      *r = _temp_r;                                                        (20)
      _helper = _helper_next;                                              (21)
      if (_helper_next < 1)                                                (22)
        _helper_next = _helper_next + 2;      // i.e. + (R%L)              (23)
      else                                                                 (24)
        _helper_next = _helper_next + (-1);   // i.e. + (R%L - L)          (25)
    }
    _temp_a = *a;                                                          (26)
    if (_mod_iter == 0)                                                    (27)
      _temp_r = _temp_a;                                                   (28)
    else                                                                   (29)
      _temp_r = _temp_r +_temp_a;                                          (30)
  }
  *r = _temp_r;                                                            (31)
}
```

the code, because from the perspective of individual replicas, it is the reduction ratio that is changing. This variable should not be confused with the value of reduction ratio *R* we use when describing the original reduction specified by the programmer, which is what the reduction tree is computing as a whole. Putting aside the additional counters, the code in Listing 5.3 uses a modulo iterator *_mod_iter* to determine when the correct number of inputs have been processed (lines 17 and 18), and an output value should be produced (lines 20 and 31). A new reduction is started when the modulo counter overflows (line 28), otherwise the current element is added to the running sum (line 30).

The code in Listing 5.3 contains two additional variables to help determine the number of input elements to process in each step: *_helper* and *_helper_next*. While the basic concept behind determining the number of elements to process by a leaf node is simple, its implementation using only automatically generated counters is more complex. Since replicas have to operate completely independently, each replica has to determine the number of elements to process independently. To better understand how the code in the listing implements this functionality, let us consider two different examples of replicated reductions. The first example is the 20-to-4 reduction implemented as a reduction tree shown in Figure 5.4c. The assignment of the first 15 input elements to the leaves of the tree are shown in the top table in Figure 5.7a. The input elements contributing to different output elements are shaded differently. The bottom table in Figure 5.7a shows how many input elements each leaf processes in each step. For example, the *sum0* replica (second row in the bottom table) processes two elements in the first two steps, and one element in the third step. This sequence repeats until all output elements have been produced. The length of the sequence is equal to the number of leaves ($L=3$), because the elements are assigned to leaves in round-robin fashion. Therefore, after L outputs have been produced, the total number of input elements assigned to the tree is a multiple of L, so the distribution now starts again from the first leaf node (*sum0*).

In this example the sequence determining the number of inputs being processed by a leaf node in each step appears simple, but it can be more complex in a general case. Consider a 120-to-10 reduction ($R=12$) which has been replicated in a way that produces a replication tree with 5 leaf nodes ($L=5$). Since the reduction ratio is not divisible by the number of leaf nodes, the nodes have to periodically change how many input elements they process. The assignment of the first 60 input elements to the leaves is depicted by the top table in Figure 5.7b. From this assignment it is easy to derive the number of input elements each leaf node processes, as shown by the bottom table in the figure. By observing the rows of this table, we can see that the sequence determining the number of inputs being processed by a leaf node in each step is more complex than in the previous example; it requires careful analysis.

As previously noted, in a general case, the first *R%L* nodes process *ceil(R/L)* elements, while the remaining nodes process *floor(R/L)* elements in the first step. In the next step, the next *R%L* nodes process *ceil(R/L)* elements, where the next nodes are selected modulo *L*, and the pattern continues until all

sum0	0	3	6	9	12
sum1	1	4	7	10	13
sum2	2	5	8	11	14

Step	0	1	2
sum0	2	2	1
sum1	2	1	2
sum2	1	2	2

a)

sum0	0	5	10	15	20	25	30	35	40	45	50	55
sum1	1	6	11	16	21	26	31	36	41	46	51	56
sum2	2	7	12	17	22	27	32	37	42	47	52	57
sum3	3	8	13	18	23	28	33	38	43	48	53	58
sum4	4	9	14	19	24	29	34	39	44	49	54	59

Step	0	1	2	3	4
sum0	3	2	3	2	2
sum1	3	2	2	3	2
sum2	2	3	2	3	2
sum3	2	3	2	2	3
sum4	2	2	3	2	3

b)

Figure 5.7 Keeping track of the number of inputs processed by leaf kernels for two different reduction replication scenarios

the elements have been processed. One way for a leaf node to keep track of how many elements it should process, is to maintain a modulo-*L* counter, which is incremented by *R%L* each time an output is produced. For example, for the reduction depicted in Figure 5.7b, with R=12 and L=5, the counter would produce a repeating sequence 0, 2, 4, 1, 3. Assuming that each leaf node contains an ID corresponding to its name (e.g. leaf *sum3* has an ID of 3), a leaf node can then compare its ID with the value of the modulo-L counter and determine the number of input elements to process. If a leaf node determines that its ID matches the modulo-L counter, or is within R%L from the counter, then it needs to process *ceil(R/L)* elements, otherwise it should process *floor(R/L)* elements in the current step. For the example in Figure 5.7b, this means that if the replica ID equals or is one greater than the modulo counter, it should process 3 elements, otherwise it should process 2 elements in the current step. As an example, the modulo counter is 0 in the first step. The *sum0* ID matches the counter and *sum1* ID is one greater than the counter, so they both process 3 elements. Other IDs are more than 1 greater than the current value of the modulo counter, so they process only 2 elements in the current step. In the second step, the modulo-L counter effectively selects the next two leaf nodes to process 3 elements. The nodes *sum2* and *sum3* compare their IDs to the counter value and determine that they are within the prescribed range and thus only these nodes process 3 elements. In the third step the modulo-L counter's value is 4, thus selecting nodes *sum4* and *sum0* (0 is one greater than 4 in modulo-5 arithmetic). To summarize, given a replicated reduction with reduction ratio *R* and replication tree with *L* leaves, where *L* does not divide *R*:

1. Leaf node replicas are labelled by IDs in the order that the round-robin distributor assigns elements to them, starting with 0.
2. Each leaf node replica maintains a modulo-L counter, which starts at 0 and is incremented by *R%L* in every step (i.e. for every output value produced).
3. In each step, a node compares its ID with the current value of the modulo-L counter. If the ID equals the counter value, or is greater than the counter value by less than *R%L*, the node processes *ceil(R/L)* input elements in the current step. Otherwise the node processes *floor(R/L)* elements in the current step. The comparison is performed in modulo-L arithmetic.

Most of the expressions in the above procedure are constants and can be precomputed by the compiler. The only non-trivial operation is the comparison in modulo-L arithmetic, where we are trying to determine whether the node's ID is within the range *[counter, (counter+R%L)%L)* in modulo-L arithmetic (square bracket denotes that the range includes the first number in the range). This comparison can be done by performing one of the following two comparisons:

$$ID \geq counter \land ID < \big((counter + R\%L)\%L\big), if\ counter < \big((counter + R\%L)\%L\big)$$
$$ID \geq counter \lor ID < \big((counter + R\%L)\%L\big), if\ counter > \big((counter + R\%L)\%L\big)$$

In the first line, the lower bound (i.e. *counter*) is lower than the higher bound (i.e. *(counter+R%L)%L*) of the range, so a normal comparison can be performed by checking that the ID is within the two bounds ($\land$ is the logical *AND* operator). However, because of the modulo arithmetic the lower bound may be higher (in ordinary arithmetic terms) than the higher bound. In such a case, the ID is within the bounds if it is higher than or equal to the lower bound, or lower than the higher bound in ordinary arithmetic terms ($\lor$ is the logical OR operator). Going back to the example in Figure 5.7b, with R=12 and L=5, the modulo counter produces a sequence 0, 2, 4, 1, 3. When the lower bound is 0 and the higher bound is 2, the first condition ($ID \geq 0$ and ID < 2) correctly selects nodes 0 and 1. However, when the lower bound is 4, and the higher bound is 1, the second condition ($ID \geq 4$ or $ID < 1$) has to be used to select nodes 4 and 0.

The procedure described above is implemented in the code in Listing 5.3. The modulo-L counter is contained in the variable *_helper*. The counter is updated to the new value every time a new output is produced (line 21). Instead of updating it directly, the program maintains another counter, *_helper_next*, which always contains the value the modulo-L counter should acquire in the next step. This is because the next value of the counter is also equal to the higher bound used to evaluate the condition that determines the number of input elements the replica node should perform in the current step. Therefore, we maintain this value in a separate counter (*_helper_next*), so that it can be reused, which saves some hardware resources. This counter is updated in lines 22 through 25. Since it is a modulo counter, it is incremented by 2 (i.e. *R%L*), unless this operation would overflow in the modulo-L arithmetic. In such a case, we

subtract 1 from the current value of the counter (i.e. add *R%L* and then subtract *L*) to obtain the correct counter value. The comparison between the counter and the replica ID is performed in lines 9 through 12, while the selection of the correct number of inputs to process is done in lines 13 through 16.

The final matter that remains to be explained is the total number of elements the leaf node processes, denoted by the *_total_elements* variable in the code in Listing 5.3. The logic behind determining this value again relies on the fact that input elements are distributed in round-robin order. Given an *X-to-Y* reduction implemented as a reduction tree with *L* leaves, *X* elements have to be distributed to *L* leaves, meaning that each leaf processes a total of *X/L* elements if *L* divides *X*. Otherwise, the first *X%L* leaf nodes process *ceil(X/L)*, while the rest of the nodes process *floor(X/L)* elements. Considering our example of 20-to-4 reduction and 3 leaves, the first two leaves (*sum0* and *sum1*) process 7 elements, while the third leaf (*sum2*) processes 6 elements in total. This explains why the *_total_elements* variable in the code is set to 7 in line 5 of the code. A similar calculation is performed if replication results in multiple trees, because elements are assigned to trees in the round-robin fashion as well.

The code in Listing 5.3 has been altered significantly compared to the code generated by the compiler to make the explanations easier. While the code in the listing represents accurately the operations performed by the compiler-generated code, the generated code is usually more cryptic.

5.3. Replicating Reductions of Two-Dimensional Streams

As discussed in section 4.2.2, two-dimensional reductions are more challenging to implement than their one-dimensional counterparts. Similarly, replicating reduction kernels that operate on 2-D streams is more challenging than replicating 1-D reduction kernels. The problem with replicating 2-D reductions is that round-robin distribution may distribute elements of the input contributing to one output to many different nodes. This is demonstrated in Figure 5.8. Figure 5.8a shows a <4,6>-to-<2,3> reduction, while Figures 5.8b and 5.8c show round-robin distribution of the input stream elements to two and three replicas, respectively. The purpose of this is to observe the distribution of input elements to the replicas and see how processing should proceed. While the distribution shown in Figure 5.8b suggests that we could simply build a reduction tree, as in the one-dimensional case, Figure 5.8c reveals that distribution of elements is in general more complex. For example, elements with indices 0, 1, 6, and 7, which all contribute to one output element, are assigned to replicas *sum0* and *sum1*, meaning that outputs of these two replicas have to be combined to produce the final result. However, elements with indices 2, 3, 8, and 9, which contribute to a different output element are assigned to replicas *sum0* and *sum2*, meaning that outputs of these two replicas have to be combined. Similarly, the distribution of elements 4, 5, 10, and 11 necessitates outputs of *sum1* and *sum2* to be combined. This means that the reduction algorithm would

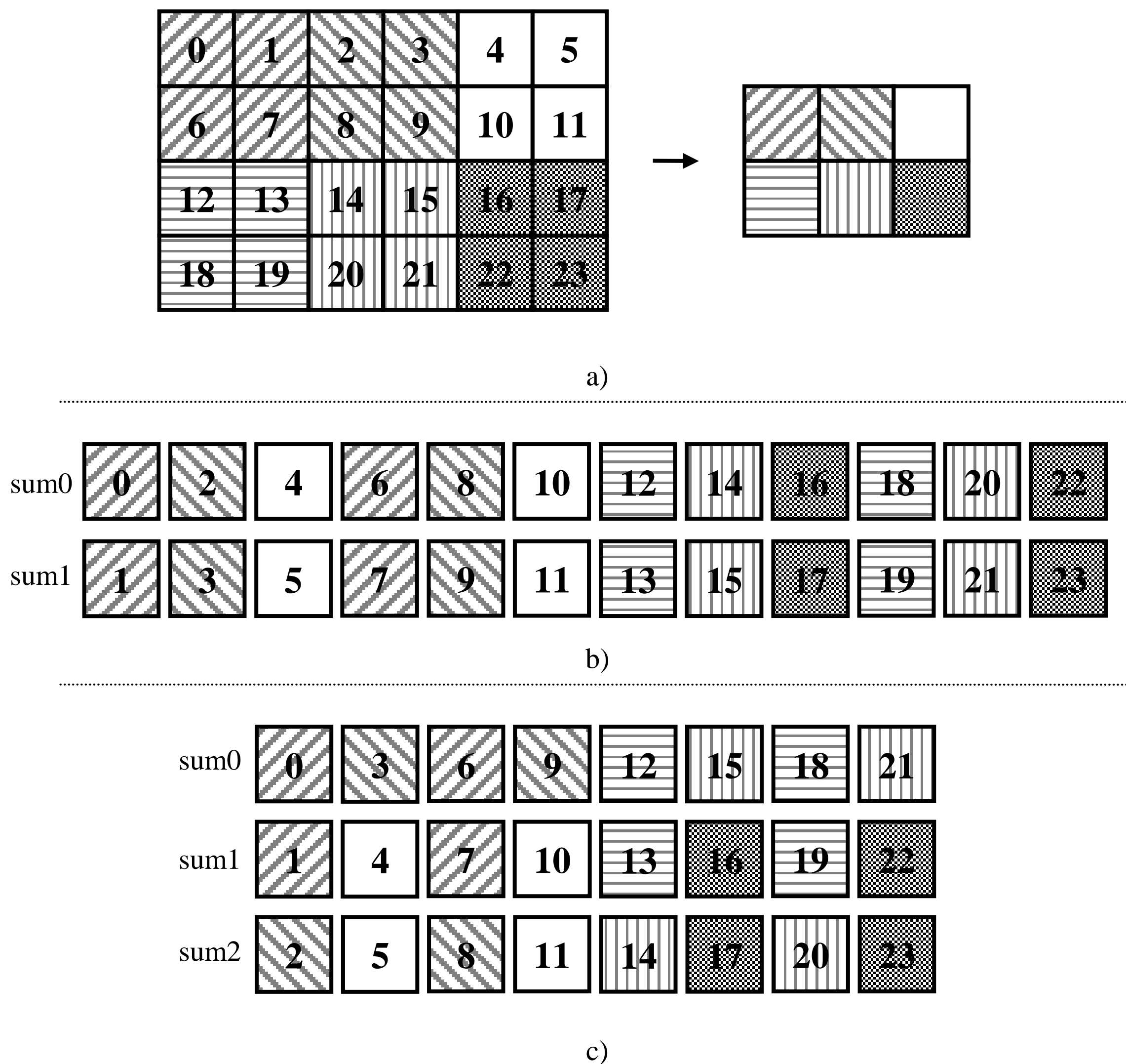


Figure 5.8 Round-robin distributions of a 2-D stream to be reduced

have to dynamically choose the replicas whose outputs should be recombined. Since such an operation spans multiple replicas, this cannot be done in C2H code, but would require specialized hardware support. While it is feasible to build such hardware, a simpler solution is desirable.

An important step in this direction is an observation that a 2-D reduction can in the general case be decomposed into two reductions. An *<X,Y>-to-<A,B>* reduction can be decomposed into two phases: *<X,Y>-to-<X,B>* reduction and *<X,B>-to-<A,B>* reduction. In other words, the first phase combines appropriate stream elements within rows, while the second phase takes the output of the first phase and combines its elements within columns. For example, a <4,6>-to-<2,3> reduction can be decomposed into <4,6>-to-<4,3> and <4,3>-to-<2,3> reductions, as depicted in Figure 5.9. While some throughput

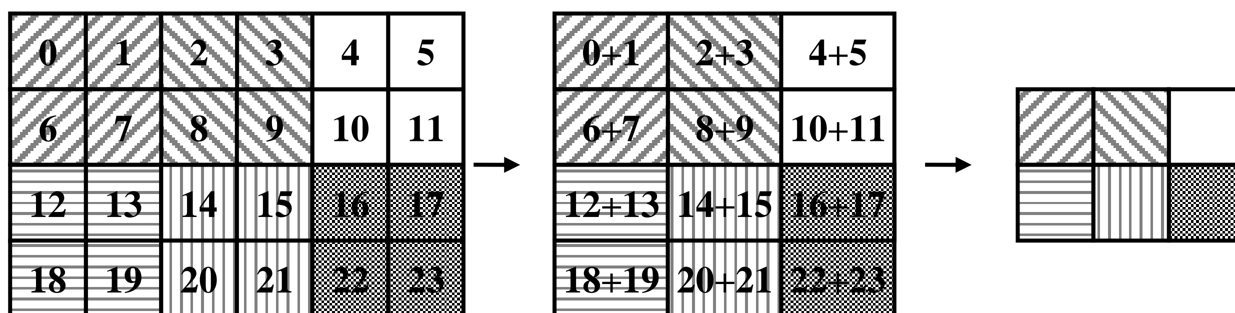


Figure 5.9 Decomposing a reduction into two phases

improvement can be achieved simply by implementing each phase as a separate hardware accelerator, further improvements are possible by further replicating each of the two phases. Since elements of the input stream arrive in a row-major order, the first phase is equivalent to 1-D reduction with $X*Y$ elements on the input and $X*B$ elements on the output. Therefore, it can be replicated as any other 1-D reduction, by building one or more reduction trees. The remaining challenge is to replicate the second phase when needed.

The second phase performs a reduction in the *<X,B>-to-<A,B>* format, meaning that it only has to combine elements within columns. Replicating the second phase may lead to the same problem we described earlier, where elements of the input contributing to one output are distributed to many different nodes. However, replicating the second phase a number of times that is equal to the number of columns (B) results in a round robin distribution that assigns one column to each replica. In such a case, each replica performs an *X-to-A* one-dimensional reduction and each replica produces A elements of the output. Furthermore, replicating the second phase a number of times that divides the number of columns, assigns the elements in the same column to the same replica, although one replica processes multiple columns in such a case. Assuming that the second phase, which performs *<X,B>-to-<A,B>* reduction is replicated Z times, where $B=k*Z$, each replica effectively processes k columns of the original *<X,B>* stream. This in turn means that each replica implements an *<X,k>-to-<A,k>* reduction.

In the example depicted in Figure 5.9, where the second phase implements <4,3>-to-<2,3> reduction, the second phase can only be replicated three times, with each replica implementing a <4,1>-to-<2,1> reduction. The limitation of this 2-phase approach is that the second phase can only be replicated a number of times that divides the number of columns. This may be a problem if the number of columns of the stream is, for example, a large prime number. A programmer aware of this limitation could work around it by padding the stream with additional columns initialized with data that does not affect the final result (for example, zeros in case of summation), which would provide more options for the replication

factor. The advantage of the two-phase approach is that it results in much simpler hardware than alternative implementations.

There are two special cases of two-dimensional reduction that do not require the two-phase approach described above. The first is an $<X,Y>$-to-<1,1> reduction, which means that the output has only one element. This reduction is equivalent to a one-dimensional ($X*Y$)-to-1 reduction, and can be implemented as such. The second special case is an <X,Y>-to-<X,B> reduction, meaning that the number of rows in the input and output are the same. This is equivalent to the first phase described above and can be implemented as a one-dimensional $X*Y$-to-$X*B$ reduction. Other forms of 2-D reductions are handled using the two-phase approach described above.

Throughput improvement in all cases described in this section results from kernel replicas processing fewer input elements per one output element. We discuss speedup achievable through replication more formally in the next section.

5.4. Theoretical Limits to Throughput Improvement due to Replication

In this section we discuss throughput improvement achievable by the different replication scenarios described in the previous sections. Understanding the theoretical limits is important for two reasons. First, the programmers can know the limits of achievable performance without having to perform exhaustive experimental evaluation. If a throughput speedup that is required for an application is theoretically feasible, the programmer can specify such speedup and run a single experiment to verify the performance. On the other hand, if the required speedup is not theoretically feasible, they can request the maximum theoretically feasible speedup and focus on other ways to further optimize performance. The second reason why understanding these limits is important is that our compiler has to produce minimal hardware that achieves the desired speedup. For this to be possible, we need to understand the speedup achievable by different replication scenarios. Our compiler also reports an error in case the speedup specified by the speedup pragma is not theoretically feasible.

Throughout the thesis we use the terms speedup and throughput improvement interchangeably. Throughput of a data processing system can be defined as an average amount of data produced by the system per time unit. In our analyses we found it more suitable to use its inverse, the average amount of time it takes to produce one data element. We also make some simplifying assumptions that make the analysis simpler. First, we consider any potential overheads due to the round-robin distribution and collection of stream elements necessary to implement replication negligible, and thus ignore them in our analyses. Our experimental results (see Chapter 6) indicate that this is a valid assumption for our benchmark set, at least for replication factors up to 16. Second, we ignore the time it takes to fill up the pipeline in a case where multiple kernel replicas are connected in a pipelined fashion, because the number

of cycles needed to fill the pipeline is negligible compared to the number of cycles required to process the whole stream in most cases. Third, we only consider on-chip throughput in our analyses, ignoring any I/O bandwidth limitations. While I/O bandwidth can often be a limiting factor when using any logic device, its effects on throughput can be analyzed independently from on-chip throughput limitations due to data processing. Therefore, our focus is on the on-chip throughput analysis. Finally, we assume that kernel processing time does not vary from one stream element to the next. While this assumption may not be realistic for many applications, we expect that any variation in processing time between input elements would average out over a large number of elements. If that is not the case, an application-specific analysis has to be performed, which takes into account characteristics of the specific application. Since such analysis relies on characteristics of the specific application, we focus on analyses that are more general and can be applied to a range of different applications.

As discussed in section 5.1, replicating an ordinary kernel results in a speedup equal to the number of replicas in an ideal case. This is because all replicas operate in parallel, so they can process proportionately more data per time unit. Determining the speedup of a replication tree implementing 1-D reduction is more complex, as we saw in section 5.2.1. The speedup can be calculated by comparing the average amount of time it takes to produce an output element for replicated and non-replicated cases. Since our source-to-source compiler does not have access to circuit timing and scheduling information, it has to estimate the processing time. Assuming that processing each input element takes an equal amount of time, we can use the amount of time required to process one input element as a *time unit*. Therefore, we can calculate speedups of reduction kernels by dividing the average number of input elements that a non-replicated kernel has to process to produce one output element with the average number of input elements that a node in the replicated tree has to process per output element. If not all nodes in the reduction tree process the same number of input elements per output element, we have to consider the node that processes the largest number of input elements, because such a node will represent a bottleneck.

In the case of a single tree the speedup can be calculated by dividing the number of input elements processed by the non-replicated kernel per output element with the highest number of input elements processed by a replica per output element. Given an *X-to-A* reduction, with reduction ratio $R=X/A$, the non-replicated kernel processes R elements of input for each output element it produces. As we saw in section 5.2.1, by building a sufficiently large tree, the reduction can be decomposed such that no node in the tree processes more than two inputs per output element. Therefore, the maximum speedup achievable by building one tree is:

$$\frac{R}{2} = \frac{X}{2*A} \tag{5-1}$$

As observed previously, if other kernels in the application achieve higher throughput than the largest possible reduction tree, we can build multiple trees up to a maximum of A trees, in which case each tree

produces one element of the output. Therefore, maximum speedup achievable by building multiple trees is *A* times higher than the speedup of one tree, which is:

$$A * \frac{R}{2} = \frac{X}{2} \tag{5-2}$$

In this analysis we determined the speedup achieved by implementing the reduction in different ways. However, the compiler has the opposite task, which is to find an implementation that can achieve a desired speedup. While the solution is again obvious for ordinary kernels, and also for speedups achievable using one tree, care must be taken when building multiple trees. Consider an example of a 20-to-4 reduction that needs to be sped up 3 times. Since the reduction ratio (R) is 5, the maximum speedup achievable by building one tree is 2.5, which is not sufficient. By building two trees, we can achieve speedup of 5. However, it is not necessary to build two trees of maximal size, where each node processes at most two elements. Instead, both trees can be built in such a way that each node processes at most three elements, in which case the speedup is $2 * (5/3) = 3.3\dot{3}$, which exceeds the required speedup of 3, and requires fewer replicas to implement. To minimize hardware resources, our compiler first selects the minimal number of trees that can meet the desired speedup, and then selects the minimal size of each tree that still meets the required speedup.

Reductions of two-dimensional streams are sped up in two phases. Decomposing the reduction into two phases and implementing each phase as a separate hardware unit reduces the number of elements each unit has to process per output element. Therefore, some speedup has been achieved even if the two phases are not further replicated. Given an *<X,Y>-to-<A,B>* reduction that was decomposed into an *<X,Y>-to-<X,B>* reduction (first phase) and an *<X,B>-to-<A,B>* reduction (second phase), we wish to calculate the speedup achieved through decomposition.

The original, non-decomposed reduction kernel processes the input stream one row at a time, as described in section 4.2.2. Assuming that rows of the input are indexed 0 through *X-1*, no output is produced until the last row of the reduction window is processed, i.e. no output is produced until the kernel processes row *X/A-1*. While processing this row, the kernel produces a burst of B output elements. To compute the average number of elements processed to produce one output element, we observe that B outputs were produced while processing X∗Y/A elements of the input. This means that the average amount of time it takes to produce one output element, assuming the definition of the time unit established earlier in this section, is:

$$\frac{X*Y}{A} * \frac{1}{B} = \frac{X*Y}{A*B} \tag{5-3}$$

Once the reduction is decomposed, the first phase combines elements within rows, so it is equivalent to *X∗Y-to-X∗B* one-dimensional reduction. Therefore, the number of input elements this kernel processes to produce one output element is:

$$\frac{Y}{B} \tag{5-4}$$

The speedup of the first phase obtained through decomposition can then be calculated by dividing the time it takes to produce one element in the non-decomposed case (equation 5-3) with this value. The speedup of the first phase compared to the non-decomposed case is therefore:

$$\frac{\frac{X*Y}{A*B}}{\frac{Y}{B}} = \frac{X}{A} \tag{5-5}$$

Similar analysis can be done for the second phase, which performs an *<X,B>-to-<A,B>* reduction. Thus, the number of input elements it processes to produce one output is:

$$\frac{X}{A} \tag{5-6}$$

Therefore, its speedup over the non-decomposed case can be calculated as:

$$\frac{\frac{X*Y}{A*B}}{\frac{X}{A}} = \frac{Y}{B} \tag{5-7}$$

If the desired speedup is lower than the speedups of each of the two phases (equations 5-5 and 5-7), no further replication is necessary. Otherwise, one or both phases have to be further sped up. We analyze possible ways of speeding up each phase and achievable speedups next.

The first phase is equivalent to an *(X*Y)-to-(X*B)* one-dimensional reduction, so it can be replicated by building one or more reduction trees. If one reduction tree of maximum size is used, each node in the tree processes 2 input elements to produce one output. Therefore, its speedup relative to the non-replicated first phase can be obtained by dividing the expression 5-4 by 2, which results in the speedup of:

$$\frac{Y}{2*B} \tag{5-8}$$

It is important to realize that this speedup is relative to the non-replicated first phase. To obtain the speedup relative to the original non-decomposed reduction, we have to divide the expression 5-3 by 2, because each node in the tree processes 2 input elements per output element. Therefore, the speedup of the first phase implemented as a single tree relative to the original reduction before decomposition is:

$$\frac{X*Y}{2*A*B} \tag{5-9}$$

The same expression could have been derived by multiplying the basic speedup of the first phase (expression 5-5) and the speedup of building a single tree (expression 5-8).

