2018

AN ARCHITECTURE EVALUATION AND IMPLEMENTATION OF A SOFT GPGPU FOR FPGAs

Kevin Andryc

Follow this and additional works at: https://scholarworks.umass.edu/dissertations_2

Part of the Computer and Systems Architecture Commons

Recommended Citation
Andryc, Kevin, "AN ARCHITECTURE EVALUATION AND IMPLEMENTATION OF A SOFT GPGPU FOR FPGAs" (2018). Doctoral Dissertations. 1322.
https://scholarworks.umass.edu/dissertations_2/1322

This Open Access Dissertation is brought to you for free and open access by the Dissertations and Theses at ScholarWorks@UMass Amherst. It has been accepted for inclusion in Doctoral Dissertations by an authorized administrator of ScholarWorks@UMass Amherst. For more information, please contact scholarworks@library.umass.edu.
AN ARCHITECTURE EVALUATION AND IMPLEMENTATION OF A SOFT GPGPU FOR FPGAs

A Dissertation Presented

by

KEVIN R. ANDRYC

Submitted to the Graduate School of the University of Massachusetts Amherst in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 2018

ELECTRICAL AND COMPUTER ENGINEERING
AN ARCHITECTURE EVALUATION AND IMPLEMENTATION OF A SOFT GPGPU FOR FPGAs

A Dissertation Presented

by

KEVIN R. ANDRYC

Approved as to style and content by:

____________________________________
Russell G. Tessier, Chair

____________________________________
Wayne P. Burleson, Member

____________________________________
Patrick A. Kelley, Member

____________________________________
J. Blair Perot, Member

C. V. Hollot, Department Head
Electrical and Computer Engineering
DEDICATION

This dissertation is dedicated to my parents, Joseph and Sandra Andryc, and to my friend and advisor Dr. Russell Tessier.
ACKNOWLEDGMENTS

This dissertation is dedicated to my parents, Joseph and Sandra Andryc, who loved and supported me through all my endeavors. Though a long and turbulent path, they were always there to help ease the road ahead. I would also like to acknowledge my sister Cindy Sheppardson, brother-in-law Paul Sheppardson, nephew Paul Sheppardson Jr., niece Kaitlyn Sheppardson, and niece Kailey Sheppardson.

I would also like to provide a special acknowledgement to my advisor, committee chair, and friend, Dr. Russell Tessier, for his encouragement and helping me reach my potential. This dissertation would not have been made possible without his guidance through many difficult times in my life. There are no words to express my sincere gratitude and what his friendship means to me. Dr. Tessier provided support and inspiration, possessing both personal and professional attributes that I strive for, and I will always consider it to be a tremendous honor to have studied under his tutelage.

I would also like to extend a special thanks to Murtaza Merchant who assisted me in my research during his time at the University of Massachusetts as a Master’s student. I will always be grateful for his dedication, commitment, and work ethic. I remember the late nights and passionate discussions, but above all, I remember being thankful for his friendship.

Dr. Neal G. Anderson, Professor of Electrical and Computer Engineering, deserves a special mention and my sincere gratitude. Dr. Anderson would always make time to talk
to me on a personal level and provided me the courage to continue on with my educational pursuit. He is a great person and friend and someone who has done more for me than he probably realizes. For that, I will always be grateful.

I would also like to acknowledge L-3 KEO and the many friends and co-workers who supported me throughout my PhD career. Some of those who deserve special mention are Gary Kelley, Mike Haley, Charlene Pellegrino, Joe Findley, Jack Ouimet, and Mark Fydenkevez. I would also like to thank Mike Rose for his support and friendship, who helped bring me to L-3 KEO and made it possible to work with all the brilliant and creative people. Dave Beaudet, who I have the pleasure to develop a friendship and whose creative mind has inspired me. A very special thanks to goes to Jim Westwell whose friendship and support has helped guide me both personally and professionally. There are moments, both good and bad, that we have shared that I will always remember.

I would like to thank all of the teachers, professors, and other professionals who educated, trained, or inspired me in some way.

A final thanks goes to all of my friends who influenced and supported me along my journey. While it would be impossible to list every name, some of those that come to mind are Christine Church, Allen Jenkins, Dave Kubera, Jayson Lacasse, Christopher Bates, George Condon, George Condon Jr., and Tim Eagleson. Lastly, I would like to provide a special thanks to my sincere friend Tom Jamate who was always there when I needed someone.
ABSTRACT
AN ARCHITECTURE EVALUATION AND IMPLEMENTATION OF A SOFT
GPGPU FOR FPGAs

SEPTEMBER 2018

KEVIN R. ANDRYC, B.S., UNIVERSITY OF MASSACHUSETTS
M.S., UNIVERSITY OF MASSACHUSETTS
Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Russell G. Tessier

Embedded and mobile systems must be able to execute a variety of different types
of code, often with minimal available hardware. Many embedded systems now come with
a simple processor and an FPGA, but not more energy-hungry components, such as a
GPGPU. In this dissertation we present FlexGrip, a soft architecture which allows for the
execution of GPGPU code on an FPGA without the need to recompile the design. The
architecture is optimized for FPGA implementation to effectively support the conditional
and thread-based execution characteristics of GPGPU execution without FPGA design
recompilation. This architecture supports direct CUDA compilation to a binary which is
executable on the FPGA-based GPGPU. Our architecture is customizable, thus providing
the FPGA designer with a selection of GPGPU cores which display performance versus
area tradeoffs.

This dissertation describes the FlexGrip architecture in detail and showcases the
benefits by evaluating the design for a collection of five standard CUDA benchmarks
which are compiled using standard GPGPU compilation tools. Speedups of 23x, on
average, versus a MicroBlaze microprocessor are achieved for designs which take
advantage of the conditional execution capabilities offered by FlexGrip. We also show FlexGrip can achieve an 80% average reduction of dynamic energy versus the MicroBlaze microprocessor.

The dissertation furthers discussion by exploring application-customized versions of the soft GPGPU, thus exploiting the overlay architecture. We expand the architecture to multiple processors per GPGPU and optimizing away features which are not needed by certain classes of applications. These optimizations, which include the effective use of block RAMs and DSP blocks, are critical to the performance of FlexGrip. By implementing a 2 GPGPU design, we show speedups of 44x on average versus a MicroBlaze microprocessor. Application-customized versions of the soft GPGPU can be used to further reduce dynamic energy consumption by an average of 14%.

To complete this thesis, we augmented a GPGPU cycle accurate simulator to emulate FlexGrip and evaluate different levels of cache design spaces. We show performance increases for select benchmarks, however, we also show that 64% and 45% of benchmarks exhibited performance decreases when L1D cache was enabled for the 1 SMP and 2 SMP configurations, and only one benchmark showed performance improvement when the L2 cache was enabled.
TABLE OF CONTENTS

ACKNOWLEDGMENTS ...........................................................................v
ABSTRACT ............................................................................................ vii
LIST OF TABLES .................................................................................. xi
LIST OF FIGURES ................................................................................ xii
CHAPTER
1. INTRODUCTION ..................................................................................1
2. BACKGROUND .................................................................................. 7
   2.1 GPGPUs ....................................................................................... 7
   2.2 Differences between GPGPUs and Vector Processors ....................... 9
   2.3 GPGPU Cache Memory .................................................................. 10
   2.4 Related Work ............................................................................... 12
   2.5 Summary ..................................................................................... 19
3. FLEXGRIP SOFT-GPGPU ................................................................. 20
   3.1 FlexGrip System Overview ............................................................ 20
   3.2 FlexGrip Streaming Multiprocessor ................................................. 21
   3.3 CUDA Instructions ....................................................................... 24
   3.4 FPGA-Specific Considerations ....................................................... 25
   3.5 Experimental Methodology ............................................................ 27
      3.5.1 Software Flow ....................................................................... 27
      3.5.2 Design Environment and Benchmarks .................................... 27
   3.6 Experimental Results .................................................................... 29
      3.6.1 Architectural Scalability .......................................................... 30
      3.6.2 Application Scalability .............................................................. 32
      3.6.3 Energy Efficiency ................................................................... 33
   3.7 Summary ...................................................................................... 33
4. FLEXGRIP SOFT-GPGPU OPTIMIZATIONS ....................................... 35
   4.1 Architectural Optimizations ............................................................ 35
      4.1.1 Conditional Branch Circuitry Optimization ................................. 35
      4.1.2 Multiple Streaming Multiprocessors ....................................... 38
      4.1.3 Source Operand Optimization .................................................. 40
   4.2 Experimental Results .................................................................... 41
      4.2.1 Comparison versus the MicroBlaze Soft-Core Processor ............... 42
   4.3 Energy Comparison versus a CPU running Ocelot ............................ 44
   4.4 Architectural Customization ............................................................. 45
   4.5 Summary ...................................................................................... 47
5. A COMPARISON OF CACHE CONFIGURATIONS FOR SOFT-GPGPUs .... 48
   5.1 Overview ...................................................................................... 48
   5.2 Motivation ...................................................................................... 49
   5.3 Modifying GPGPU-Sim for FlexGrip .............................................. 52
      5.3.1 SIMT Core (SMP) .................................................................. 54
   5.4 Memory Hierarchy ........................................................................ 61
      5.4.1 SMP Memory and Level-1 Data Cache .................................... 62

Page
xii 62 61 54 52 49 48 48 44 42 41 38 35 35 35 33 32 30 29 27 27 27 27 25 24 21 20 20 19 12 10 9 7 7 1 vii v

ix
5.4.2 Level-2 Data Cache ................................................................. 64
5.5 Experimental Methodology .......................................................... 67
  5.5.1 Baseline Configuration ......................................................... 67
  5.5.2 Benchmarks .................................................................. 68
5.6 Cache Configuration Trade-Offs .................................................... 68
  5.6.1 L1D Cache Performance ....................................................... 69
  5.6.2 L2 Cache Performance ......................................................... 81
5.7 Summary .............................................................................. 91

6. CONCLUSION, FUTURE CONSIDERATIONS AND PUBLISHED WORK
  6.1 Conclusion ........................................................................... 93
  6.2 Future Considerations .............................................................. 94
  6.3 Published Work .................................................................. 95

BIBLIOGRAPHY ............................................................................ 96
LIST OF TABLES

Table                                      Page

Table 1: FlexGrip-Supported CUDA Instructions ................................................. 25
Table 2: Area comparison of FlexGrip implementations ........................................ 30
Table 3. FPGA Power Estimates (W) at 100 MHz .................................................. 33
Table 4. MicroBlaze vs. FlexGrip Energy Consumption: 256 data size .................... 33
Table 5: FlexGrip Physical Limits ......................................................................... 38
Table 6: Area comparison of FlexGrip implementations for 1 and 2 SMs ................. 41
Table 7: Comparison of FlexGrip implementations ............................................... 43
Table 8. Speedup of 2 SM versus 1 SM for input data size 256 ............................. 43
Table 9. Ocelot Energy Consumption: 256 data size ............................................. 44
Table 10. Results of FlexGrip optimizations for an 1 SM, 8 SP system ..................... 45
Table 11. Default GPGPU-Sim Configuration ......................................................... 67
Table 12. GPGPU Cache Benchmarks .................................................................... 68
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1: Overview of a GPGPU architecture. The architecture can support multiple streaming multiprocessors</td>
<td>7</td>
</tr>
<tr>
<td>Figure 2. NVIDIA GPGPU representative architecture</td>
<td>11</td>
</tr>
<tr>
<td>Figure 3: Overview of the system architecture showing the FlexGrip GPGPU connected to the MicroBlaze processor via the AXI bus</td>
<td>21</td>
</tr>
<tr>
<td>Figure 4: Block diagram depicting the details of the FlexGrip Streaming Multi-processor</td>
<td>23</td>
</tr>
<tr>
<td>Figure 5: Software Flow for the FlexGrip Soft GPU</td>
<td>28</td>
</tr>
<tr>
<td>Figure 6: Percent of instruction operations executed for each benchmark</td>
<td>29</td>
</tr>
<tr>
<td>Figure 7. Speedup vs. MicroBlaze for variable scalar processors and input data size 256 for 1 SM</td>
<td>31</td>
</tr>
<tr>
<td>Figure 8. Speedup of 1 SM, 32-SP GPGPU vs. MicroBlaze for varying problem size</td>
<td>32</td>
</tr>
<tr>
<td>Figure 9: FlexGrip conditional branch and warp stack architecture. There is one stack and one set of predicate registers for each of the eight warps</td>
<td>36</td>
</tr>
<tr>
<td>Figure 10: FlexGrip read stage and execute unit</td>
<td>41</td>
</tr>
<tr>
<td>Figure 11. Speedup vs. MicroBlaze for variable scalar processors and input data size 256 for 2 SM</td>
<td>42</td>
</tr>
<tr>
<td>Figure 12. GPU architecture highlighting the multi-level cache hierarchy</td>
<td>49</td>
</tr>
<tr>
<td>Figure 13. Different memory access patterns resulting in either (a) a single transaction or (b) two transactions</td>
<td>51</td>
</tr>
<tr>
<td>Figure 14. Top level architecture modeled by GPGPU-Sim</td>
<td>53</td>
</tr>
<tr>
<td>Figure 15. Detailed architecture of the GPGPU-Sim SIMT Core. The labels listed above are the analogous FlexGrip SMP pipeline stages</td>
<td>54</td>
</tr>
<tr>
<td>Figure 16. GPGPU-Sim operand collector microarchitecture [48]</td>
<td>59</td>
</tr>
<tr>
<td>Figure 17. FlexGrip read, execute and write pipeline stages</td>
<td>60</td>
</tr>
</tbody>
</table>
Figure 18. Representative FlexGrip block diagram exhibiting the details of the GPPGU memory hierarchy. ................................................................. 62

Figure 19. L1 data cache and supporting memory components. ....................... 63

Figure 20. FlexGrip SMP depicting the integration of L1 cache into the architecture... 64

Figure 21. L2 data cache memory partition..................................................... 65

Figure 22. Execution time speedup relative to the 1 SMP baseline system for various L1D cache configurations. ................................................................. 70

Figure 23. Execution time speedup relative to the 2 SMP baseline system for various L1D cache configurations. ................................................................. 71

Figure 24. Performance increase of the architecture with 1-cycle perfect memory access versus baseline architecture with global memory modeled and no cache. ....... 72

Figure 25. Normalized global memory traffic with and without L1D cache for 1 SMP and 2 SMP configurations, normalized to no cache. ............................. 74

Figure 26. Execution time speedup of Breadth First Search relative to 1 SMP and 2 SMP baseline systems for various L1D cache configurations....................... 75

Figure 27. Execution time speedup of Autocorrelation relative to 1 SMP and 2 SMP baseline systems for various L1D cache configurations.......................... 77

Figure 28. Execution time speedup of Matrix Multiply relative to 1 SMP and 2 SMP baseline systems for various L1D cache configurations............................. 78

Figure 29. CUDA kernel code for the Matrix Multiply benchmark........................ 79

Figure 30. Calculation of a row of a tile in matrix C using a single row in matrix A and an entire tile of matrix B................................................................. 80

Figure 31. Execution time speedup relative to the 1 SMP baseline system for various L2 cache configurations. ................................................................. 82

Figure 32. Execution time speedup relative to the 2 SMP baseline system for various L2 cache configurations. ................................................................. 82

Figure 33. Normalized network latency with and without L2 cache for 1 SMP and 2 SMP configurations, normalized to no cache (baseline). .................................... 84

Figure 34. CUDA kernel code for the Matrix Transpose benchmark..................... 84

Figure 35. Transpose from A to C, whereby an entire row of tiles of matrix A are used to output the transposed column elements in C. ................................................. 85
Figure 36. A fraction of the CUDA kernel code for the k-means benchmark.............. 87

Figure 37. A simple example of two threads executing a memory load into the features array for the k-means benchmark. ................................................................. 89
CHAPTER 1

INTRODUCTION

FPGAs are used in a wide variety of embedded systems, such as automotive applications, appliances, and other consumer products. Most of the processing is performed by low-end embedded microprocessors and FPGAs. In some cases, just an FPGA is used and one or more microprocessors are fashioned from FPGA logic to execute specific code types. The benefits of this approach include the ability of software designers to specify functionality in a familiar high-level language (e.g. C) and the flexibility to modify this functionality for the FPGA device without the need to recompile FPGA logic, a time-consuming process that can range from minutes to days.

