Improving CGRA Utilization by Enabling Multi-threading for Power-efficient Embedded Systems

by

Jared Pager

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved November 2011 by the
Graduate Supervisory Committee:

Aviral Shrivastava, Chair
Sandeep Gupta
Gil Speyer

ARIZONA STATE UNIVERSITY

December 2011

ABSTRACT

Performance improvements have largely followed Moore's Law due to the help from technology scaling. In order to continue improving performance, power-efficiency must be reduced. Better technology has improved power-efficiency, but this has a limit. Multi-core architectures have been shown to be an additional aid to this crusade of increased power-efficiency. Accelerators are growing in popularity as the next means of achieving power-efficient performance.

Accelerators such as Intel SSE are ideal, but prove difficult to program. FPGAs, on the other hand, are less efficient due to their fine-grained reconfigurability. A middle ground is found in CGRAs, which are highly power-efficient, but largely programmable accelerators. Power-efficiencies of 100s of GOPs/W have been estimated, more than 2 orders of magnitude greater than current processors.

Currently, CGRAs are limited in their applicability due to their ability to only accelerate a single thread at a time. This limitation becomes especially apparent as multi-core/multi-threaded processors have moved into the mainstream. This limitation is removed by enabling multi-threading on CGRAs through a software-oriented approach. The key capability in this solution is enabling quick run-time transformation of schedules to execute on targeted portions of the CGRA. This allows the CGRA to be shared among multiple threads simultaneously.

Analysis shows that enabling multi-threading has very small costs but provides very large benefits (less than 1% single-threaded performance loss but nearly 300% CGRA throughput increase). By increasing dynamism of CGRA scheduling, system performance is shown to increase overall system performance of an optimized system by almost 350% over that of a single-threaded CGRA and nearly 20x faster than the same system with no CGRA in a highly threaded environment.

DEDICATION

"I believe God made me for a purpose, but he also made me fast.

And when I run I feel His pleasure."

-Chariots of Fire

To those who were more convinced of my abilities than I was myself.

To those who told me to run faster because they believed.

Specifically, I am thinking of my dear mother and father.

ACKNOWLEDGEMENTS

Those who knew me as an undergraduate student knew how I was used to approaching my work. They knew that I did a lot, accomplished a lot, but did so by accepting things as 'good enough.' That all changed as I began working with Prof. Aviral Shrivastava. Suddenly, 'good enough' became 'perfect' and, somehow, it became attainable. He always knew when I could do better and inspired that extra bit in me. Without his continual guidance, this work would never have been completed.

I have had opportunity to learn experience from many great people. I appreciate every member of Compiler Microarchitecture Lab (CML), Abhishek Rhisheekesan, Bryce Holton, Chuan Huang, Di Lu, Jian Cai, Jing Lu, Ke Bai, Mahdi Hamzeh, Tushar Rawat, Yooseong Kim, and others for their support and making such a wonderful and enjoyable environment. It is in this lab that I was able to interact with some of the most brilliant people I have known in my life. Much special thanks to Reiley Jeyapaul who spent more time than I'm sure he wanted in order to guide and mentor me through the whole process. This work could have been that much better if only he could have written it himself instead of me.

The process of researching and writing this has been one of the most rewarding/difficult/stressful/complex undertakings I have ever done. Somehow I bounced between thinking I would never complete this work, that this work was going to drain every bit of energy I had left, to finally seeing the completion of this work. During this whole process, I have been grateful for my adult mentors growing up that taught me how to obtain balance in my life. Probably the most influential person I can remember is Dwayne Farnsworth, who taught me how to relax and enjoy the moment, even when it feels like you have just made a dumb mistake like diving head first into shallow water.