Higher speedup of the first phase can be achieved by building multiple trees, up to a maximum of X*B trees, in which case X*B output elements are produced in 2 units of time. In such a case the maximum achievable speedup compared to non-replicated first phase is:

$$\frac{\frac{Y}{B}*X*B}{2} = \frac{X*Y}{2} \tag{5-10}$$

Similarly, the maximum achievable speedup compared to the original reduction before decomposition is:

$$\frac{\frac{X*Y}{A*B}*X*B}{2} = \frac{X^2*Y}{2*A} \tag{5-11}$$

The second phase of the decomposed reduction is an *<X,B>-to-<A,B>* reduction. As previously discussed, it can be replicated a number of times that divides the number of columns *B*. Let this number be *Z*, with *B=k*Z*. In such a case, each replica implements an *<X,k>-to-<A,k>* reduction. Therefore, *Z* outputs are produced every *X/A* time units, which means that the speedup relative to the non-replicated second phase is:

$$\frac{\frac{X}{A}}{\frac{X}{A}} * Z = Z \tag{5-12}$$

Similarly, speedup relative to the original reduction before decomposition is:

$$\frac{\frac{X*Y}{A*B}}{\frac{X}{A}} * Z = \frac{Y*Z}{B} \tag{5-13}$$

When the maximum number of replicas is used (*B*=*Z*), the above two speedups become *B* and *Y*, respectively. Then each replica implements an *X-to-A* one-dimensional reduction, which can be further sped up, until the point where each of the B replicas produces A outputs every 2 time units. In such a case, the speedup relative to the non-replicated second phase is:

$$\frac{\frac{X}{A}}{2} * A * B = \frac{X*B}{2} \tag{5-14}$$

Finally, the speedup of this configuration relative to the original reduction before decomposition is:

$$\frac{\frac{X*Y}{A*B}}{2} * A * B = \frac{X*Y}{2} \tag{5-15}$$

Our compiler considers different possible replication strategies in the order presented in this section, until it finds a strategy that meets the desired speedup. In the case of two-dimensional reductions, the maximum possible speedups of the two phases are different (expressions 5-11 and 5-15). Since *X* is always greater than or equal to *A*, the expression 5-11 is greater than or equal to the expression 5-15. Therefore, a speedup is theoretically possible only if it is lower than or equal to the expression 5-15.

We also note that aggressive replication, such as building the maximum possible number of reduction trees, would likely result in diminishing returns because of overheads we ignored in our analysis. However, it is still useful to understand the theoretical limits to achievable performance to be able to determine how far from optimal the system is and whether further improvements may be profitable.

This section derived expressions that help our compiler build structures that achieve speedups specified by the programmers. Our experimental results presented in the next chapter indicate that these expressions work well for our benchmarks and for speedups up to 16.

5.5. Strength Reduction of Modulo and Division Operations in the Presence of Replication

In section 4.2.3 we discussed strength reduction of modulo and division operations in ordinary and reduction kernels. However, that discussion did not consider how replication affects the implementation of these operations. We considered modulo and division operations that depend on loop iterators and can thus be implemented as counters related to the iterators. This occurs in one of two scenarios: in reduction kernels, to determine the reduction boundaries, and modulo and division operations involving the *indexof* operator. We analyze how each of these scenarios is affected by replication next.

Modulo and division operations are fundamental to implementing reduction kernels, because they are needed to determine when one reduction should end and the next one should begin. When reduction kernels are replicated they are implemented as a composition of smaller reductions operating independently. More specifically, the reduction ratios of the smaller reductions are known when the replica code is generated and they are independent from one another. Therefore, replicas of reduction kernels can be implemented as equivalent smaller reductions and do not require special handling of modulo and division operations. FPGA Brook does not presently support the *indexof* operator within reduction kernels, because such reduction kernels can usually be decomposed into an ordinary kernel that uses the *indexof* operator and a reduction kernel that does not use the operator. Therefore, there is no need to handle modulo and division operations involving the *indexof* operator in the reduction kernels.

The *indexof* operator can be used in ordinary kernels, and requires additional handling when such a kernel is replicated. As previously discussed, the *indexof* operator is used when the operation to be performed on a stream element depends on the position of the element within the stream. One example of this is the convolution operation used to implement FIR filters, discussed in section 2.2, where the coefficient a sample should be multiplied by depends on the sample's index. In Listing 2.2 we showed a simple kernel implementing a 4-tap FIR filter, which performs filtering on an input stream containing 4 samples. Since data parallelism in streaming applications is directly proportional to the length of the stream, this example exposes parallelism poorly. To expose more parallelism, we observe that filtering operations are normally performed repeatedly over many sequences of samples. In our example, this means that multiple sequences of 4 samples are processed one after another. To take advantage of parallelism, a longer stream can be formed that contains many sequences of 4 samples. In such a case, the

Listing 5.4 Modified FIR filter code in FPGA Brook

```
kernel void fir_mul (int x<>, out int y<>) {
  const int h[4] = {1, 2, -3, -1};
  int expr = 4;

  y = h[indexof(x)%expr] * x;
}
```

FIR filter kernel cannot use the value of the *indexof* operator directly to determine the index of the coefficients to use, because the stream has more elements than the number of coefficients. The kernel code, modified to account for this, is shown in Listing 5.4. The modulo operation involving the *indexof* operator ensures that the four coefficients are used repeatedly over sequences of four consecutive input samples. The *expr* variable is used instead of directly using the constant 4 to illustrate how our compiler handles the modulo operation in a general case when this value is not a constant. When the kernel is not replicated, the modulo operation can be implemented using counters, as described in section 4.2.3. This code has to be adjusted when the kernel is replicated, as shown in Listing 5.5.

Listing 5.5 shows the C2H code generated for replica 0 of the *fir_mul* kernel when the kernel is replicated five times. As with other code listings in this thesis, it has been modified from the original, compiler-generated code to better illustrate how modulo operations are handled. Given the expression *indexof(x)%expr*, the initial value of the modulo iterator that replaces this expression should be equal to *_replica_id%expr*. The *_replica_id* variable contains a unique ID assigned to each replica, as discussed in section 5.1. If the value of *expr* is known at compile time, the initial value of the modulo iterator is precomputed by our compiler; otherwise, our compiler generates a *while* loop that computes the correct initial value (lines 10 through 12).

In every iteration of the main loop that processes stream elements, the value of the modulo iterator is incremented by the value equal to the replication factor (line 15). Since this is also done in the first iteration, before any elements have been processed, an adjustment is made to the initial value to compensate for this (line 13). Another *while* loop is used to ensure that the result is within range of the modulo operation (lines 16 and 17). The *while* loop is needed because the replication factor can be higher than the value of *expr*, so the modulo counter can exceed the value of *expr* by more than the value of *expr*, which means that the subtraction performed in the *while* loop has to occur more than once. There are two additional optimizations that can be performed when the value of *expr* is known at compile time. First, if the replication factor is lower than the value of expr, the *while* loop can be replaced by a simple *if* condition. Second, if the replication factor equals the value of *expr*, that means that the value of the modulo iterator never changes, because the stride of indices matches the value of *expr*, so lines 15 through 17 in the code can be simply omitted.

Listing 5.5 Implementation of the modulo operation in a replicated kernel

```
void  fir_mul_replica_0 () {                                              (1)
  volatile int *x;                                                        (2)
  volatile int *y;                                                        (3)
  int  _temp_x, _temp_y;                                                  (4)
  const int h[4] = {1, 2, -3, -1};                                        (5)
  int _iter, _mod_iter;                                                   (6)
  int expr = 4;                                                           (7)
  int _replication_factor = 5;                                            (8)
  int _replica_id = 0;                                                    (9)

  _mod_iter = _replica_id;                                               (10)
  while (_mod_iter >= expr) {                                            (11)
    _mod_iter -= expr;                                                   (12)
  }
  _mod_iter -= _replication_factor;                                      (13)

  for (_iter=0; _iter < INPUTS; _iter+=_replication_factor) {            (14)
    _mod_iter += _replication_factor;                                    (15)
    while (_mod_iter >= expr) {                                          (16)
      _mod_iter -= expr;                                                 (17)
    }
    _temp_x = *x;                                                        (18)
    _temp_y = h[_mod_iter] * _temp_x;                                    (19)
    *y = _temp_y;                                                        (20)
  }
}
```

Modulo operations involving the *indexof* operator are also useful in conjunction with kernels processing two-dimensional streams. For example, two-dimensional convolution, commonly used in image processing, takes a small subset of the input image, known as a *window*, and multiplies the pixels of the window by a mask, which is a two-dimensional array of constant coefficients. The window consists of the pixel currently being considered and its neighbouring pixels. The size of the window matches the size of the two-dimensional array holding constants. This window should not be confused with the term reduction window used elsewhere in this thesis. The results of multiplications are added up to produce the output value for the current pixel. The multiplication part of the 2-D convolution application utilizing window size of 3x3 is shown in Listing 5.6. It is assumed that the kernel is invoked with a 2-D stream as a parameter, so the *indexof* operator returns a two-dimensional vector. The modulo operation is applied to both dimensions, and the result is used to address the two-dimensional array. The modulo operation ensures that the indices stay within the bounds of the array. The C2H code generated for a replica of this kernel is shown in Listing 5.7.

The code associated with handling of the modulo operation is shown in bold in Listing 5.7. The remaining code implements other functionality necessary for correct functioning of a replicated kernel handling 2-D streams, as described in section 5.1. The modulo iterators are first initialized to be equal to

Listing 5.6 Using the modulo operation for the multiplication part of the 2-D convolution application

```
kernel void conv_2d (int x<>, out int y<>) {
  const int h[3][3] = {{1, 2, 1}, {2, -1, 3}, {-2, 1, 1}};
  int expr = 3;

  y = h[indexof(x)%expr]*x;
}
```

their respective loop iterators (lines 12 and 13), and their values are then adjusted using while loops until the correct initial value is obtained (lines 15 through 18). If the value of *expr* is known at compile time, this adjustment is not necessary and the initial values of these iterators can be precomputed by the compiler. The value of the column iterator is further adjusted in line 19 to compensate for the first increase in its value before any computation is performed in line 34. The modulo iterators have to be updated at the same time as their respective loop iterators. The modulo iterator *_mod_iter_1* is updated inside the inner loop by first being increased by the replication factor (line 34) and then computing the result of the modulo operation (lines 35 and 36). The modulo iterator *_mod_iter_0* is updated in the outer loop (lines 26 through 29) whenever the loop iterator *_iter_0* is updated. The modulo iterators are used to address the two-dimensional array in line 38.

The code in lines 30 and 31 requires further analysis. Every time the loop iterator *_iter_0* is incremented a new row is being selected. This may affect the value of the column iterator in some circumstances, as illustrated by Figure 5.10a. The figure shows the expected values of the modulo counters for a 4x5 stream and the expression *indexof(x)%3*. We use the notation (*_mod_iter_0*, *mod_iter_1*) to denote the values of the corresponding modulo iterators. Let us assume that the replication factor is 4 and let us consider the modulo iterator values for the replica with the ID of 0. The stream elements assigned to replica 0 are shaded in Figure 5.10a. In the first iteration through the loop the values of the modulo counters are (0,0). In the second iteration *_mod_iter_1* is first increased by the value equal to the replication factor (4), and then decreased by 3 to compute the modulo operation. This results in the modulo iterators acquiring values (0,1), as expected, because the iterator *_mod_iter_0* does not change in this step. If we disregard the code in lines 30 and 31, in the following step, *_mod_iter_0* is incremented once and *_mod_iter_1* is again increased by 4 and then decreased by 3, which makes the pair (1,2), which is not the desired value.

To understand why this occurs and how to compensate for the discrepancy, let us consider the example in Figure 5.10b. This example differs from the example in Figure 5.10a only in the number of columns of the stream, which is 6 in this case. If we repeated the same exercise, we would find that the values of the modulo iterators would produce the desired result. To understand why, one can observe the values of the *_mod_iter_1* iterator across rows Figure 5.10b. The values of this iterator form a continuous sequence

Listing 5.7 Implementation of the modulo operation in a replicated kernel handling 2-D streams

```
void  conv_2d_replica_0 () {                                              (1)
  volatile int *x;                                                        (2)
  volatile int *y;                                                        (3)
  int _temp_x, _temp_y;                                                   (4)
  int h[3][3] = {{1, 2, 1}, {2, -1, 3}, {-2, 1, 1}};                      (5)
  int _ROWS = 1920, _COLS = 1440;                                         (6)
  int _replica_id = 0;                                                    (7)
  int _replication_factor = 4;                                            (8)
  int _iter_0 = _replica_id/_COLS;                                        (9)
  int _iter_1 = _replica_id%_COLS;                                       (10)
  int expr = 3;                                                          (11)
  int _mod_iter_0 = _iter_0;                                             (12)
  int _mod_iter_1 = _iter_1;                                             (13)
  int _cols_mod_expr = _COLS;                                            (14)

  while (_mod_iter_0 >= expr) {                                          (15)
    _mod_iter_0 -= expr;                                                 (16)
  }
  while (_mod_iter_1 >= expr) {                                          (17)
    _mod_iter_1 -= expr;                                                 (18)
  }
  _mod_iter_1 -= _replication_factor;                                    (19)
  while (_cols_mod_expr >= expr) {                                       (20)
    _cols_mod_expr -= expr;                                              (21)
  }
  for ( ; _iter_0 <  _ROWS; ) {                                          (22)
    while (_iter_1 >= _COLS) {                                           (23)
      _iter_0++;                                                         (24)
      _iter_1 -= _COLS;                                                  (25)
      if (_mod_iter_0 == (expr-1))                                       (26)
        _mod_iter_0 = 0;                                                 (27)
      else                                                               (28)
        _mod_iter_0 = _mod_iter_0 + 1;                                   (29)
      if (_cols_mod_expr != 0)                                           (30)
        _mod_iter_1 += expr - _cols_mod_expr;                            (31)
    }
    if (_iter_0 <  _ROWS) {                                              (32)
      for ( ; _iter_1 < _COLS; _iter_1+=_replication_factor) {           (33)
        _mod_iter_1 += _replication_factor;                              (34)
        while (_mod_iter_1 >= expr) {                                    (35)
          _mod_iter_1 -= expr;                                           (36)
        }
        _temp_x = *x;                                                    (37)
        _temp_y = h[_mod_iter_0][_mod_iter_1] * _temp_x;                 (38)
        *y = _temp_y;                                                    (39)
      }
    }
  }
}
```

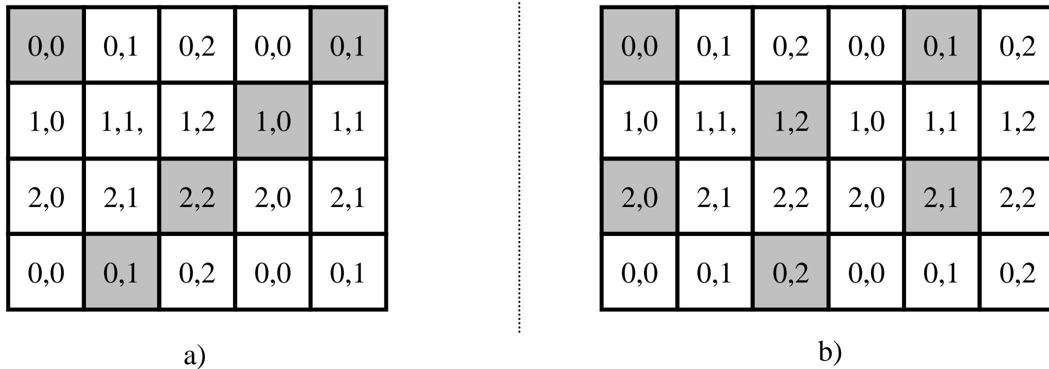


Figure 5.10 Distribution of modulo values in a two-dimensional stream

across row boundaries in modulo-3 arithmetic. That is, its value is 2 for the last element in a row, and 0 for the first element in the next row. Going back to Figure 5.10a, one can see that the sequence is interrupted when going from one row to the next, because the value of *_mod_iter_1* for the last element in a row is 1, and the first value in the next row is 0, rather than 2. This is because the number of columns is not divisible by 3. In general, if the number of columns is divisible by the number the modulo operation is being computed by (i.e. *expr*), there is no need to adjust the value of *_mod_iter_1* across row boundaries, because the values are continuous across row boundaries. On the other hand, if this is not the case, the correct value of *_mod_iter_1* can be obtained by "padding" the stream with extra columns, so that the total number of columns is divisible by *expr*. Going back to the code in Listing 5.7, this means that if the number of columns *_COL* is not divisible by *expr*, there are trailing *_COL%expr* columns. To make the number of columns divisible by *expr*, we "pad" the stream with *expr-(_COL%expr)* empty columns. Since these columns do not actually exist in the stream, the value of the *_mod_iter_1* iterator has to be increased at the end of each row by the number of columns that the stream was "padded" with, which skips the non-existing columns and resets the modulo counter to the correct starting value for the next row.

In our original example, we obtained the incorrect value of the modulo iterator pair (1,2). Given the adjustment we derived above, the *_mod_iter_1* is increased by one, because only one column is missing to make the number of columns divisible by 3. This is implemented by the code in line 31 in Listing 5.7. Let us calculate the sequence of values of the modulo iterator pair again, taking this code into account. In the first two iterations the values of the modulo counters are (0,0) and (0,1), as previously discussed. In the following step, after *_mod_iter_0* is incremented, *_mod_iter_1* is increased by 1, because *expr-(_COL%expr)* evaluates to 1. This makes the modulo iterator pair acquire values (1, 2). However, before these values are used, *_mod_iter_1* is again increased by 4, which makes it 6, and then decreased by 3 twice to calculate the modulo value, which makes the pair (1,0), equal to the expected value. The code

presented in the listing is general enough to handle the case where the value of *expr* is not known at compile time. If this value is known at compile time, the code can be simplified by precomputing various values and substituting them in appropriate locations.

In this section we described how our compiler implements modulo operations involving the *indexof* operator in the presence of replication. While we have not specifically addressed division operations, they can be implemented in a similar fashion. Our compiler currently does not support division operations involving *indexof* operators in replicated kernels, because none of our benchmarks required such operations.

5.6. Putting It All Together: Building the Replicated Streaming Dataflow Graph

In the preceding sections we described various components needed to support replication of FPGA Brook kernels. In this section we discuss several issues that arise when the components are put together to implement the original application.

FPGA Brook allows each kernel to be replicated an arbitrary number of times. As a result, different kernels in the streaming dataflow graph may have different replication factors. If a kernel processes an output from another kernel, and they have different replication factors, the stream elements have to be redistributed to match the number of replicas in the destination kernel. This functionality can be implemented in several ways. Given the collector and distributor nodes described in section 5.1.1, we could use a collector FIFO at the output of the source kernel to gather stream elements, and a distributor FIFO at the input to the destination node to distribute stream elements. The problem with this approach is that both collector and distributor FIFOs are passive elements, meaning that they only contain Avalon slave ports, and require master ports to read or write to the FIFOs. This could be resolved by either redesigning one of these FIFOs, or by inserting another hardware accelerator that reads from one FIFO and writes to the other. A better solution is to build a hardware module that performs the functionality of both collector and distributor node. We call such a FIFO node the *collector/distributor FIFO*. The module is automatically generated by our compiler with the appropriate number of input and output ports. A similar issue occurs when a replicated kernel is followed by a replicated reduction kernel that contains multiple trees. Since multiple trees require a double-distributor, we generate Verilog code for a special FIFO module that we call the *collector/double-distributor*, which performs the required operations. While the collector/distributor FIFOs have potential to become bottlenecks when the number of replicas is large, we found their performance satisfactory for the replication factors used in our experiments.

The performance of applications consisting of many kernels, possibly with different replication factors may be difficult to understand in some cases. To help, our compiler generates streaming dataflow graphs

in graphical format. Two streaming dataflow graphs are generated for each application, one obtained through dataflow analysis of the original application, which ignores replication, and also the replicated dataflow graph. These graphs are produced in the *dot* format compatible with the open source Graphviz graph visualization software [72], which can be easily rendered in graphical form for visual representation of the streaming dataflow graph. The produced graphs were also useful while developing our compiler and helped with debugging.

In this chapter we presented algorithms utilized in replicating kernel code to exploit data parallelism in streaming applications. We also described the hardware modules necessary for implementation of replicated kernels and their integration into the Nios II-based system. The next chapter presents experimental results obtained using our design flow.

Chapter 6

Experimental Evaluation

In the previous chapters we described an automated design flow for implementation of FPGA Brook streaming programs in FPGA logic. In this chapter we present results of experimental evaluation of the design flow we developed. The evaluation has three main goals. First, we wish to validate our design and demonstrate that it can process realistic applications and implement them correctly. Second, we wish to compare performance and area of the designs produced by our design flow to alternative design flows for FPGAs. Third, we wish to explore how kernel performance scales with replication. To this end, we implemented a suite of benchmark programs in FPGA Brook. In the following sections we discuss implementation details of these benchmarks and results of our experiments.

6.1. Development of the FPGA Brook Benchmark Set

To support testing and performance evaluation of the FPGA Brook design flow, we selected a number of applications and developed them in FPGA Brook. In this section we describe the algorithms used for each of the benchmarks, and describe challenges when implementing them in FPGA Brook. Some of these applications have already been discussed in earlier chapters. We provide references to those sections and expand on previous descriptions. For brevity, we do not provide full source code of all benchmarks, only the key parts that help understanding the issues related to FPGA Brook. The source code of all benchmarks will be made publicly available at the author's web-site in the near future. We hope that this section will be useful to programmers unfamiliar with streaming who wish to learn how to effectively express their applications in FPGA Brook.

6.1.1. Autocorrelation

The autocorrelation application is described in detail in section 3.2. To summarize, autocorrelation is the correlation of a signal with a time-shifted version of itself. If the original signal is represented as a stream and the time-shifted version of the signal as another stream, the length of the time-shifted stream is different for different values of the time-shift because the signal is assumed to have a finite number of elements. This represents a small challenge for FPGA Brook implementation, because stream lengths have to be static. A simple workaround is to set the stream length to the largest size needed, and pad the elements that are not needed for a particular time-shift value with zeros, which does not change the result. The source code of the autocorrelation application is shown in Listing 3.1. In our experiments, the

autocorrelation is performed on a signal consisting of 100,000 samples for 8 different shift distances, with samples represented as 32-bit integers. Practical applications of autocorrelation usually involve a much larger number of shift distances. Due to the structure of the application, using a larger number of shift distances would proportionately increase run-time of all implementations and would not affect the relative performance of different implementations, which is what we are primarily interested in. Therefore, a small number of shift distances is sufficient for our experiments.

6.1.2. FIR Filter

FIR filters are commonly used in DSP systems for signal filtering. The FIR filter implements the convolution operation, presented in section 2.2, represented by the equation:

$$y(n) = \sum_{k=0}^{M-1} h(k)x(n-k) \quad (6\text{-}1)$$

For each input sample *x(n)*, a sequence of past *M* samples, including the sample *x(n)* are multiplied by a set of coefficients *h(0,…M-1)*, and the products are added to produce the output *y(n)*. Hence, the filter's current output always depends on the signal's current value and *M-1* past values. The procedure is repeated for all input samples. The number of coefficients (*M*) is usually referred to as the number of taps in the filter. There are two potential problems when attempting to implement a FIR filter in a streaming language. First, data parallelism exposed by equation 6-1 is limited. The number of taps in practical applications can range from only a few to several hundred [17]. However, the number of samples being processed is usually several orders of magnitude larger. As discussed in section 5.5, given an *M*-tap FIR filter and a sequence of *X* input samples, a stream with *M∗X* samples can be formed. Each sequence of *M* consecutive samples contains a sample from the input sequence and *M-1* of its predecessors. Such functionality can be achieved by using the stencil operator in the original Brook specification, or a streamRead-type operator in FPGA Brook. The streamRead-type operator can implement this functionality using a simple circular buffer. The multiplication kernel shown in Listing 5.4 is then used to multiply the samples by the coefficients. Finally, a reduction kernel, such as the *sum* reduction kernel shown in Listing 3.1, performs the summation of *M* products to produce one element of the output. This means that the reduction kernel performs an *M∗X*-to-*X* reduction, which can be achieved by calling the reduction kernel with a stream with *M∗X* elements on the input and *X* elements on the output. FIR filter functionality is usually depicted in graphical form as shown in Figure 6.1. The figure shows the mapping of the convolution operation (equation 6-1) into kernels and streams for a 4-tap filter. The delay elements represent memory that holds previous values of the input, elements labelled with ∗ and + are multipliers and adders, respectively. Dotted rectangles enclosing the elements in the diagram denote the streamRead-type operator and two kernels. Finally, dotted lines between the kernels denote streams.

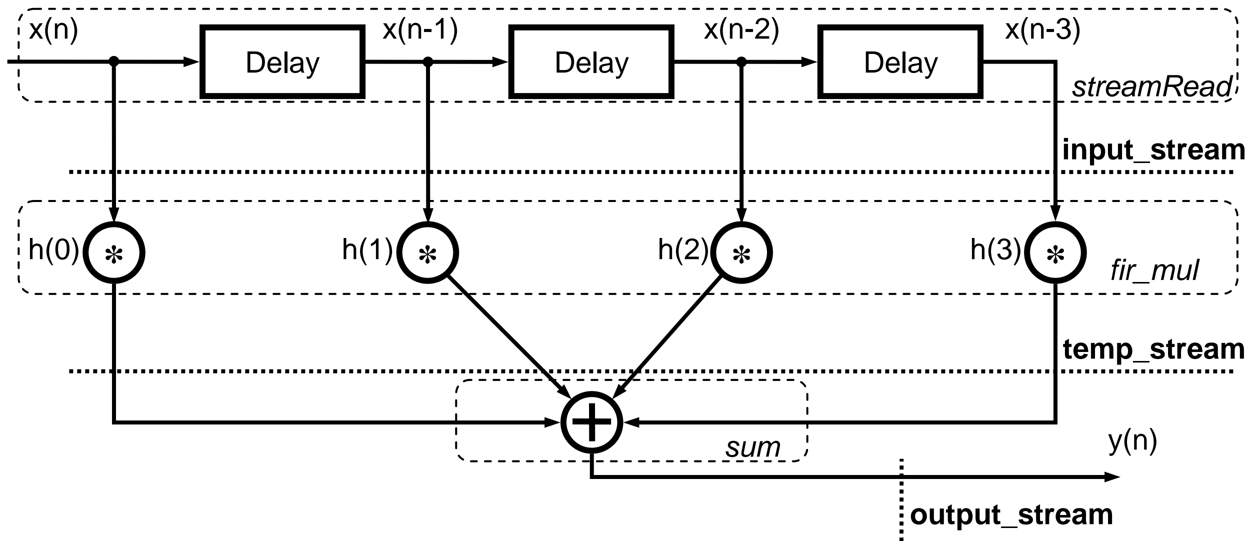


Figure 6.1 Depiction of a 4-tap FIR filter and its mapping to FPGA Brook

We implemented two versions of the FIR filter benchmark, with 8 and 64 taps, to explore how the number of taps affects performance. In both cases, the applications process an input of 100,000 samples, represented as 32-bit integers.