A recent trend in FPGA design is the use of overlay architectures. An overlay design implements a soft, synthesizable version of an architecture which is customarily implemented in fixed ASIC logic, such as a microprocessor, vector processor, or multiprocessor. For example, soft microprocessors have become ubiquitous in FPGA design and they are used for a variety of embedded applications ranging from I/O and system control that do not demand high performance to data processing with higher computational demands.

Most FPGA designs use soft processors for sequential tasks, such as I/O interfacing and control that do not demand high performance. The benefits of soft processor usage include the ability of software designers to specify functionality in a familiar high-level language (e.g. C) and the flexibility to modify this functionality for
the FPGA device without the need to recompile FPGA logic, a time-consuming process that can range from minutes to days.

The success of soft microprocessors led to alternative compute models which follow a similar simple program-compile design flow. Recently, soft vector processors [1] [2], which provide performance benefits for applications exhibiting significant data parallelism have appeared. Although soft vector processors address a portion of the data parallel spectrum, they are limited in their support for significant multithreaded and conditional program execution. Multithreaded soft processors have been reported although they have generally been constrained to executing a small number of threads [3] or have limited parallelism [4].

Graphics processing units for general purpose computing (GPGPUs) are now widely-accepted computing platforms for a broad range of multi-threaded, conditional computation. The programming languages created to program GPGPUs, CUDA and OpenCL, are now in wide use for other computing platforms, and creating a code and knowledge base for programmers. The critical benefit of GPGPUs, besides their inherent parallelism, is their ability to automatically manage the execution of highly multi-threaded applications in hardware, freeing the programmer to focus on achieving maximum parallelization by writing efficient CUDA code. Although a number of previous projects have explored mapping GPU languages directly to FPGA hardware [5] [6], "GPU-like" soft FPGA architectures [7] [8], and soft multi-cores [9], a soft GPGPU architecture which allows for direct execution of CUDA binary code following
compilation with the CUDA compile-time environment has not been reported. Previous architectures also primarily consider hardware synthesis for each application, which is a lengthy and potentially infeasible option for designers which desire to execute a number of GPGPU applications on the same FPGA substrate.

This dissertation focuses on an exploration of soft GPGPU architectures in FPGAs. We describe the implementation of FlexGrip [10] [11] (FLEXible GRaphIcs Processor for general-purpose computing), a fully CUDA binary-compatible integer GPGPU, optimized for FPGA implementation. The amount of parallelism is customizable at multiple levels including the number of parallel operations per instruction (processors) per multiprocessor.

FPGA implementation allows for additional optimization for classes of computation which may not require all components in a standard GPGPU. As part of our work we explore the possibility of creating a small set of soft GPGPUs with varying architectural parameters (e.g. number of functional units, size of memory structures) which can be swapped into the FPGA as needed. The interaction between FlexGrip and an on-chip MicroBlaze soft processor is coordinated allowing for the seamless execution of sequential and parallel portions of a CUDA program. The hardware can be used for numerous CUDA programs without hardware resynthesis. Different versions of FlexGrip can be created that can be optimized for specific classes of application requirements, such as the number of conditionals or the need for a multiply-accumulate operation.
The architecture has been implemented in VHDL for a variety of parameters and evaluated in hardware using an ML605 [12] Virtex-6 FPGA platform which includes DRAM. A total of five CUDA benchmarks have been directly compiled to the architecture using standard NVIDIA compiler products. We show dynamic energy savings versus a soft-core processor of 66% on average. Through optimization of per-application on-chip resources, an additional 14% dynamic energy reduction is possible. We also provide a dynamic energy comparison for CUDA code compiled to FlexGrip versus a high-end Intel processor. For some benchmarks, FlexGrip provides reduced dynamic energy consumption versus the much larger, fixed microprocessor.

Within this dissertation we describe the effect of architectural optimizations including reducing the numbers of functional units, conditional execution hardware, and memory interfaces on energy consumption. The effects of using multiple SMs to perform computation are also explored. Results for each of these experiments versus a baseline FlexGrip architecture are presented to quantify the results of the optimizations. Additionally, energy consumption comparisons versus a high-end Intel processor are made to provide an additional energy comparison.

While FPGAs provides significant flexibility afforded to the designer, it is done so at the expense of fixed resources that is chip dependent. A number of look-up tables (LUTs), flip-flops, block RAM, and DSPs are dependent upon the packaging selected and therefore, the designer may sometimes need to make architectural decisions regarding optimizations. Many modern day processor and GPGPU designs now include multiple
hierarchies of cache in an effort to improve throughput and overcome limitations of main memory. By evaluating different cache designs and parameter choices, we can determine an optimized strategy based on a fixed number of block RAM.

Specific contributions of our work include:

- We provide a detailed analysis of the operation and resources consumed by the FlexGrip design as we vary the number of scalar processors, characterizing the performance, energy and power consumption.

- We analyze tradeoffs as we vary the amount of conditional execution hardware, number of scalar processor operands and functions supported by the scalar processors. These characterizations allow for the optimization of area and energy.

- We consider FPGA performance tradeoffs as the number of scalar processors in the soft GPGPU and the number of streaming multiprocessors are varied. The variation in compute density also effects the energy consumption of the design.

- We explore GPGPU cache designs by emulating the FlexGrip architecture on a cycle accurate GPGPU simulator. The total size of the cache memory is constrained to emulate the number of block RAMs available on the FPGA. By evaluating different cache hierarchy and parameter designs, we can provide trade-off analysis and characterize performance.

The remainder of this dissertation is structured as follows. Section 2 provides background on similar overlay and synthesized architectures and an overview of relevant features of GPUs. Section 3 describes the architecture of FlexGrip and provides an
overview of the entire FlexGrip system including the soft GPGPU, MicroBlaze, and the DRAM interface. Architectural optimizations are a specific focus in Section 4. Section 5 describes our work of comparing cache configurations for a soft-GPGPU. Finally, in Section 6 we provide our conclusion including our published papers.
CHAPTER 2

BACKGROUND

2.1 GPGPUs

GPGPUs have a many-core device architecture and possess substantial parallel processing capabilities. As shown in Figure 1, a typical GPGPU consists of an array of multiprocessors (each with two or more processors) enabling the device to execute numerous threads in parallel. In a GPGPU, a majority of the silicon area is dedicated to data processing units with only a small portion assigned to data caching and flow control circuitry. Such a design architecture makes a GPGPU suitable for solving streaming compute-intensive problems.

Figure 1: Overview of a GPGPU architecture. The architecture can support multiple streaming multiprocessors
Although several different companies manufacture GPGPUs, in describing the devices we will use terminology commonly used with NVIDIA devices. A GPGPU is primarily made up of an array of streaming multiprocessors (SMs), with each multiprocessor consisting of multiple scalar processor (SP) cores that generally use 32-bit operands. The term streaming multiprocessor implies that scalar processors in an SM perform the same operation, SIMD style. The vector register file contains a pool of registers that is strictly partitioned across scalar processors. This way, every processor uses its own set of registers to store operands and intermediate results, steering them clear of any data dependent hazards. A shared memory serves as a communication medium between different cores residing in the same SM. In addition, there is a read-only constant memory accessible by all the threads. The constant memory space is a cache for each SM, thus allowing fast data access as long as all threads read the same memory address.

In the CUDA programming model, the host program launches a series of kernels organized as a grid of thread blocks. A thread block represents a collection of operations which can be performed in parallel. The NVIDIA device architecture partitions thread blocks and groups them into warps, where a warp is a smaller set of simultaneous operations, some of which may be performed conditionally. Multiple warps may be assigned to a single SM and scheduled over time. To manage fine-grained scheduling, each SM is architected as a single instruction, multiple-thread (SIMT) processor. A single instruction is mapped to the scalar processors in the SM and each processor maintains its own program counter (PC). Every thread performs the same operation on a different set of data, but is free to independently execute data-dependent branches. Branching threads
diverge from the normal execution flow and scalar processors which do not execute the branch must be marked (deactivated) during this execution. The thread instructions executed as a taken branch are executed serially, while the non-branching threads are *masked* until they are executed later.

### 2.2 Differences between GPGPUs and Vector Processors

In general, GPGPUs and vector processors have many similarities and a few differences [13]. Both architectures support wide data parallel, SIMD-style computation using multiple parallel compute lanes, provide support for conditional operations, and require optimized interfaces to on-chip and off-chip memory. However, soft vector processors contain a number of limitations regarding implementation and compiler support that are addressed by GPU architectures.

- **Scalable thread counts**: In general, GPGPUs provide support for significant amounts of compute threads both within an SM and across SMs. Vector processors are generally limited to a single thread per SIMD processor (similar to an SM). Our architecture supports the implementation of numerous threads.

- **Hardware support for conditional operations**: The conditional branch mechanism for GPGPUs is typically implemented in *hardware* to simplify both the user programming model and the associated compiler. This approach allows for run-time determined levels of loop nesting and data-dependent branching.
The burden for handling conditional operations in vector processors generally falls on both the programmer and the compiler with minimal hardware support provided.

- **Overcoming memory latency:** The memory system for GPGPUs is architected to take advantage of the presence of numerous threads which can be switched with low overhead by a thread scheduler. Vector processors generally rely on deep pipelining to overcome memory latency.

Our architecture addresses each of these points using an implementation which is optimized for FPGAs. We show that a soft GPGPU implementation allows a designer to trade off the amount of SMs and conditional branch hardware as needed for classes of applications.

### 2.3 GPGPU Cache Memory

GPGPUs have the capability to execute thousands of threads concurrently and thus rely on high memory throughput. In an effort to provide sufficient memory bandwidth, GPU designers have begun to implement cache memory as part of the architecture.
Figure 2 shows a reference GPGPU architecture representative of modern NVIDIA GPUs such as Fermi [14] and Kepler [15].

Prior to execution, data is transferred to the GPGPUs global memory, which typically consists of many gigabytes of off-chip GDDR5 memory. The global memory is shared by all the SMPs and is partitioned, with each partition containing an L2 cache bank. Each SMP contains four different types of on-chip memory: shared memory, data cache, local cache and texture cache. Shared memory is a fast, on-chip multi-banked scratchpad memory that is not backed by any cache and facilitates communication between threads in a single block. The L1 data cache is a private, per-SMP first level cache and includes a Miss Status Holding Register (MSHR) to track cache misses in flight. Constant cache stores constant and parameter data, similar to the L1 data cache with the exception that it is read-only. Texture cache utilizes a unique pre-fetching mechanism [16] for storage and retrieval of graphics data.
2.4 Related Work

Our soft GPGPU is part of a larger trend in FPGA usage to eliminate the long FPGA compile times and difficult hardware design cycles for many designers. Instead of application-specific custom hardware, an architectural overlay [17] is implemented in FPGA hardware. An overlay circuit typically has the features of a common ASIC-based architecture (microprocessor, vector processor, GPGPU, etc.). Designers can specify applications in more familiar languages (e.g., C, CUDA) which require modest compile times.

Although these architectures exhibit lower performance and higher energy consumption than their full custom counterparts, they can be swapped into the FPGA on-demand, providing the flexibility needed by embedded systems. Over the past ten years, the implementation of soft vector processors on FPGAs has matured significantly. A number of projects have examined the implementation of data parallel applications on FPGAs using these architectures. The VEGAS [2] and VENICE projects [18] examined the implementation of soft vector processors on a range of FPGAs. These architectures support a customizable number of operations performed in parallel, an optimized memory interface, and a compiler. VENICE supports a simple, mask-based approach to conditionally execute specific data-parallel operations. Conditional operations are explicitly managed with code generated via compilation. The VESPA project [19] explored a soft vector processor approach that considers the customization of the soft vector processor instruction set and data bit widths. A later project [20] exploited the pipeline parallelism found in FPGAs to create custom modules that can be integrated into
the soft vector processor datapath. As mentioned in the Section 2.2, although similar, vector processors have a more constrained operating model compared to GPGPUs. Specifically, vector processors require a compiler to perform strip mining of vector accesses and explicitly manage the implementation of multiple threads.

Several FPGA-targeted projects considered the mapping of GPGPU applications represented in OpenCL to multi-threaded implementations. The OpenRCL project [9] focused on a compiler for a multi-core architecture. The results for a single application mapped to a 30-core architecture using this LLVM-based compiler showed a 5x power improvement versus a commercial GPU for similar performance. This implementation does not implement multiple threads on a processor at the same time. Labrecque and Steffan [4] described the multithreading of a single processor core. Hazard logic is removed from the processor and hazards are avoided by switching between up to seven different threads. Another work [3] considered an extension of this idea to include multiple cores of these simple multi-threaded processors operating in parallel. Kingyens and Steffan [21] described a GPU-like architecture that has some similarities to our architecture. Their GPU-like architecture includes multithreading across 32 “batches”, small cores which contain ALUs. This architecture was described in the context of a graphics application although it was not fully implemented in RTL or in hardware. The architecture is notable for its multiple execution cores, support for up to 256 threads, and limited support for conditional data parallel execution. The architecture does not support nested conditionals, multiple clusters of multi-processors, or a direct compile path for CUDA or OpenCL GPGPU languages. Although these projects examined a similar goal
to ours, FlexGrip employs the ability to target CUDA or OpenCL code to FPGAs without hardware recompile. In addition, the earlier architectures and compilers do not take advantage of the dynamic thread scheduling and hardware-controlled parallel branch mechanisms, including deeply nested loops, commonly found in GPGPUs and expected by GPGPU compilers. FlexGrip is also scalable to multiple multi-processor clusters. Our implementation is fully compatible with CUDA integer binaries and typical GPGPU operation.

Many recent projects, including commercial offerings, have examined synthesizing designs specified in CUDA and OpenCL to application-specific circuits implemented in FPGAs. The MARC architecture [22], a multi-core with custom datapaths, was optimized on a per-application basis to achieve competitive performance versus full-custom FPGA implementation. The FCUDA project [5] developed a tool which converts CUDA programs to a synthesizable version of C. A high-level synthesis tool and FPGA compiler then converts this code to hardware circuits. This work was later extended to consider the synthesis of multiple dependent kernels [23]. Owaida et al. [6] presented an approach which converts OpenCL code to a synthesizable RTL template. This approach is appropriate for applications and programmer coding styles which match well with the template. Similarly, Shagritchaya et al. [24] developed an OpenCL compiler with a library that supports the OpenCL host API. Finally, Altera has developed an OpenCL compiler [25] which converts OpenCL programs to a series of custom parallel compute cores. Although all of these approaches generate circuits which are optimized for a specific application and reap the associated area, performance, and energy benefits,
they all require the substantial compile time associated with FPGA synthesis, place, and route. The migration of a new application to the FPGA requires substantially more time than the few seconds normally found when targeting CUDA programs to GPGPUs.

In direct relation with the approach discussed, a proliferation of work has been done recently in the area of soft GPGPUs, attempting to implement GPGPU functionality with configurable or application specific processing soft cores. Al-Dujaili, et al. [8] implemented a simple soft-GPU based on the LEON3 processor with eight threads which requires hand-compilation of GPU programs and tested only for matrix multiplication. The implementation achieved speedups of up to 3x over the LEON3 architecture, with the memory interface being the limiting performance factor. While this was an early attempt at a soft-GPGPU, there are many recent papers that have cited and leveraged the work of FlexGrip. Siddiqui, et al. [26] developed an architecture called Image Processing PROcessor (IPPro) using small, reconfigurable soft-core scalar RISC processors. Similar to FlexGrip, it features a five stage pipeline utilizing Xilinx DSP48E1 primitives as the base design for the Arithmetic Logic Unit (ALU) and distributed block RAM to support the memory hierarchy. A feature of IPPro is the ability to configure the cores as a multi-core heterogeneous architecture, enabling the user to build either SIMD or MIMD computational models. Unlike FlexGrip, the architecture specifically targets image processing algorithms, using the RVC-CAL [27] dataflow language which is then converted to IPPro binary code. An open source RTL implementation of a GPGPU called MIAOW (Many-core Integrated Accelerator of Wisconsin) [28] was developed with an architecture similar to AMD’s Southern Island (SI) ISA [29]. The authors devised a
hybrid strategy, with L2 cache, on-chip networks (OCN), and memory controllers developed as C/C++ behavioral models and the remaining architecture implemented as RTL. It is able to run applications written in standard OpenCL, supporting a subset of the SI ISA, eliminating any graphic-related instructions. The MIAOW compute unit (similar to NVIDIA’s SM) was synthesized using 32nm technology and is able to run at 222MHz.

An FPGA implementation, called Neko, was developed as part of the effort. Due to the size, a single compute unit (CU) was implemented along with a Microblaze processor on a Xilinx VC707 evaluation board with each CU requiring 195,285 LUTs (64% of available resources) and 137 BRAMs (16% of available BRAMs). As Neko was based on MIAOW, there was no RTL implementation of a memory controller, therefore the Microblaze processor was used as an intermediary for accessing memory. However, there were no timing or performance results reported for Neko. Similar to MIAOW, Kadi et al. [30] developed FGPU (FPGA general purpose Graphical Processing Unit), using a custom ISA, extended from MIPS and inspired from the OpenCL execution model. While early versions required the user to write assembly code, a later version provided compiling directly from OpenCL language. Each FGPU compute unit features a scheduling unit, memory controller, runtime memory and eight processing elements. Each processing element consists of a vector register file and an ALU. FGPU was implemented on a Xilinx ZC706 development board which can support up to 8 compute units. Results showed speedups between 10.6x and 48.5x over Microblaze and compared to an equivalent ARM with the NEON SIMD engine achieved 3.5x when in the 8 CU configuration. Nyami [31] is another open source soft-GPGPU implementation developed in Verilog with an associated simulation model. It features an in-order, single issue,
unified scalar and vector pipeline with a register-to-register RISC ISA execution model. A baseline configuration of Nyami consisting of DRAM, video controller, and a was synthesized for the Altera Cyclone IV E (EP4CE115F29C7) FPGA occupying a total of 92,186 logic elements (81% of the device logic). It was deployed on the Cyclone FPGA, therefore no results were published, however, static timing analysis reported 30MHz.

There have been many recent studies with regard to GPGPU cache design and optimization. The authors in [32] and [33] have looked at improvement methods of the last level cache (LLC) to optimize data transfers and reduce latency between the CPU and GPGPU. In [34], L2 cache locking techniques are examined in an effort to improve time predictability for real-time applications.

There have been other studies that have looked at optimization of caches on a per-SMP basis. Huangfu et al. in [35] increase cache performance by bypassing the cache determined by profiling accesses. Thread mapping and scheduling techniques have also been explored in order to dynamically quantify and improve performance, such as those in [36], [37], [38], and [39]. Sankaranarayanan et al. [40] introduce an additional, shared per-SP incoherent cache called tinyCache. They claim the ability to filter out 62% of memory requests serviced by the L1 data cache, and almost 81% of requests to shared memory providing a 37% energy reduction within the on-chip memory resources.

While these techniques are relevant to our work, FPGAs feature specific cache design challenges, especially in highly multithreaded processors, such as soft GPGPUs
and soft vector processors (SVPs). Previous work on FPGA caches include TputCache [41] which focused on implementing a highly-pipeline cache design operating at near maximum frequency of BRAMs for throughput processing. The approach features a replay-based architecture, the ability to support multiple outstanding misses, write coalescing and arbitrary associativity. The design used the XMP SVP on the Cyclone IV and Stratix IV FPGAs and achieved speedups of up to 10.5x versus the non-cache architecture.

Other works includes Yiannacouras et al. [42] which analyzed the performance of the memory subsystem by adjusting the cache depth, line size, and hardware prefetch mechanism of their VESPA soft vector processor. They show an average performance increase of nearly 2x for 1.8x the system design area.

Our approach attempts to effectively support the CUDA programming and compile environments available to GPU programmers on FPGAs without the need for costly hardware compilation or remapping to parallel RISC-style integer cores. We envision such a system as being particularly useful for environments such as embedded processing where compute nodes contain reconfigurable logic that may be used for many different purposes at different times. In these cases, the extra cost, complexity, or power consumption of an off-the-shelf GPGPU in the nodes may be unwanted or unnecessary. The soft GPGPU can be swapped into the FPGA as needed and used to execute recently-compiled (perhaps on-the-fly compiled) CUDA code. Several custom versions of the soft GPGPU can also be available and swapped in based upon requirements, resource
availability in the FPGA, or architectural parameters needed by the application. Our approach provides a fast way to target CUDA programs to these environments.

In addition, the previous works on caches focused on optimizing throughput for highly data-parallel architectures, however do not address the cache hierarchy associated with many designs. Similarly, to this date there have been no research into the design and analysis of caches on soft-GPGPUs.

2.5 Summary

This chapter provided an introduction of the GPGPU architecture and described terminology essential for understanding the concepts of execution. It outlined how parallel execution occurs as seen from the programmer and the GPGPU hardware. The discussion included comparing GPGPUs to vector processors, an FPGA overlay architecture that can execute SIMD-style computations. The chapter concludes exploring research related to our work, showing the various methodologies used to implement GPGPUs on FPGAs.
CHAPTER 3

FLEXGRIP SOFT-GPGPU

3.1 FlexGrip System Overview

Our FlexGrip soft GPGPU detailed in this section is part of our published work [10]. The design is used in concert with a Xilinx MicroBlaze to execute parallel operations. The FlexGrip soft GPGPU is attached to the Xilinx MicroBlaze soft-core microprocessor via the AXI bus as shown in Figure 3. During execution of a program, the MicroBlaze processor loads a driver that communicates control, status, and data to the AXI bus interface logic. The control logic acts as an interface between the AXI bus and the FlexGrip GPGPU. It executes functions depending on the values written to the control register. Once the driver is loaded, it dispatches CUDA instructions and data which in turn are loaded into system and global memory, respectively, by the control logic. In addition, the driver loads parameters associated with the CUDA kernel program such as thread block and grid dimensions, number of thread blocks per SM, the number of registers used per thread, and the shared memory size. These parameters are stored in the GPGPU configuration registers. After initialization, control flow is passed to the GPGPU to execute the CUDA kernel. During this period, the MicroBlaze processor can continue execution concurrently with the GPGPU.

FlexGrip follows a SIMT model in which an instruction is fetched and mapped onto multiple scalar processors simultaneously. The block scheduler is responsible for scheduling thread blocks in a round-robin fashion. The number of thread blocks scheduled at the same time is determined by the number of scalar processors in an SM.
and the number of SMs. After scheduling the thread blocks, the block scheduler signals the warp unit to initiate scheduling the warps, which are contained within the respective thread blocks. The maximum number of thread blocks that can be scheduled to a SM is restricted by the available shared memory and SM registers. The GPGPU controller acts as the interface between the block scheduler and the SM. It initializes registers in the vector register file with respective thread IDs.

Figure 3: Overview of the system architecture showing the FlexGrip GPGPU connected to the MicroBlaze processor via the AXI bus

### 3.2 FlexGrip Streaming Multiprocessor

For this custom FPGA implementation, we have developed a five-stage pipelined SM architecture, shown in Figure 4. The SM includes Fetch, Decode, Read, Execute and
Write stages. The *warp unit* at the front of the pipeline coordinates the execution of instructions through the pipeline. The following sections elaborate on the different blocks used in this architecture. Once the block scheduler assigns thread blocks to a specific SM, the warp unit assigns threads to specific scalar processors (SP). This unit schedules warps in a round-robin fashion. Each warp includes a program counter (PC), a *thread mask*, and state. Each warp maintains its own PC and can follow its own conditional path. The mask is used to prevent thread execution within a warp for threads which do not meet specific conditions. The warp state indicates the status of the warp: Ready, Active, Waiting or Finished. The Ready state indicates that the warp is idle and is ready to be scheduled, while the Active state indicates that the warp is currently active in the pipeline.

Within a warp, threads are arranged in rows depending on the number of scalar processors (SP) instantiated within an SM. For example, for an 8-SP configuration, a warp with 32 threads would be arranged in four rows with each row containing 8 threads. Similarly, for a 16-SP configuration, a warp would be arranged in two rows with 16 threads each. The maximum parallelism is achieved with 32 SPs and one row.

The Fetch stage is the initial stage of the execution pipeline and is responsible for fetching four or eight-byte CUDA binary instructions from system memory. After fetching the instruction, the PC value is incremented (by 4/8 bytes) to point to the next instruction. The Decode stage decodes the binary instruction to generate several output tokens such as the operation code, predicate data, source and destination operands.
Figure 4: Block diagram depicting the details of the FlexGrip Streaming Multiprocessor

In the Read stage, source operands are read from the vector register file or shared/global memory blocks depending on the decoded inputs. The vector register file is partitioned, with each thread assigned a set of general-purpose registers. The address register file stores memory addresses for load and store instructions. All instructions can include an optional predicate flag that controls conditional execution of the instruction (predicate instructions). The predicate register file is used to store these predicate flags, each of which is then used as an index into a predicate look-up table which obtains the predicated instruction (i.e.: less than, greater than, etc.). The active-thread mask is updated by combining the thread mask with the predicated instruction. The constant memory is a read-only memory which is initialized by the host.

The Execute stage consists of multiple scalar processors and a single control flow unit. This unit operates on control flow instructions such as branch and synchronization instructions which are described in more detail in the next section. Each thread is mapped to one scalar processor, enabling parallel execution of threads. The scalar processors
support integer-type addition, subtraction, multiplication, multiply and add, data type convert operations, shifting operations and Boolean logic operations.

The Write stage stores intermediate data in the vector register file, memory addresses in the address register file, and predicate flags in the predicate register file. Final results are stored in the global memory. All pipeline stages output a stall signal that is fed to the preceding stage. The stall signal indicates that the stage is busy and not ready to accept new data.

3.3 CUDA Instructions

The soft GPGPU supports the NVIDIA G80 instruction set with compute capability 1.0. Instructions were tested based on the requirements of the selected benchmarks. We tested 27 integer CUDA instructions as a part of this research. The list of all supported instructions is shown in Table 1. All instructions needed by our benchmark circuits are supported.
Table 1: FlexGrip-Supported CUDA Instructions

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I2I</td>
<td>Copy integer value to integer with conversion</td>
</tr>
<tr>
<td>IMUL/IMUL32/IMUL32I</td>
<td>Integer multiply</td>
</tr>
<tr>
<td>SHL</td>
<td>Shift left</td>
</tr>
<tr>
<td>IADD</td>
<td>Integer addition between two registers</td>
</tr>
<tr>
<td>GLD</td>
<td>Load from global memory</td>
</tr>
<tr>
<td>R2A</td>
<td>Move register to address register</td>
</tr>
<tr>
<td>R2G</td>
<td>Store to shared memory</td>
</tr>
<tr>
<td>BAR</td>
<td>Barrier synchronization</td>
</tr>
<tr>
<td>SHR</td>
<td>Shift right</td>
</tr>
<tr>
<td>BRA</td>
<td>Conditional branch</td>
</tr>
<tr>
<td>ISET</td>
<td>Integer conditional set</td>
</tr>
<tr>
<td>MOV/ MOV32</td>
<td>Move register to register</td>
</tr>
<tr>
<td>RET</td>
<td>Conditional return form kernel</td>
</tr>
<tr>
<td>MOV R, S[]</td>
<td>Load from shared memory</td>
</tr>
<tr>
<td>IADD, S[], R</td>
<td>Integer addition between shared memory and register</td>
</tr>
<tr>
<td>GST</td>
<td>Store to global memory</td>
</tr>
<tr>
<td>AND C[], R</td>
<td>Logical AND</td>
</tr>
<tr>
<td>IMAD/IMAD32</td>
<td>Integer multiply-add; all register operands</td>
</tr>
<tr>
<td>SSY</td>
<td>Set synchronization point; used before potentially divergent instructions</td>
</tr>
<tr>
<td>IADDI</td>
<td>Integer addition with an immediate operand</td>
</tr>
<tr>
<td>NOP</td>
<td>No operation</td>
</tr>
<tr>
<td>@P</td>
<td>Predicated execution</td>
</tr>
<tr>
<td>MVI</td>
<td>Move immediate to destination</td>
</tr>
<tr>
<td>XOR</td>
<td>Logical XOR</td>
</tr>
<tr>
<td>IMADI/ MAD32I</td>
<td>Integer multiply-add with an immediate operand</td>
</tr>
<tr>
<td>LLD</td>
<td>Load from local memory</td>
</tr>
<tr>
<td>LST</td>
<td>Store to local memory</td>
</tr>
<tr>
<td>A2R</td>
<td>Move address register to data register</td>
</tr>
</tbody>
</table>

3.4 FPGA-Specific Considerations

All circuitry described in this section has been implemented in a Virtex-6 FPGA and has been shown to operate correctly. While a strength of the FlexGrip architecture is its ability to execute numerous CUDA binaries without the need for FPGA design recompilation, a user may select to create a new FlexGrip implementation, if desired. The FlexGrip architecture is designed such that different counts of scalar processors per SM,
SMs per GPGPU, warp stack size, and multiplier/third operand usage can be implemented by modifying parameters in a configuration file and rerunning Xilinx tools. Depending on the target FPGA platform, the user can customize FlexGrip to maximize performance or area. For a specific FlexGrip hardware implementation, a small set of in-design registers are used to store application specific configuration information, such as thread block count.

Most of FlexGrip source code was written in custom VHDL code to provide for fine-grained control, although MATLAB’s Simulink was used for coarse-grained functions. Xilinx System Generator converts MATLAB Simulink blocks to RTL code for rapid development of FPGA designs. For example, Simulink was used to connect DSP, adder, and multiply blocks together to form SP functional units. To minimize data latency, we heavily utilize dual-ported block RAMs throughout the design. In the case of the warp unit scheduler, the state information and the data are stored in block RAM indexed by the warp ID. This allows warps to be scheduled every clock cycle after an initial one clock cycle of latency. Similarly, the vector, predicate, and address registers use dual-port block RAM providing simultaneous read and write access. To support the numerous integer arithmetic instructions, the scalar processors take advantage of Xilinx’s DSP48E1 digital signal processing blocks. A single DSP slice can support add/subtract, multiply, multiply add, shift, and bitwise logic instructions.
3.5 Experimental Methodology

3.5.1 Software Flow

The complete CUDA binary code generation flow is illustrated in Figure 5. At compile time, the input program is divided by the CUDA front-end (*cudafe*) into C/C++ host code and the GPU device code. The GPU code is fed to the host compiler (e.g.: *gcc*, *cl*) to generate a filehash containing device code descriptors. The device descriptors are evaluated by runtime libraries whenever device code is invoked by the system. The NVIDIA CUDA compiler (*nvcc*) converts this information to PTX assembly instruction code which is then converted to CUDA binary instructions (*.cubin*). This code, along with the device code descriptors, are merged (*fatbin*) and compiled together with the host compiler to produce a final executable. Microsoft Visual Studio 2008 and NVIDIA Toolkit v2.3 are used together to compile the CUDA code file. The NVIDIA toolkit is comprised of the NVIDIA CUDA compiler (*nvcc*), and the CUDA driver and runtime API libraries required for building the executable and the cubin file.