6.1.3. Two-Dimensional Convolution

Two-dimensional convolution is an operation commonly used in image processing. The operation consists of multiplying the intensity of a pixel and its neighbouring pixels, referred to as a window, by a two-dimensional array of constant coefficients, referred to as a mask. The mask size normally matches the size of the window. The results of multiplications are added to produce the output value of the current pixel. Two-dimensional convolution is similar to its one-dimensional counterpart used by the FIR filter application. As a result, their implementations in FPGA Brook are similar. To avoid dependences between stream elements and increase parallelism, the original image has to be transformed into a stream that for each pixel of the original image contains a window consisting of the pixel and its neighbouring pixels. This process is depicted in Figure 6.2, which shows the result of this transformation when applied to a 3x4 image, with the window size of 3x3 pixels. As with the FIR filter implementation, a streamRead-type operator is used to implement this functionality. For pixels that do not have neighbours on all sides, the window is padded with zero pixels in the corresponding positions in the expanded image. While other techniques for handling this border condition exist, their discussion is beyond the scope of this thesis.

After creating the input stream using the streamRead-type operator, a multiplication kernel, such as the one shown in Listing 5.6, can be used to multiply each element of the stream by an appropriate constant

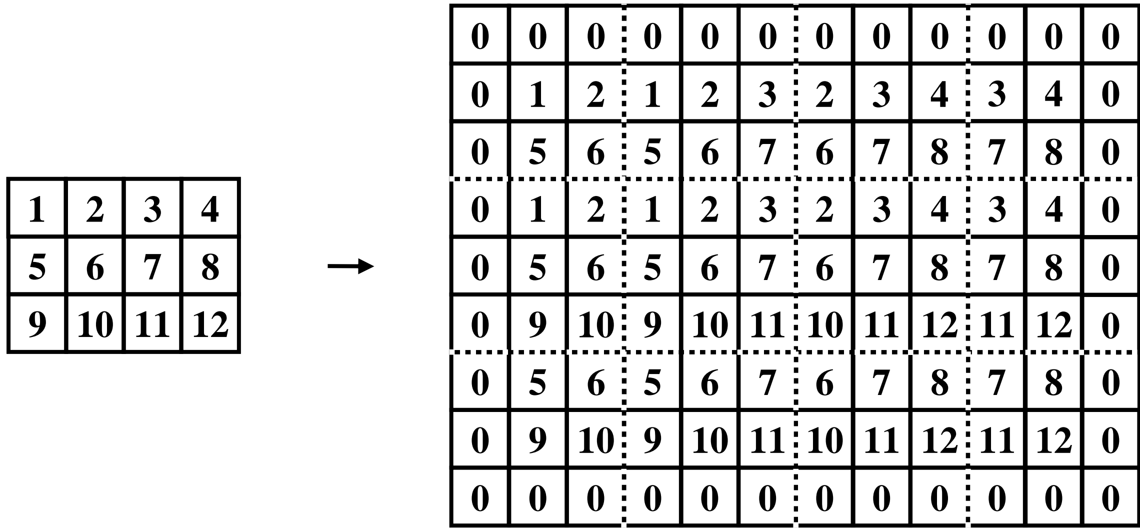


Figure 6.2 Applying the window function to a 3x4 pixel image

from the mask. Finally, the output of the multiplication is processed by a summation reduction kernel, such as the *sum* reduction kernel shown in Listing 3.1. As previously discussed, the same reduction kernel can be used for both 1-D and 2-D streams. The final result is a stream whose dimensions match the original image size. In our experiments an image size of 640x480 and mask size of 3x3 pixels are used. As a result the stream that is created by the streamRead-type operator has dimensions of 1920x1440. The stream is multiplied by the mask and then reduced to a 640x480 stream, which is the final result. Each pixel is represented as a 32-bit integer.

6.1.4. Matrix-Vector and Matrix-Matrix Multiplication

Matrix-vector and matrix-matrix multiplication are operations at the core of many other algorithms, and are commonly used for benchmarking of computer systems. In section 2.2 we described how matrix-vector multiplication is implemented in GPU Brook. The FPGA Brook implementation is similar, except that the streamRead and streamWrite operators have to be replaced with FPGA Brook streamRead-type and streamWrite-type operators. Since FPGA Brook does not automatically resize streams, the streamRead-type operator handling the input vector has to create a two-dimensional stream from the vector, by replicating the vector across rows, as discussed in section 2.2. In our experiments we multiply a 1000x1000 matrix by a 1000-element vector, both containing 32-bit integers.

Matrix-matrix multiplication is more complex to implement than matrix-vector multiplication because of the increased number of dependencies. Consider a simple example of multiplying a 3x2 matrix by a 2x3 matrix:

$$\begin{bmatrix} x_{00} & x_{01} \\ x_{10} & x_{11} \\ x_{20} & x_{21} \end{bmatrix} \cdot \begin{bmatrix} y_{00} & y_{01} & y_{02} \\ y_{10} & y_{11} & y_{12} \end{bmatrix} = \begin{bmatrix} x_{00}y_{00} + x_{01}y_{10} & x_{00}y_{01} + x_{01}y_{11} & x_{00}y_{02} + x_{01}y_{12} \\ x_{10}y_{00} + x_{11}y_{10} & x_{10}y_{01} + x_{11}y_{11} & x_{10}y_{02} + x_{11}y_{12} \\ x_{20}y_{00} + x_{21}y_{10} & x_{20}y_{01} + x_{21}y_{11} & x_{20}y_{02} + x_{21}y_{12} \end{bmatrix} \quad (6\text{-}2)$$

By observing the result of the matrix multiplication, we can form a plan for a streaming implementation. Given properly formatted streams, the result can be decomposed into a multiplication and summation. For the above example, the streamRead-type operators have to transform the two matrices into streams as follows:

$$\begin{bmatrix} x_{00} & x_{01} \\ x_{10} & x_{11} \\ x_{20} & x_{21} \end{bmatrix} \rightarrow \begin{matrix} x_{00} & x_{01} & x_{00} & x_{01} & x_{00} & x_{01} \\ x_{10} & x_{11} & x_{10} & x_{11} & x_{10} & x_{11} \\ x_{20} & x_{21} & x_{20} & x_{21} & x_{20} & x_{21} \end{matrix}$$

$$\begin{bmatrix} y_{00} & y_{01} & y_{02} \\ y_{10} & y_{11} & y_{12} \end{bmatrix} \rightarrow \begin{matrix} y_{00} & y_{10} & y_{01} & y_{11} & y_{02} & y_{12} \\ y_{00} & y_{10} & y_{01} & y_{11} & y_{02} & y_{12} \\ y_{00} & y_{10} & y_{01} & y_{11} & y_{02} & y_{12} \end{matrix}$$

The two streams can be multiplied and then reduced to a 3x3 stream through summation reduction. In a general case, given two matrices M_1 and M_2, with AxB and BxC elements, respectively, multiplication of these matrices in FPGA Brook proceeds as follows. Two streamRead-type operators are used to create two 2-D streams, each having A rows and $B*C$ columns. The first stream is created using the elements of matrix M_1, by repeating each row of the matrix C times across the rows of the stream being created, such that the first row of the matrix is repeated over the first row of the stream, the second row of the matrix over the second row of the stream and so on. Another way to describe this operation is by saying that the whole matrix M_1 is repeated C times in the resulting stream. The second stream is created using the elements of matrix M_2, by repeating each column of the matrix C times, such that each column of the matrix appears in every row of the stream. The two streams are then multiplied and the multiplication result is reduced to a stream with A rows and C columns by a summation reduction kernel. The resulting stream is the final result of the matrix-matrix multiplication. In our experiments, the two matrices being multiplied have shapes 60x30 and 30x40, and their elements are 32-bit integers.

6.1.5. Mandelbrot Set Generator

Mandelbrot set is a type of fractal model generator [34], which can be described as a set of points in the complex plane, in its simplest form. When the points belonging to the set are plotted in a two-dimensional plane, they produce a complex pattern resembling complex structures found in nature. Fractal models are useful in the study of various systems that exhibit seemingly chaotic behaviour [34]. Given a point C in the complex plane the following recurrence is used to determine whether the point belongs to the Mandelbrot set:

$$Z_0 = 0$$
$$Z_{n+1} = Z_n{}^2 + C \tag{6-3}$$

The point C is said to be in the Mandelbrot set if the recurrence in equation 6-3 converges when n goes to infinity, otherwise the point is not in the set. In practical implementations the recurrence is evaluated a finite number of times. If the absolute value of Z_n exceeds a given threshold in any iteration, the evaluation for that point terminates and the point is declared not to be in the set. If the absolute value of Z_n does not exceed the threshold until the last iteration, the point is declared to be in the set.

Evaluation of the Mandelbrot set is suitable for implementation in a streaming language because computation on a specific point in the complex plane is independent from computations on all other points. Additionally, Mandelbrot set generation is a computationally intensive operation, so it can benefit from parallelization. Since the quality of results depends on the number of iterations, a high number of iterations is desirable. Furthermore, the computation has to be performed over many points in the complex plane if the goal is to produce an image of high resolution. Finally, the computation is often repeated for different regions of the plane to study various properties of the Mandelbrot set.

Implementing Mandelbrot set generation in FPGA Brook presents two challenges. First, the algorithm requires complex number manipulation. While FPGA Brook does not directly support complex numbers, a sequence of complex numbers can be represented by two streams: one containing the real, and the other containing the imaginary part of the number. The complex addition and multiplication can be easily implemented on the components of the number. Second, the algorithm implementation requires computation that supports fractions, but FPGA Brook does not support floating-point numbers. The alternative is to use fixed-point number representation [34]. Fixed-point representation uses a predefined number of bits in a register for the fractional part of the number, while the remaining bits store the integer part of the number and the sign bit. The number of bits used for each part depends on the required precision and the range of numbers that need to be represented to implement a given application. Fixed-point representation is an attractive alternative to floating-point representation because it does not require specialized processing units, but uses ordinary integer arithmetic units instead. Evaluating the recurrence in equation 6-3 requires addition and multiplication of fixed-point numbers. Addition of fixed point numbers can be performed by ordinary integer adders. Multiplication requires integer multiplication, followed by arithmetic right-shift by a constant number of bits, both of which can be efficiently implemented in FPGAs. Discussion of effects due to using the fixed-point representation for implementation of Mandelbrot set generation is beyond the scope of this thesis. We note that choosing a different fixed-point format would have no effect on the performance of the algorithm, as long as the number of bits used for the number representation does not change. To investigate the effect the number of bits has on performance and area of the FPGA Brook implementation of the algorithm, we implement

two versions of the Mandelbrot benchmark, using 16-bit and 32-bit data types. In the 16-bit version, 2 bits are used for the integer part and 13 bits are used for the fraction part of the number. In the 32-bit version, 7 bits are used for the integer part and 24 bits are used for the fraction part.

The Mandelbrot benchmark is implemented in FPGA Brook through two streamRead-type operators, the *mandelbrot* kernel and a streamWrite-type operator. The two streamRead-type operators create two streams, which represent real and imaginary parts of the points in the complex plane to be tested for inclusion in the Mandelbrot set. The *mandelbrot* kernel computes the value of Z_n in a loop, until its value exceeds the threshold or the maximum number of iterations is reached, whichever comes first. As a result, the execution time of the kernel can vary significantly between stream elements. For every point that is found to be in the Mandelbrot set, 0 is written to the output stream. For points that are found not to be in the Mandelbrot set, the output stream contains the number of iterations it took to determine that Z_n exceeds the threshold. The *mandelbrot* kernel code is similar to the equivalent C code, the only difference being that references to the array containing the points and storing the result have to be replaced by references to input and output streams, respectively. The kernel code was based on the fixed-point implementation of the algorithm by Pellerin and Thibault [34]. Finally, the streamWrite-type operator is used to write the results to the memory.

In our experiments, 250,000 points in the complex plane were tested, while the maximum number of iterations was set to 2000 and the threshold was 3.5.

6.1.6. Zigzag Ordering in MPEG-2

The zigzag ordering algorithm is a part of the MPEG-2 video compression standard [57]. The algorithm operates on a stream of independent arrays, each array consisting of 64 elements, representing an 8x8 section of an image in the video sequence. Each array's elements are ordered based on the positions specified in the ordering matrix. A detailed description of the zigzag ordering algorithm is given in section 3.3. In our experiments the code similar to the one shown in Listing 3.3 is used, except that a different ordering matrix is used. This is because the code in Listing 3.3 implements the ordering operation performed by the encoder, whereas our benchmark implements the operation performed by the decoder. In our experiments we use a stream size of 10,000 elements, with array elements of the *short* (16-bit) data type.

6.1.7. Inverse Quantization in MPEG-2

During the MPEG-2 encoding process, data blocks representing images from a video sequence are transformed into the spatial frequency domain. This transformation is performed because data in the frequency domain can be processed in a way that takes advantage of the properties of the human visual system. One such property is the relative insensitivity to high spatial frequency components in the image.

Therefore, high spatial frequency components can be transferred with lower precision than low frequency ones. The *quantization* step, performed during encoding, divides each frequency component by a predefined constant coefficient, discarding any remainder. Higher frequency components are usually divided by higher coefficients, often causing many of them to be reduced to zero. As a result, the amount of information that has to be stored is reduced, which is the main goal of video compression. In the decoder, the *inverse quantization* step performs the inverse operation by multiplying the components by the same coefficients they were divided by in the encoder. The coefficient values are given in the form of a quantization matrix, whose size matches the size of the block of data being processed.

The FPGA Brook implementation of the inverse quantization algorithm consists of only one processing kernel and a streamRead-type and a streamWrite-type operator. The kernel operates on a stream of arrays, each array consisting of 64 elements, representing an 8x8 section of an image in the spatial frequency domain. The kernel multiplies each array element by a corresponding coefficient in the quantization matrix. Because of the way the quantization matrix is normally represented in the MPEG-2 standard, the multiplication is followed by an arithmetic right shift. In our experiments we use a stream size of 10,000 elements, with array elements of the *short* (16-bit) data type and the quantization matrix of the *char* (8-bit) data type.

6.1.8. Saturation in MPEG-2

Due to limited precision and rounding issues in MPEG-2 encoding and decoding, data values of pixels and frequency components may get out of the allowed range. Saturation tests whether a value is outside the prescribed range and if so, sets it to the minimum or maximum allowed value as appropriate [56]. The operation is performed over all pixels of an image. Saturation is sometimes also referred to as clamping [57]. As with other MPEG-2 components, saturation can be implemented as a simple kernel processing streams of arrays, each array consisting of 64 elements. The kernel tests each array element and saturates its value if necessary. In our experiments we use the stream size of 10,000 elements, with array elements of the *short* (16-bit) data type.

6.1.9. Mismatch Control in MPEG-2

The conversions to and from the spatial frequency domain in MPEG-2 are lossless in an ideal case. As discussed in section 3.3, the conversion to the spatial frequency domain is performed by a transformation called discrete cosine transform (DCT), while the inverse operation is known as inverse DCT (IDCT). DCT is used in MPEG-2 encoding, while IDCT is used in decoding. The two operations are perfect inverses of one another if calculated to infinite precision, meaning that if DCT were performed on a block of data followed by IDCT, the result would be the original block of data [57]. However, because of finite precision and rounding errors, IDCT does not always produce the result identical to its input. The

resulting error is known as the IDCT mismatch error [57]. The goal of the mismatch control is to detect such errors and make corrections to reduce possible distortions in the decoded video, without significantly increasing the size of the compressed video sequence. The mismatch control algorithm used in MPEG-2 adds all the elements of one 64-element block and if the resulting sum is even, the last bit of the last element in the block is toggled by either adding or subtracting a value of 1. A detailed discussion of merits of this algorithm is beyond the scope of this thesis, but can be found in [57]. In the FPGA Brook implementation, a single kernel performs the described operations. In our experiments the kernel operates on a stream of arrays, with the stream size of 10,000 elements and array elements of the *short* (16-bit) data type.

6.1.10. Inverse Discrete Cosine Transform (IDCT)

IDCT is used in MPEG-2 decoders to reconstruct the original pixel values from the spatial frequency components of the encoded signal. IDCT takes an 8x8 block of data containing frequency components from the compressed video sequence and restores the original pixel values. Since other compression techniques, such as quantization, are used between DCT and IDCT, the restored values will be only an approximation of the original image. In its general form, 2-D IDCT computation can be expressed through the following equation [57]:

$$f(x,y) = \sum_{\mu=0}^{7}\sum_{\nu=0}^{7} \frac{C(\mu)}{2}\frac{C(\nu)}{2} F(\mu,\nu) \cos\left[\frac{(2x+1)\mu\pi}{16}\right] \cos\left[\frac{(2y+1)\nu\pi}{16}\right]$$

$$C(\mu) = \frac{1}{\sqrt{2}}, if\ \mu = 0 \qquad (6\text{-}4)$$

$$C(\mu) = 1, if\ \mu > 0$$

The computation takes as an input the 8x8 two-dimensional array *F*, representing the frequency components, and produces an 8x8 two-dimensional array *f*, representing the restored pixel values. Therefore, μ and ν are indices in the input array, while *x* and *y* are indices in the output array.

The IDCT computation would require 1024 multiplications and 896 additions if computed directly as expressed by equation 6-4, and is thus computationally very intensive. However, more efficient implementations of IDCT exist [57]. We base the FPGA Brook implementation of 2-D IDCT on the C code provided with the MPEG-2 reference implementation available online [73]. Briefly, the two-dimensional IDCT is decomposed into 16 one-dimensional IDCT operations. First, 8 one-dimensional IDCTs are performed over each row of the input block of data, followed by another 8 one-dimensional IDCTs performed over the columns of the input block. The 1-D IDCT used was based on the work of Wang [74]. Fixed-point arithmetic is used to make the computation faster and reduce the area cost.

Listing 6.1 Source code of the first phase of the FPGA Brook implementation of 2-D IDCT

```
kernel void idctrow (short a<>[64], out short c<>[64]) {                    (1)
  int x0, x1, x2, x3, x4, x5, x6, x7, x8, i;                                (2)

  for (i=0; i<8; i++) {                                                     (3)
    x1 = a[4]<<11;                                                          (4)
    x2 = a[6];                                                              (5)
    x3 = a[2];                                                              (6)
    x4 = a[1];                                                              (7)
    x5 = a[7];                                                              (8)
    x6 = a[5];                                                              (9)
    x7 = a[3];                                                             (10)
    if (!(x1 | x2 | x3 | x4 | x5 | x6 | x7)) {                             (11)
      c[0]=c[1]=c[2]=c[3]=c[4]=c[5]=c[6]=c[7]=a[0]<<3;                     (12)
    } else {                                                               (13)
      x0 = (a[0]<<11) + 128;                                               (14)
      /* first stage */
      x8 = W7*(x4+x5);                                                     (15)
      x4 = x8 + (W1-W7)*x4;                                                (16)
      x5 = x8 - (W1+W7)*x5;                                                (17)
      x8 = W3*(x6+x7);                                                     (18)
      x6 = x8 - (W3-W5)*x6;                                                (19)
      x7 = x8 - (W3+W5)*x7;                                                (20)
      /* second stage */
      x8 = x0 + x1;                                                        (21)
      x0 -= x1;                                                            (22)
      x1 = W6*(x3+x2);                                                     (23)
      x2 = x1 - (W2+W6)*x2;                                                (24)
      x3 = x1 + (W2-W6)*x3;                                                (25)
      x1 = x4 + x6;                                                        (26)
      x4 -= x6;                                                            (27)
      x6 = x5 + x7;                                                        (28)
      x5 -= x7;                                                            (29)
      /* third stage */
      x7 = x8 + x3;                                                        (30)
      x8 -= x3;                                                            (31)
      x3 = x0 + x2;                                                        (32)
      x0 -= x2;                                                            (33)
      x2 = (181*(x4+x5)+128)>>8;                                           (34)
      x4 = (181*(x4-x5)+128)>>8;                                           (35)
      /* fourth stage */
      c[0] = (x7+x1)>>8;                                                   (36)
      c[1] = (x3+x2)>>8;                                                   (37)
      c[2] = (x0+x4)>>8;                                                   (38)
      c[3] = (x8+x6)>>8;                                                   (39)
      c[4] = (x8-x6)>>8;                                                   (40)
      c[5] = (x0-x4)>>8;                                                   (41)
      c[6] = (x3-x2)>>8;                                                   (42)
      c[7] = (x7-x1)>>8;                                                   (43)
    }
    a = a+8;                                                               (44)
    c = c+8;                                                               (45)
  }
}
```

The FPGA Brook implementation of 2-D IDCT consists of two kernels. The first kernel implements 8 one-dimensional IDCTs along rows, while the second kernel implements the remaining operations along columns. The code of the first kernel implementing processing along rows is shown in Listing 6.1. We focus on the usage of a stream of arrays to implement the desired functionality. The outer *for* loop (line 3) is a helper loop that controls iterations through the rows of the array. Each block of input data is stored in a one-dimensional array with 64 elements, organized in standard C, row-major order. Since variables *a* and *c* are declared as streams of arrays, referring to these variables inside the kernel code has the meaning of accessing an array. Therefore, all operations that are normally allowed on arrays in C are supported in FPGA Brook as well, including modifying the array pointers. This is used in lines 44 and 45 to advance the pointers to the next row.

The rest of the code inside the *for* loop implements the 1-D IDCT computation and will not be discussed in detail. This code is shown to demonstrate the complexity of the IDCT computation. We emphasize that the code presented in Listing 6.1 is the source FPGA Brook code, not C2H code. The C2H code of the *idctrow* kernel is generated according to the rules described in the previous chapters. In our experiments the 2-D IDCT benchmark operates on a stream of arrays with 10,000 elements, each array containing 64 elements of the *short* (16-bit) data type.

6.1.11. MPEG-2 Spatial Decoding

The spatial decoding part of the MPEG-2 decoder restores the images of the video sequence from their compressed form into a form suitable for display. Other types of decoding in MPEG-2 include variable-length decoding and temporal decoding [56], which are not part of the FPGA Brook benchmark set and will not be discussed further in this chapter. MPEG-2 spatial decoding benchmark is the most complex benchmark in our suite, consisting of the five component algorithms described in the previous five sections, as shown in Figure 6.3 [56]. The FPGA Brook implementation of this benchmark is simply a composition of the individual kernels interconnected by streams. Kernel code was based on the C code found on the MPEG.ORG web-site [73], StreamIt code included with the StreamIt benchmarks [59], and the pseudocode from [57]. Figure 6.3 can also be seen as a streaming dataflow graph, if we keep in mind that the IDCT algorithm actually consists of two kernels. In our experiments, the MPEG-2 benchmark operates on a stream of arrays with 10,000 elements, each array containing 64 elements of the *short* (16-bit) data type.

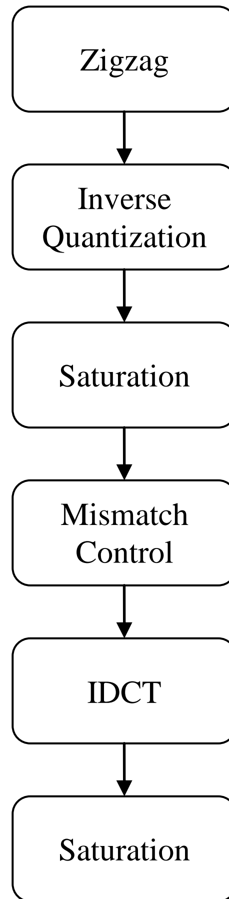


Figure 6.3 MPEG-2 spatial decoding dataflow graph [56]

6.2. Experimental Methodology

To validate the correctness of our design flow and explore the performance of the resulting circuits, we performed a series of experiments using the FPGA Brook benchmark set. We compiled each benchmark using our source-to source compiler and Nios II EDS 7.2, which includes the C2H compiler. The output of this flow was synthesized by Quartus II 7.2 to produce the FPGA programming file. Our experimental system is based on the Nios II processor f(ast) version, with instruction and data caches (4 Kbytes each), and a hardware multiplier unit, connected to an off-chip 8-MB SDRAM module. The system also contains a timer unit to measure performance, a JTAG UART serial port for communication with a terminal program to display the results, and a PLL to generate a system clock of desired frequency. The experiments were run on Altera's DE2 development board [61] with a *Cyclone II EP2C35F672C6* FPGA device [19]. The device has 33,216 LEs, 483,840 bits of memory split into 105 *M4K blocks*, which are memory blocks containing 4 Kbits of memory each. The device also contains seventy 9-bit hard

multipliers, 4 PLLs and up to 475 user I/O pins. The development board contains an 8-MB SDRAM module, which was used to store the Nios II code and data.

We implemented and measured the throughput of each benchmark in three different ways: a software function running on the Nios II processor, the same software function implemented as a C2H accelerator, and the same benchmark implemented as an FPGA Brook program. The correctness of results of the FPGA Brook and C2H implementations was verified by comparing their outputs to the outputs produced by the same application running on the Nios II soft processor. FPGA Brook programs were first implemented without replication, and then the replication factor was increased until no further performance improvement could be obtained, or the design could not fit into the FPGA device used. We implemented each benchmark in three different ways, because we wish to compare the throughput of FPGA Brook implementations with alternative methodologies targeting FPGAs; a soft processor and a behavioural synthesis tool. These comparisons are relevant because our design flow presents the programmer with a compiler interface that is similar to these alternatives. We also compare the area and the maximum operating frequency (F_{max}) of the systems.

FPGA Brook implementations contain one or more streamRead-type operators that generate input data streams, and one streamWrite-type operator that writes the resulting stream to the memory. Since the on-chip memory has limited capacity, the results are stored in the off-chip SDRAM. We do not use the off-chip memory as a source of input data since that would limit the throughput of our implementations, because streamRead-type operators would compete for memory access with streamWrite-type operators. Therefore, it would be hard to distinguish when throughput is limited by the off-chip bandwidth, versus limitations of our design flow. To avoid this, the streamRead-type operator generates input data using a simple *for* loop and setting stream elements to the values of the loop iterator. To make the comparison with software and C2H fair, we used the same method when implementing these two approaches.

In most of our FPGA Brook experiments we used FIFOs with depth 4. This is because most benchmarks have little variability in processing time between the elements, so they would not benefit from deeper FIFOs. Hormati et al. [42] also found that FIFO depths when implementing StreamIt streaming programs do not need to be high to achieve optimum performance. In fact, they found that "it is common that a queue size of one element is sufficient for correct execution that is also as efficient as a rate-matched static scheduled." We use higher FIFO depths only for benchmarks using streams of arrays, when replication factor is higher than 4, in which case we use FIFO depth equal to the replication factor to fully exploit parallelism, as discussed in section 5.1.2.