3.5.2 Design Environment and Benchmarks

Synthesis was performed using the Xilinx ISE 14.2 toolkit and Modelsim SE 10.1 was used for simulation and verification. A block-level simulation approach was adopted, where each block was individually verified using logic simulation in addition to a system level verification. Inputs were stimulated using CUDA binary instructions and data stored in block RAM. To rapidly evaluate a variety of benchmarks and data, we generated Memory Initialization Files (*.mif*) that were used to populate Xilinx Block RAM cores.
We have evaluated five CUDA applications, *bitonic sort* (bs), *autocorrelation* (ac), *matrix multiplication* (mm), *parallel reduction* (pr) and *transpose* (tr) from the University of Wisconsin [43] and the NVIDIA Programmer’s Guide [44], using FlexGrip. The mix of data-parallel (e.g. multiply, transpose) and control-flow intensive (e.g. bitonic sort) benchmarks helped us evaluate our platform. Figure 6 provides a breakdown of the instruction operations by type for each of the benchmarks.
3.6 Experimental Results

The FlexGrip soft GPGPU design was implemented on a Xilinx ML605 development board [12] which utilizes a Virtex-6 VLX240T device. The device area and design operating frequency for designs with a varying number of scalar processors are annotated in Table 2.

We performed experiments and compared performance and energy results against a Xilinx MicroBlaze soft-core processor with about 3,000 LUTs running at 100 MHz using C versions of the same benchmarks. For the purposes of this paper, a design with a single SM and 8 scalar processors was implemented and benchmarked on the ML605 board, while 16- and 32-SP designs were simulated. The FlexGrip design implemented in
hardware could successfully run all five benchmarks using the same FPGA bitstream. The CUDA compile times for all benchmarks were less than one second.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Freq. (MHz)</th>
<th>LUTs</th>
<th>FFs</th>
<th>BRAM</th>
<th>DSP48E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 SM – 8 SP</td>
<td>100</td>
<td>60,375</td>
<td>103,776</td>
<td>124</td>
<td>156</td>
</tr>
<tr>
<td>1 SM – 16 SP</td>
<td>100</td>
<td>113,504</td>
<td>149,297</td>
<td>132</td>
<td>300</td>
</tr>
<tr>
<td>1 SM – 32 SP</td>
<td>100</td>
<td>231,436</td>
<td>240,230</td>
<td>156</td>
<td>588</td>
</tr>
</tbody>
</table>

### 3.6.1 Architectural Scalability

We ran experiments by varying the number of scalar processors within a single SM, which effectively varies the number of threads that can be executed in parallel. Figure 7 shows application speedups versus a MicroBlaze for a varying number of SPs per SM. Application speedups range from 7x to 29x with an average speedup close to 12x for 8 SPs, 18x for 16 SPs, and 22x for 32 SPs. Since they are highly data parallel, matrix multiplication and reduction show the largest speedups. Reduction has a highly symmetric data flow graph consisting of multiple iterations. The number of array elements in the benchmark is halved with each iteration, progressively leading to smaller number of scheduled warps. Considering the array size to be a multiple of 32 (the warp size), all active threads remain tightly packed within a warp in every iteration, thus fully utilizing the warp at all times. In bitonic, the sorting network consists of a fixed number of swapping operations that are performed at every stage. Though the warp divergence increases with an increased number of parallel threads, the divergence cost is amortized by performing more swapping operations in parallel. Transpose shows less speedup due to low arithmetic intensity and memory bandwidth limitations. Matrix multiply has better
performance than transpose, as the former has higher arithmetic density which amortizes the number of required memory accesses.

![Image](318x39)\[108x702]

Figure 7. Speedup vs. MicroBlaze for variable scalar processors and input data size 256 for 1 SM

One common limitation to cycle speedup for all benchmarks targeted to our architecture is memory access. Memory operations are most effective when the burst data is written and read in parallel. This action requires the memory to be split up into multiple banks and coalesced, such that consecutive memory addresses fall into consecutive banks. Most data parallel CUDA kernels include neighboring threads that access consecutive memory locations. However, for control flow intensive applications where data accesses are not sequential, memory mapping is more of a challenge, especially if multiple threads access the same memory location. For the sake of architectural simplicity, enhanced support for memory coalescing was not included in our
soft GPGPU prototype. The matrix benchmarks pay a slightly larger penalty for memory bandwidth limitations due to a larger number of scatter-gather memory operations.

3.6.2 Application Scalability

Experiments were conducted to observe the performance of the soft GPGPU in comparison to MicroBlaze for varying problem (input data array) sizes of each benchmark. The speedup results are shown in Figure 8. Due to its regular kernel structure, reduction reaps the steepest performance benefits of almost 30x as the size of the array becomes large. With increasing array size, performance increases gradually for both autocorrelation and bitonic up to a certain point and then begins to taper off. This result can be attributed to the accumulation of the warp divergence penalty over the execution time of larger arrays, amortizing the parallel processing benefits. Matrix multiply shows a speedup of about 27x, with transpose showing an average speedup of 22x. The flat curve of both benchmarks is due to limitations of the memory bandwidth.

Figure 8. Speedup of 1 SM, 32-SP GPGPU vs. MicroBlaze for varying problem size
3.6.3 Energy Efficiency

We used Xilinx's XPower power estimator tool to determine static and dynamic power for the designs (Table 3). Since static power is largely a function of the device size, we evaluate the dynamic energy consumption of the implementations. This value is determined by multiplying dynamic power by application execution time. In Table 4, it is shown that the baseline FlexGrip dramatically reduces dynamic energy consumption versus the MicroBlaze, primarily due to reduced execution time. FlexGrip also uses the same instruction for many scalar processors, limiting instruction memory accesses. For a 1 SM, 8 SP configuration, the dynamic energy reduction is about 80%, on average.

Table 3. FPGA Power Estimates (W) at 100 MHz

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Dynamic</th>
<th>Static</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 SM, 8 SP</td>
<td>0.84</td>
<td>3.45</td>
<td>4.29</td>
</tr>
<tr>
<td>1 SM, 16 SP</td>
<td>1.08</td>
<td>3.46</td>
<td>4.54</td>
</tr>
<tr>
<td>1 SM, 32 SP</td>
<td>1.39</td>
<td>3.46</td>
<td>4.85</td>
</tr>
<tr>
<td>MicroBlaze</td>
<td>0.37</td>
<td>2.00</td>
<td>2.37</td>
</tr>
</tbody>
</table>

Table 4. MicroBlaze vs. FlexGrip Energy Consumption: 256 data size

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>MicroBlaze</th>
<th>8 SP</th>
<th>16 SP</th>
<th>32 SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>ac</td>
<td>277</td>
<td>102.49</td>
<td>40.28</td>
<td>33.84</td>
</tr>
<tr>
<td>bs</td>
<td>118</td>
<td>43.66</td>
<td>9.39</td>
<td>7.88</td>
</tr>
<tr>
<td>mm</td>
<td>186041</td>
<td>68835.17</td>
<td>14098.02</td>
<td>11842.34</td>
</tr>
<tr>
<td>rd</td>
<td>11</td>
<td>4.07</td>
<td>0.66</td>
<td>0.55</td>
</tr>
<tr>
<td>tr</td>
<td>705</td>
<td>260.85</td>
<td>57.79</td>
<td>48.54</td>
</tr>
</tbody>
</table>

3.7 Summary

In this chapter, the FlexGrip soft GPGPU architecture was described. The scalable design was shown to be fully implemented and tested on a Xilinx ML605 development board. A novel design aspect of GPGPUs versus microprocessors and vector processors
is the ability to handle thread divergence and barrier synchronization in hardware. The FlexGrip soft GPGPU provides control circuitry, which can automatically handle complex conditional control operations in hardware, similar to the GPGPU programming model. We showed our design has been validated using five benchmarks which were compiled from CUDA to a binary representation. All five benchmarks were executed using the same FlexGrip design (no need to create a new bitstream). The binary was executed on the soft GPGPU without any per-application hardware modifications. Experimental results demonstrate application speedups of up to 30x versus a MicroBlaze soft processor for highly parallel benchmarks.
CHAPTER 4

FLEXGRIP SOFT-GPGPU OPTIMIZATIONS

This section expands upon our previous work in [10] to show how the support of fine-grained control access allows for features to be optimized from the FlexGrip design, which is described in our Architectural Evaluation publication [11]. We provide the details of the optimizations and their results within this section.

4.1 Architectural Optimizations

4.1.1 Conditional Branch Circuitry Optimization

A key contribution of the FlexGrip soft GPGPU is its ability to support thread-level branching in hardware. These resources provide an opportunity for architectural optimization for specific classes of applications which may exhibit less control-intensive behavior. The execution of threads in a warp diverges if the results of a conditional operation are different for different threads. In case of divergence, execution for some threads proceeds along one path (e.g., not-taken) while other threads are idle. When instructions for the not-taken path complete, the execution switches to the alternative execution path (taken path) for the remaining threads while the first set of threads are idle. When both execution paths are finished, a reconvergence point in the code is reached. At this point, execution is resynchronized across all threads and the same set of instruction operations is unconditionally performed by all threads once again. The reconvergence point is generally identified by a set synchronization (SSY) instruction that is executed just prior to the execution of the instruction which sets the branch condition.
To synchronize multiple warps within a thread block at a reconvergence point, CUDA supports explicit *barrier synchronization*. Warps that reach the barrier instruction first have to wait for other warps to reach the same checkpoint. At that point, they are marked as *Waiting* in warp state memory stored in the warp unit (not shown in Figure 4). When all the threads in a warp finish executing the kernel, the warp is declared *Finished*. The warp state memory holds the state of each warp and warp data memory (also in the warp unit) holds the active-thread mask and the warp PC.

![Figure 9: FlexGrip conditional branch and warp stack architecture. There is one stack and one set of predicate registers for each of the eight warps](image)

To handle conditional execution, each of the eight warps per SM has its own *warp stack* that includes an instruction address (32 bits), type identifier (2 bits), and an active-thread mask (32 bits) in each stack entry [13] (Figure 9). The instruction address of the
taken branch and the active-thread mask prior to evaluation of the conditional operation is stored on a warp stack for safekeeping. The stored mask contains one bit for each thread in the warp and the type identifier indicates if the instruction address is a reconvergence point or the start address of taken branch instructions. When the taken path of the branch is reached, the stack is popped and the active-thread mask for the warp is inverted to allow for execution of this second path. When the reconvergence point is reached, the original active-thread mask is retrieved by popping the stack.

A complete view of the hardware architecture used to control conditional execution in FlexGrip is shown in Figure 9. The execution of a conditional (predicate) instruction results in the generation of a four-bit predicate for each instruction (sign, zero, carry, and overflow). This four-bit instruction result for each thread is assigned to a predicate register. Each thread has 4 four-bit predicate registers ($p0$ through $p3$) assigned to it. For each thread, the value in the selected predicate register and the condition for the instruction executed for the branch (e.g. $<$, $>$, $=$) are used as in index into a lookup table to generate an instruction mask. One mask bit is generated for each thread. This mask is combined with a thread mask (e.g. thread not Finished or Waiting) to generate the active-thread mask for the warp. Warp stack pushing and popping of this information is controlled by the control flow unit state machine.

In the GPGPU control architecture, nested conditionals are possible, requiring a deep stack to hold nested address and mask information. In the worst case, only one of 32 threads may execute at a specific time, requiring support for conditional nesting up to 32
entries deep. However, for many applications, a much smaller stack depth is required. This depth can be determined by examining the amount of control nesting in the program or by profiling the application with representative data sets. In our optimizations, we consider the application warp stack depth as an optimization parameter. In Section 4.2, several architectures with varied warp stack depths are made available for execution. The size of the warp and associated control circuitry is reduced from a stack depth of 32 based on application needs. This reduction saves associated memory and logic resources, leading to energy savings.

### 4.1.2 Multiple Streaming Multiprocessors

A notable feature of our architecture is its support for multiple SMs. A thread block of up to 256 threads can be assigned to any available SM by the block scheduler (Figure 1). The number of thread blocks is specified by the programmer and passed to the FlexGrip architecture by the MicroBlaze driver at run-time. The allocation of SM shared memory and the number of registers required per block are also determined during scheduling. The values are determined during compilation and stored in GPGPU configuration registers. After assignment by the block scheduler, the warp unit in the SM uses the parameters to generate and schedule warps.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads per Warp</td>
<td>32</td>
</tr>
<tr>
<td>Warps per SM</td>
<td>24</td>
</tr>
<tr>
<td>Threads per SM</td>
<td>768</td>
</tr>
<tr>
<td>Thread Blocks per SM</td>
<td>8</td>
</tr>
<tr>
<td>Total Number of 32-bit Registers per SM</td>
<td>8,192</td>
</tr>
<tr>
<td>Shared Memory per SM (bytes)</td>
<td>16,384</td>
</tr>
</tbody>
</table>
At the start of kernel execution, the maximum number of thread blocks that can be scheduled is calculated. This value is limited by the number of allocated warps per SM, the number of registers per SM, and the size of the shared memory per SM. As an example, consider a kernel with 256 threads per thread block. The block requires 4 KB memory and each thread requires 8 registers. Table 5 lists the physical limits of the FlexGrip GPGPU. With 256 threads per block, the number of blocks per SM, 3, is calculated with the following formula:

\[
BlocksPerSM_{Warp} = \frac{\text{WarpsPerSM}}{\frac{\text{ThreadsPerBlock}}{\text{ThreadsPerWarp}}} = \frac{24}{\frac{256}{32}} = 3
\] (1)

Next, we determine the number of blocks that can be scheduled based on the number of allocated registers:

\[
BlocksPerSM_{Registers} = \frac{\text{TotalRegs}}{\text{ThreadsPerBlock} \times \text{RegsPerThread}} = \frac{8192}{256 \times 8} = 4
\] (2)

Finally, the number of blocks per SM based on the requested shared memory size is calculated by the following:

\[
BlocksPerSM_{SharedMem} = \frac{\text{SharedMemPerSM}}{\text{SharedMemSize}} = \frac{16384}{4096} = 4
\] (3)

The maximum number of blocks that can be scheduled to each SM is the minimum number of the three values calculated, which in our example is 3.

Control signals from the SM notify the block scheduler when all thread blocks have completed and scheduling of subsequent blocks can begin. Once all thread blocks have successfully executed, the block scheduler signals the GPGPU, which will notify the driver that execution has completed.
4.1.3 Source Operand Optimization

Figure 10 depicts the detailed view of the read stage, which consists of a read controller, parallel read source operand units, and interface controllers to memory subsystems and registers. The arithmetic portion of the execute stage is shown on the right side of the figure. The read controller takes in data from the decode stage, performs pre-processing depending on the operation, and then directs the data to each of the read operand units. These units are functionally identical, allowing for read operations to be performed in parallel. However, they can perform different functions depending on the instruction passed to them at run time. For example, one of the modules may perform a read operation from global memory, while the others perform a read operation from the register file.

The modular independence of the read hardware allows for the removal of one of the read operand modules and the multiplier if they are not needed by an application. For example, if an application does not perform multiply or multiply-accumulate operations, a version of the GPGPU which does not include these features could be used. This hardware is represented by the shaded blocks in Figure 10. The area and energy benefits of removing this hardware for selected applications are explored in Section 4.2.
4.2 Experimental Results

We extended the design described in Section 3 to compare a single SM versus two SMs, each with 8, 16, and 32-SP via simulation. The same baseline FlexGrip design with no architectural optimizations implemented in hardware could successfully run all five benchmarks using the same FPGA bitstream. The device area and design operating frequency for designs with a varying number of scalar processors and streaming multiprocessors are annotated in Table 6.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Freq. (MHz)</th>
<th>LUTs</th>
<th>FFs</th>
<th>BRAM</th>
<th>DSP48E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 SM – 8 SP</td>
<td>100</td>
<td>60,375</td>
<td>103,776</td>
<td>124</td>
<td>156</td>
</tr>
<tr>
<td>1 SM – 16 SP</td>
<td>100</td>
<td>113,504</td>
<td>149,297</td>
<td>132</td>
<td>300</td>
</tr>
<tr>
<td>1 SM – 32 SP</td>
<td>100</td>
<td>231,436</td>
<td>240,230</td>
<td>156</td>
<td>588</td>
</tr>
<tr>
<td>2 SM – 8 SP</td>
<td>100</td>
<td>135,392</td>
<td>196,063</td>
<td>238</td>
<td>306</td>
</tr>
<tr>
<td>2 SM – 16 SP</td>
<td>100</td>
<td>232,064</td>
<td>287,042</td>
<td>262</td>
<td>594</td>
</tr>
<tr>
<td>2 SM – 32 SP</td>
<td>100</td>
<td>413,094</td>
<td>468,959</td>
<td>310</td>
<td>1170</td>
</tr>
</tbody>
</table>

Table 6: Area comparison of FlexGrip implementations for 1 and 2 SMs
4.2.1 Comparison versus the MicroBlaze Soft-Core Processor

4.2.1.1 Architecture Scalability

We ran experiments by varying the number of scalar processors within a single SM and across 2 SMs which effectively varies the number of threads that can be executed in parallel. Benchmarks autocorr, bitonic, and reduction used input data sets of 32, 64, 128, and 256 values. Benchmarks matrix multiplication and transpose used input data sets of 32x32, 64x64, 128x128, and 256x256 for experimentation.

![Bar chart showing speedup vs. MicroBlaze for variable scalar processors and input data size 256 for 2 SM](image)

Figure 11. Speedup vs. MicroBlaze for variable scalar processors and input data size 256 for 2 SM

For experiments performed with 2 SMs, the block scheduler logic equally and automatically distributed thread blocks to the multiple SMs, thus reducing the workload of each SM to roughly half of the 1 SM cases. All benchmarks exhibited additional speedups versus the 1 SM case for the same number of SPs per SM. As shown Figure 11 and Table 7, the peak speedups for the 2 SM, 32-SP implementations of the benchmarks offer over a 40x speedup for four out of the five benchmarks.
Table 7: Comparison of FlexGrip implementations

<table>
<thead>
<tr>
<th></th>
<th>AutoCorr</th>
<th>Bitonic</th>
<th>MatrixMul</th>
<th>Reduction</th>
<th>Transpose</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (ms)</td>
<td>Speed Up</td>
<td>Time (ms)</td>
<td>Speed Up</td>
<td>Time (ms)</td>
</tr>
<tr>
<td>32 or 32x32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>uBlaze</td>
<td>5.0</td>
<td>6.0</td>
<td>374.0</td>
<td>-</td>
<td>1.0</td>
</tr>
<tr>
<td>8 SP</td>
<td>1.8</td>
<td>2.9</td>
<td>29.0</td>
<td>12.9</td>
<td>0.1</td>
</tr>
<tr>
<td>16 SP</td>
<td>1.0</td>
<td>4.8</td>
<td>0.6</td>
<td>10.7</td>
<td>0.1</td>
</tr>
<tr>
<td>32 SP</td>
<td>0.8</td>
<td>6.3</td>
<td>0.5</td>
<td>13.1</td>
<td>0.1</td>
</tr>
<tr>
<td>64 or 64x64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>uBlaze</td>
<td>18.0</td>
<td>-</td>
<td>2947.0</td>
<td>-</td>
<td>2.0</td>
</tr>
<tr>
<td>8 SP</td>
<td>3.4</td>
<td>5.3</td>
<td>225.4</td>
<td>13.1</td>
<td>0.2</td>
</tr>
<tr>
<td>16 SP</td>
<td>2.7</td>
<td>6.7</td>
<td>166.3</td>
<td>17.7</td>
<td>0.1</td>
</tr>
<tr>
<td>32 SP</td>
<td>2.1</td>
<td>8.4</td>
<td>110.6</td>
<td>26.6</td>
<td>0.1</td>
</tr>
<tr>
<td>128 or 128x128</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>uBlaze</td>
<td>70.0</td>
<td>-</td>
<td>23368.0</td>
<td>-</td>
<td>5.0</td>
</tr>
<tr>
<td>8 SP</td>
<td>11.1</td>
<td>6.3</td>
<td>1776.0</td>
<td>13.2</td>
<td>0.3</td>
</tr>
<tr>
<td>16 SP</td>
<td>8.8</td>
<td>8.0</td>
<td>1311.5</td>
<td>17.8</td>
<td>0.2</td>
</tr>
<tr>
<td>32 SP</td>
<td>6.9</td>
<td>10.2</td>
<td>870.3</td>
<td>26.9</td>
<td>0.2</td>
</tr>
<tr>
<td>256 or 256x256</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>uBlaze</td>
<td>277.0</td>
<td>-</td>
<td>186041.0</td>
<td>-</td>
<td>11.0</td>
</tr>
<tr>
<td>8 SP</td>
<td>20.9</td>
<td>13.3</td>
<td>7120.3</td>
<td>26.1</td>
<td>0.4</td>
</tr>
<tr>
<td>16 SP</td>
<td>16.6</td>
<td>16.7</td>
<td>4412.1</td>
<td>42.2</td>
<td>0.3</td>
</tr>
<tr>
<td>32 SP</td>
<td>12.8</td>
<td>21.6</td>
<td>3486.9</td>
<td>53.4</td>
<td>0.2</td>
</tr>
</tbody>
</table>

Table 8 shows the scalability of our architecture. Speedups for 2 SM versus 1 SM versions of the same benchmark ranged from 1.77 (Reduction) to 1.98 (Transpose and Matrix Multiply). The block scheduler was able to distribute thread blocks more evenly between the two SMs for the latter two applications due to a smaller number of conditional statements in the applications versus the other three applications.

Table 8. Speedup of 2 SM versus 1 SM for input data size 256

<table>
<thead>
<tr>
<th></th>
<th>8 SP</th>
<th>16 SP</th>
<th>32 SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Autocorr</td>
<td>1.94</td>
<td>1.94</td>
<td>1.94</td>
</tr>
<tr>
<td>Bitonic</td>
<td>1.82</td>
<td>1.83</td>
<td>1.85</td>
</tr>
<tr>
<td>MatrixMul</td>
<td>1.98</td>
<td>1.98</td>
<td>1.98</td>
</tr>
<tr>
<td>Reduction</td>
<td>1.78</td>
<td>1.77</td>
<td>1.77</td>
</tr>
<tr>
<td>Transpose</td>
<td>1.98</td>
<td>1.98</td>
<td>1.98</td>
</tr>
</tbody>
</table>
4.3 Energy Comparison versus a CPU running Ocelot

In general, the use of a power-hungry multicore processor is not an option for many embedded systems. However, since Ocelot [45] offers a non-GPGPU platform for direct CUDA execution, we compared energy and performance results for Ocelot running under Ubuntu 12.04 versus FlexGrip implementations on the Virtex-6 FPGA. The tested microprocessor is a 2.6 GHz Intel Core i7 2960-XM microprocessor with 16 GB of DRAM. For the purpose of our experiments, we ran Ocelot using the LLVM option and all optimizations enabled. To gather energy statistics, we used the Intel Power Gadget [46] which is capable of monitoring real-time power usage. The dynamic energy results for each of the benchmarks is listed in Table 9.

Table 9. Ocelot Energy Consumption: 256 data size

<table>
<thead>
<tr>
<th></th>
<th>Core i7</th>
<th>8 SP</th>
<th>16 SP</th>
<th>32 SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Autocorr</td>
<td>0.26</td>
<td>9.6</td>
<td>2.50</td>
<td>40.28</td>
</tr>
<tr>
<td>Bitonic</td>
<td>0.85</td>
<td>9.9</td>
<td>8.39</td>
<td>9.39</td>
</tr>
<tr>
<td>MatrixMul</td>
<td>260.97</td>
<td>10.4</td>
<td>2800.79</td>
<td>14098.02</td>
</tr>
<tr>
<td>Reduction</td>
<td>2.41</td>
<td>10.3</td>
<td>24.95</td>
<td>0.66</td>
</tr>
<tr>
<td>Transpose</td>
<td>0.84</td>
<td>10.4</td>
<td>8.81</td>
<td>57.79</td>
</tr>
</tbody>
</table>

As shown in Table 9, two of the five benchmarks, bitonic sort and reduction, were found to require 34% and 88% less energy for the 32-SP implementation, respectively, although, not surprisingly, all benchmarks executed considerably faster on the microprocessor. The energy reduction for these benchmarks can be attributed to regular memory accesses that limit FlexGrip stalling.
4.4 Architectural Customization

To limit the need for dynamic FlexGrip recompile during runtime, we would expect that user would have several precompiled FlexGrip bitstreams available for download to an embedded FPGA to execute target CUDA applications. The needed FlexGrip design characteristics represented by the bitstreams were explored via experimentation.

Table 10. Results of FlexGrip optimizations for an 1 SM, 8 SP system

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Num. of Operands</th>
<th>Warp Depth</th>
<th>Slice LUTs</th>
<th>Flip Flops</th>
<th>Black RAM</th>
<th>DSP</th>
<th>% Area Reduction</th>
<th>% Dyn. Energy Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>3</td>
<td>32</td>
<td>60,375</td>
<td>103,776</td>
<td>124</td>
<td>156</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Autocorr</td>
<td>3</td>
<td>16</td>
<td>52,121</td>
<td>82,017</td>
<td>124</td>
<td>156</td>
<td>14%</td>
<td>3%</td>
</tr>
<tr>
<td>MatrixMul</td>
<td>3</td>
<td>0</td>
<td>42,536</td>
<td>60,161</td>
<td>124</td>
<td>156</td>
<td>30%</td>
<td>9%</td>
</tr>
<tr>
<td>Reduction</td>
<td>3</td>
<td>0</td>
<td>42,536</td>
<td>60,161</td>
<td>124</td>
<td>156</td>
<td>30%</td>
<td>9%</td>
</tr>
<tr>
<td>Transpose</td>
<td>3</td>
<td>0</td>
<td>42,536</td>
<td>60,161</td>
<td>124</td>
<td>156</td>
<td>30%</td>
<td>9%</td>
</tr>
<tr>
<td>Bitonic</td>
<td>3</td>
<td>2</td>
<td>39,189</td>
<td>57,301</td>
<td>124</td>
<td>156</td>
<td>35%</td>
<td>15%</td>
</tr>
<tr>
<td>Bitonic</td>
<td>2</td>
<td>2</td>
<td>22,937</td>
<td>27,136</td>
<td>120</td>
<td>12</td>
<td>62%</td>
<td>38%</td>
</tr>
</tbody>
</table>

To evaluate the possible benefits of removing unneeded features from FlexGrip, we ran several experiments to determine the minimum required architectural configuration for area and energy optimization for each application. As described in Section 4.1, the specific optimizations include reducing the size of the warp stack (and associated control logic), removing the multiplier, and removing the third-operand read circuitry from the read stage of the SM pipeline. Table 10 lists the optimizations performed for each of the benchmarks. By performing an instruction analysis, we can determine the minimal set of functions needed to support each benchmark. The baseline scalar processor supports all instructions listed in Table 1 with no optimizations. Of the five benchmarks, we were able to remove the multiplier/third operand for bitonic, since
the benchmark does not require multiply operations. Effectively, any benchmark which performs multiplies could use this FlexGrip version and obtain the 23% dynamic energy reduction versus FlexGrip with a reduced warp stack and 38% dynamic energy reduction versus baseline FlexGrip. We note that only the multiply-add (MAD) instruction requires three operands, therefore by eliminating the multiply unit the need for support of a third operand is removed. A total of 12 DSP blocks are still used for address calculation in the FlexGrip control circuitry.

Table 10 indicates that the necessary depth of the warp stack for applications varies. As noted in Section 4.1.1, each warp has its own warp stack, which is configured as 32 registers of 66-bits each. For short instruction sequences, such as if statements without a corresponding else, the compiler uses condition codes to avoid managing divergence, reducing the need for significant warp stack depth. In cases with longer sequences of conditional code, conditional branches are used. For matrix multiplication, reduction, and transpose, conditional branches are minimized, limiting warp stack usage. By customizing the warp stack, a LUT area reduction of up to 35% and a dynamic energy reduction of up to 15% can be realized.

In an embedded system, one could consider compiling and storing the bitstreams for four separate FlexGrip GPGPUs. The baseline system would include a multiplier and a full 32-depth warp stack. A second system would include a 16-depth warp stack and a third system would have a 2-depth stack. Finally, the fourth system would include a 2-deep warp stack and no multiplier/third operand fetch unit.
4.5 Summary

In this chapter we explored the possibility of providing a small set of FlexGrip soft
GPGPU implementations that could be targeted to classes of applications with different
execution characteristics (e.g., reduced conditional operation, no multiplication). We
showed that architectural optimization can reduce dynamic energy consumption by 14%
and LUT area by 33%, on average. Experimental results demonstrated application
speedups of up to 55x for a FlexGrip design with two streaming multiprocessors (SMs)
versus a MicroBlaze soft processor operating at the same clock frequency for highly
parallel benchmarks.
CHAPTER 5

A COMPARISON OF CACHE CONFIGURATIONS FOR SOFT-GPGPUs

5.1 Overview

The effectiveness of GPGPUs relies on their ability to execute thousands of threads in parallel, however the peak performance is typically bandwidth limited. One technique employed to overcome this limitation is to utilize thread switching to hide memory latency. In an effort to provide sufficient memory bandwidth, GPU designers have begun to implement cache memory as part of the architecture, a technique commonly used in CPUs. In CPUs, cache memory is used to bridge the performance gap by mitigating long accesses to memory [13]. Adding cache accesses to prevent long latency off-chip memory accesses allows for efficient and fast access to data, especially for data that exhibit good temporal and spatial locality. However, in the case of GPUs, caches are used to reduce the amount of in-flight data requests caused by massive multithreading. The amount of data reused can provide significant speedup, especially for those applications that provide regular memory access patterns.

The GPU processors found today, such as NVIDIA’s GPUs, adopt a multilevel cache design structure as shown in Figure 12. Level-1 cache is private per streaming multiprocessor and installed on the same physical module as the shared memory. Introduced with the Fermi architecture, the amount of cache and shared memory can be configured by the programmer allowing either 48kB of shared memory and 16kB of L1 cache or as 48kB of L1 cache and 16kB of shared memory. In the case of L2 cache, it is shared among all the streaming multiprocessors on a GPU device.
Figure 12. GPU architecture highlighting the multi-level cache hierarchy

5.2 Motivation

The FlexGrip soft-GPGPU contains one or more highly threaded streaming multiprocessors, each with a number of scalar processors which have the ability to execute assigned threads in a parallel fashion. Upon launch of a GPGPU kernel and prior to execution, the runtime uses the block and grid parameters to create a massive number of threads that are organized hierarchically. The threads are then assigned consecutive IDs (thread identifiers or \( tid \)) which are then grouped into warps, with 32 threads per warp. Multiple warps are then assigned to a thread block with all thread blocks comprising a grid. During execution, warps are assigned to an SM and are then scheduled, with each thread within a warp executing in lockstep fashion.