We implemented each benchmark in C and compiled it for the Nios II processor to obtain the relative performance compared to the FPGA Brook implementation. We also compiled each of the benchmarks written in C using the C2H compiler. In most cases, only one version of each application was implemented using C2H, because C2H does not support any direct way of exploiting data parallelism,

unlike FPGA Brook. To make the comparison between the three approaches fair, we adjust the code of the C and C2H implementations where appropriate. First, we do not read the input dataset from memory, but rather use loop iterators as the input data. While the same is done for the streamRead-type operators in the FPGA Brook implementations, data generation is a separate hardware unit from the kernels performing computation in FPGA Brook. In software and C2H implementations, using iterators sometimes results in trivially simple code that can be perfectly pipelined, so it processes one data unit per clock cycle. While this approach is sometimes unfair to FPGA Brook implementations, our experiments show that even under such circumstances our design flow can greatly exceed the performance of the C code running on Nios II and C2H compiled hardware for all but the simplest benchmarks. When writing C code to be compiled by the C2H compiler, we also perform two additional optimizations. First, we avoid using modulo or division operations in the code by manually modifying the code to achieve the same functionality. This is in contrast with our source-to-source compiler, which performs many of these transformations automatically. However, we believe that the comparison would be unfair to the C2H implementation if we did not perform this manual step. Another optimization we perform is modifying the code for local array initialization inside hardware accelerators, such as the ordering matrix in the zigzag benchmark. If the C function that will be accelerated using C2H contains a local array declaration with an initializer list (i.e. list of initial values in curly braces), the C2H compiler allocates such an array in the main processor's memory, so that it can be correctly initialized before the accelerator is called [27]. Since the main processor's memory is in the off-chip SDRAM, this would result in a bottleneck when accessing the SDRAM, because it is also used to store results. To avoid this, we modify the code to declare the array without the initializer, which causes it to be implemented in on-chip memory, and then initialize all of its elements through manual assignments at the beginning of the code. We also note that our source-to-source compiler performs this transformation automatically for arrays declared inside kernels.

To measure throughput of a benchmark, we first implement it in a system using the Nios II processor with debug support, which allows us to download code to the Nios II processor and start its execution. We execute each benchmark and measure its running time. The benchmarks are executed in a system running at a relatively low clock frequency of 50 MHz, which allows us to run all the experiments uniformly, regardless of the actual F_{max} of the system.

To measure the area and F_{max} of each benchmark, we re-create the system in a configuration that we expect would be used in production environments, by removing unnecessary components that are useful only for debugging. We remove the debug support for the Nios II processor, and we also remove all unnecessary connections from the Nios II data master to the FIFO buffers in the system. As discussed in section 4.4, these connections are inserted so that the system passes the integrity checks performed by the C2H compiler and are useful for system debugging. In the current implementation, these connections

have to be removed manually by editing the system in the SOPC Builder component of Quartus II and regenerating the system after that. Next, we set the clock speed to 135 MHz, which is the frequency we determined experimentally to be a reasonable target for most systems generated using our flow.

We also modify several Quartus II settings to better optimize the circuits. We set the *Analysis and Synthesis* optimization option to optimize for speed (rather than area), because our main goal is high performance. Similarly, we set the *Fitter effort* option to "Standard Fit (highest effort)", meaning that placement and routing algorithms optimize the design as much as possible, even after meeting the target clock speed. We also disable the "Smart compilation" option to ensure that the system is always synthesized completely. Finally, we disable generation of *OpenCore hardware evaluation*, which is extra logic generated to allow evaluation of hardware modules for which the proper license does not exist. We disable generation of this logic because we ran our experiments on several computers, some of which did not have the C2H license. This option prevents generation of the extra logic so that the area results obtained after compilation are accurate. At the end, we compile the system using the *DSE Explorer* component of Quartus II, which we configure to automatically compile the design five times, each time with a different seed. We report the F_{max} and area results of the compilation that achieved the highest F_{max}. We use this F_{max} to calculate the achievable throughput. We note that the SDRAM chip on the DE2 board cannot run at speeds higher than 100 MHz, so the throughput we report cannot be fully realized on the DE2 board. However, faster memories that operate at higher clock frequencies are readily available and could be used with the FPGA chip we target.

6.3. Experimental Results

In this section we present the experimental results obtained when running our benchmarks on the DE2 board. The goal of our experiments is to determine how kernel performance scales with replication and to compare FPGA Brook implementations to C and C2H implementations. In this section, we only show results relative to the baseline FPGA Brook implementation without replication. Detailed results are available in Appendix A. The baseline FPGA Brook implementation is labelled *FBROOK* in all the graphs. Replicated FPGA Brook implementations are labelled by the desired speedup specified by the speedup pragma in the code (e.g. 8X). Finally, C2H implementation and code run on the Nios II soft processor are labelled as *C2H* and *SOFT*, respectively. In this section we show two throughput results; *wall-clock throughput* and *cycle throughput*. *Wall-clock throughput* is the throughput achievable when the system runs at the maximum possible operating frequency for that system. *Cycle throughput* is the throughput achieved when the system runs at 50 MHz. We call it cycle throughput because it correlates with the number of clock cycles it takes to process a predefined amount of data, without taking clock frequency into account. When comparing two implementations of the same algorithm it is useful to

consider the cycle throughput in addition to wall-clock throughput to understand whether the difference in wall-clock throughput is due to the difference in F_{max}, due to the difference in the number of clock cycles needed for processing, or both. Consequently, we use the term *wall-clock speedup* to refer to the improvement in wall-clock throughput, and term *cycle throughput improvement* to refer to the improvement in cycle throughput.

Area results are shown in terms of the number of logic elements (LEs) and the number of 9x9 hard multipliers required to implement the benchmark. In this section, we do not show the amount of memory used by applications, because memory was not critical for any of the benchmarks. Memory utilization for our benchmarks can be found in Appendix A.

6.3.1. Autocorrelation

Performance and area results for the autocorrelation application are shown in Figure 6.4. The results show that the replicated FPGA Brook implementation scales well up to 8X speedup, but fails to scale to 16X. This is because at 16X this application is already processing one sample per clock-cycle. This is the maximum achievable performance in our framework, because streamRead-type operators cannot be replicated, so the highest throughput they can achieve is one data element per clock cycle. Similarly, streamWrite-type operators can only write data to the memory one element per cycle. Interestingly, C2H achieves this maximum performance directly from the C code at a much lower area cost. This benchmark is heavily biased in favour of C2H because of its simplicity. As discussed in section 6.2, we use loop iterators for sample values, which results in a perfectly parallel loop for C2H, so processing is performed

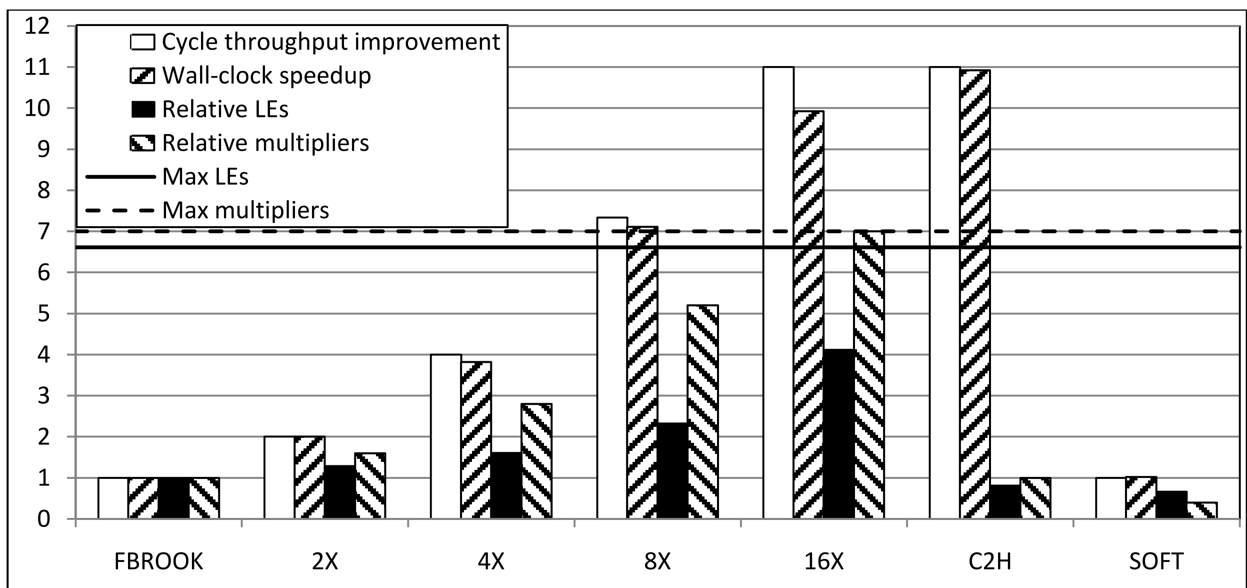


Figure 6.4 Relative throughput and area of the autocorrelation application

at a rate of one data element per clock cycle. The Brook implementation has overheads associated with communicating data through FIFOs and maintaining modulo counters to implement reductions, so it only achieves the maximum performance after replication.

The C2H implementation requires less area than even the baseline FPGA Brook implementation. The FPGA Brook implementation uses FIFOs to pass stream elements between kernels, and accessing the FIFOs requires Avalon master ports inside of accelerators, which take a significant amount of area. Since the C2H implementation generates input data in the accelerator, it only requires one Avalon master port for access to the off-chip memory to write the results, so it uses less overall area.

The results in Figure 6.4 show that replication does not have a significant effect on the F_{max} of the system, except for the system with speedup of 16. This can be observed by comparing the values of cycle throughput improvement and wall-clock speedup; any difference between the two is due to differences in the F_{max} between the corresponding system and the baseline system. The system with speedup of 16 has slightly lower F_{max} because it requires more multipliers than the number of hard multipliers available in the FPGA. As a result, some multipliers are implemented in soft-logic, which affects the system's F_{max}. This also explains the sudden rise in the number of LEs required to implement the system with speedup of 16, compared to the system with speedup of 8.

The lines indicating the maximum number of LEs and multipliers in Figure 6.4 are shown relative to the number of LEs and multipliers needed to implement the baseline FPGA Brook implementation, just like all the other values in the graph. As a result, these lines will appear at different values for different benchmarks (e.g. in Figure 6.5), because the baseline implementation has different logic utilization for different benchmarks. These lines are useful indicators of the LE and multiplier utilization. For example, looking at the multiplier utilization for the 16X implementation in Figure 6.4, it is easy to see that all the multipliers have been utilized, and that just over 2/3 of LEs have been utilized.

6.3.2. FIR Filter

Performance and area results of the FIR filter application with 8 and 64 taps are shown in Figures 6.5 and 6.6, respectively. These benchmarks exhibit different behaviour than the autocorrelation application. The FPGA Brook implementation of the 8-tap FIR filter scales to speedup of 2X, but not to 4X. This is because the streamRead-type operator in this application simulates delay elements using a circular memory buffer. The circular buffer implementation requires maintaining two modulo counters, so the implementation of this buffer quickly becomes a bottleneck. This conclusion is confirmed by the performance of the 64-tap FIR filter. This filter has a larger circular buffer, so one of the modulo counters is incremented less frequently, providing better performance. As expected, scalability of this design is better than that of the 8-tap filter, with the 4X implementation almost achieving a speedup of four,

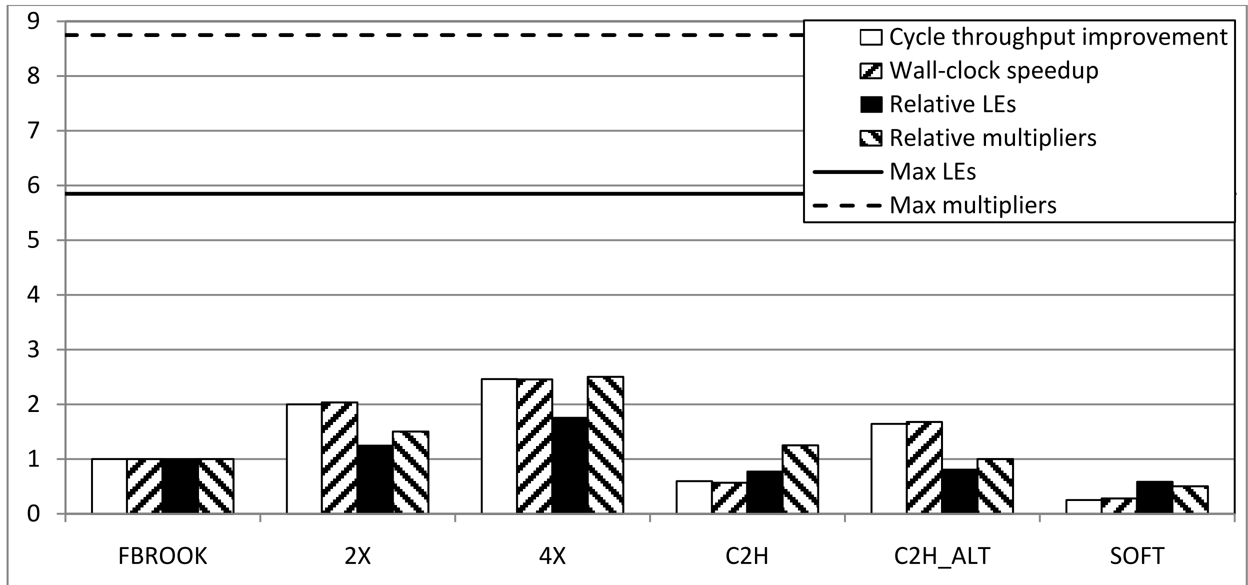


Figure 6.5 Relative throughput and area of the FIR filter application with 8 taps

although the speedup does not scale further. We expect that the performance of FIR filters with more taps would scale even further. This is significant, because filters with higher numbers of taps are computationally more intensive.

We use the two FIR filter implementations to demonstrate how the coding style of C programs affects C2H performance. At first, we used the same C code for both the C2H and soft processor implementations, except that we replaced the modulo operations with modulo iterators in C2H

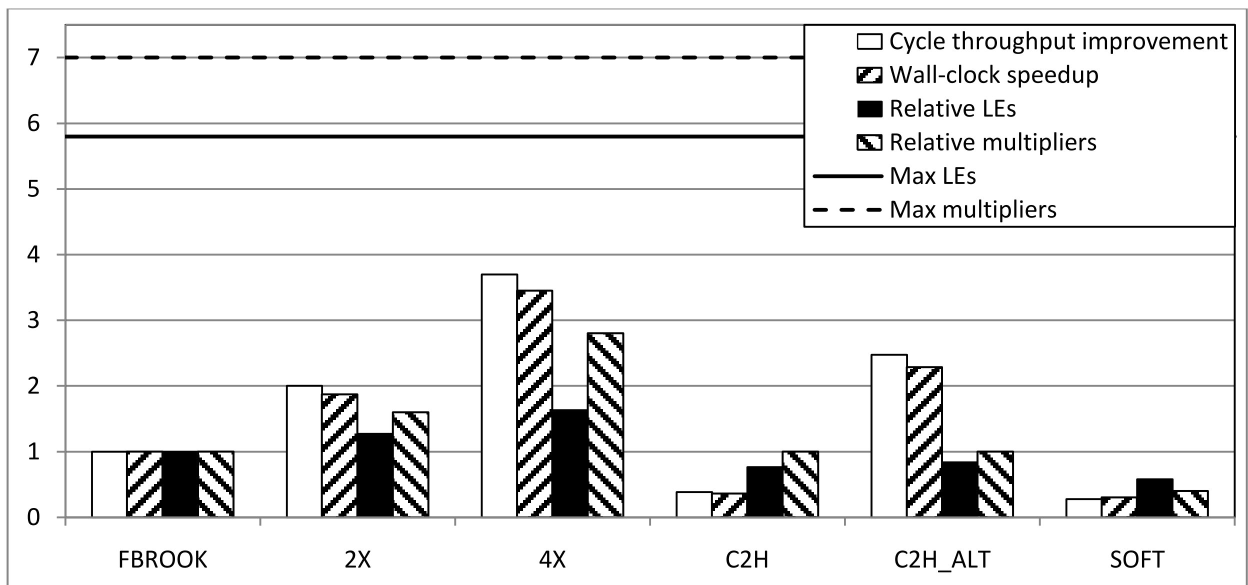


Figure 6.6 Relative throughput and area of the FIR filter application with 64 taps

implementations. Results for this code are labelled as C2H in Figures 6.5 and 6.6. As we learned more about the C2H compiler, we found out that a locally declared array is implemented in off-chip memory if it contains an initializer list. This is because all such arrays are initialized by the Nios II processor, and C2H does not make arrays local to accelerators accessible to the processor. The FIR filter implementation contained two such arrays; one storing the coefficients, and another implementing the circular buffer. We modified the code to declare both arrays without the initializer lists, which caused them to be implemented in private on-chip memories, local to the hardware accelerator. To initialize these arrays, we added an explicit assignment statement into the C2H code for each element of the array. These assignments are performed only once before any processing is performed because both arrays are read-only. Results for this alternative implementation are labelled as C2H_ALT in Figures 6.5 and 6.6. The results show that this small change had drastic effect on performance; the 8-tap FIR filter achieved almost 3 times higher throughput, while the 64-tap FIR filter achieved more than 6 times higher throughput.

Comparison of the FPGA Brook implementations and the basic C2H implementations shows that the baseline FPGA Brook implementation outperforms the basic C2H implementation, although the FPGA Brook implementation uses more LEs. The FIR filter is a more complex algorithm than autocorrelation, so the C2H compiler cannot easily optimize the C code implementing it. The FPGA Brook implementations benefit from both task and data parallelism. Task parallelism is exploited because processing is divided among the streamRead-type operator, two kernels, and the streamWrite-type operator. Data parallelism is exploited because multiple kernel replicas perform computation in parallel, while C2H does not support such an option. Thus, the FPGA Brook implementations achieve higher performance than C2H at the expense of increased area. C2H achieves higher performance than software running on the Nios II soft processor because it exploits some instruction level parallelism (ILP) in the C code, and also because the software running on the processor uses the modulo operation to implement the circular buffer.

The alternative C2H implementation outperforms the baseline FPGA Brook implementation of the 8-tap FIR filter, and even the 2X implementation of the 64-tap FIR filter. However, 4X FPGA Brook implementations outperform both C2H implementations. We note that two optimizations that favour C2H implementations were performed manually on the C code in the basic C2H implementation; iterator values were used as input sample values (instead of being read from memory), and modulo operations were replaced by iterators. An additional optimization on local arrays was performed for the alternative implementation, as described above. We conclude that while carefully crafted C code compiled with the C2H compiler can achieve good performance, a sufficiently replicated FPGA Brook implementation can still outperform it. This is because the FPGA Brook language encourages a coding style that exposes parallelism and avoids pitfalls that reduce effectiveness of C2H results. We will revisit this topic in more detail in the next chapter.

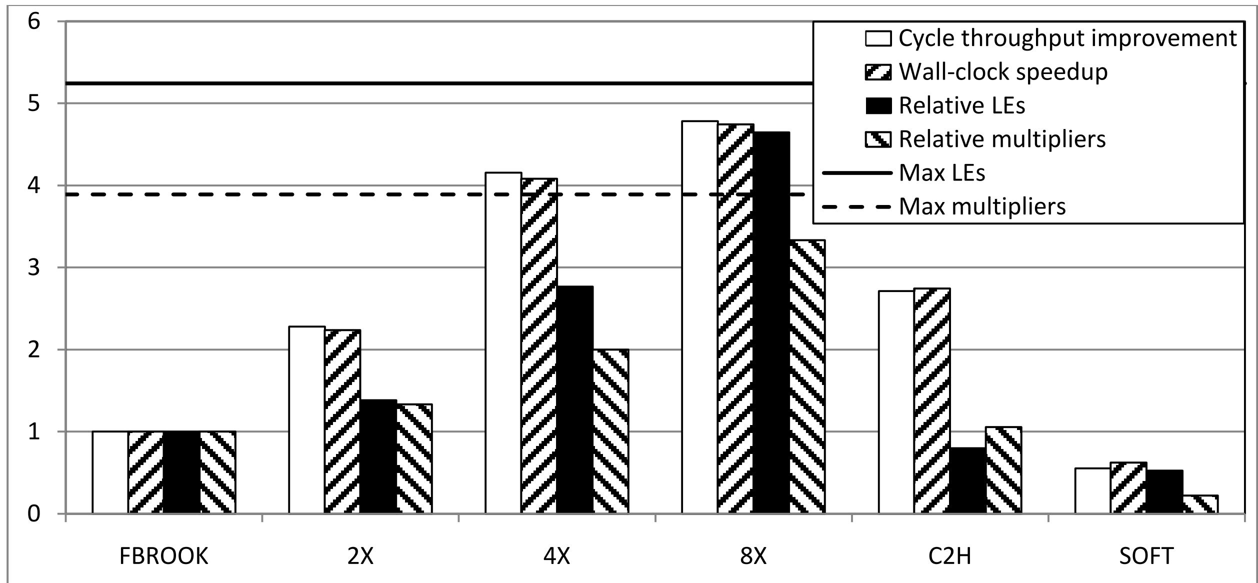


Figure 6.7 Relative throughput and area of the two-dimensional convolution application

6.3.3. Two-Dimensional Convolution

Performance and area results of the two-dimensional convolution application are shown in Figure 6.7. This benchmark exhibits performance trends similar to the FIR filter; it scales well up to the speedup of 4, but achieves only marginally higher performance when speedup of 8 is specified in the code. This is because of a relatively complex streamRead-type operator that expands the original image as discussed in section 6.1.3, which is a bottleneck in this application. C2H performance is better than the baseline FPGA Brook implementation because it can take advantage of some ILP optimizations due to the loop iterators being used as input values. However, the code is more complex than for the autocorrelation application, so the C2H compiler cannot optimize it to the same degree. Although this benchmark is also biased in favour of C2H, FPGA Brook replicated implementations 4X and 8X outperform it significantly by exploiting data parallelism. Finally, all hardware implementations outperform software, as expected.

6.3.4. Matrix-Vector and Matrix-Matrix Multiplication

Performance and area results of the matrix-vector and matrix-matrix multiplication are shown in Figures 6.8 and 6.9, respectively. One surprising result in these experiments is the apparent super-linear speedup due to replication. This effect occurs because of the way 2-D reductions are replicated. In a non-replicated case, 2-D reductions are implemented using three modulo and division iterators, as discussed in section 4.2.3. When such a reduction is replicated, it is decomposed into two phases, as discussed in section 5.3. In the case of matrix-vector and matrix-matrix multiplication, all reduction replicas implement 1-D reductions, which use fewer modulo counters and thus achieve better performance than

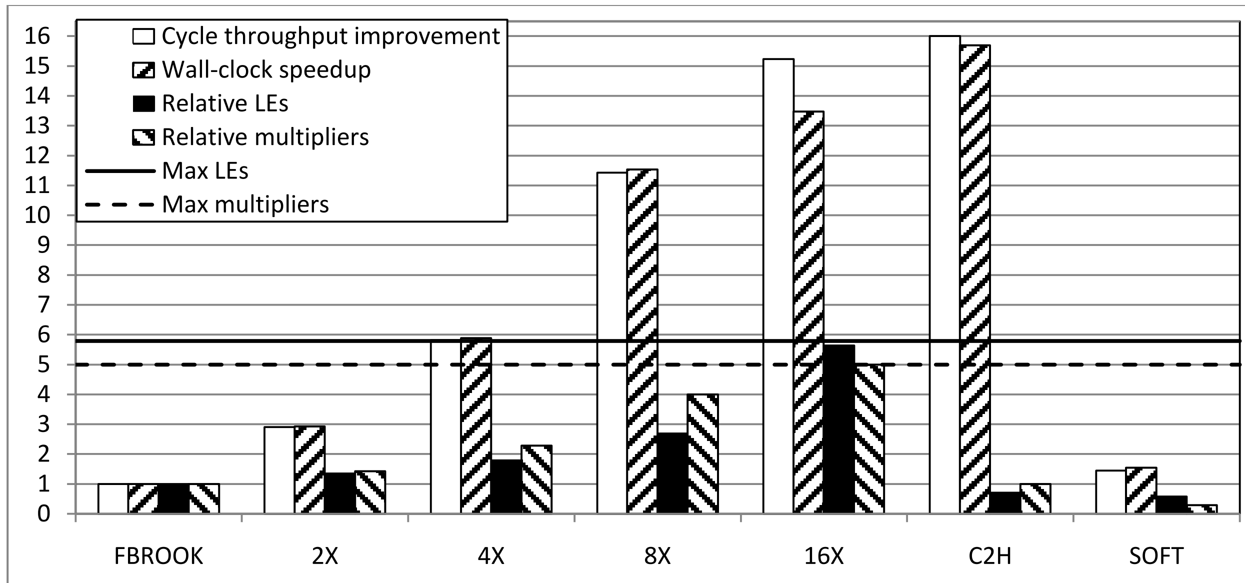


Figure 6.8 Relative throughput and area of the matrix-vector multiplication application

their 2-D counterparts. Since our replication algorithm does not account for this effect, as described in section 5.4, the obtained speedup is higher than expected.

Matrix-vector multiplication scales well up to 16X, at which point it processes one element per clock cycle. The 16X implementation suffers from decreased F_{max}, because all hard multipliers are utilized and some are implemented in soft logic. Matrix-matrix multiplication scales well up to 8X, also processing one data element per clock cycle at this replication factor. Similar to the autocorrelation application,

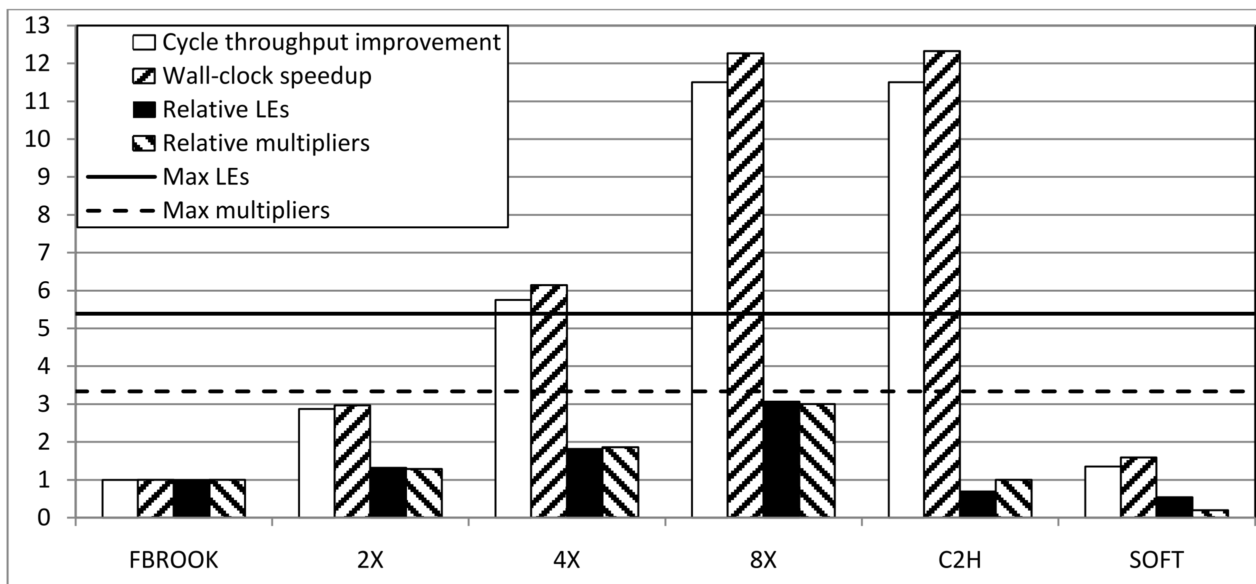


Figure 6.9 Relative throughput and area of the matrix-matrix multiplication application

C2H implementation achieves performance that matches the best FPGA Brook implementation, because the loop nest implementing the applications is perfectly parallel, due to loop iterators being used as data elements.