In NVIDIA’s CUDA GPGPU architecture, when the 32 threads in a warp access global memory, the addresses are coalesced into one or more memory transactions. If
memory addresses are scattered or not concurrent such that they align on a cache boundary, multiple memory transactions are required to fulfill the request. The result may be reduced global efficiency through increased network traffic and latency waiting for all transactions to be serviced and completed. To understand this more effectively, assume that each thread needs to fetch 4 bytes of data. If the data needed by each thread are well coalesced, for example, each thread accesses adjacent 4-byte word aligned on a cache line boundary, then a single 128-byte memory transaction can be serviced. Even if the accesses by the threads are permuted within the warp, a single 128-byte transaction will still take place, as shown in Figure 13 (a). However, when threads in a warp access sequential memory locations not aligned with the cache lines or if the memory access pattern is altered, two 128-byte memory transactions will be requested, as shown in Figure 13 (b). Assuming that the data is not reused, the result is an over-fetch where only half of the data is useful. While this represents a simple case, scenarios such as these can significantly degrade both performance and energy efficiency. This scenario directly extends into the CUDA programming model where, by default, when backed by L2, data cached in both L1 and L2 will use 128-byte memory transactions [47]. However, to reduce over-fetching data, the CUDA programming model provides the ability to bypass L1, in which case 32-byte segments are used to transfer data. In cases such as those depicted in Figure 13 (b), assuming no re-use of data, bypassing L1 and using 32-byte transactions would reduce the amount of memory traffic.
Unlike an NVIDIA GPGPU, a soft GPGPU provides extreme flexibility, enabling the ability to optimize performance by trading off different aspects of the design. While caches have been used in previous designs, to date there has been no research into cache optimization for soft GPGPUs, taking into consideration area and performance. This section extends our soft GPGPU work, FlexGrip [10] [11], to perform a comparative analysis of different cache hierarchies evaluating the trade-offs between performance and area. More explicitly, the following will be assessed:

- **L1 Performance Analysis**: Each SMP contains individual L1 cache for data, local memory and texture memory. As a warp executes, it will request data for each of the individual threads. As multiple warps are issued, intra-warp contention can occur as data is swapped in and out of cache by warps who share memory. This is
exacerbated when multiple SMPs are requesting data from the lower memory hierarchy. To evaluate the impacts, we vary the number of SMPs versus the cache size, number of sets, and associativity of the cache.

- **L2 Performance Analysis**: As the L2 cache is shared among the SMPs, we vary the number of SMPs and evaluate the performance effects. In addition, we vary the L2 cache size, number of sets, and associativity.

Experiments are performed using GPGPU-Sim [48], a cycle-accurate simulator based on NVIDIA’s microarchitecture. Our approach modified GPGPU-Sim to emulate the architecture and performance of FlexGrip. This facilitated the ability to rapidly analyze different cache designs and hierarchies.

### 5.3 Modifying GPGPU-Sim for FlexGrip

The baseline architecture used to implement and test our cache configurations is a modified version of GPGPU-Sim that is representative of the FlexGrip microarchitecture. Figure 14 illustrates the top level GPU architecture modeled by GPGPU-Sim. The GPGPU-Sim architecture is comprised of multiple Single Instruction Multiple Thread (SIMT) core clusters, each consisting of multiple SIMT cores connected via an on-chip interconnection network that interfaces to the off-chip memory subsystem. A SIMT core is roughly equivalent to FlexGrip’s Streaming Multiprocessor as shown in Figure 4. For the purposes of our experiments, we limit testing to a single SIMT core cluster and vary the number of SIMT cores.
GPGPU-Sim provides support for four independent clock domains: one for the SIMT Core Cluster, one for the interconnection network, one for the L2 cache and one for the DRAM. Currently, FlexGrip supports a global system clock domain for each of the SMPs and a separate clock domain for the SPs. While the SPs have the ability to be clocked at a higher rate, it currently runs at the system clock rate of 100 MHz. We configure GPGPU-Sim to run the SIMT core, interconnect, and L2 clocks at 100 MHz to reflect the same clock speed configured in FlexGrip. In addition to providing a separate clock domain for DRAM, GPGPU-Sim also provides timing parameters to accurately model the DRAM memory. The ML605 development board, which was used to implement FlexGrip, is configured with 512 MB of DDR3 SODIMM memory which was run at a clock frequency of 400MHz from the Virtex-6 XC6VLX240T-1FFG1156 FPGA’s memory interface [49] as shown in Figure 3. We use this clock rate and the timing parameters per the specifications for the ML605’s Micron Technology MT4JSF6464HY-1G1 DDR3 DRAM as input into GPGPU-Sim to accurately model the memory interactions. For the purposes of our experiments, we did not modify the
interconnection network and utilize the default simulation configuration within GPGPU-Sim.

5.3.1 SIMT Core (SMP)

Figure 15 illustrates details of the GPGPU-Sim SIMT core architecture, which is roughly analogous to the five stage pipeline of FlexGrip as shown in Figure 4. The labels provided above each of the stages depict the analogous FlexGrip pipeline stages.

---

5.3.1.1 Fetch and Decode Stage

Recall that FlexGrip stores instructions in system memory (Figure 4), which is implemented as dual port block RAM in the FPGA, and can be thought of as a large cache. Instructions are fetched and decoded when a ready warp is issued to the pipeline by the Warp Unit. As all the instructions are loaded into block RAM, there are no cache
misses and instructions can be read from the block RAM every clock cycle. Once an instruction is fetched, it can be decoded and stored in the pipeline registers, which occurs in a single clock cycle. Assuming no pipeline stalls, fetching and decoding can occur in three clock cycles for short (32-bit) instructions or six clock cycles for long (64-bit) instructions.

As shown in Figure 15, GPGPU-Sim uses an instruction buffer (I-Buffer) to buffer instruction data once it has been fetched from cache, with each warp containing two entries. Each entry in the buffer contains a single decoded instruction along with a valid and ready bit. The valid bit indicates a non-issued decoded instruction while the ready bit indicates a decoded instruction is ready to be issued into the pipeline. At a conceptual level, the scoreboard logic sets the ready bit depending on the availability of hardware logic, although, GPGPU-Sim performs a readiness check rather than physically setting the ready bit. An instruction fetch will occur if a warp does not have any valid instructions in the I-Buffer, in which case a read request will be sent to the I-Cache with the PC of the currently scheduled warp. Two consecutive instructions are fetched from the cache by default.

The instruction cache in GPGPU-Sim is represented as a read-only set associative cache with the ability to simulate both FIFO and LRU replacement with on-miss or on-fill allocation policies. Requests to the instruction cache result in either a hit, miss or reservation fail which occurs when either the miss status holding register (MSHR) is full or there are no replaceable blocks in the cache set that exist. A cache miss will result in a
read request from memory, causing the warp to send an additional read request to the cache. A cache hit sends the instruction to the decode stage, whereby the instruction is decoded and stored in the buffer awaiting to be scheduled into the pipeline.

Matching the performance of FlexGrip for the fetch and decode stage in GPGPU-Sim is largely a function of the instruction cache effectiveness. Instruction data exhibit high spatial and temporal locality due to warps being issued in a round robin fashion. Therefore, we did not modify the I-Cache as the miss rate was insignificant compared to the FlexGrip model. In addition, the decode stage was unmodified as the I-Buffer mimics FlexGrip’s pipeline register, albeit holding several ready decoded instructions instead of one.

5.3.1.2 Warp Unit

The instruction issue in GPGPU-Sim utilizes a round-robin scheduler to issue warps into the pipeline assuming that the warp is not waiting at a barrier, it has valid instructions in the I-Buffer (i.e.: the valid bit is set), the scoreboard passes its check, and the pipeline is not stalled. The scheduler will then issue a warp to either the memory or ALU pipeline, which consists of scalar and floating point unit pipelines. With the exception of scoreboard and individual operation pipelines, issuing warps into a pipeline works in a similar fashion as described in Section 3.2. The scoreboard checks for read after write (RAW) and write after write (WAW) dependency hazards by tracking which registers have pending writes, or more succinctly, instructions that have been issued but have not yet written their results back. FlexGrip is architected to avoid RAW and WAW
dependency hazards due to how warps and instructions are issued through the pipeline and the induction of pipeline stalls. With respect to the individual pipelines for memory and ALU operations, we modified GPGPU-Sim to issue memory and ALU instructions serially. This is done by checking each memory and ALU pipeline to determine if a warp exists. If an active warp with either a memory or ALU operation exists in the pipeline, our modified version of GPGPU-Sim will stall the current warp waiting to be issues. Once the warp in the pipeline has moved to the next stage, the scheduler will issue the stalled warp.

Both GPGPU-Sim and FlexGrip handle the execution of branch divergence using a per-warp stack, or SIMT stack as shown in Figure 15. A detailed description of FlexGrip’s branch divergence and stack model is explained in Section 4.1.1. Conceptually, both GPGPU-Sim and FlexGrip implement branch divergence in a similar fashion with the exception of its location within the execution pipeline, therefore, we did not perform any modifications to this area of the architecture.

5.3.1.3 Execute Stage

The ALU pipeline in GPGPU-Sim models scalar processor (SP) functional units, which handle all ALU instructions except transcendental functions, and special function units (SFU) which execute transcendental instructions (i.e.: Sine, Cosine, Log, etc.). The ALU functional units are pipelined and operate in a SIMD fashion allowing for execution of one warp instruction per clock cycle for SP units and two or more for SFU units, depending on the type of instruction.
The execute stage in FlexGrip consists of multiple scalar processors which support the domain of integer instructions only, the details of which are outlined in Section 3.2. While the SPs are highly pipelined, they do not operate in a SIMD fashion and each instruction has a variable number of execution latency cycles. GPGPU-Sim provides a configuration file that allows for the adjustment of execution latencies based on the instruction type (e.g.: add, multiply, max, etc.), thus the performance of each of the SPs can be accurately duplicated. We configured the integer, floating point and double instruction latencies and initiation intervals to match FlexGrip.

5.3.1.4 Read and Write Stage

Control of read and writes within GPGPU-Sim are performed through a structure called the operand collector, as described through various NVIDIA patents. The operand collector, as depicted in Figure 16, is architected as an arbitrator along with multiple banks of on-chip single port RAM and register files, denoted as collector units. Once an instruction is received from the decode stage, it is allocated to an available collector unit in addition to setting the warp ID, operands, register identifier, ready and valid bits. Performed in parallel, read requests are queued by the arbitrator for each of the register file banks until access is granted. Once the data has been read, the arbitrator selects up to four non-conflicting collector unit accesses from the queue to send to the register file banks. For each clock cycle, an operand is read out from each of the banks, storing it in the corresponding collector unit and setting the ready bit. When all the ready bits are set in the collector unit for a particular instruction, it is issued to the execution stage. Data
from the execution stage is sent to the arbitrator and is queued until it is selected for write back.

Figure 16. GPGPU-Sim operand collector microarchitecture [48].

In contrast to GPGPU-Sim, the FlexGrip architecture partitions the read and write into separate stages, as shown in Figure 17. Within the read stage there are three operand units which calculate the source address, depending upon the type [50] of instruction. Collectively, the three operand units can be seen as a single collector unit. The read requests are queued in the operand unit until there is an available memory controller to service the corresponding type of memory or register access. The arbitrator will grant requests to one or more non-conflicting memory controllers and store the resulting data in a register to be sent to the execution stage. Data from the execute stage are stored in registers and queued until the arbitrator grants access for write back.
We leverage the existing GPGPU-Sim implementation and made architectural adjustments to match the functional capability of FlexGrip. Each of the register file banks and collector units can be represented as a stage in the FlexGrip pipeline, as shown in Figure 17. Therefore, we configured GPGPU-Sim to use only a single collector unit, emulating FlexGrip. The arbitration unit in GPGPU-Sim will select non-conflicting read accesses from the register bank from each of the decoded warp instructions in the
collector units. This is in contrast to FlexGrip which will only service the current warp, causing a stall to the pipeline until the read request has been serviced. The collector unit was modified to only service a single warp, stalling other warps that could be issued.

5.4 Memory Hierarchy

The following sections form the basis of our experiments with the prior sections describing the foundation for us to build upon. Before delving into the cache specifics, we will first describe the various memory spaces within the GPGPU as exposed by NVIDIA’s CUDA. The architecture described is within the context of how it would be physically implemented in FlexGrip and its relation to GPGPU-Sim. The following sections will describe the operational concept behind the level-1 and level-2 data caches along with the details of how it would be implemented within the FlexGrip architecture. For each of the level-1 and level-2 caches, we provide a description of the GPGPU-Sim model that is used to perform the experiments.

In Section 3.1, an overview of the FlexGrip architecture was provided outlining the interaction between the GPGPU and the MicroBlaze processor. Data was stored in DRAM and accessed by the processor which would stream it to the on-chip block memory for storage and processing by the GPGPU. The memory hierarchy consisted of global, system and constant memory that was shared by each of the SMs, and a private, per-SM shared memory space.
Injecting cache into the FlexGrip architecture requires us to augment the architecture as shown in Figure 18. While we previously had global, system and constant memory located on-chip, we now migrate those memory spaces to direct access off-chip DRAM. Each streaming multiprocessor still consists of private shared memory, however, we also add data, texture and constant cache to encompass our level-1 cache structure, each of which is backed by L2 cache. The following sections describe the L1 and L2 data caches in detail.

Figure 18. Representative FlexGrip block diagram exhibiting the details of the GPPGU memory hierarchy.

5.4.1 SMP Memory and Level-1 Data Cache

The Level-1 cache is a private, per SMP, non-blocking cache for both local and global memory accesses. A high-level block diagram depicting the memory hierarchy is shown in Figure 19. The size of the cache, along with the associativity, number of sets and cache line size are configuration parameters within GPGPU-Sim. We note that memory accesses that are generated by the address generation unit does not span two or more cache lines, therefore, individual requests are made for each cache line. A memory
access hit in the cache can be serviced in one clock cycle while a miss will be inserted into a FIFO miss queue. If the interconnection injection buffers for the DRAM are able to accept data, on each clock cycle, a fill request is generated by the L1 data cache. Upon a memory access miss, an entry is inserted into the Miss Status Holding Registers (MSHR) to track the status of cache misses in flight and a fill request is generated, pending there is currently no request for that cache line. The MSHR is configured as a fully-associative array with a fixed number of entries in the table, with each entry being able to service a fixed number of miss requests per cache line. If a request to access a memory location is currently in-flight, the request will be combined within the MSHR table. Once the fill response is received, the data is inserted into the cache line and the MSHR is marked as filled. The fill responses for MSHR entries are generated at one request per cycle. Upon servicing and responding to all waiting requests, the MSHR entry is freed.

Figure 19. L1 data cache and supporting memory components.
For our FlexGrip soft GPGPU, both the Local and Global memory are accessed and serviced by the L1 data cache, which is a private, non-blocking, per streaming multiprocessor cache. Figure 20 depicts a representation of integrating L1 cache into the FlexGrip architecture. On the FPGA, this would be implemented as true dual-port 36kB block RAM memories. The L1 data cache is not banked and is capable of servicing two coalesced memory request per clock cycle. As cache coherency poses significant challenges with GPUs [51], L1 data caches are not coherent. For global memory access, the L1 data cache follows a write-evict [50], write no-allocate policy while local memory cache acts as a write-back cache with write no-allocate policy. Both can be configured prior to run-time.

![Diagram of FlexGrip SMP](image)

Figure 20. FlexGrip SMP depicting the integration of L1 cache into the architecture.

5.4.2 Level-2 Data Cache

The Level-2 data cache, similar to the L1 data cache, is a unified last level cache (LLC) that is shared by all the SMs. For local memory access, the L2 cache write policy
exhibits a write-back, no-allocate policy while global memory access elects a write-evict, write no-allocate policy. As with the L1 data cache, a memory request cannot span across two cache lines, this ensures that requests from a lower level cache can be serviced by a higher cache.

Figure 21 below shows the components that service memory requests from the SMs and represents the model that will be used to simulate memory access within GPGPU-Sim. Memory requests from the interconnection network (ICNT) are entered into the ICNT->L2 queue. As configured, the L2 cache bank can service one request per clock cycle from the ICNT->L2 queue. If a miss occurs in the L2 cache bank, a request is made to the off-chip DRAM and entered into the L2->DRAM queue. Data that returned from the off-chip DRAM is then entered into the DRAM->L2 queue and placed in the L2 cache. For read requests, data is sent through the L2-ICNT queue and returned to the SM.