6.3.5. Mandelbrot Set Generator

Performance and area results of the Mandelbrot set generator application using 16- and 32-bit data representations are shown in Figures 6.10 and 6.11, respectively. The results show that the 16-bit Mandelbrot implemented in FPGA Brook scales well up to 8X. At 16X, cycle speedup is 14.7, but because of decreased F_{max} the actual speedup is approximately 11X. As in other benchmarks, F_{max} is decreased because some multipliers are implemented in soft logic. An interesting observation is that at 16X the cycle throughput improvement is not closer to 16. Considering that computation on individual points is completely independent, and that the streamRead-type operator is very simple, we expected to obtain perfectly linear cycle throughput improvement. The reason is the nature of distributor and collector FIFOs, which block reads or writes from all ports other than the one currently selected for reading or writing, as discussed in section 5.1.1. Since the Mandelbrot benchmark exhibits significant variation in processing times between elements, this can impact overall performance. Therefore, a collector and distributor FIFO design proposed in Figure 5.2b should be used for large replication factors, particularly for programs whose kernel processing time varies between elements.

The performance of the 32-bit Mandelbrot application implemented in FPGA Brook scales up to 2X, but suffers from decreased F_{max} already at 2X, because 32-bit multiplication requires more multipliers than 16-bit multiplication. The FPGA we used could not fit designs larger than 4X for Mandelbrot32. It is

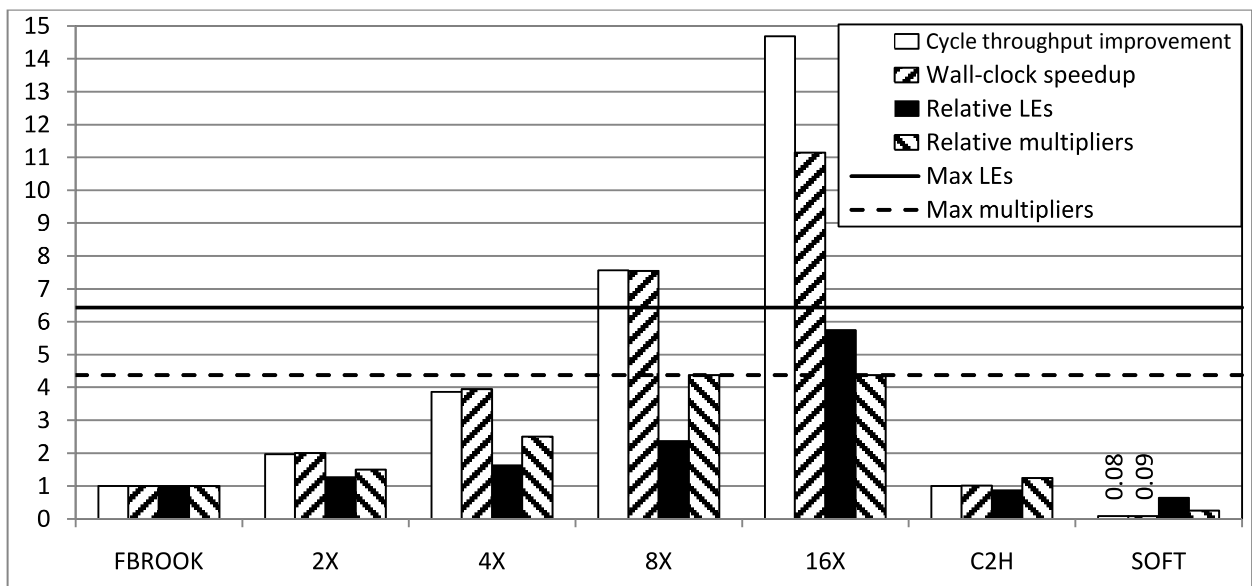


Figure 6.10 Relative throughput and area of the Mandelbrot application with 16-bit data

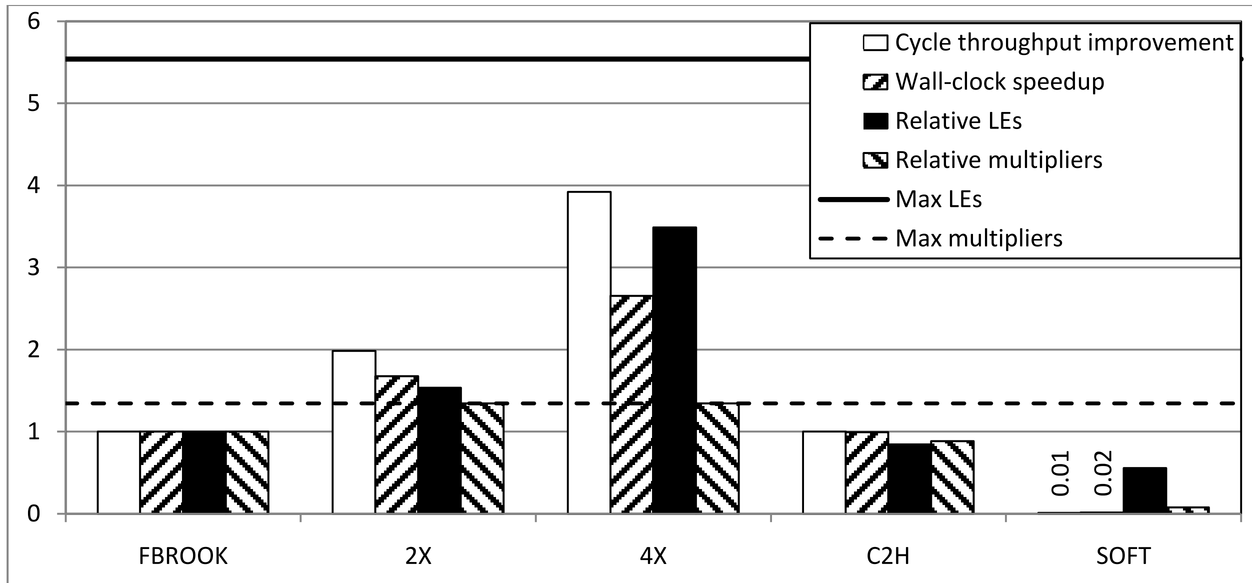


Figure 6.11 Relative throughput and area of the Mandelbrot application with 32-bit data

interesting to observe that Mandelbrot32 achieves much higher speedups over software than Mandelbrot16. This is because multiplication of two 32-bit numbers requires the result to be stored in a 64-bit register, which is more easily implemented in hardware, than in software.

C2H achieves the same performance as the basic streaming implementation, because it does not take advantage of data parallelism. While Altera has reported better speedups for Mandelbrot than our C2H results [75], this is because their implementation was manually designed to perform processing in parallel, which included creating a sufficient number of parallel processing elements and procedures for dividing and merging data. In principle, this is similar to how streaming implementations operate, although their actual implementation differs from our approach by using only one C2H accelerator. Our compiler performs data partitioning and instantiation of a sufficient number of processing elements automatically and thus simplifies the design process.

6.3.6. Zigzag Ordering, Inverse Quantization and Mismatch Control

Performance and area results of the zigzag ordering application are shown in Figure 6.12. The FPGA Brook implementations scale reasonably well up to 4X. The 8X implementation does not scale to speedup of 8 because this implementation already processes close to one data element per clock cycle, which is the maximum possible in our framework. The C2H implementation exceeds the performance of the baseline and 2X FPGA Brook implementations because we use iterator values as data inputs. Even so, our 4X and 8X implementations outperform C2H. Since this application does not perform any multiplications, all

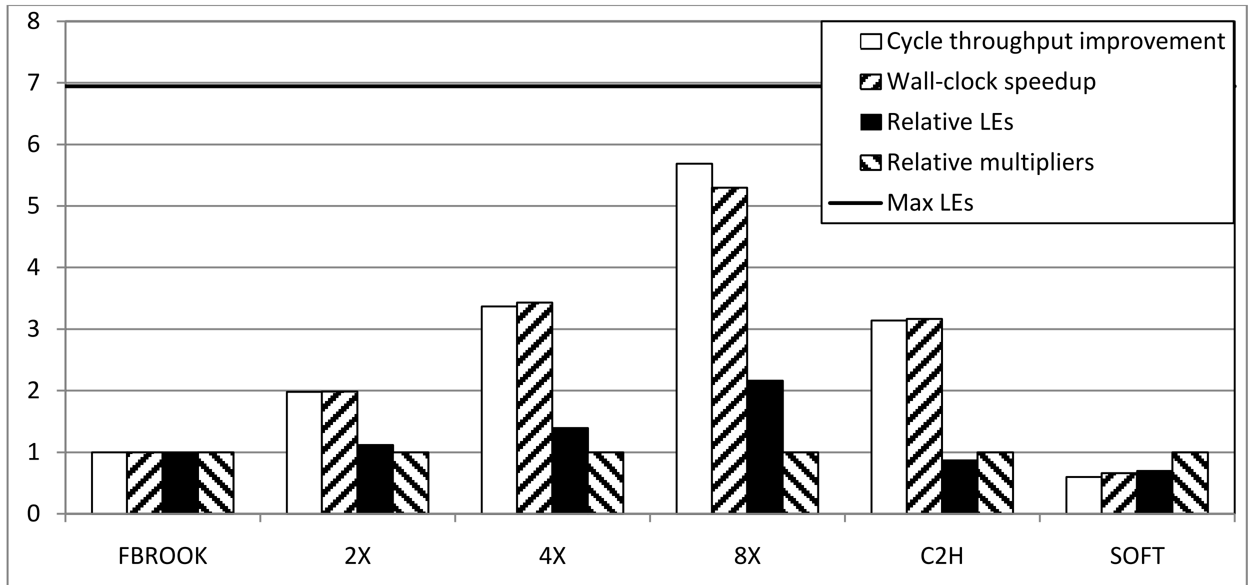


Figure 6.12 Relative throughput and area of the zigzag ordering application

implementations use the same number of multipliers (4 out of 70 available on the chip). Hence, we do not show the maximum number of multipliers in Figure 6.12 to keep the other results clearly displayed.

Results for the inverse quantization application, shown in Figure 6.13, exhibit similar performance trends as zigzag ordering. The only difference is that inverse quantization uses more multipliers, because the algorithm involves multiplication. Finally, Figure 6.14 shows the results for the mismatch control application, whose performance trends are also similar to the zigzag application. The only exception in this case is the unusually high performance of the C2H implementation. This is because the inner loop of

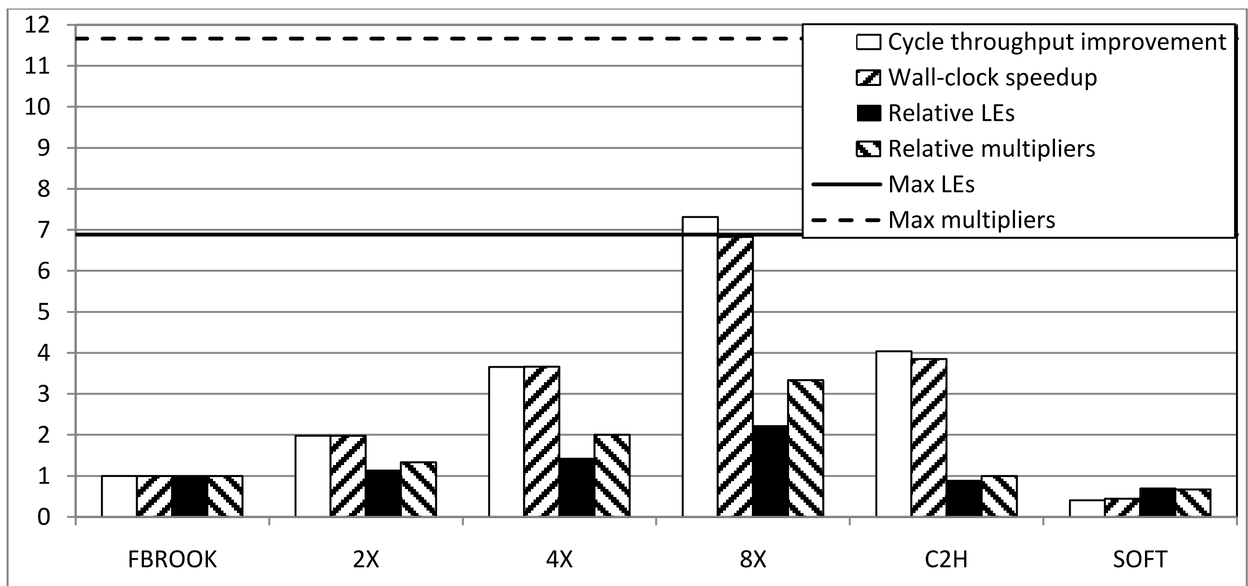


Figure 6.13 Relative throughput and area of the inverse quantization application

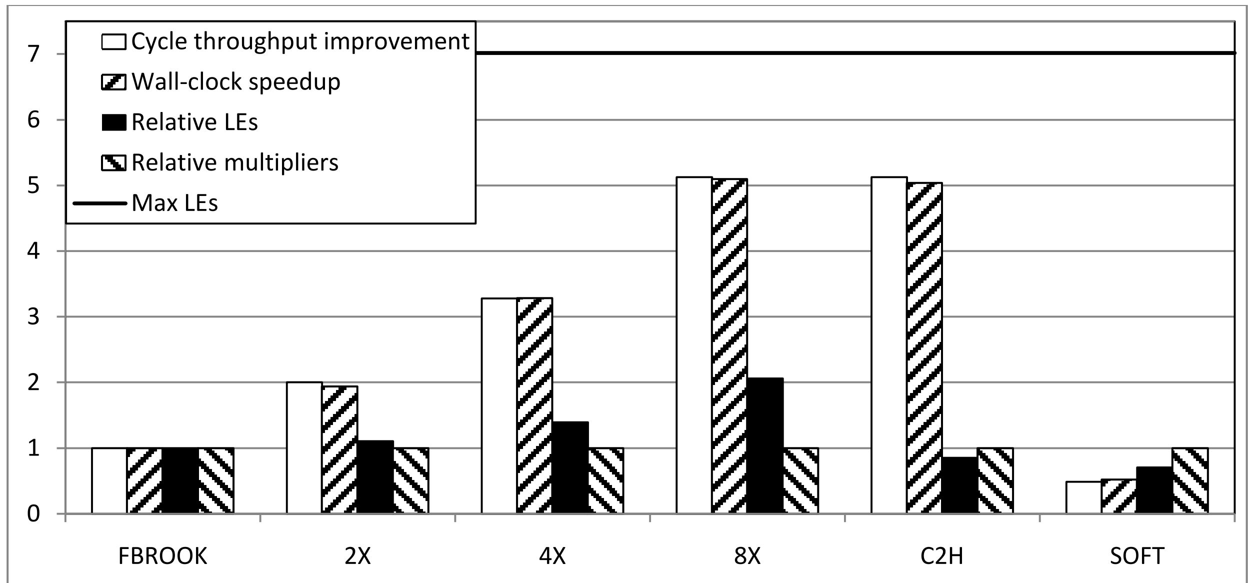


Figure 6.14 Relative throughput and area of the mismatch control application

this benchmark is perfectly parallel due to loop iterators being used as data inputs, so the C2H implementation comes close to processing one element per clock cycle. This is similar to the behaviour of the C2H implementation of the autocorrelation application.

6.3.7. Saturation

Performance and area results of the saturation application are shown in Figure 6.15. This application exhibits similar performance trends as the zigzag application up to 8X speedup. However, cycle throughput improvement of this application does not scale to 16X. Looking at the absolute performance, we find that at 16X this application does not yet reach the limit of one data element per clock cycle. Therefore, there has to be another explanation for this lack of scalability. As discussed in section 5.1.2, array FIFO collectors and distributors use FIFO buffers whose depth equals the number of their inputs or outputs. This means that collectors and distributors for the 16X implementation use FIFO buffers that are 16 slots deep. To explore whether this may be limiting parallelism to some degree, we created an alternative implementation of the 16X design, which uses FIFO buffers that are 32 slots deep. Results for this implementation are labelled as 16X_ALT in Figure 6.15. As can be seen, cycle throughput improvement now scales to 16X, although wall-clock speedup actually suffers compared to the regular 16X implementation. This is because of a significantly lower F_{max} for the 16X_ALT design, which is caused by the large 32-to-1 multiplexer necessary to implement such large array FIFO collectors and distributors. In fact, even the ordinary 16X implementation has somewhat lower F_{max} than implementations with lower speedups. Timing analysis shows that this is also due to the multiplexers

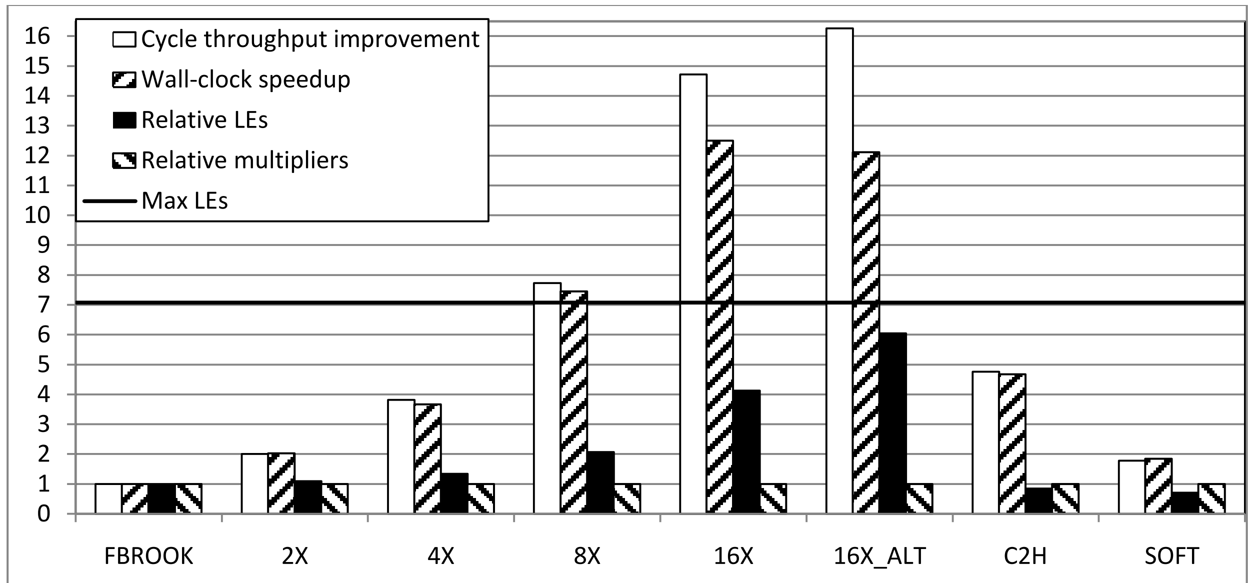


Figure 6.15 Relative throughput and area of the saturation application

inside the array FIFO collectors and distributors. This is not a surprising result, as large multiplexers are known to be challenging to implement efficiently in FPGAs. While this has only a small effect on our results, the effect would become more pronounced for larger replication factors. Therefore, we expect that array FIFO collectors and distributors will have to be modified by pipelining the multiplexers, if replication factors higher than 16X are needed.

Considering our discovery that collectors and distributor FIFOs with FIFO depth equal to the number of inputs and outputs do not provide perfectly linear scalability, we observe that this effect is noticeable, although to a lesser extent, for replication factors of 4 and 8 and also for applications zigzag, inverse quantization and mismatch control. Replication factor of 2 always provides the cycle throughput improvement of 2, because we use FIFO depth of 4 for these implementations. We conclude that FIFO buffers in array collectors and distributors should have the depth slightly higher than the number of inputs or outputs of the module to provide linearly scalable speedup. The exact depth needed to achieve linear speedups likely depends on the characteristics of the application, such as the amount of variability in processing times between elements.

Another surprising result is the low absolute throughput of the baseline FPGA Brook implementation of the saturation application, which is approximately 3 times lower than the throughput of the zigzag and inverse quantization applications, although the algorithms they perform are of similar computational complexity. The reason for this is the coding style used when writing the processing kernel for the saturation benchmark. Briefly, the kernel compares every input value with the lower and the higher bound of the range. If the input value is outside this range, the output value for that element is set to either the minimum or the maximum value, as appropriate. If the value is within the range, the output value is set

equal to the input value. The algorithm requires access to the input value three times, and in our benchmark implementation each access is performed by accessing it directly from the array FIFO preceding the processing kernel. While this coding style was a mere coincidence, and the performance of this benchmark could be easily improved by modifying the code to read this value into a local register only once, we decided to leave it as is, because it represents an interesting real-world scenario. For example, if a designer who designed one or more kernels left the company, the (poorly written) code left behind may be too costly to optimize by another designer unfamiliar with the code. FPGA Brook offers to solve this problem through replication. Although the poor coding style causes low throughput of the baseline implementation, replication successfully improves the throughput and brings it close to the throughput of other kernels of similar complexity. While it is true that the improvement comes at an increased area cost, such a trade-off is sometimes desirable.

6.3.8. Inverse Discrete Cosine Transform (IDCT)

Performance and area results for the IDCT application are shown in Figure 6.16. IDCT performs complex computations and thus requires a large amount of logic for its implementation. The largest replicated implementation that could fit in the FPGA device we used was 4X. The cycle throughput improvement scales well for both 2X and 4X implementations, although wall-clock speedup does not scale for 4X because of decreased F_{max}. The F_{max} decreases for the 4X implementation because some multipliers are implemented in soft logic. The baseline FPGA Brook implementation achieves higher performance than C2H because the FBROOK implementation implements the IDCT computation as two

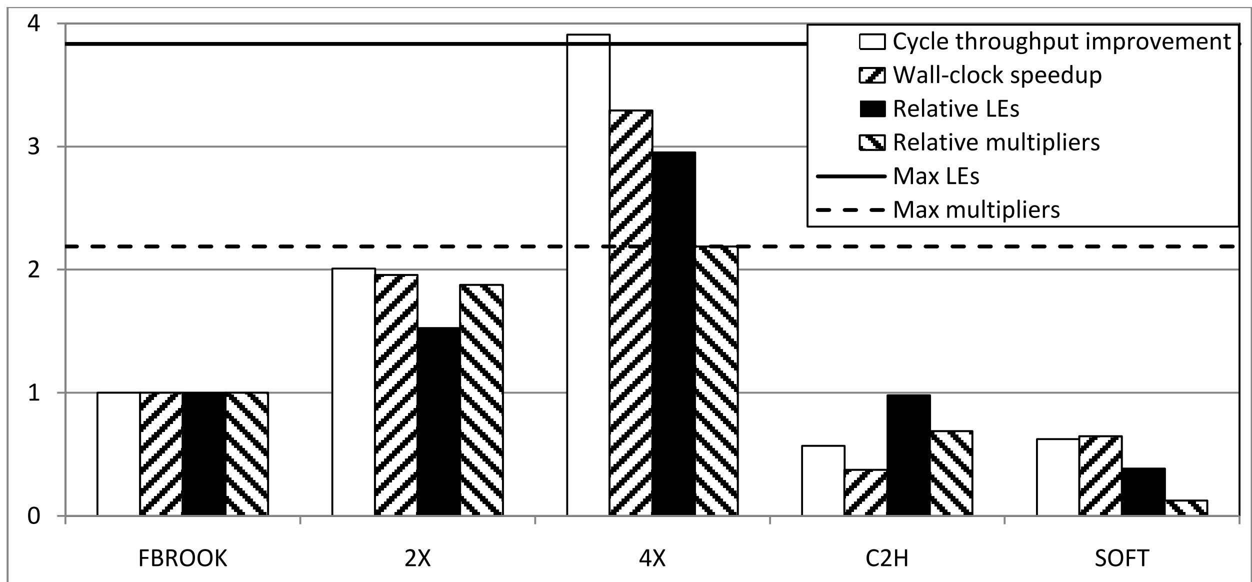


Figure 6.16 Relative throughput and area of the IDCT application

kernels, as described in section 6.1.10, so it benefits from task parallelism. C2H achieves performance similar to the C code running on the soft processor because of many dependencies in the code, which limit instruction level parallelism.

6.3.9. MPEG-2 Spatial Decoding

As discussed in section 6.1.11, MPEG-2 spatial decoding is a composition of algorithms whose performance has been presented in the previous sections. Therefore, we expect performance trends of spatial decoding to be related to performance trends of its components. Figure 6.17a shows the streaming dataflow diagram of MPEG-2 spatial decoding when no kernels are replicated. This diagram is equivalent to the dataflow diagram in Figure 6.3, except that kernel names have been abbreviated and the IDCT node has been split into two kernels, IDCTR and IDCT, representing IDCT row and column computation, respectively. The numbers next to the nodes show the throughput in KB/s of the corresponding kernel when implemented separately from other kernels. The exception is the IDCT kernel, for which we show the same throughput value next to both parts of the algorithm. This is because we only measured the performance of the overall IDCT and did not perform experiments with each kernel separately, since the two kernels work together to implement IDCT.

The number on top of the diagram shows the wall-clock throughput measured when the MPEG-2 spatial decoding application is implemented on the DE2 development board according to the streaming dataflow graph shown in the figure. This throughput is equal to the lowest throughput of a node in the dataflow graph, which are the two saturation kernels in this case. The small difference in throughput is due to a difference in F_{max} between the system implementing the saturation kernel and the system implementing MPEG-2 spatial decoding (see Tables A.11 and A.14); cycle throughputs of the two systems are identical.

To improve the performance of the system in Figure 6.17a, the saturation kernel, which is the bottleneck, should be replicated. This scenario is shown in Figure 6.17b. The number next to the saturation kernel nodes in this figure shows the throughput of the saturation kernel when implemented independently with 2X speedup. In this scenario, the bottleneck is the IDCT implementation, and throughput of the system matches the throughput of the IDCT kernels; the small difference being completely accounted for by the difference in F_{max} between these two systems.