Figure 21. L2 data cache memory partition.
In order to model DRAM latency, a DRAM latency queue is used whereby the request access is held for a fixed number of clock cycles. The number of clock cycles is configurable and depends on the hardware being benchmarked. Each DRAM clock cycle, a memory access from the latency queue can be serviced and can push the results to the DRAM->L2 queue. It should be noted that ICNT->L2 queue operate at the L2 clock domain frequency while the L2->ICNT queue operates on the interconnect network domain frequency.

Integrating the memory partition within FlexGrip and on the FPGA, the L2 cache bank would be represented as a dual-port block RAM with read and write port 0 (r0 and w0) operating at the interconnect frequency and read and write port 1 (r1 and w1) operating at the L2 clock frequency. The L2->ICNT queue and the ICNT->L2 queue would represent single-port block RAMs with the L2->ICNT block RAM operating at the interconnect frequency and the ICNT->L2 operating at the L2 clock frequency. The DRAM->L2 queue and L2-> DRAM queue would be modeled as dual-port block RAMs to allow for each port to operate on the two different clock domains. The DRAM-> L2 queue read port would operate at the interconnect frequency, matching that of the L2->ICNT queue, while the write port would operate at the L2 frequency. The DRAM access scheduler would represent a combination of control logic and a memory controller configured by Xilinx’s Memory Interface Generator (MIG) tool. Both the DRAM latency queue and timing model are used for simulation purposes only and therefore would not be part of the implementation.
5.5 Experimental Methodology

In this section we describe the components of our infrastructure for use in evaluating our architectural decisions. Specifically, we describe our baseline configuration, which encompasses the representative hardware platform used as a model for the system, in addition to the benchmarks that are used for the experiments.

5.5.1 Baseline Configuration

As described, we augmented GPGPU-sim to create a representative architecture of FlexGrip implemented on a Xilinx ML605 development board. The ML605 hardware platform has a single Virtex-6 XC6VLX240T FPGA which contains 14,976 Kb of Block RAM. The FPGA is connected to a single Micron MT4JSf6464HY-1G1 512MB 8-channel DDR3 SODIMM memory module. The features of the ML605 are used as input into the configuration of GPGPU-Sim as described in Table 11. In our baseline configuration, there is no L1D or L2D cache, and therefore it is disabled in GPGPU-Sim. In addition, we turned off memory coalescing and shared memory bank conflict resolution, as both of these features are not implemented in FlexGrip.

Table 11. Default GPGPU-Sim Configuration

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core clock frequency</td>
<td>100MHz</td>
</tr>
<tr>
<td>Interconnection network</td>
<td>Mesh</td>
</tr>
<tr>
<td>Number of SMPs</td>
<td>1 – 2</td>
</tr>
<tr>
<td>Number of SPs per SMP</td>
<td>32</td>
</tr>
<tr>
<td>Total on-chip memory size</td>
<td>14,976KB</td>
</tr>
<tr>
<td>Number of 32-bit registers per SMP</td>
<td>32,768</td>
</tr>
<tr>
<td>Shared memory size per SMP</td>
<td>16KB</td>
</tr>
<tr>
<td>DRAM size</td>
<td>512MB</td>
</tr>
</tbody>
</table>
5.5.2 Benchmarks

To analyze how enabling different cache levels affects the performance quantitatively, we conduct simulations using several GPGPU applications with varying architectural parameters. The representative GPGPU application encompass synthetic benchmarks from the NVIDIA CUDA SDK [52] in addition to the Rodinia [53] benchmark suite. The benchmarks selected are shown in Table 12 and represent a variety of memory behaviors. The simulations are performed using cycle-accurate GPGPU-Sim augmented to mimic the performance of FlexGrip. The following sections describe the details of our experiments.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>NVIDIA CUDA SDK</strong></td>
</tr>
<tr>
<td>AC</td>
<td>Autocorrelation</td>
</tr>
<tr>
<td>BS</td>
<td>Bitonic Sort</td>
</tr>
<tr>
<td>FWT</td>
<td>Fast Walsh Transform</td>
</tr>
<tr>
<td>MM</td>
<td>Matrix Multiply</td>
</tr>
<tr>
<td>RD</td>
<td>Reduction</td>
</tr>
<tr>
<td>TP</td>
<td>Transpose</td>
</tr>
<tr>
<td></td>
<td><strong>Rodinia</strong></td>
</tr>
<tr>
<td>BFS</td>
<td>Breadth First Search</td>
</tr>
<tr>
<td>KMN</td>
<td>K-Means Clustering</td>
</tr>
<tr>
<td>GE</td>
<td>Gaussian Elimination</td>
</tr>
<tr>
<td>LUD</td>
<td>LU Decomposition</td>
</tr>
<tr>
<td>SRAD</td>
<td>Speckle Reducing Anisotropic Diffusion</td>
</tr>
</tbody>
</table>

5.6 Cache Configuration Trade-Offs

This section describes the exploration of trade-offs for different data cache configurations. We would like to answer the question, given a fixed amount of memory, what is the optimal cache configuration. As part of our experiments, we will vary the L1
data cache size from 1KB to 64KB and the L2D cache size from 24KB to 164KB. With respect to memory, recall that our environment is modeled after hardware from the Xilinx ML605 development board. The Virtex 6 FPGA used on the ML605 board contains a total of 416 36kB block RAMS (14,976 Kb of on-chip memory), of which, 156 block RAMs are used for the register file, shared memory per each SMP, and various other components for each of the SMPs. Therefore, as we increase the number of SMPs, the amount of on-chip memory that can be used for cache decreases, providing us with a constrained memory size. In the current FlexGrip configuration, we only have enough resources to support 2 SMPs, resulting in approximately 9,360 Kb for 1 SMP and 3,744 Kb of on-chip memory for 2 SMs. Even taking in consideration block RAM used by the memory controller and additional logic, there is more than enough memory to implement even the largest cache configurations. Of final note, we only explore a cache line size of 128 bytes, which is the amount of data required to service requests from a warp.

5.6.1 L1D Cache Performance

Thrashing in L1 cache can be caused by intra- or inter-warp contention [54], [55]. As the number of active concurrent threads increase, the effective cache size per thread decreases. Warps within an SM share L1 cache space which can lead to inter-warp contention as data is continually replaced. A secondary effect is an increase in memory traffic due to over-fetching of data not used by other threads caused by low temporal and special locality in L1D cache, as described earlier. We evaluate these effects by varying the number of SMs versus the cache size, number of sets, and associativity for each of the benchmarks listed in Table 12. We note that while separate cache is also used for texture,
constant and instruction cache, our focus here is only on data cache. Texture cache is only used for certain graphical applications. Constant cache is used to store read only configuration information and typically stores small amounts of data, thereby would not exhibit the same level of performance implications as with data cache. The following provides a performance comparison to the baseline 1 SM and 2 SM configurations along with an analysis of the results.

Figure 22. Execution time speedup relative to the 1 SM baseline system for various L1D cache configurations.
Figure 23. Execution time speedup relative to the 2 SM baseline system for various L1D cache configurations.

Figure 22 and Figure 23 compare the performance of various L1D cache configurations against the 1 SM and 2 SM baseline configuration, respectively. Recall that the baseline configuration has no L1D or L2D cache, therefore, data requests to global memory are stored in queues prior to entering the memory interconnect and being serviced by SDRAM. We notice there are several benchmarks where there are no significant performance benefits and varying cache configurations exhibit no changes. Specifically, bs, fwt, lud, and rd show constant speedups of 1.0, 1.04, 1.0, and 0.96 across all cache configurations, with lud exhibiting minor deviations. Compared to the performance without the cache in the baseline system, only four benchmarks (ac, mm, bfs, and ge) show performance improvements for the 1 SM configuration and six benchmarks showed improvement in the 2 SM configuration (ac, fwt, mm, tp, bfs and ge).
Before we delve into the specifics, it is important to understand how the baseline architecture with no cache compares to the same architecture with perfect memory, in other words, global memory access returns data immediately the next cycle. This will help us gain insight into how much speedup we can expect when adding cache into our baseline configuration. Figure 24 shows the performance increase of the architecture with perfect memory versus the baseline architecture with global memory and no cache enabled. We can see large performance increases when perfect memory is enabled for benchmarks ac, mm, tp, bfs and kmn. In fact, this directly correlates with the large variations of speedups shown in Figure 22 and in Figure 23. For example, Matrix Multiply (mm) shows a performance increase of 49% and 93% when enabling perfect memory for 1 SM and 2 SM configurations, respectively. When showing the speedup of L1D cache versus the baseline architecture, as shown in Figure 22 and in Figure 23, we see a correlation, where the maximum speedup of 1.7 and 2.96 is represented for both 1 SM and 2 SM configurations, respectively. On the other hand, for small performance
increases, such as in the case for bs, fwt, rd, ge, lud, and srad (with the exception of ge), speedups of L1D versus the baseline architecture are around 1.0 and vary slightly across all cache configurations. The primary reason for the lack of speedup and variation is due to shared memory usage in the latter benchmarks. Recall that shared memory is fast cache that can be used to store shared data and is managed by the programmer, unlike L1 and L2 cache which is managed by the hardware. However, like the L1D cache, when used effectively, it can significantly reduce the amount of global traffic. Thus, there is minimal performance impact between perfect memory, the baseline architecture of modeled global memory with no cache, and when L1D cache is enabled. As we will discuss, the small impact in performance is due to the additional traffic caused by L1D global loads and shared memory stalls. However, when not using shared memory, L1D cache can a significant impact on performance, as seen in the case for both Autocorrelation (ac) and Matrix Multiply (mm).

There are several benchmarks that exhibit decreased performance when L1D cache is enabled, specifically benchmarks bs, fwt, rd, kmn, lud and srad. The decreases in performance are attributed to two factors, the first is from enabling L1D cache, which can significantly increase memory traffic. Figure 25 shows the normalized global memory traffic for bs, fwt, kmn, lud and srad which is normalized to that with no cache for both 1 SM and 2 SM configuration. We note that we left out reduction (rd), as the performance is tied to the effectiveness of shared memory. The results in Figure 25 show that global memory load traffic can be reduced by 43% when not enabling cache. When L1D is enabled, 128-byte memory transactions are performed versus 32-byte transactions when
not enabled, thus contributing to the additional memory access traffic when spatial and temporal data reuse is low. In fact, the majority of benchmarks in Figure 25 exhibit low temporal and spatial locality, lending to over-fetching of data and an increase in memory traffic.

As discussed, benchmarks that use shared memory exhibit only small variations in performance and typically show no performance benefits (speedups of around 1.0). In most cases, when not using shared memory and enabling L1D cache, there are potentials for large performance gains, as in the case of Autocorrelation and Matrix Multiply. However, in the case of \( k \)-Means Clustering (\( kmn \)), this is not the case, as enabling L1D cache significantly decreases performance. This leads to the second factor in decreased performance, which is due to pipeline stalls at the memory stage due to shared memory.

Figure 25. Normalized global memory traffic with and without L1D cache for 1 SM and 2 SM configurations, normalized to no cache.
bank conflicts and non-coalesced memory accesses. In the case of \textit{kmn}, almost all (i.e.: close to 100\%) of memory accesses were misses in the L1D cache. This not only causes a significant increase in traffic, but also causes additional stalls due to compulsory misses. In fact, the number of non-coalesced memory stalls was more than double that over the baseline architecture without L1D cache enabled, directly correlating to the decrease in performance. In the case of the reduction (\textit{rd}) benchmark that was referenced earlier, disabling the cache generated 3x more memory traffic than with L1D, yet the performance with L1D was only slightly worse at 0.96 speedup for 1 SM and 2 SM configurations. The reduction of memory traffic when L1D cache is enabled is due to the low cache miss rate of 3.3\%, exhibiting high data reuse amongst warps. However, this is offset by the fact that there was a 33\% increase in non-coalesced memory stalls versus the disabled cache configuration.

![Figure 26. Execution time speedup of Breadth First Search relative to 1 SMP and 2 SMP baseline systems for various L1D cache configurations.](image)

Figure 26. Execution time speedup of Breadth First Search relative to 1 SMP and 2 SMP baseline systems for various L1D cache configurations.
As illustrated in Figure 22 and Figure 23, Autocorrelation (ac), Matrix Multiply (mm), and to a lesser extent, Breadth First Search (bfs) benefit most from L1D cache. The bfs algorithm performs a search on a graph where each vertex is associated to a thread. The benchmark exhibits high levels of data reuse as arrays are used to determine if a node has been visited and to update cost information. As illustrated in Figure 26, we see that increases in speedup over the baseline occur for L1D cache sizes above 32KB, as smaller cache sizes lead to higher cache miss rates, and thus, data is continuously swapped in and out causing significant increase in memory traffic and global non-coalesced stalls. On average, memory traffic is 1.6x greater for cache sizes less than 32KB and similarly, global memory access stalls are 1.5x greater. It is of note that the L1D miss rate has a significant adverse effect on the 2 SM case where the amount of traffic generated causes excessive latency in the network, thus it is unable to efficiently service both SMs. In the perfect memory configuration, where global data is services in one clock cycle, speedup of the 2 SM system is almost 1.9x over the 1 SM system. In effect, this architecture works similar to having one SM being serviced while the other waits for data, operating in a serial fashion. Finally, from the graph, we see the largest performance gains occur when set sizes are equal to 64, specifically when our cache size if 64KB. This is due to our chosen input graph size of 65,535 nodes, which can take full advantage of the cache size, correlating to higher cache hit rates.
Figure 27. Execution time speedup of Autocorrelation relative to 1 SM and 2 SM baseline systems for various L1D cache configurations.

The results of the Autocorrelation benchmark are shown in Figure 27. We can see that for small cache sizes and associativity less than 2, speedup varies slightly, and in fact, the maximum speedup occurs with only 16KB of cache. The performance increase is attributed to two reasons; one is the high spatial and temporal data reuse, which is reflected in the data cache hit rate of almost 1.0. The second is due to the array size chosen of 4,096 integers, resulting in a maximum data storage size of 16KB. Thus, we expect performance increase would track with the size of the input up until where the size of the dataset is more than the size of the cache, where data miss rates would begin to cause performance impacts.

Figure 28 illustrates the results of the Matrix Multiply benchmark, using a matrix dimension size of \( N = 128 \), for both 1 SMP and 2 SMP configurations across a series of
data cache configurations. To better understand the performance, it is essential we describe the details of memory accesses for the Matrix Multiply benchmark.

![Matrix Multiply](image)

Figure 28. Execution time speedup of Matrix Multiply relative to 1 SM and 2 SM baseline systems for various L1D cache configurations.

The standard matrix multiplication calculation is performed by multiplying each row of matrix A with each column of matrix B, with each element of matrix C defined as:

\[ C_{i,j} = \sum_{k=0}^{n} A_{i,k} \times B_{k,j} \quad j \in [0, m], i \in [0, n] \tag{4} \]

In CUDA, a natural decomposition is to define matrix A of dimension \( M \times w \) and matrix B of dimension \( w \times N \), with \( w \) defined as the tile size which is set to 16. This value was chosen based on the block size of the architecture. A \( w \times w \) tile results in 256 threads, or 8 warps, which is the maximum number of warps in a thread block for our architecture (compute capability 2.0), ensuring maximum occupancy. The result is matrix C of dimension \( M \times N \), where \( M, N \) are both set to 128 for our benchmark.
Figure 29. CUDA kernel code for the Matrix Multiply benchmark.

```c
__global__ void MatrixMultiply(int *a, int *b, int *c, int N) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int sum = 0;
    for (int i = 0; i < N; i++) {
        sum += a[row*TILE_DIM+i] * b[i*N+col];
    }
    c[row*N+col] = sum;
}
```

In the CUDA kernel code listed in Figure 29, variables a, b, and c are pointers to global memory which reference matrices A, B, and C respectively. The values of blockDim.x, blockDim.y, and TILE_DIM are all equal to \( w \), which in our case, is set to 16. The `row` and `col` variables are calculated using the block and grid dimensions and reference the row and column elements used by a thread (threadIdx) to calculate the result, referenced in element C. As such, each thread within the \( w \times w \) (16 x 16) thread block calculates a single result in a tile of matrix C. The `for` loop iterates over the tile dimension, performing the matrix multiply calculation by multiplying the row of A by the column of B, finally storing the result in matrix C.