Continuing this process, we replicate the IDCT kernels, and obtain the system shown in Figure 6.17c. In this system, the saturation kernels are again the bottleneck, and the system throughput is close to the throughput of the saturation kernels. The difference between the two throughputs is mostly due to differing F_{max} of the two systems, although there is also a slight difference in cycle throughput. Further improvement can be achieved by replicating the saturation kernels further, obtaining the system in Figure 6.17d. In this figure two system throughputs are given. Using our regular design flow, we obtained the

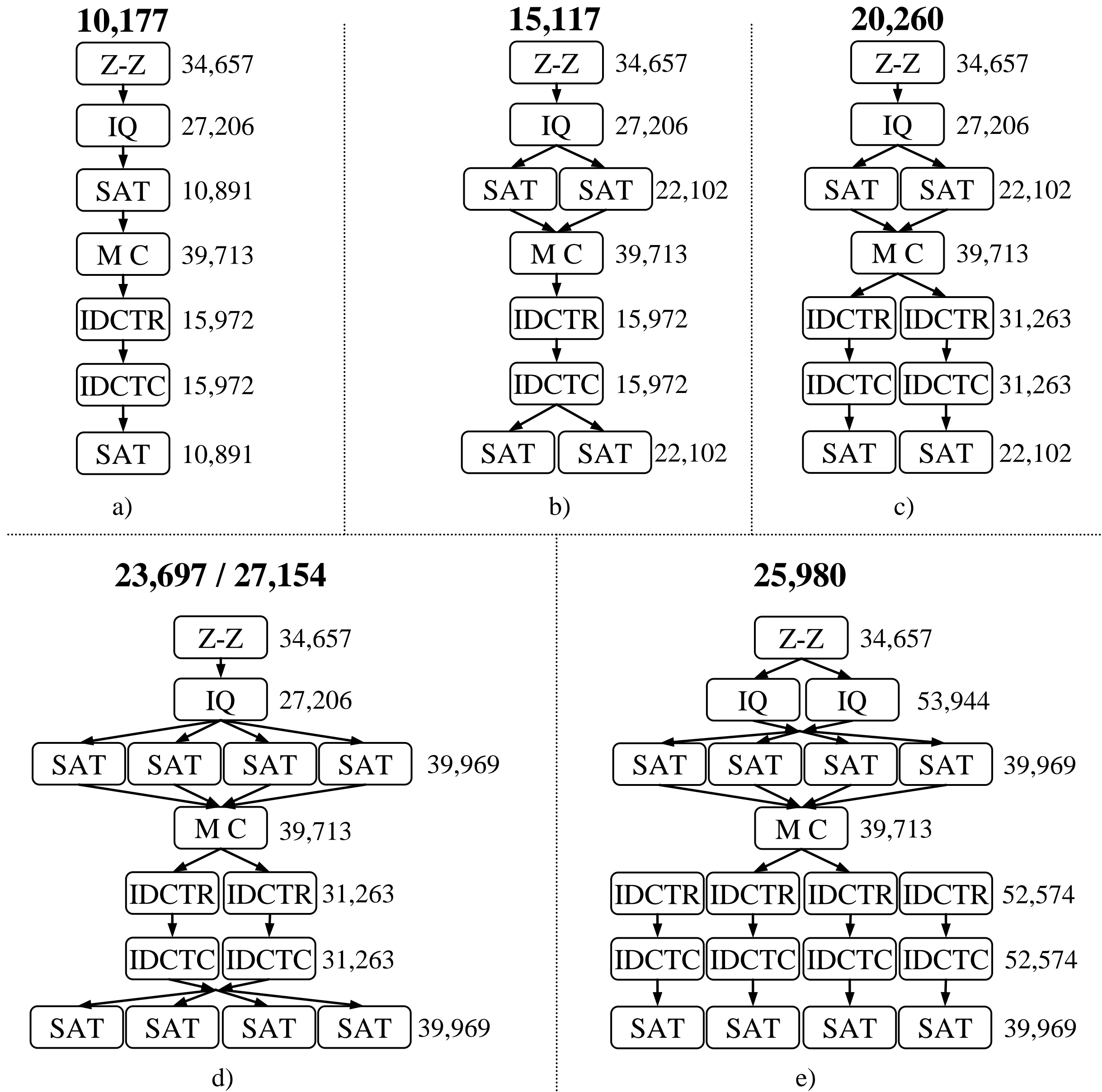


Figure 6.17 Wall-clock throughput of different replicated configurations of MPEG-2 spatial decoding implemented on the DE2 development board (KB/s)

throughput of 23,697 KB/s, while the bottleneck kernel, which is the inverse quantization in this case, can sustain 27,206 KB/s. This discrepancy is higher than in the previous cases because of declining F_{max}, but also because of growing discrepancy in the cycle throughput. This is due to the effect of collector and distributor nodes with FIFO depth of 4, as discussed in section 6.3.7. To improve on this, we modified all array collectors, distributors and collector/distributor nodes with 4 inputs or outputs to have FIFO depth of 8. This resulted in the throughput of 27,154 KB/s. Cycle throughput of this improved system also

matches the cycle throughput of the bottleneck kernel perfectly. This confirms our assumption that array collectors and distributors limit performance when FIFO depth equals the number of inputs or outputs.

Additional throughput improvement should be possible by further replicating the IDCT kernels and the inverse quantization kernel to obtain the system in Figure 6.17e. This implementation used FIFO depth of 4 throughout the design. Increasing the FIFO depths further was not feasible because the FPGA device was nearly full. Interestingly, this is the only scenario in which Quartus II reported that the number of M4K memory blocks available in the target FPGA is not sufficient to implement the desired functionality. However, this occurred at a point where other logic resources were almost fully utilized as well, so the design would not have fit even if sufficient on-chip memory was available. While the design in Figure 6.17e achieved higher cycle throughput than all previous configurations, the wall-clock throughput was actually lower than the wall-clock throughput of the design in Figure 6.17d, because of declining F_{max} (see Table A.14). The F_{max} of this system was lower because some multipliers were implemented in soft logic.

In this section we explored throughput improvements by gradually increasing the throughput of bottleneck kernels in an application. This was done for illustrative purposes only. In a real world scenario the system designer can skip these steps and simply specify the desired speedup for each kernel so that the overall throughput of the system matches the requirement for a given application. Our exploration has revealed that this process is very precise up to a point where the structure of collector and distributor nodes begins to skew the results. We have demonstrated that changing the structure of the collector and distributor nodes moves this point towards higher replication factors. We have also shown that further improvements may be possible even beyond this point, given a sufficiently large FPGA device. Using our current system, throughput improvement of more than 2.5 was achieved simply by inserting a few pragma statements into the code of a complex application.

We also implemented MPEG-2 spatial decoding in C and compiled it to run on the Nios II soft processor, and compiled it using C2H as well. Due to the algorithm complexity, these implementations did not achieve good performance compared to the FPGA Brook implementations. The Nios II soft processor achieved the wall clock throughput of only 9%, and C2H implementation achieved only 10% of the wall-clock throughput of the best FPGA Brook implementation. Detailed results of this and other experiments can be found in Appendix A.

6.4. Results Summary

In the preceding sections we presented detailed results of our experimental evaluation. In this section, we summarize our findings about scalability of replicated kernels in FPGA Brook applications, and also summarize the comparison of FPGA Brook to C2H and the Nios II soft processor.

6.4.1. Replication Scalability

Figure 6.18 shows how the wall-clock speedup scales with replication. Vertical axis in the graph is shown in logarithmic scale. It can be seen that most applications easily achieve 2X speedup, the only exception being the 32-bit Mandelbrot implementation, which consumes all hard multipliers even with this low replication factor. As the desired speedup is increased, fewer applications are able to meet the requested speedup. This occurs for one of four reasons:

1. The first and most frequently occurring case is that the replicated implementation of an application requires more multipliers than are available in the FPGA device used. As a result, soft logic is utilized to implement some of the multipliers, which results in poor F_{max}. Since the Cyclone II FPGA used in our experiments is a relatively small device, this problem can be avoided by using a larger device with more hard multipliers.
2. Some applications achieve maximum performance supported by our design flow, processing one data element per clock cycle. This limitation can be addressed by modifying our design flow to support globally asynchronous, locally synchronous (GALS) designs. In such a configuration, streamRead-type operators, which serve as sources of input data, can run at significantly higher clock rates and thus provide data to kernel replicas faster.
3. Applications utilizing array FIFO collectors and distributors experience slightly reduced scalability due to limitations of the current implementation of these modules. This issue can be resolved by modifying the modules to use deeper FIFOs, which is relatively straightforward, and by pipelining the multiplexers inside these modules for speedups of 16X and above, which is more complex.
4. Some algorithms do not expose sufficient data parallelism because of dependencies inherent in the algorithm. This problem can be addressed in one of three ways. First, the algorithm could be redesigned by finding an implementation that exposes more parallelism. Second, increasing the problem size may increase the available parallelism for some applications. For example, a 64-tap FIR filter scaled better than an 8-tap filter in our experiments. Finally, if the algorithm is found not to be suitable for a streaming implementation, an alternative programming paradigm should be explored.

Two applications (matrix-matrix and matrix-vector multiplication) achieve super-linear speedups up to 8X, which occurs because the replication algorithm does not take into account different runtimes for one-dimensional and two-dimensional reductions. Our experimental results could be used to guide modifications to the replication algorithm so that this effect is taken into account. As a result, the obtained speedup would better match the speedup requested by the designer through the speedup pragma.

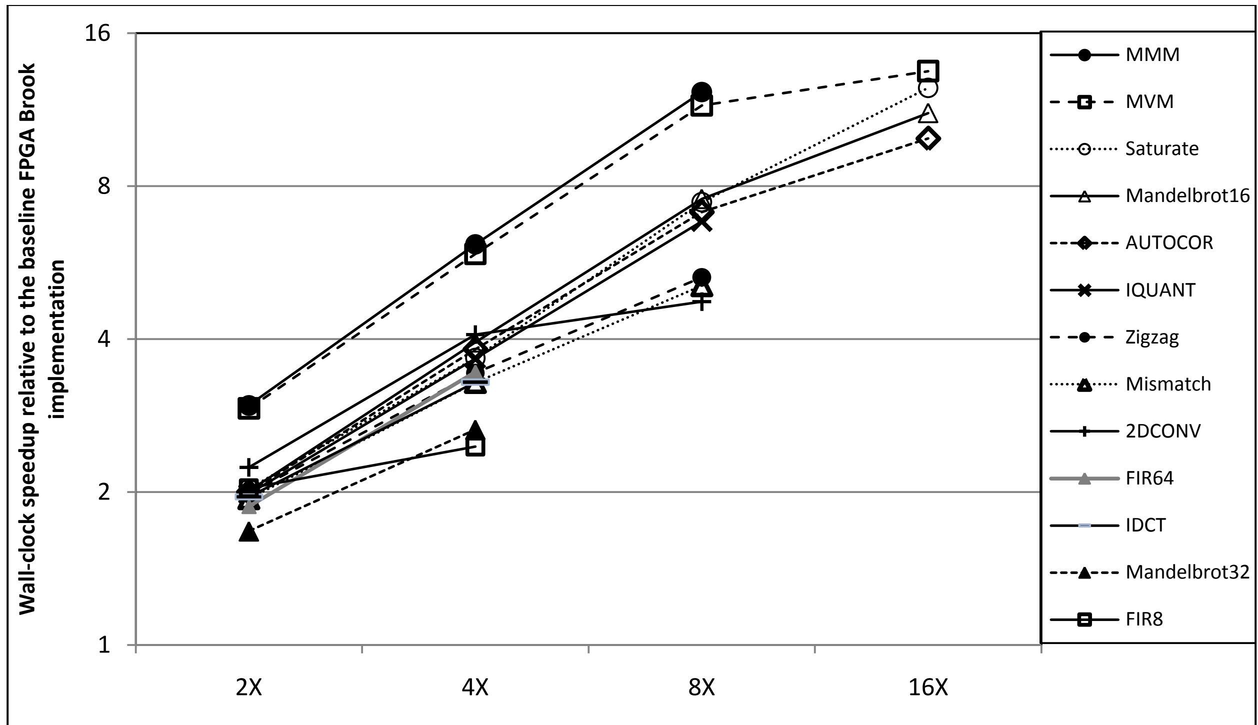


Figure 6.18 Wall-clock speedup of replicated FPGA Brook implementations

We found that the amount of on-chip memory available in the Cyclone II device was sufficient for all our experiments. The only design that required more M4K blocks than were available in the FPGA also required additional logic resources, so it could not be considered memory limited.

6.4.2. Comparison of FPGA Brook to C2H and Soft Processor

Performance and area comparison between C2H and FPGA Brook is summarized in Figure 6.19. The figure shows performance and area of C2H implementations relative to the best performing FPGA Brook implementation of the same benchmark. Two extreme trends can be seen in the figure; at one end there are several applications that C2H implements equally well as FPGA Brook, while at the other end there are several applications that achieve significantly better performance when implemented in FPGA Brook.

For simple applications, C2H achieves or exceeds the throughput of the best FPGA Brook implementation, while using fewer logic elements. This occurs because the code of these benchmarks is very simple, so C2H can implement them as fully parallel loops. This is partly because iterator values are used as input data values. We expect that the C2H performance would be lower in more realistic scenarios, although we acknowledge that for some simple algorithms, C2H may provide a better performance-area trade off. This is because FPGA Brook always decomposes applications into at least three parts: a streamRead-type operator that serves as a source of data to be processed, one or more

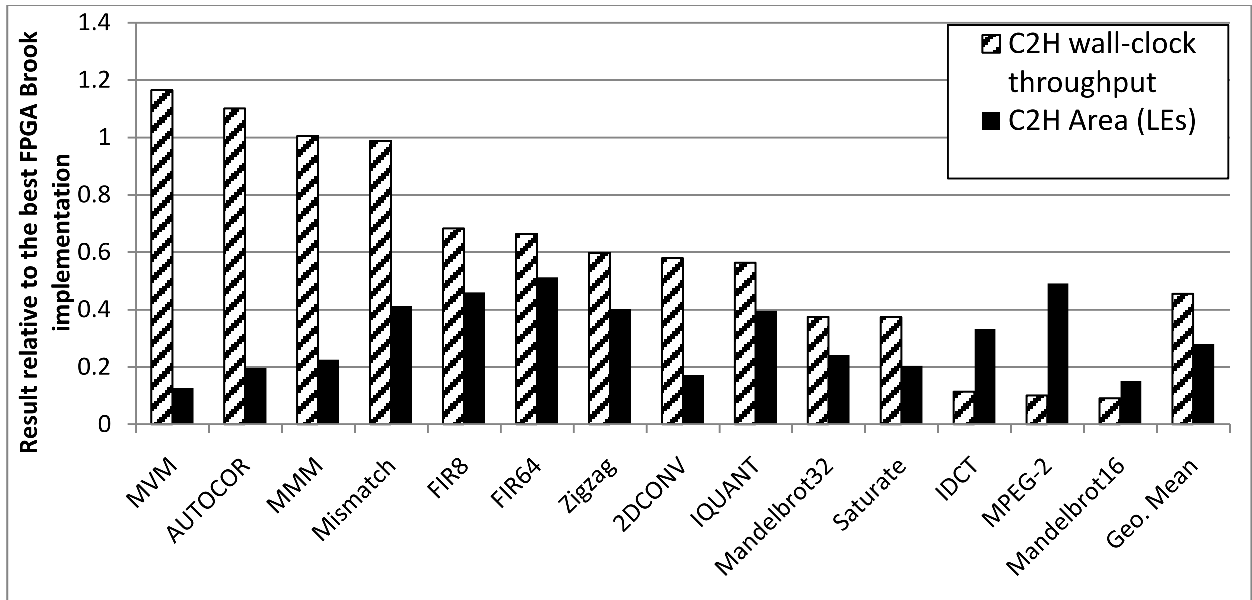


Figure 6.19 Results of C2H implementations relative to the best FPGA Brook implementation

processing kernels, and a streamWrite-type operator that serves as a data sink. For simple applications this decomposition may introduce unnecessary overhead that is better handled by a single hardware block, such as the one produced by C2H.

The other extreme in Figure 6.19 are large applications, such as IDCT, MPEG-2 spatial decoding and Mandelbrot set generator. For these applications, C2H provides only 10% of the performance achieved by

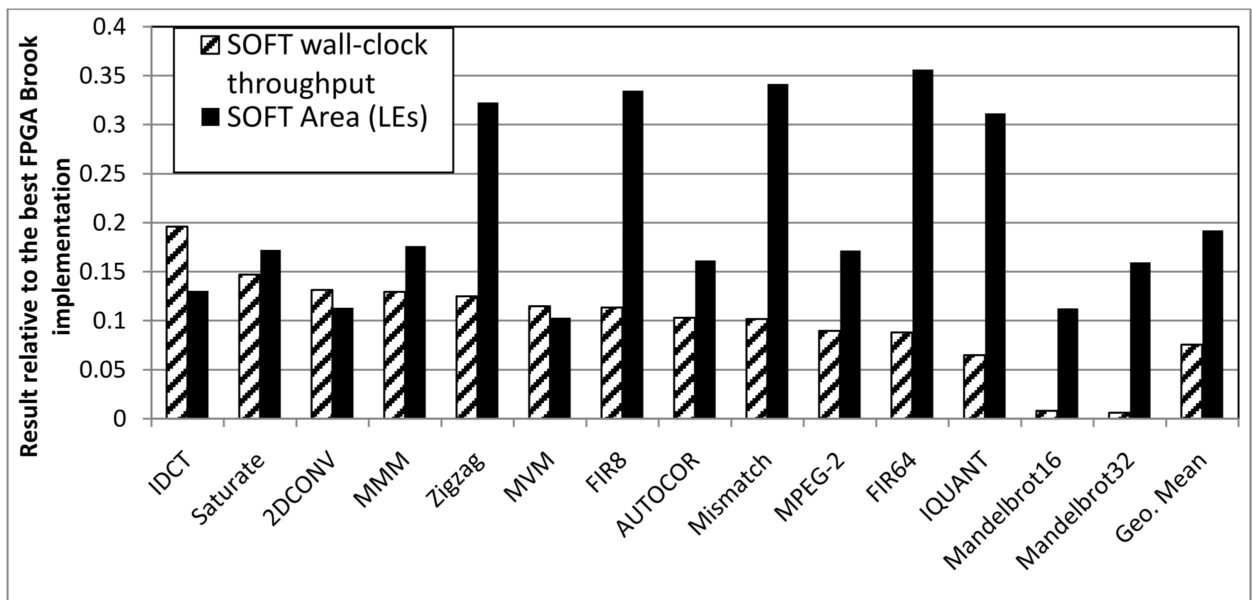


Figure 6.20 Results of soft processor implementations relative to the best FPGA Brook implementation

the best streaming implementation. The main goal of our work is simplifying the FPGA design process, which is particularly important for large applications, because developing such applications is complex. Therefore, it is encouraging to see that our design flow achieves good performance improvements over an alternative design methodology when it matters the most.

The applications that fall between the two extremes perform better when implemented in FPGA Brook, but not by a very large margin. This is mostly because scalability of these applications is limited for one of the reasons outlined earlier in this section, where we also outlined how to overcome these limitations. We note that given a larger FPGA device, FPGA Brook can achieve even higher performance, while C2H does not provide such an option. Therefore, the gap between C2H and FPGA Brook would increase if a larger device were used. C2H uses fewer LEs than FPGA Brook for all implementations. This was expected, because FPGA Brook trades-off area to achieve better performance. Furthermore, in this comparison, we always selected the best performing FPGA Brook implementation, even if this meant a large area increase for a small performance gain, because the assumption was that performance is the main objective. Even under these conditions, FPGA Brook achieves better throughput per area ratio than C2H for large applications, which can be concluded by observing the ratio of the relative wall-clock throughput and area in Figure 6.19. For example, the IDCT benchmark uses 33% of the area used by the best FPGA Brook implementation, but achieves only 11% of the throughput achieved by the best FPGA Brook implementation, thus providing three times lower throughput per area ratio. For completeness, we provide the geometric mean of all results in Figure 6.19, although this value is not very useful in comparing C2H to FPGA Brook, because it hides the trends discussed above.

Figure 6.20 shows the performance and area of C code running on the Nios II soft processor, relative to the best performing FPGA Brook implementation of the same benchmark. It is not surprising that Nios II performs poorly for all applications, with the relative wall-clock throughput being anywhere between 20% to less than 1% of the throughput of the best FPGA Brook implementation. The advantage of using a soft processor is that code is easy to develop, and the processor uses few logic resources. However, soft processors are not appropriate for applications where performance is important, and their throughput per area ratio is poor for most applications.

In this chapter we described our experimental methodology and presented results of our experiments. In the next chapter we discuss issues we encountered while developing our design flow and propose some improvements.

Chapter 7

Discussion

In this chapter we describe our experiences with the FPGA Brook design flow and discuss potential optimizations and future extensions, which would make the design flow easier to use and improve the performance and reduce the area of resulting circuits.

7.1. Ease of Use

Kernel code in FPGA Brook is often similar to the C code implementing equivalent functionality. This is because Brook and its variants are based on the C programming language. In C, large amounts of data are usually stored in arrays. If application data can be organized as a stream of independent elements, C code can often be ported to one or more Brook kernels with only small changes. Array references are usually transformed into stream references, which make array indices unnecessary, or simplify pointer manipulation for streams of arrays. Drake [56] put it succinctly by stating that in C "buffer management details dominate every line of code and obscure its functional purpose." The quote refers to a code segment of the MPEG-2 decoder implemented in C, and buffer management refers to pointer manipulation necessary to determine the address of the next block to be processed. The FPGA Brook compiler implements the functionality that advances stream elements without programmer intervention, thus simplifying buffer management and making the code easier to understand. Any manipulation of stream shape, size, or element ordering is isolated to stream operators, while kernels perform only computation. This results in a design that clearly distinguishes between computation and communication, and has an added benefit that FPGA Brook code closely resembles dataflow diagrams often used in early stages of application design.

To test our premise that FPGA Brook is easy to use, we offered it as a platform for a course project in a graduate-level course in our Department. The goal of the project was to develop an application of the student's choice in FPGA Brook and explore its performance. A junior graduate student with no previous exposure to streaming, and no previous experience with FPGA design, used our design flow to develop the Mandelbrot application [34]. The student also had no previous familiarity with the Mandelbrot algorithm. With a little guidance, the student found it easy to express the original C application in Brook and use our design flow to implement it on the DE2 board [61], achieving good performance, as outlined in the previous chapter. This was achieved in the course of a few weeks available for the course project.

This experience confirmed our belief that FPGA Brook is relatively easy to use, compared to traditional design methodologies for FPGAs.

7.2. Compilation Time

Our source-to-source compiler takes only seconds to compile each program in our benchmark suite, regardless of the replication factor. However, as the number of replicas grows, C2H compile times become excessively long. For example, C2H took more than 20 hours to compile a design with a total of 66 hardware accelerators. In this case, the C2H compiler was running on a 2GHz, Pentium IV computer with 1GB of memory. Quartus II compilation usually takes an additional hour or more on the same computer. While this was a very large design, and better computer configurations are common nowadays, the compilation time is still long. We believe that this is partly because the C2H compiler was not originally designed for a large number of accelerators and that the compiler could be further optimized to reduce the compilation time. Alternatively, a different behavioural synthesis compiler, with shorter compile times, could be integrated into our design flow. However, we note that a typical application does not contain such a large number of accelerators prior to replication. Since we expect replication to be applied only towards the end of the development cycle, compilation time for the most part of the application development would be closer to one hour.

Another way to avoid long compile times would be to develop and debug code in one of the Brook compilers targeting ordinary microprocessors [10] or shared memory multiprocessors [12]. While neither of these projects currently supports FPGA Brook, they demonstrate that such an approach is possible. Once the application is developed and its correct functioning has been verified, it can be implemented using our design flow, at which point a long compile time is less important because compilation is not expected to be performed many times. We believe that more research into fast compilation techniques for behavioural and hardware synthesis is needed, because of the ever-growing design complexity.

7.3. Debugging

One of the important features of any software or hardware design flow is debugging support. Contemporary behavioural synthesis tools provide very limited support for debugging of generated circuits. The underlying assumption is that development and debugging is performed in an ordinary software compiler, so the resulting circuit should be bug-free. However, behavioural compilers often include additional functionality that is normally not supported by software compilers, so debugging using the software compiler may not always be feasible. Similarly, the generated circuit usually has to interact with other components in the system, whose behaviour cannot be simulated in a software compiler. If there is a bug at the interface between the two components, it may be difficult to find. For example, while

code targeting the C2H compiler can often be tested in a software compiler, a system containing multiple accelerators that interact with other on-chip and off-chip devices can be debugged only using the SignalTap II on-chip logic analyzer [22]. However, using a logic analyzer requires an in-depth understanding of principles of digital logic design, and is generally more time consuming than debugging at the software level.

The FPGA Brook design flow does not currently support the kind of debugger interface commonly available in software development tools. However, we found that it is possible to take advantage of the structure of streaming programs and C2H features to perform simple debugging tasks. In FPGA Brook, design functionality is divided into multiple kernels interconnected by FIFO buffers. The FIFO buffer ports are implemented as Avalon slaves, accessible to the main processor, so they can be used as convenient access points for debugging. For example, consider the autocorrelation application, whose FPGA Brook code is shown in Listing 3.1. If this application produced an incorrect result, the programmer might want to inspect the data at the output of one of the kernels to determine the location of the problem. In our current system this can be done by manipulating the C2H code generated by our source-to-source compiler, parts of which are shown in Listing 4.1. For example, assume that the programmer wants to observe the output of the *mul* kernel. This can be achieved by reading the *out* port of the *mul_result_fifo* buffer. As mentioned in section 4.3, the base address of every Avalon slave port is defined in the *system.h* file, which is automatically generated by the C2H compiler. Therefore, the *main* function in the C2H code, which runs on the Nios II processor, can be modified to perform reads from this memory address using an appropriately initialized pointer. By repeatedly performing reads from this location, all elements of the stream produced by the *mul* kernel can be observed. There are two technicalities that have to be paid attention to. First, the *sum* kernel should not be called in the above scenario, to prevent it from reading the values from the *mul_result_fifo* FIFO buffer, since reads from FIFO buffers are destructive. If the *sum* kernel were active while debugging, some stream elements would be read by the Nios II processor, while others would be read by the *sum* kernel, because the processor would compete with the kernel for access to the FIFO. Second, if the Nios II processor includes the data cache, reads from the FIFO should be performed using a special macro which prevents caching of memory values, so that each read from the FIFO returns a new element [60]. Another debugging technique could be implemented using a similar procedure to inject data elements into a FIFO, by writing to its *in* port.

We used the above debugging procedure to debug several applications in our benchmark set. Usually, the bug was caused by incorrect C2H code generated by our source-to-source compiler during its development. Observing the outputs of different kernels helped us narrow down the problem and discover it sooner. Often it was sufficient to simply count the number of elements in the stream to determine that the kernel did not produce a sufficient number of elements. This usually indicated that loop boundaries in

the C2H code generated for the kernel were not set correctly, or the code emitting the output was not correctly placed within the loops. A particularly appealing aspect of this debugging technique is that only the software has to be recompiled to monitor the contents of different FIFOs. As long as the code for all the accelerators remains the same, the hardware does not have to be regenerated. This significantly reduces the debug turnaround time, because multiple parts of the system can be analyzed without having to regenerate the hardware. However, once the bug is found and the kernel code is modified to fix it, the design has to be fully recompiled, including hardware generation.

While the debugging procedure outlined above requires manually changing the C2H code and is not user friendly, we believe that the procedure could be used as a baseline for a more advanced debugging system. For example a GUI similar to the ones provided by software compilers could be developed, so that it hides the implementation details. If the system designer chose to observe data in a stream, the system's backend could make the necessary modifications to the code, so that the next time the application is run, the Nios II processor reads the values from the appropriate FIFO and presents them to the designer. To allow multiple streams to be observed simultaneously, the FIFO buffers could be modified to include a dedicated debugging port, which would allow the processor to observe stream values without interrupting the flow of data to the destination kernel. Further modifications to FIFO buffers could allow "single-stepping" through the program, by allowing only one stream element to pass at a time. A new stream element would be allowed to pass only when triggered by the controlling processor. We note that this is not equivalent to single-stepping through the lines of source code. Instead, it could be described as single-stepping through one kernel at a time. While this is one of the ways debugging could be done in our system, we believe that more research is needed in this area to develop effective methods for debugging streaming systems implemented in hardware.