The analysis of performance follows with an illustration of how threads in warps access memory within the `for` loop of Figure 29. Consider the illustration in Figure 30, where threads in each warp calculate one row of a tile of C. Of particular note, the result
only depends on using a single row of $A$, but an entire tile of $B$, which effects how data is brought into the cache and utilized.

Figure 30. Calculation of a row of a tile in matrix $C$ using a single row in matrix $A$ and an entire row of tiles of matrix $B$.

As previously noted, only a single tile of $A$ is required to calculate a row of $C$. Threads from tile $A$ will read values from global memory, which will also be brought into L1D cache. As calculations proceed along a row, this data is constantly being re-used, making it unlikely that it will be evicted from cache. In the case of matrix $B$, threads in a warp require reading an entire row of tiles. To maximize performance, the entire row of tiles will need to remain in cache. Using a matrix width of 128 and a tile size of 16 ($2,048$ threads), a total of $8,192$ bytes of data need to be brought in from memory and stored in cache. The memory accesses from tile $A$ and tile $C$ are insignificant and contribute very little to the overall performance impact. As we can see from Figure 28, this correlates
with a significant increase in performance when using 8KB or more of L1D cache. Using
less cache requires data to be continuously swapped in and out of memory, degrading
performance. This also explains why there are performance degradations when not using
multiples of 8 KB as the entire row of tiles cannot fit into cache. For example, in the case
with 12 KB of cache, one entire row of tiles can be brought in, however, only half of the
second row, causing the other half of data to be continuously swapped in and out of
memory.

5.6.2 L2 Cache Performance

GPGPUs exhibit massive multithreading causing resource congestion which can
severely limit system performance and cache efficiency. As each SM is connected to the
L2 cache via the Network on Chip (NoC), requests can quickly saturate the bandwidth.
Furthermore, all requests from the SMs are now serviced by the L2 cache. For working
sets larger than the cache size, thrashing, or contention can cause inefficiency and
degrade system performance. As we are concerned with cache performance, we will not
perform an analysis on the NoC design. The following provides a performance
comparison and analysis of L2 cache designs versus the baseline 1 SM and 2 SM
configurations.
Figure 31. Execution time speedup relative to the 1 SM baseline system for various L2 cache configurations.

Figure 32. Execution time speedup relative to the 2 SM baseline system for various L2 cache configurations.
As we can see in Figure 31 and Figure 32, with the exception of the Reduction (\(rd\)) benchmark, enabling L2 cache either causes no performance benefits or seriously degrades performance, as in the case of Transpose (\(tp\)) and \(k\)-means (\(kmn\)). In the case of the Reduction benchmark, performance increase is negligible. Recall that that our architecture does not coalesce data transactions, thus memory transactions generated by threads in warps can quickly saturate the network. This is especially true in our case as the system employs only a single memory interface, whereby typical GPGPU architectures feature multiple memory interfaces and highly banked DDR RAM. By not using L2 cache, the additional latency to fetch data from RAM helps reduce contention on the network, decreasing the overall latency to service memory requests. Therefore, the reduced access time afforded by L2 cache is negated by the extra latency caused by additional traffic and contention on the network. In addition, as increased latency is directly correlated to increase miss rates, as additional time is needed to fetch data from main memory. Figure 33 shows the normalized network latency with L2 cache and without (baseline), normalized to the baseline architecture. With the exception of the Gaussian (\(ge\)) benchmark for the 2 SM case, every benchmark exhibits additional traffic latency over the baseline system. There are two interesting cases to explore further, Transpose and \(k\)-means benchmarks, each exhibiting significant degradation and variation in performance over the baseline system.
Figure 33. Normalized network latency with and without L2 cache for 1 SMP and 2 SMP configurations, normalized to no cache (baseline).

In Figure 34, we provide the CUDA kernel implementation of the Matrix Transpose algorithm, $C = A^T$.

```c
__global__ void MatrixTranspose(int* idata, int* odata, int width)
{
    int x = blockIdx.x * TILE_DIM + threadIdx.x;
    int y = blockIdx.y * TILE_DIM + threadIdx.y;

    for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS) {
        odata[x*width + (y+j)] = idata[(y+j)*width + x];
    }
}
```

Figure 34. CUDA kernel code for the Matrix Transpose benchmark.
We note that for our particular benchmark, we use a matrix of size 1024 x 1024. In the CUDA code provided in Figure 34, the variables `idata` and `odata` are pointers to global memory, which reference the input matrix `A` and output matrix `C`, respectively. The value of `TILE_DIM` and `BLOCK_ROWS` are set to 32 and 8, respectively. These values were chosen to optimally fit into a thread block. As previously noted, our architecture is based on a maximum thread block size of 8 warps, or 256 threads. The `x` and `y` variables are the row and column elements used by a particular thread to copy values from the row of `idata` to the column of `odata`, with the `for` loop iterating over the tile dimension. In our example we note that our 1024 x 1024 matrix has a stride of 1024 elements or 4,096 bytes between contiguous threads.

![Figure 35. Transpose from A to C, whereby an entire row of tiles of matrix A are used to output the transposed column elements in C.](image)

The analysis of performance follows with an illustration of how threads in warps access memory within the `for` loop of Figure 34. Consider the illustration in Figure 35, where threads in each warp access an entire tile of `A`. To transpose a row from matrix `A` to
a column in C, threads in a warp read an entire row of A tiles which is brought into L2 cache. A matrix width of 1024 and a tile size of 32 equates to 32,768 threads, or 131,072 bytes of data that need to be brought in from memory. As we can see from Figure 31 and Figure 32, this correlates with a significant increase in performance when using cache sizes of 128KB or more versus when using smaller cache sizes. Using less cache requires data to be continuously swapped in and out of memory, degrading performance.

The second benchmark we explore is the $k$-means algorithm, a popular clustering algorithm used in a variety of fields such as statistical analysis, pattern recognition, image analysis and bioinformatics. The primary goal of clustering is to group data points into sets such that each set share similar characteristics. The algorithm can be described as the following clustering problem: given a set of points \{$P_1, \ldots, P_n$\} with each point a vector of size $d$, the goal is to partition $n$ points into $k$ clusters \{$S_1, \ldots, S_k$\} ($k < n$) with centroids \{$C_1, \ldots, C_k$\} such as to minimize the sum of squares of distances within the clusters. This can be described as follows:

$$\arg \min_{S} \sum_{i=1}^{k} \sum_{P_j \in S_i} \|P_j - C_i\|^2$$  \hspace{1cm} (5)

Our analysis begins with the following CUDA kernel code of the $k$-means algorithm shown in Figure 36. We note that we did not provide the full kernel code, only the relevant pieces for our analysis.
Figure 36. A fraction of the CUDA kernel code for the k-means benchmark.

Abstracting the details of the kernel code, our primary focus is global memory transactions. To put the kernel code into perspective, we note that for our benchmark we provided the following input values: nclusters = 5, nfeatures = 34, and npoints = 65,536. Within the two for loops, highlighted between lines 16 through 33, there are two global memory loads, one for features[addr] and one for...
c_clusters[cluster_base_index + j]. Our goal is to understand memory usage and the level of data reuse for each thread that executes the kernel code. We first analyze the memory load in line 23, c_clusters[cluster_base_index + j]. The address for c_clusters is calculated using the cluster_base_address and the index, j, inside the for loop in line 20. Line 17 calculates the cluster_base_index, which, when iterated over the outer for loop and multiplied by the index value, results in the following set of values: 0, 34, 68, 102, and 136. Each of these values are added to the inner loop iteration index value, j, inside the second for loop (line 20), which is used to calculate the set of addresses {0, 1, 2, 3, …, 168, 169}. Thus with 4 bytes of data per request and 170 data requests, a total of 680 bytes is loaded from memory into the c_clusters array. Due to the linear addressing, the data can be loaded into 6 L2 cache lines. The data exhibits high temporal locality since every thread is requesting the exact same data in parallel, making it highly unlikely the data being evicted from cache.

Our analysis continues with the loading of features data into an array as shown on line 24. In this instance, the address (addr) is calculated (line 22) using the point_id, calculated on line 8, which is essentially the thread index. Each thread will iterate over the inner for loop (line 20) and calculate nfeatures (34) of distinct features addresses (line 22) from memory, as shown on line 24, for a total of 136 bytes requested. In our architecture, a maximum of three thread blocks (8 warps per thread block) with a total of 768 threads can be scheduled on a single SM. Therefore, a total of 102KB of data is requested for each iteration of the outer for loop in line 16. If
we assume a large enough L2 cache to store the data, each thread will exhibit a 100% cache hit rate on each subsequent iteration starting on line 16. As a note, we ignore the small amount data needed to store the `c_clusters` array. However, if the L2 cache size is smaller, such that not all data can be brought into cache, serious performance degradation will occur as data is constantly being swapped in and out. We illustrate this through a simple example.

Figure 37. A simple example of two threads executing a memory load into the features array for the k-means benchmark.

Figure 37 shows an example of two threads executing line 24 of the k-means kernel code. For brevity, the features array variable is denoted as `f` in the example. In step 1, the first iteration of the outer `for` loop (line 16) and inner `for` loop (line 20) is executed by thread 0, which requests data from memory with address `[T0 + (j*npoints)]`, where `T0` is the thread index value and `j` is equal to 0. As no data exists in cache, a compulsory cache miss occurs, resulting in a request from main
memory to populate the L2 cache location with the serviced data. In the same step, thread 1 performs a request with address \([T1 + (j*npoints)]\). Similarly, a compulsory miss occurs causing a request from main memory, which is serviced and written to L2 cache. Step 2 increments index \(j\), with each thread making a data request from memory, populating the remaining locations in L2 cache. In step 3, thread 0 requests address \([T0 + (1*npoints)]\) from cache, which results in a cache miss. The least recently used (LRU) replacement policy evicts the first cache location and replaces it with the appropriate requested data from memory. Thread 1 then requests address \([T1 + (1*npoints)]\), which also results in a cache miss, requiring data to be evicted from cache, and filled with data requested from memory. As with previous steps, step 4 starts with thread 0 requesting data from address \([T0 + (0*npoints)]\), however, this data was previously brought into cache in step 1, but was evicted in the prior step. This also occurs when thread 1 requests \([T1 + (0*npoints)]\). In fact, this will continue to occur over \(i\) number of iterations, resulting in 100% cache miss rate, despite high temporal data reuse.

Based on the analysis, we see a direct correlation in Figure 31 when cache sizes are greater than or equal to 96KB, translating to cache miss rates of 21.7%, 3.4% and 3.1% for cache sizes of 96KB, 128KB and 160KB, respectively for the 1 SM case. In the 2 SM case, slight improvement occurs as cache size increases, however, the cache size would need to be at least 192KB to exhibit the jump in performance shown in the 1 SM case. However, as with other benchmarks, even with very small cache miss rates, network traffic limits potential speedup factors.
5.7 Summary

In this chapter, we investigated L1 and L2 cache configurations for soft GPGPUs by modifying GPGPU-Sim, a cycle accurate GPGPU simulator, to emulate the functionality of FlexGrip. It was shown that the baseline system with no cache performs better than the system with L1 cache for 7 out of the 11 benchmarks in the 1 SM case and five out of the eleven benchmarks in the 2 SM case, with only autocorrelation and matrix multiply showing significant improvement. The decrease in performance using L1D cache led to two contributing factors: global network traffic and pipeline stalls at the memory stage. When L1D cache is enabled, 128-byte transactions are used versus 32-byte transactions used in our baseline configuration without cache. If applications exhibit poor temporal or spatial data reuse, over-fetching could occur, leading to increased network traffic. We showed that global memory load traffic increases 74% when enabling L1D cache. While misses in L1D cache can cause a significant increase in traffic, it also causes additional stalls due to compulsory misses, resulting in non-coalesced memory stalls, directly correlating to a decrease in performance.

We concluded the section with an analysis of L2 cache performance versus the baseline system of no cache over varying cache configurations. Over the 11 benchmarks, only Reduction showed improved performance when enabling L2 cache. We showed that with L2 cache enabled, network latency increased on average 23% and 28% over the baseline system for 1 SM and 2 SM configurations, respectively. When benchmarks
exhibit good data reuse, fast service times from L2 cache can quickly saturate the network causing significant latency.

While we showed certain benchmarks benefited from cache, many benchmarks showed no performance gains and in some cases, significant decreases in performance. Overall, the results and analysis showed the network as a key factor in determining the performance. Therefore, we conclude that future research and resources should focus on improving the underlying network and memory infrastructure, with cache considered on an individual basis.
CHAPTER 6

CONCLUSION, FUTURE CONSIDERATIONS AND PUBLISHED WORK

6.1 Conclusion

This dissertation has outlined the implementation and architectural evaluation of a soft-GPGPU on an FPGA. We described in detail the scalable architecture which was implemented and tested on the Xilinx ML605 development board. The FlexGrip architecture utilized features of the FPGA to allow for different implementations to target certain classes of application depending on their execution characteristics. Experimental results against five benchmarks showed speedups of up to 30x versus a MicroBlaze soft processor for a single streaming multiprocessor and 55x for two streaming multiprocessors for highly parallel benchmarks. When implementing architectural optimization, we found we can reduce the dynamic energy consumption by 14% and LUT area by 33% on average.

We concluded with an evaluation of the performance of cache designs within a GPGPU by varying key parameters. The uniqueness of this approach lies within the context of the FPGA design, understanding the application and design space parameters that would optimize the performance of the GPGPU. To that extent, we modified GPGPU-Sim, a cycle accurate GPGPU simulator, to match FlexGrip’s functionality. When evaluating 11 benchmarks against designs with varying configurations of L1 cache or L2 cache enabled versus a baseline system with no cache, we found that 64% and 45% of benchmarks exhibited performance decreases when L1D cache was enabled for the 1 SM and 2 SM configurations, and only one benchmark showed performance
improvement when the L2 cache was enabled. Our analysis concluded that improving network throughput would provide significant benefits over using resources to implement cache memory in the design.

### 6.2 Future Considerations

To maximize effectiveness of cache, future considerations should include researching and implementing memory coalescing by grouping memory requests into a minimal set of transactions to reduce network traffic. An extension of that effort should also include shared memory banks and memory bank resolution. The existing baseline FlexGrip design does not incorporate banked memory; therefore, reads and writes are performed serially. In order to take advantage of banked memory, a method to resolve bank addresses must be included such that a single memory transaction can service multiple banks.

In the previous section, we discussed limitations of the network and memory structure leading to contention and latency. Future work should include optimizing the network and memory structure to maximize bandwidth and minimize latency. The design space should take into consideration taking advantage of FPGA resources such as those included on System on Chip (SoC) devices.

Another design space area to consider is the ability to replace or add custom processors. For example, executing an image processing algorithm, such as Local Area Contrast Enhancement (LACE), requires threads to iterate through the pipeline multiple
times for each instruction. By implementing special purpose processors, a single processor would execute the LACE algorithm, minimizing the number of memory transactions and time required to schedule and service warps.

As a final area to consider, recall that FlexGrip schedules warps in a round robin fashion and will stall during memory transactions. Techniques should be researched on issuing warps into the pipeline that can proceed to execute, hiding latencies of other warps as they wait for long memory transactions.

6.3 Published Work

The following section lists our published work.


BIBLIOGRAPHY