7.4. Streaming Limitations and Heterogeneous System Development

While streaming languages are suitable for expressing many types of applications, we found that some application types may not be suitable for streaming implementations. Algorithms that are by their nature serial may be hard to express in streaming languages, because the main purpose of streaming languages is to allow programmers to effectively express parallelism. For example, our initial plan was to implement the complete MPEG-2 decoding algorithm in FPGA Brook. However, as we analyzed the different components involved in decoding, we found that some components would be challenging, if not impossible to express in FPGA Brook in a way that would provide performance benefits. As an example, consider temporal encoding in MPEG-2. Unlike spatial encoding, where each picture is encoded independently from all others, temporal encoding exploits similarities between pictures in the video to achieve high compression rates. Simply put, temporal encoding works by fully encoding some pictures,

and only encoding differences from the fully-encoded picture for all others. Fully-encoded pictures are known as *reference pictures*. For all other pictures, differences from the temporally nearest reference picture in the video sequence are encoded. In the decoder, the original values of all pixels in all pictures have to be restored. To achieve this, modules in the decoder handling non-reference pictures require access to the temporally nearest reference picture. Since the temporally nearest reference picture changes periodically, this creates a complex dependency that is challenging to handle in a streaming language. StreamIt introduced a technique called teleport messaging specifically to deal with this issue.

Teleport messaging is a method of passing relatively small amounts of data between filters in StreamIt programs asynchronously [56]. As discussed in section 2.5, StreamIt filters are similar to Brook kernels. Teleport messaging is used by the filter that detects a reference picture to send messages to all other filters that require the reference picture for their operation. Since filters operate independently, the timing of message delivery is critical for correct algorithm execution. The message with the new reference picture cannot be delivered instantaneously, because the picture currently being processed, and one or more pictures held in FIFO buffers, may require the old reference picture. To accommodate this, each teleport message contains timing information, which directs the message recipients to process the message only after a certain number of input stream elements have been consumed [56]. Brook and GPU Brook do not currently support such functionality, and consequently, FPGA Brook does not support it either. This makes it difficult to express temporal decoding in FPGA Brook. Potential benefits and drawbacks of adding teleport messaging, or another similar technique to FPGA Brook should be explored in future versions of FPGA Brook.

Another part of MPEG-2 that presents a challenge for streaming implementations is variable-length coding (VLC). Values in MPEG-2 encoded files are stored as variable-length codes, based on the Huffman encoding algorithm [71]. This means that different values are encoded using code words of different lengths, so that commonly occurring values use shorter code words. This presents a challenge for streaming decoder implementations because the incoming data cannot be easily divided into fixed size blocks and stored in a stream. Drake [56] has considered this problem in the context of the StreamIt language and concluded that this kind of processing is simply not suitable for expression in streaming languages and should be implemented in an alternative language [56]. We agree with this sentiment, and note that although some new way of expressing these applications in streaming languages may be found, we expect that there would always be some applications that would be difficult to express as streaming.

To deal with applications that cannot be effectively expressed in a streaming language, Drake [56] proposes introducing "clean interfaces" that would allow code written in a traditional language, such as C, to be combined with StreamIt code. A similar sentiment is expressed by Neuendorffer and Vissers [6], who suggest that "alternative models of computation" should be used "in concert with dataflow". While FPGA Brook does not currently support a dedicated interface for such heterogeneous development, it

does support combining ordinary C code with FPGA Brook code, due to its reliance on Nios II and C2H compilers. This is because any code outside of kernels and streamRead-type and streamWrite-type operators is simply passed down to the Nios II compiler. As a result, programmers can include ordinary C functions in their code, and they will be executed by the Nios II soft processor. Furthermore, due to our reliance on the C2H compiler, some of these functions could be implemented as hardware accelerators, regardless of the fact that they are not kernels. While FPGA Brook does not currently support this directly, a skilled programmer could modify the system generation scripts to achieve such functionality. Furthermore, it would be relatively easy to add a new type of pragma statement to FPGA Brook, which would allow the programmer to mark certain functions for implementation as C2H accelerators. Such a pragma statement would be comparable to the *asm* directive in the C programming language, which allows embedding of assembly instructions into C source code [76]. Most C programmers never use this directive, because doing so requires detailed understanding of the underlying computer architecture and reduces code portability. However, it can play an important role in implementing certain applications. A pragma statement that would allow embedding of C2H accelerators into FPGA Brook programs would be similar, in that it could be disregarded by an average FPGA Brook developer and reserved for advanced users. In theory, this principle could even be extended to support the inclusion of Verilog or VHDL code in the FPGA Brook code, but interfacing such modules would be more challenging than interfacing C2H accelerators. This kind of heterogeneous development environment would allow easier integration of design components from different development teams working on a project, and thus warrants further investigation.

7.5. Comparing FPGA Brook and C2H

While the FPGA Brook design flow utilizes the C2H compiler, our results show that FPGA Brook implementations outperform the C2H results in most cases. This indicates that FPGA Brook adds significant value over C2H. As discussed earlier, this is because our source-to-source compiler exploits both task and data parallelism exposed by the structure of FPGA Brook streaming programs. If the application contains multiple kernels, task parallelism is exploited by implementing each kernel as a separate hardware unit and interconnecting kernels through FIFOs, so that they can operate in a pipelined fashion. At the kernel level, data parallelism is exploited by creating multiple instances of a kernel that perform processing in parallel. The same cannot be achieved by the C2H compiler because C code is not well suited to express parallelism. While some behavioural synthesis tools support special pragmas that allow unrolling of loops without cross-iteration dependencies, the burden is on the programmer to ensure that there are no such dependencies, which involves analysis at the code level. Furthermore, unrolling a loop that contains many memory accesses would not improve performance of FPGA implementations

because memory accesses would become a bottleneck. Our source-to-source compiler uses collector and distributor FIFOs to deal with this problem. This is possible because FPGA Brook raises the abstraction level by forcing the programmer to think about data and task parallelism at the algorithm level, so that the compiler can automatically exploit the parallelism in a way that best takes advantage of the underlying platform, which in our case is FPGA hardware.

Since the FPGA Brook design flow uses the C2H compiler in its back-end, it is obvious that skilled programmers could structure their C2H code in the same way that our source-to-source compiler does, and achieve identical results. While this is true, it would require a significant manual effort that is more suitable for an automated compiler to perform. Operations such as strength reduction of modulo and division operations, SOPC system generation, kernel replication and design and implementation of custom collector and distributor hardware modules in support of replication would take excessively long time to produce manually. The manual implementation would also likely contain bugs, so additional time would have to be dedicated to debugging. Furthermore, generating the hardware modules requires hardware design expertise, whereas the FPGA Brook design flow can be used by software programmers without hardware expertise. Finally, if the application requirements change so that different throughput is required, a significant manual effort would be necessary to modify the hardware modules and the SOPC system to meet the changing requirements. FPGA Brook performs replication automatically, and only requires changing one or more speedup pragma statements in the code to support such a change. We conclude that FPGA Brook helps programmer productivity by allowing the programmer to focus on the algorithm design, particularly expressing the parallelism in the algorithm, while the compiler can take advantage of the parallelism automatically.

7.6. Comparing FPGA Brook and Manual IDCT Implementation

While our results show that FPGA Brook designs outperform both Nios II and C2H results, it is interesting to also compare our results to designs developed manually by a skilled HDL designer. We use the IDCT benchmark as a case study, because it is readily available as an IP core. We use the IDCT core provided by the CAST, Inc. company for comparison [77]. Their core achieves throughput of one sample per clock cycle running at 150 MHz when implemented in the EP2C5-C6 Cyclone II device, which is a smaller device in the same FPGA family, and of the same speed grade that we used. The implementation uses 1,379 LEs, 16 hard multipliers and one M4K memory block and achieves throughput of 150 Msamples/s. The best FPGA Brook IDCT implementation achieves throughput of 27 Msamples/s, which is 5.6 times lower than the CAST's implementation. At the same time, the best FPGA Brook implementation uses 18.6 times more LEs, 4.4 times more hard multipliers and 56 times more M4K memory blocks.

Assuming that a larger FPGA were available for the FPGA Brook implementation, we expect that IDCT could achieve throughput similar to other benchmarks that are part of the MPEG-2 spatial decoding. To determine the throughput that could be achieved in such a case, we extrapolate the results we obtained for IDCT and other benchmarks. As shown in Table A.13, the running time of the FPGA Brook implementations of the IDCT benchmark decreases linearly with replication. A similar trend can be observed for other benchmarks that are part of the MPEG-2 spatial decoding, as shown in Tables A.9 through A.12. The results indicate that this running time scalability stops at the running time of 0.016 seconds, and that no further improvement is possible. Extrapolating the IDCT results shown in Table A.13, we conclude that the IDCT benchmark would achieve the running time of 0.016 seconds with the replication factor of 16. Assuming that a sufficient number of hard multipliers were available, the F_{max} of this system would be limited by the multiplexers in the collector and distributor FIFOs, as discussed in section 6.3.7. Because of this, we can assume that the F_{max} of the IDCT implementation with the replication factor of 16 would be the same as the F_{max} of the saturation application with the same replication factor, which is 100MHz. From this, we conclude that implementing the IDCT benchmark in a larger FPGA device could achieve throughput of 80 Msamples/s, which is 1.9 times lower than the CAST's IP core. Extrapolating the area results, we find that such a design would consume approximately 80,000 LEs, 450 hard multipliers and approximately one hundred M4K memory blocks. This means that such an implementation would use 58 times more LEs, 28 times more hard multipliers and 100 times more M4K memory blocks. These results are achievable using our current infrastructure. As discussed in section 6.4.1, further throughput improvements are possible by modifying some parts of our design flow. In such a case we expect that our design flow could achieve throughput that is close to, or exceeds the throughput of the CAST's IP core, although at a significantly higher area cost.

The difference in the area utilization of the FPGA Brook implementation of the IDCT benchmark and its manual implementation is large. The main reason for such a large difference is the ability of the HDL designer to specify the exact data width of all registers in the design, while FPGA Brook limits available data widths to several predefined data types, similar to the C programming language. The MPEG standard uses 12-bit values at the input and 9-bit values at the output of the IDCT [78]. The FPGA Brook implementation of IDCT uses the *short* (16-bit) data type to represent both input and output values, because that is the smallest data type that can fit the required number of bits. The FPGA Brook implementation uses the *integer* (32-bit) data type to represent the intermediate values. The CAST's IP core uses 11-bit values at the input and 8-bit values at the output of the IDCT to "satisfy the accuracy criteria of the JPEG standard" [77,79]. Since the HDL designer can specify the exact number of bits for inputs, outputs and intermediate results, this results in significant area savings over the FPGA Brook implementation. In addition, FPGA Brook uses replication to improve throughput, so the area difference between the manual implementation and the FPGA Brook implementation is multiplied by the replication

factor. We conclude that the FPGA Brook design flow could be improved by modifying the language to allow data types with arbitrary data widths. While the inability to specify the exact data width in FPGA Brook accounts for a large portion of the area difference, other reasons for the area difference include overheads associated with collector and distributor nodes and lack of area optimizations in the FPGA Brook design flow. We also note that area results for FPGA Brook are slightly lower if the *char* (8-bit) data type is used for the output data, which makes it equivalent to the CAST's core. However, the difference is small (less than 5%) because all other computations still have to be performed using the *short* and *int* data types.

In this section we compared the FPGA Brook implementation of the IDCT benchmark with a commercially available, hand-optimized version of the same benchmark. Our results indicate that there are significant opportunities for improving our results, both in terms of performance and area requirements. We discuss some of these optimization opportunities in the following section.

7.7. Design Flow Limitations and Optimization Opportunities

In section 6.3.2 we demonstrated that coding style can have a significant impact on performance of C2H hardware accelerators. Since FPGA Brook relies on C2H for hardware generation, performance of FPGA Brook code depends on the coding style as well, as demonstrated by the performance results of the saturation application, presented in section 6.3.7. In the case of the saturation application, poor performance was caused by repeated reads of the same value from the array FIFO memory, where one read would have sufficed. While replication was successful in making up for the lost performance, the area cost was significant; the same outcome could have been achieved by a simple compiler optimization. There are many other analyses and optimizations that contemporary optimizing compilers routinely perform, which are currently not performed by either our source-to-source compiler or the C2H compiler. This represents a major opportunity for performance improvement of FPGA Brook programs, and would also reduce the dependence of performance on the coding style. We also note that the FPGA Brook language by its construction alleviates the coding style problem somewhat, because it encourages the programmer to break down the application into independent kernels, thus reducing the chance of performance being limited by dependencies in the code. Both C2H and FPGA Brook limit the set of features of the C language that are allowed inside accelerated functions and kernels, respectively. However, there is a significant overlap between the limitations of the two languages, which makes mapping between them easier, and avoids many problems that would occur if C2H had many more limitations than Brook. We conclude that, while coding style is an issue in the current version of FPGA Brook, many techniques exist that could be applied to alleviate the problem.

The code generated by our source-to-source compiler is often general enough to handle all possible cases, and could be further optimized by generating simpler code for special cases. As an example, the code generated for kernels processing 2-D streams maintains both row and column iterators, as described in section 4.2.2. While this is necessary when the kernel code uses the *indexof* operator, it is not needed for other kernels. Therefore, the two loops could be replaced by a single loop with appropriately adjusted loop limit, which would save area by eliminating one register and potentially improve performance. Other such opportunities should be explored and optimized in future versions of our compiler.

Stream operators present a potential performance bottleneck in Brook streaming programs. While FPGA Brook currently implements the functionality of stream operators within streamRead-type and streamWrite-type operators, stream operators should be implemented in future versions of FPGA Brook, as discussed in section 3.2. Their potential impact on performance is relevant to our work. For example, consider the stencil stream operator, which selects a stream element and several of its neighbours from the input. This operator can be applied to a two-dimensional stream, as in the example of two-dimensional convolution, in which case the operator selects the current element of the stream and its neighbours in both dimensions. Considering that stream elements are passed between kernels in the row-major order, this means that the operator would have to wait for several rows of data before it can produce the output for the current element. This implies that any downstream kernels also have to wait for data and may become idle. Furthermore, the operator has to buffer several rows of data while waiting for the data elements it needs to produce the output for the current element. This is because each element of the stream arrives to the operator only once. If the number of elements in a row (i.e. number of columns) is large, a large memory buffer would be necessary to support such functionality. However, if the number of columns in the stream is much higher than the number of its rows, a simple compiler optimization could change how streams are passed between the kernels by passing them in the column-major order. This would reduce the amount of time the stream operator has to wait, and would also reduce the size of the memory buffer in the accelerator implementing the stream operator. To support such an optimization the programmer would have to provide two implementations of any custom streamRead-type and streamWrite-type operators; one using row-major, and one using the column-major element ordering. This is because these operators currently support only row-major ordering. A special language construct would have to be introduced to unambiguously identify the two implementations. In cases when a stream operator provides inadequate performance despite all optimizations, an alternative to streaming may have to be considered, as discussed in section 7.4. We also note that the *peek* operator in StreamIt presents a similar challenge, because it allows filters to access FIFO elements with arbitrary distance from the FIFO head. If the *peek* operator is used to access a FIFO element with a large distance from the FIFO head, it would have the same negative impact on performance as the stencil stream operator in Brook. We

conclude that stream operators may present a performance bottleneck in Brook programs, so they should be optimized where possible and used with care.

Our source-to-source compiler is currently operating independently of the C2H compiler, and the information passed from our compiler to C2H is limited to pragma statements specifying connections between kernels and FIFO buffers. If our compiler were more closely integrated with the C2H compiler, information extracted from the FPGA Brook source code could be used by the C2H compiler to better optimize the hardware accelerators. For example, accelerators are currently connected to FIFOs through Avalon master-slave port pairs, which could be simplified if the two tools were combined into one. As another example, the *main* function in the C2H code is executed on the Nios II soft processor. Since the *main* function in many applications only starts the accelerators and does little else, the Nios II processor could be replaced by a simple state machine implementing this functionality, thus reducing the total area. Generally, we expect that closer integration of the two compilers would open opportunities for additional optimizations, which would result in further area savings and performance improvements.

Another option that should be considered when discussing improvements to our design flow is the possibility of using a different behavioural synthesis compiler. While we obtained compelling results using C2H, other behavioural compilers that have emerged offer some attractive options. For example, C2H does not currently support floating-point arithmetic because FPGA designs did not typically involve floating-point computations. However, rapid increases in logic densities of these devices has made it viable to implement algorithms relying on floating-point arithmetic in FPGAs [80]. Therefore, using a behavioural synthesis compiler that supports floating-point arithmetic in our design flow would increase the range of applications that FPGA Brook could support.

While developing the FPGA Brook design flow, our main goal was improving the performance at the expense of increased area. While this is often appropriate, there are applications where a balance between area and performance is more important. While our current design flow provides some freedom to balance area and performance through selection of kernel replication factors, additional flexibility could be provided by the behavioural synthesis tool used at the back-end of the design flow. The C2H compiler translates C statements directly into hardware by instantiating a hardware unit for each operator in the code, usually without sharing logic among statements, as discussed in section 2.4. While some logic sharing is possible by separating common functionality into a subfunction, this would require our source-to-source compiler to generate different code specifically for this purpose, which our compiler does not currently perform. Since other behavioural synthesis compilers implement logic sharing, using such a compiler as a back-end compiler for our design flow would provide more flexibility in trading-off area and performance.

Another advantage of using a different behavioural synthesis compiler is support for loop unrolling, which C2H does not support. For example, Impulse C supports a pragma statement that instructs the

compiler to unroll a given loop in the code [34]. Before using this pragma, the programmer has to ensure that there are no dependencies between the loop iterations, so that the unrolled loop produces the same result as the original loop. Loop unrolling could be used as an alternative to kernel replication, at least for simple kernels, as long as our source-to-source compiler could determine that no dependencies between loop iterations exist. Loop unrolling would not be applicable to kernels with complex dataflow and most reduction operations, in which case replication could still be used. In summary, we believe that there are potential benefits of using other behavioural synthesis tools as our back-end, instead of the C2H compiler we currently use, and that more research is needed in this area.

In this section we discussed optimizations that have the potential to further improve our results. Our main conclusion is that the results we obtained, though compelling, are still far from the results that could be obtained if additional compiler and other optimizations were combined in a production-grade design tool flow. We believe that more research in the area of stream computing on FPGAs is needed to uncover its full potential.

Chapter 8

Conclusions and Future Work

Complexity of digital systems being implemented in FPGA devices has been growing rapidly over the past decade. For this growth to be sustainable, sophisticated design tools are needed, which are capable of efficiently implementing high-level application descriptions in FPGA logic with minimal designer intervention. In this thesis we proposed using the Brook streaming language as a high-level language suitable for designing digital systems targeting FPGAs. A streaming language was chosen because many applications targeting FPGAs perform computations that are well suited for streaming implementations. Brook was chosen because it is based on the C programming language, so it is easy to adopt by many programmers who are already familiar with C. We based our work on the existing Brook and GPU Brook languages, and developed FPGA Brook, which is a variant of these languages suited for designs targeting FPGAs. In FPGA Brook data is organized into streams, which are independent collections of data, and computation is organized into kernels, which are functions operating on streams.

We developed a fully automated design flow that implements an application described in FPGA Brook in FPGA logic, using a combination of our source-to-source compiler and commercial behavioural and hardware synthesis tools. We demonstrated that our design flow is easy to use, as evidenced by the relatively smooth transition from C code to FPGA Brook code, and based on the experiences of another graduate student who used our design flow successfully. Furthermore, we found that FPGA Brook code is better structured and easier to understand than the corresponding C code. We developed a set of benchmark applications in FPGA Brook and evaluated their performance when implemented using our design flow. We found that our design flow effectively takes advantage of data and task parallelism exposed by expressing the application in FPGA Brook. For simple applications with only one kernel, data parallelism is exploited by replicating the kernel so that multiple hardware units perform processing in parallel. For more complex applications with multiple kernels, task parallelism is exploited by interconnecting the kernels using FIFO buffers, so that computation is performed in a pipelined fashion. Data parallelism can still be exploited by replicating individual kernels to achieve desired throughput. Our experiments show that the application throughput is limited by the throughput of the kernel with the lowest throughput, which indicates that pipelining exploits task parallelism effectively.

We found that the FPGA Brook design flow produces circuits that often outperform both the Nios II soft processor and the circuits produced by the C2H compiler when streaming is not used. FPGA Brook implementations outperform the soft processor up to 100 times. While simple applications can be effectively implemented using the C2H compiler, complex applications achieve up to 10 times higher

throughput when implemented using our design flow. We also found that our results could be further improved by using larger FPGAs, and also through various compiler and other optimization techniques. We conclude that the FPGA Brook design flow developed in this thesis presents an attractive design platform for targeting FPGAs from high-level descriptions.

8.1. Contributions

This thesis work offers the following contributions:

- Definition of the FPGA Brook streaming language, suitable for implementation of streaming applications in FPGA logic.
- One of the first implementations of a design flow that can automatically implement an application described in a high-level streaming language in an FPGA. The design flow is fully automated, easy to use, and achieves significantly better data throughput than alternative approaches.
- The source-to-source compiler developed as a part of our design flow, which converts FPGA Brook programs into C code suitable for compilation by a behavioural synthesis compiler. Altera's C2H compiler was used in our design flow as an example. The source-to-source compiler uses a kernel replication technique to exploit data parallelism in streaming applications by creating many parallel instances of the hardware implementing the kernel.
- A novel technique for parallel implementation of complex reduction operations using hardware units interconnected by FIFOs. One or more reduction trees are built to implement reductions, as needed to achieve throughput specified by the programmer in the FPGA Brook source code. Algorithms are described that minimize the number of reduction trees, number of nodes in each tree and latency within each tree.
- Implementation of hardware modules necessary to interconnect the kernel replicas into a system that implements the functionality described by the FPGA Brook source code.
- Demonstration of how strength reduction of modulo operations can be performed in the presence of replication. While strength reduction of modulo operations has been explored in earlier works, the handling of these operators in presence of replication is novel.
- Implementation of twelve benchmark applications in FPGA Brook and their performance evaluation and analysis, including exploration and analysis of their throughput improvement due to replication.
- Discussion of a number of options for further improvement of our results, spanning the design of streaming languages, optimizations applicable to our source-to-source compiler, and possible integration with other behavioural synthesis tools.

The source code for our source-to-source compiler, along with the FPGA Brook benchmark set, will be made publicly available at the author's web-site in the near future.

8.2. Future Work

While our existing design flow achieves good performance compared to other design flows targeting FPGAs, and it is easy to use, there is room for improvement. We suggest the following areas for improvement of our design flow and future research. First, support for stream operators defined in the original Brook specification should be added to FPGA Brook. More research is needed on the kinds of operators that are required to effectively express applications targeting FPGAs. Second, our current design flow limits the throughput of some applications to processing one data element per clock cycle; this limitation could be overcome by introducing support for globally asynchronous locally synchronous (GALS) designs, so that each kernel can operate at a different clock frequency. Third, a number of compiler optimizations common in contemporary optimizing compilers could be implemented in our source-to-source compiler to further improve the results. Finally, our design flow could be modified by replacing the C2H compiler by a different behavioural synthesis compiler.

Currently, our design flow allows programmers to use the speedup pragma to specify that a kernel's throughput should be increased by a certain factor. There are two ways this process could be modified to make it more user friendly. First, the compiler could use performance estimation techniques to estimate the absolute performance of kernels, so that the desired throughput could be specified as an absolute value (e.g. KB/s). The performance estimates could then be used in combination with the replication techniques we described in this thesis to achieve the desired throughput. Alternatively, the compiler could use area estimation techniques to implement a design flow that maximizes performance given an area constraint. Each of these approaches is desirable for certain applications.

Another important addition to our design flow would be implementation of a user-friendly debugging interface, which would in its simplest form allow the system designer to monitor the state of all FIFOs in the system at run-time. Other, more advanced debugging features should also be considered. The FPGA Brook design flow could also be extended by adding support for complex applications that cannot be fully described using streaming. In such a case, special directives could be used to denote code that does not abide by the streaming design paradigm, which could then be processed by an alternative compiler. Finally, more research is needed to determine the optimal way for implementing different streaming applications in FPGAs. It is possible that some applications may provide better performance-area trade-off if implemented using multiple generic streaming processors, similar to the Merrimac supercomputer [9], instead of generating custom hardware accelerators for all kernels. The architecture of a streaming processor suitable for FPGA implementation would likely be the focus of such a research effort.

It is our opinion that streaming systems and their implementation in FPGAs are an exciting research area, offering significant potential for improving the FPGA design methodology. We expect that this research area will experience growth in the foreseeable future.

Appendix A

Results of Experimental Evaluation

Results of our experimental evaluation are shown in Tables A.1 through A.14. The baseline FPGA Brook implementation is labelled *FBROOK* in all the tables. Replicated FPGA Brook implementations are labelled by the desired speedup specified by the speedup pragma in the code (e.g. *8X*). Finally, C2H implementations and code run on the Nios II soft processor are labelled as C2H and SOFT, respectively. For each implementation, the following results are reported:

- Running time (s) – Application execution time when the circuit runs at 50 MHz.
- F_{max} (MHz) – Maximum operating frequency of the circuit.
- Wall-clock throughput (KB/s) – Throughput achievable if the circuit runs at the maximum operating frequency. It is computed by dividing the amount of application input data (in bytes) by the application execution time pro-rated to account for the maximum operating frequency (F_{max}).
- Area (LEs) – Number of logic elements utilized by the circuit.
- Multipliers – Number of 9x9 hard multipliers utilized by the circuit.
- M4K blocks – Number of M4K memory blocks utilized by the circuit.
- Cycle throughput improvement – Cycle throughput relative to the baseline FPGA Brook implementation.
- Wall-clock speedup – Wall-clock throughput relative to the baseline FPGA Brook implementation.
- Relative LEs – Number of logic elements relative to the baseline FPGA Brook implementation.
- Relative multipliers – Number of 9x9 hard multipliers relative to the baseline FPGA Brook implementation.

	Running Time(s)	Fmax (MHz)	Wall-Clock Throughput (KB/s)	Area (LEs)	Multipliers	M4K Blocks	Cycle Throughput Improvement	Wall-Clock Speedup	Relative LEs	Relative Multipliers
FBROOK	0.176	136	6,019	5,025	10	28	1.00	1.00	1.00	1.00
2X	0.088	136	12,046	6,442	16	30	2.00	2.00	1.28	1.60
4X	0.044	130	23,000	8,085	28	32	4.00	3.82	1.61	2.80
8X	0.024	131	42,804	11,678	52	36	7.33	7.11	2.32	5.20
16X	0.016	122	59,743	20,665	70	44	11.00	9.93	4.11	7.00
C2H	0.016	135	65,744	4,062	10	24	11.00	10.92	0.81	1.00
SOFT	0.177	139	6,153	3,333	4	24	0.99	1.02	0.66	0.40

Table A.1 Performance and area results for the autocorrelation application

	Running Time(s)	Fmax (MHz)	Wall-Clock Throughput (KB/s)	Area (LEs)	Multipliers	M4K Blocks	Cycle Throughput Improvement	Wall-Clock Speedup	Relative LEs	Relative Multipliers
FBROOK	0.128	125	7,636	5,679	8	27	1.00	1.00	1.00	1.00
2X	0.064	127	15,548	7,071	12	29	2.00	2.04	1.25	1.50
4X	0.052	125	18,761	9,957	20	33	2.46	2.46	1.75	2.50
C2H	0.214	118	4,307	4,391	10	25	0.60	0.56	0.77	1.25
C2H_ALT	0.078	128	12,802	4,582	8	24	1.64	1.68	0.81	1.00
SOFT	0.512	139	2,127	3,333	4	24	0.25	0.28	0.59	0.50

Table A.2 Performance and area results for the 8-tap FIR-filter application

	Running Time(s)	Fmax (MHz)	Wall-Clock Throughput (KB/s)	Area (LEs)	Multipliers	M4K Blocks	Cycle Throughput Improvement	Wall-Clock Speedup	Relative LEs	Relative Multipliers
FBROOK	1.024	128	979	5,730	10	29	1.00	1.00	1.00	1.00
2X	0.512	120	1,833	7,271	16	32	2.00	1.87	1.27	1.60
4X	0.277	120	3,375	9,353	28	36	3.70	3.45	1.63	2.80
C2H	2.653	119	351	4,379	10	25	0.39	0.36	0.76	1.00
C2H_ALT	0.414	119	2,239	4,793	10	26	2.47	2.29	0.84	1.00
SOFT	3.669	139	297	3,333	4	24	0.28	0.30	0.58	0.40

Table A.3 Performance and area results for the 64-tap FIR-filter application

	Running Time(s)	Fmax (MHz)	Wall-Clock Throughput (KB/s)	Area (LEs)	Multipliers	M4K Blocks	Cycle Throughput Improvement	Wall-Clock Speedup	Relative LEs	Relative Multipliers
FBROOK	0.885	123	3,343	6,334	18	32	1.00	1.00	1.00	1.00
2X	0.388	121	7,477	8,767	24	34	2.28	2.24	1.38	1.33
4X	0.213	121	13,649	17,532	36	39	4.15	4.08	2.77	2.00
8X	0.185	122	15,856	29,437	60	48	4.78	4.74	4.65	3.33
C2H	0.326	125	9,180	5,050	19	25	2.71	2.75	0.80	1.06
SOFT	1.604	139	2,086	3,333	4	24	0.55	0.62	0.53	0.22

Table A.4 Performance and area results for the two-dimensional convolution application

	Running Time(s)	Fmax (MHz)	Wall-Clock Throughput (KB/s)	Area (LEs)	Multipliers	M4K Blocks	Cycle Throughput Improvement	Wall-Clock Speedup	Relative LEs	Relative Multipliers
FBROOK	0.32	130	31,739	5,736	14	28	1.00	1.00	1.00	1.00
2X	0.11	131	92,754	7,766	20	30	2.91	2.92	1.35	1.43
4X	0.055	131	186,752	10,250	32	32	5.82	5.88	1.79	2.29
8X	0.028	131	365,969	15,448	56	36	11.43	11.53	2.69	4.00
16X	0.021	115	427,819	32,312	70	29	15.24	13.48	5.63	5.00
C2H	0.02	127	498,053	4,067	14	24	16.00	15.69	0.71	1.00
SOFT	0.222	139	49,073	3,333	4	24	1.44	1.55	0.58	0.29

Table A.5 Performance and area results for the matrix-vector multiplication application

	Running Time(s)	Fmax (MHz)	Wall-Clock Throughput (KB/s)	Area (LEs)	Multipliers	M4K Blocks	Cycle Throughput Improvement	Wall-Clock Speedup	Relative LEs	Relative Multipliers
FBROOK	0.023	119	1,212	6,162	21	29	1.00	1.00	1.00	1.00
2X	0.008	123	3,590	8,102	27	30	2.88	2.96	1.31	1.29
4X	0.004	127	7,447	11,164	39	29	5.75	6.14	1.81	1.86
8X	0.002	127	14,864	18,909	63	33	11.50	12.26	3.07	3.00
C2H	0.002	127	14,938	4,273	21	24	11.50	12.32	0.69	1.00
SOFT	0.017	139	1,922	3,333	4	24	1.35	1.59	0.54	0.19

Table A.6 Performance and area results for the matrix-matrix multiplication application

	Running Time(s)	Fmax (MHz)	Wall-Clock Throughput (KB/s)	Area (LEs)	Multipliers	M4K Blocks	Cycle Throughput Improvement	Wall-Clock Speedup	Relative LEs	Relative Multipliers
FBROOK	35.18	130	36	5,163	16	27	1.00	1.00	1.00	1.00
2X	17.842	133	73	6,513	24	27	1.97	2.01	1.26	1.50
4X	9.09	133	142	8,402	40	27	3.87	3.94	1.63	2.50
8X	4.654	130	273	12,233	70	27	7.56	7.55	2.37	4.38
16X	2.396	99	402	29,643	70	27	14.68	11.15	5.74	4.38
C2H	35.14	132	37	4,472	20	24	1.00	1.01	0.87	1.25
SOFT	423.146	139	3.22	3,333	4	24	0.08	0.09	0.65	0.25

Table A.7 Performance and area results for the 16-bit Mandelbrot application

	Running Time(s)	Fmax (MHz)	Wall-Clock Throughput (KB/s)	Area (LEs)	Multipliers	M4K Blocks	Cycle Throughput Improvement	Wall-Clock Speedup	Relative LEs	Relative Multipliers
FBROOK	34.486	107	60	5,995	52	27	1.00	1.00	1.00	1.00
2X	17.385	90	101	9,206	70	27	1.98	1.67	1.54	1.35
4X	8.795	72	160	20,905	70	27	3.92	2.65	3.49	1.35
C2H	34.448	106	60	5,063	46	24	1.00	0.99	0.84	0.88
SOFT	2817.994	139	0.97	3,333	4	24	0.01	0.02	0.56	0.08

Table A.8 Performance and area results for the 32-bit Mandelbrot application

	Running Time(s)	Fmax (MHz)	Wall-Clock Throughput (KB/s)	Area (LEs)	Multipliers	M4K Blocks	Cycle Throughput Improvement	Wall-Clock Speedup	Relative LEs	Relative Multipliers
FBROOK	0.091	126	34,657	4,785	4	33	1.00	1.00	1.00	1.00
2X	0.046	127	68,786	5,338	4	34	1.98	1.98	1.12	1.00
4X	0.027	128	118,937	6,659	4	36	3.37	3.43	1.39	1.00
8X	0.016	117	183,500	10,333	4	48	5.69	5.29	2.16	1.00
C2H	0.029	127	109,692	4,160	4	25	3.14	3.17	0.87	1.00
SOFT	0.152	139	22,930	3,333	4	24	0.60	0.66	0.70	1.00

Table A.9 Performance and area results for the zigzag ordering application

	Running Time(s)	Fmax (MHz)	Wall-Clock Throughput (KB/s)	Area (LEs)	Multipliers	M4K Blocks	Cycle Throughput Improvement	Wall-Clock Speedup	Relative LEs	Relative Multipliers
FBROOK	0.117	127	27,206	4,825	6	33	1.00	1.00	1.00	1.00
2X	0.059	127	53,944	5,461	8	34	1.98	1.98	1.13	1.33
4X	0.032	128	99,624	6,831	12	36	3.66	3.66	1.42	2.00
8X	0.016	119	185,724	10,694	20	48	7.31	6.83	2.22	3.33
C2H	0.029	122	104,747	4,241	6	25	4.03	3.85	0.88	1.00
SOFT	0.29	139	12,018	3,333	4	24	0.40	0.44	0.69	0.67

Table A.10 Performance and area results for the inverse quantization application

	Running Time(s)	Fmax (MHz)	Wall-Clock Throughput (KB/s)	Area (LEs)	Multipliers	M4K Blocks	Cycle Throughput Improvement	Wall-Clock Speedup	Relative LEs	Relative Multipliers
FBROOK	0.082	130	39,713	4,733	4	32	1.00	1.00	1.00	1.00
2X	0.041	126	77,028	5,242	4	32	2.00	1.94	1.11	1.00
4X	0.025	131	130,514	6,596	4	32	3.28	3.29	1.39	1.00
8X	0.016	130	202,423	9,760	4	40	5.13	5.10	2.06	1.00
C2H	0.016	128	200,013	4,028	4	24	5.13	5.04	0.85	1.00
SOFT	0.169	139	20,623	3,333	4	24	0.49	0.52	0.70	1.00

Table A.11 Performance and area results for the mismatch control application

	Running Time(s)	Fmax (MHz)	Wall-Clock Throughput (KB/s)	Area (LEs)	Multipliers	M4K Blocks	Cycle Throughput Improvement	Wall-Clock Speedup	Relative LEs	Relative Multipliers
FBROOK	0.309	135	10,891	4,693	4	32	1.00	1.00	1.00	1.00
2X	0.154	136	22,102	5,105	4	32	2.01	2.03	1.09	1.00
4X	0.081	130	39,969	6,292	4	32	3.81	3.67	1.34	1.00
8X	0.04	130	81,095	9,705	4	40	7.73	7.45	2.07	1.00
16X	0.021	114	136,086	19,372	4	56	14.71	12.50	4.13	1.00
16X_ALT	0.019	100	131,869	28,365	4	88	16.26	12.11	6.04	1.00
C2H	0.065	132	50,868	3,958	4	24	4.75	4.67	0.84	1.00
SOFT	0.174	139	20,030	3,333	4	24	1.78	1.84	0.71	1.00

Table A.12 Performance and area results for the saturation application

	Running Time(s)	Fmax (MHz)	Wall-Clock Throughput (KB/s)	Area (LEs)	Multipliers	M4K Blocks	Cycle Throughput Improvement	Wall-Clock Speedup	Relative LEs	Relative Multipliers
FBROOK	0.211	135	15,972	8,667	32	38	1.00	1.00	1.00	1.00
2X	0.105	131	31,263	13,218	60	44	2.01	1.96	1.53	1.88
4X	0.054	114	52,574	25,582	70	56	3.91	3.29	2.95	2.19
C2H	0.371	89	5,973	8,494	22	27	0.57	0.37	0.98	0.69
SOFT	0.338	139	10,312	3,333	4	24	0.62	0.65	0.38	0.13

Table A.13 Performance and area results for the IDCT application

Table A.14 shows results of our experiments for the MPEG-2 application. The baseline FPGA Brook implementation without any replication is labelled *FBROOK*. The implementation with the saturation kernels replicated two times, corresponding to Figure 6.17b is labelled *FBROOK 2*. The implementation with the saturation and IDCT kernels replicated two times, corresponding to Figure 6.17c is labelled *FBROOK 3*. Two implementations with the saturation kernels replicated four times and the IDCT kernels replicated two times, corresponding to Figure 6.17d are labelled *FBROOK 4* and *FBROOK 5*. The *FBROOK 5* implementation uses collectors, distributors and collector/distributor nodes with FIFO depth of 8, as discussed in section 6.3.9. Finally, the implementation with the saturation and the IDCT kernels replicated four times, and inverse quantization kernel replicated two times, corresponding to Figure 6.17e is labelled *FBROOK 6*.

	Running Time(s)	Fmax (MHz)	Wall-Clock Throughput (KB/s)	Area (LEs)	Multipliers	M4K Blocks	Cycle Throughput Improvement	Wall-Clock Speedup	Relative LEs	Relative Multipliers
FBROOK	0.309	126	10,177	11,214	34	60	1.00	1.00	1.00	1.00
FBROOK 2	0.211	128	15,117	12,118	34	60	1.46	1.49	1.08	1.00
FBROOK 3	0.155	126	20,260	16,554	62	70	1.99	1.99	1.48	1.82
FBROOK 4	0.127	120	23,697	18,910	62	66	2.43	2.33	1.69	1.82
FBROOK 5	0.117	127	27,154	19,435	62	82	2.64	2.67	1.73	1.82
FBROOK 6	0.106	110	25,980	31,715	70	91	2.92	2.55	2.83	2.06
C2H	0.739	81	2,744	9,551	24	30	0.42	0.27	0.85	0.71
SOFT	1.431	139	2,436	3,333	4	24	0.22	0.24	0.30	0.12

Table A.14 Performance and area results for the MPEG-2 application

Bibliography

[1] G. DeMichelli, *Synthesis and Optimization of Digital Circuits*. McGraw Hill, New York, NY, 1994.

[2] Altera Corporation, "Nios II C-to-Hardware Acceleration Compiler," [Online Document, Cited: 2009, December 3], Available HTTP: http://www.altera.com/products/ip/processors/nios2/tools/c2h/ni2-c2h.html

[3] Y.Y. Leow, C.Y. Ng, and W.F. Wong, "Generating Hardware from OpenMP Programs," in Proc. of the IEEE International Conference on Field Programmable Technology, Bangkok, Thailand, 2006, pp 73-80.

[4] R. Stephens, "A Survey of Stream Processing," Acta Informatica, vol. 34, no. 7, pp 491-541, July 1997.

[5] I. Buck, "Brook Spec v0.2," Technical Report CSTR 2003-04 10/31/03 12/5/03, Stanford University, 2003.

[6] S. Neuendorffer and K. Vissers, "Streaming Systems in FPGAs", in Proc. of the 8th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation, Samos, Greece, 2008, pp 147-156.

[7] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: Stream Computing on Graphics Hardware," ACM Transactions on Graphics, vol. 23, no. 3, pp. 777-786, August 2004.

[8] Brook for GPUs Forum. [Online Document, Cited: 2009, December 3], Available HTTP: http://www.gpgpu.org/forums/index.php?c=5

[9] W.J. Dally et al., "Merrimac: Supercomputing with Streams," in Proc. of the 2003 ACM/IEEE Conference on Supercomputing, Phoenix, AZ, 2003, pp. 35-42.

[10] J. Gummaraju and M. Rosenblum, "Stream Programming on General-Purpose Processors," in Proc. of the 38th International Symposium on Microarchitecture, Washington, DC, 2005, pp. 343-354.

[11] T. J. Purcell, "Ray Tracing on a Stream Processor," Ph.D. Dissertation, Stanford University, 2004.

[12] S-W. Liao, Z. Du, G. Wu, and G-Y. Lueh, "Data and Computation Transformations for Brook Streaming Applications on Multiprocessors," in Proc. of the International Symposium on Code Generation and Optimization, Washington, DC, 2006, pp. 196-207.

[13] Advanced Micro Devices, Inc. “ATI Stream Software Development Kit,” [Online Document, Cited: 2009, December 3], Available HTTP: http://developer.amd.com/gpu/atistreamsdk/pages/default.aspx

[14] J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed., Morgan Kaufmann Publishers, San Francisco, CA, 2003.

[15] I. Buck, “Stream Computing on Graphics Hardware,” Ph.D. Dissertation, Stanford University, 2006.

[16] F. Plavec, Z. Vranesic, and S. Brown, “Enhancements to FPGA Design Methodology Using Streaming,” in Proc. of the International Conference on Field Programmable Logic and Applications (FPL ‘09), Prague, Czech Republic, 2009, pp. 294-301.

[17] R.G. Lyons, *Understanding Digital Signal Processing*, 2nd ed., Prentice Hall, Upper Saddle River, NJ, 2004.

[18] GPU Brook source code, [Online Document, Cited: 2009, December 4], Available HTTP: http://sourceforge.net/projects/brook/

[19] Altera Corporation, “Cyclone II Device Handbook,” [Online Document, Cited: 2009, December 5], Available HTTP: http://www.altera.com/literature/hb/cyc2/cyc2_cii5v1.pdf

[20] Xilinx Inc., “Virtex-5 Family Overview,” [Online Document, Cited: 2009, December 5], Available HTTP: http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf

[21] V. Betz, J. Rose, and A. Marquardt, *Architecture and CAD for Deep-Submicron FPGAs*, Kluwer Academic Publishers, Norwell, MA, 1999.

[22] Altera Corporation, “Quartus II Handbook Version 9.1,” [Online Document, Cited: 2009, December 5], Available HTTP: http://www.altera.com/literature/hb/qts/quartusii_handbook.pdf

[23] Altera Corporation, “Nios II Processor: The World's Most Versatile Embedded Processor,” [Online Document, Cited: 2009, December 5], Available HTTP: http://www.altera.com/products/ip/processors/nios2/ni2-index.html

[24] Xilinx Inc., “MicroBlaze Soft Processor Core,” [Online Document, Cited: 2009, December 5], Available HTTP: http://www.xilinx.com/tools/microblaze.htm

[25] K. Compton and S. Hauck, “Reconfigurable Computing: A Survey of Systems and Software,” ACM Computing Surveys, vol. 34, no. 2, pp. 171-210, June 2002.

[26] Mentor Graphics, “Catapult C Synthesis,” [Online Document, Cited: 2009, December 7], Available HTTP: http://www.mentor.com/products/esl/high_level_synthesis/catapult_synthesis/

[27] Altera Corporation, “Nios II C2H Compiler User Guide,” [Online Document, Cited: 2009, December 7], Available HTTP: http://www.altera.com/literature/ug/ug_nios2_c2h_compiler.pdf

[28] Altera Corporation, “Avalon Interface Specifications,” [Online Document, Cited: 2009, December 7], Available HTTP: http://www.altera.com/literature/manual/mnl_avalon_spec.pdf

[29] Altera Corporation, “Optimizing Nios II C2H Compiler Results,” [Online Document, Cited: 2009, December 7], Available HTTP: http://www.altera.com/literature/hb/nios2/edh_ed51005.pdf

[30] Impulse Accelerated Technologies, “Impulse CoDeveloper C-to-FPGA Tools,” [Online Document, Cited: 2009, December 7], Available HTTP: http://www.impulseaccelerated.com/products_universal.htm

[31] Forte Design Systems, “Cynthesizer Closes the ESL-to-Silicon Gap,” [Online Document, Cited: 2009, December 7], Available HTTP: http://www.forteds.com/products/cynthesizer.asp

[32] Agility Design Solutions Inc., “Handel-C: C for Rapid FPGA Implementation,” [Online Document, Cited: 2009, December 7], Available HTTP: http://agilityds.com/products/c_based_products/dk_design_suite/handel-c.aspx

[33] Synfora Inc., “PICO for FPGA,” [Online Document, Cited: 2009, December 7], Available HTTP: http://www.synfora.com/products/fpga.html

[34] D. Pellerin and S. Thibault, *Practical FPGA Programming in C*, Prentice Hall, Upper Saddle River, NJ, 2005.

[35] L.W. Howes, P. Price, O. Mencer, O. Beckmann, and O. Pell, “Comparing FPGAs to Graphics Accelerators and the Playstation 2 Using a Unified Source Description,” in Proc. of the International Conference on Field Programmable Logic and Applications (FPL ’06), Madrid, Spain, 2006, pp. 1-6.

[36] N. Bellas, S.M. Chai, M. Dwyer, and D. Linzmeier, “Template-Based Generation of Streaming Accelerators from a High Level Presentation,” in Proc. of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM ’06), Napa, CA, 2006, pp. 345-346.

[37] N. Bellas, S.M. Chai, M. Dwyer, and D. Linzmeier, “An Architectural Framework for Automated Streaming Kernel Selection,” in Proc. of the International Parallel and Distributed Processing Symposium (IPDPS ’07), Long Beach, CA, 2007, pp. 1-7.

[38] R. Mukherjee, A. Jones, and P. Banerjee, “Handling Data Streams while Compiling C Programs onto Hardware,” in Proc. of the IEEE Computer Society Annual Symposium on VLSI, Lafayette, LA, 2004, pp. 271-272.

[39] M.I. Gordon et al., “A Stream Compiler for Communication-Exposed Architectures,” in Proc. of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, 2002, pp. 291-303.

[40] M.B. Taylor et al., “Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams,” in Proc. of the 31st International Symposium on Computer Architecture, Munich, Germany, 2004, pp. 2-13.

[41] W. Thies, “Language and Compiler Support for Stream Programs,” Ph.D. Dissertation, Massachusetts Institute of Technology, 2009.

[42] A. Hormati, M. Kudlur, S. Mahlke, D. Bacon, and R. Rabbah, “Optimus: Efficient Realization of Streaming Applications on FPGAs,” in Proc. of the 2008 International Conference on Compilers, Architectures and Synthesis For Embedded Systems (CASES ‘08), Atlanta, GA, 2008, pp. 41-50.

[43] D. Tarditi, S. Puri, and J. Oglesby, “Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses,” Microsoft Research Technical Report, MSR-TR-2005-184, 2006.

[44] M. McCool, S. Du Toit, T. Popa, B. Chan, and K. Moule, “Shader Algebra.” in Proc. of the International Conference on Computer Graphics and Interactive Techniques, Los Angeles, CA, 2004, pp. 787-795.

[45] NVIDIA Corporation, “CUDA Zone,” [Online Document, Cited: 2009, December 9], Available HTTP: http://www.nvidia.com/object/cuda_home.html

[46] J. Cong, G. Han, and W. Jiang, “Synthesis of an Application-Specific Soft Multiprocessor System,” in Proc. of the 15th International Symposium on Field Programmable Gate Arrays (FPGA ’07), Monterey, CA, 2007, pp. 99-107.

[47] M. Saldaña, D. Nunes, E. Ramalho, and P Chow, “Configuration and Programming of Heterogeneous Multiprocessors on a Multi-FPGA System Using TMD-MPI,” in Proc. of the 3rd International Conference on Reconfigurable Computing and FPGAs (ReConFig ‘06), San Luis Potosi, Mexico, 2006, pp. 1-10.

[48] L. Shannon, B. Fort, S. Parikh, A Patel, M. Saldaña, and P. Chow, “A System Design Methodology for Reducing System Integration Time and Facilitating Modular Design Verification,” in Proc. of the International Conference on Field-Programmable Logic and Applications (FPL ‘06), Madrid, Spain, 2006, pp. 289-294.

[49] B. Fort, D. Capalija, Z. G. Vranesic, and S. D. Brown, “A Multithreaded Soft Processor for SoPC Area Reduction,” in Proc. of the 14th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM ’06), Napa, CA, 2006, pp. 131-142.

[50] M. Labrecque and J.G. Steffan, “Improving Pipelined Soft Processors with Multithreading,” in Proc. of the 17th International Conference on Field Programmable Logic and Applications (FPL ‘07), Amsterdam, Netherlands, 2007, pp. 210-215.

[51] P. Yiannacouras, J.G. Steffan, and J. Rose, “VESPA: Portable, Scalable, and Flexible FPGA-Based Vector Processors,” in Proc. of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES ’08), Atlanta, GA, 2008, pp. 61-70.

[52] J. Yu, G. Lemieux, and C. Eagleston, “Vector Processing as a Soft-Core CPU Accelerator,” in Proc. of the 16th International Symposium on Field Programmable Gate Arrays (FPGA ’08), Monterey, CA, 2008, pp. 222-232.

[53] J. Kingyens, “A GPU-Inspired Soft Processor for High-Throughput Acceleration,” Masters Thesis, University of Toronto, 2008.

[54] S-K. Lam and T. Srikanthana, "Rapid Design of Area-Efficient Custom Instructions for Reconfigurable Embedded Processing," Journal of Systems Architecture, vol. 55, no. 1, pp. 1-14, January 2009.

[55] Altera Corporation, "Nios II Processor Reference Handbook," [Online Document, Cited: 2009, December 8], Available HTTP: http://www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf

[56] M.H. Drake, "Stream Programming for Image and Video Compression," Masters Thesis, Massachusetts Institute of Technology, 2006.

[57] J.L. Mitchell, W.B. Pennebaker, C.E. Fogg, and D.J. LeGall, *MPEG Video Compression Standard*, Chapman & Hall, New York, NY, 1997.

[58] GPU Brook: Current Issues and Restrictions, [Online Document, Cited: 2009, December 17], Available HTTP: http://graphics.stanford.edu/projects/brookgpu/issues.html

[59] StreamIt Benchmarks, [Online Document, Cited: 2009, December 17], Available HTTP: http://groups.csail.mit.edu/cag/streamit/apps/library_only/mpeg2/streamit

[60] Altera Corporation, "Nios II Software Developer's Handbook," [Online Document, Cited: 2009, December 18], Available HTTP: http://www.altera.com/literature/hb/nios2/n2sw_nii5v2.pdf

[61] Altera Corporation, "DE2 Development and Education Board," [Online Document, Cited: 2009, December 18], Available HTTP: http://www.altera.com/education/univ/materials/boards/unv-de2-board.html

[62] F. Plavec, Z. Vranesic, S. Brown, "Towards Compilation of Streaming Programs into FPGA Hardware," In Proc. of the Forum on Specification and Design Languages (FDL '08), Stuttgart, Germany, 2008, pp. 67-72.

[63] F. Plavec, Z. Vranesic, S. Brown, "Challenges in Compilation of Brook Streaming Programs for FPGAs," Workshop on Soft Processor Systems (WoSPS '08), Toronto, Canada, 2008.

[64] F. Plavec, Z. Vranesic, S. Brown, "Stream Programming for FPGAs," In Languages for Embedded Systems and their Applications, Springer, 2009, pp. 241-253.

[65] CTool Library. [Online Document, Cited: 2010, January 2], Available HTTP: http://ctool.sourceforge.net

[66] J. Sheldon, W. Lee, B. Greenwald, and S. Amarasinghe, "Strength Reduction of Integer Division and Modulo Operations," In Languages and Compilers for Parallel Computing, 2003, pp. 1-14.

[67] Altera Corporation, "SCFIFO and DCFIFO Megafunctions User Guide," [Online Document, Cited: 2010, January 2], Available HTTP: http://www.altera.com/literature/ug/ug_fifo.pdf

[68] M.J. Quinn, *Parallel programming in C with MPI and openMP*, McGraw-Hill, Dubuque, IA, 2004.

[69] I. Foster, *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*, Addison Wesley, Reading, MA, 1995.

[70] M. Golumbic, "Combinatorial merging," IEEE Transactions on Computers, vol. C-25, no. 11, pp. 1164-1167, November 1976.

[71] D.A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," Proceedings of the Institute of Radio Engineers (IRE), vol. 40, no. 9, pp. 1098-1101, September 1952.

[72] Graphviz - Graph Visualization Software. [Online Document, Cited: 2010, January 23], Available HTTP: http://www.graphviz.org/

[73] MPEG.ORG, "Mpeg2play Source Code," [Online Document, Cited: 2010, January 23], Available HTTP: http://www.mpeg.org/pub_ftp/mpeg/mssg/old/mpeg2play_v1.1b.tar.gz

[74] Z. Wang, "Fast algorithms for the discrete W transform and for the discrete Fourier transform," IEEE Transactions on Acoustics, Speech and Signal Processing, vol. 32, no 4, pp. 803-816, August 1984.

[75] Altera Corporation, "C2H Compiler Mandelbrot Design Example," [Online Document, Cited: 2010, January 31], Available HTTP: http://www.altera.com/support/examples/nios2/exm-c2h-mandelbrot.html

[76] B. Eckel, *Thinking In C++*, 2nd ed., Prentice Hall, Upper Saddle River, NJ, 2000.

[77] CAST, Inc., "2-D Inverse Discrete Cosine Transform Megafunction," [Online Document, Cited: 2010, March 24], Available HTTP: http://www.cast-inc.com/ip-cores/multimedia/idct/cast_idct-a.pdf

[78] C. Schneider, M. Kayss, T. Hollstein, and J. Deicke, "From Algorithms to Hardware Architectures: A Comparison of Regular and Irregular Structured IDCT Algorithms," In Proc. of Design, Automation and Test in Europe (DATE '98), Paris, France, 1998, pp. 186-190.

[79] N.D. Zervas, Alma Technologies S.A., Private Communication, March 2010.

[80] A. Roldao and G. Constantinides, "A High Throughput FPGA-Based Floating Point Conjugate Gradient Implementation for Dense Matrices," ACM Transactions on Reconfigurable Technology and Systems, vol. 3, no. 1, pp. 1:1-1:19, January 2010.