TABLE OF CONTENTS

Page

LIST OF TABLES viii

LIST OF FIGURES ix

CHAPTER

1 INTRODUCTION 1

2 BACKGROUND AND TERMINOLOGY 5

2.1 Kernel Mapping is Complex 6

2.2 Mapping Terminology 8

3 MOTIVATION 10

3.1 Compiler Increases CGRA Applicability and Usability. 10

3.2 Multi-threading Provides Better CGRA Utilization. 10

4 RELATED WORK 12

4.1 Compiler Framework 12

4.2 Multi-threading 12

Polymorphic Pipeline Arrays 13

MT-ADRES 13

5 CONTRIBUTIONS OF THIS WORK 14

5.1 Compiler Framework 14

5.2 Multi-threading Framework 14

6 CGRA COMPILER FRAMEWORK 15

6.1 Compiler Operation 15

Identification 15

CHAPTER Page

Transformation 16

Compilation 16

Data Communication 17

6.2 Implementation Details 17

7 MULTI-THREADING FRAMEWORK 20

7.1 Overview of Framework 21

7.2 CGRA Hardware Specification 21

Connection Topologies 21

Register Requirements 22

7.3 Compilation Constraints 22

Data Flow Constraints 22

Register Constraints 23

7.4 Problem Definition 23

7.5 Fast Runtime Transformation 24

Transforming a Schedule 25

Initialization Stage 25

Filling Remaining Schedule 26

Page Mirroring 27

7.6 Example Transforms 28

Example One: Mapping to Transform 28

Example Two: Advanced Transform 30

7.7 Implementation Details 31

CHAPTER Page

8 ANALYSIS AND RESULTS 33

8.1 Compiler Framework 33

8.2 Multi-threading Framework 33

Small Cost in Single-threaded Performance 33

Multi-threading Increases Utilization and Benefits Performance . 34

A page size of 4 PEs minimizes utilization 36

A lower utilization increases throughput (IPC) 37

9 MULTI-THREADING CASE STUDY 39

9.1 Multi-threading Performance Factors 39

Multi-threading vs Single-threading 39

Direct Memory Access (DMA) Issues 40

Transform Time 40

9.2 Multi-threading System Benchmark 41

9.3 Multi-threading Performance Results 44

CPU-only Results Show Benefits of a Multi-threading Ability . . 46

Page Size Greatly Effects Overall Performance 48

Increasing CGRA Size Benefits only Multi-threaded CGRAs . . . 50

Increasing Threads Highlight DMA Needs of CGRA 52

9.4 Designing an Optimized System 54

Bottlenecks 54

An Optimal System Needs Sufficient DMA Bandwidth 54

Optimal System shows expected trends 55

CHAPTER Page

9.5 Power Reduction Analysis 56

Utilization is Maximized 57

Overall Power is Reduced 59

10 FUTURE WORK 62

10.1 Compiler Framework 62

10.2 Multi-threading Framework 63

11 CONCLUSION 64

BIBLIOGRAPHY 65

LIST OF TABLES

Table Page

8.1 *Abbreviations for benchmarks used in this work. Refer to Figure 8.1* . . 34

LIST OF FIGURES

Figure Page

2.1 *A basic CGRA architecture, with processing elements (PEs) connected by a fabric to each other, local data memory and local instruction memory. An expanded view of a PE is given, showing the input multiplexers and the rotating register file (RF).* 6

2.2 *A basic example of mapping on a CGRA. Shown in (a) is a DFG from a kernel for MPEG2. (b) gives the mapping on a the CGRA. In order to correctly map to the device, the kernel must be unrolled and software pipelined. This is illustrated in (c).* 8

2.3 *DFG in (a) is mapped to a 2x2 CGRA in (b) with an II of 2. DFG is unrolled once in (c) and mapped in (d) with an II of 4, for an effective II of 2.* 9

6.1 *Overview of compiler operation.* 16

6.2 *An example of mapping basic matrix multiplication to a CGRA. First, the code (a) is gimplified (b) to expose needed operations, a DFG is formed (c), and the operations are mapped in time and space on the CGRA (d). Note that in (c) the dotted lines between 2048 and a4. This is a target specific DFG, which requires a store node's data and address to be ready at the same time. This extra information is stored in the DFG for the CGRA compiler to use.* 19

Figure Page

7.1 *a) An example of possible page divisions. Note that transforms for the first division (a1) are slightly more complex than the second (a2). This is explained in Section 7.5 on page 27. b) An illustration of page-level data flow that the compiler is allowed to generate*. 22

7.2 *a) Due to the page divisions, page mirroring must be performed. Pages must be 'folded' along dotted axises when moving locations of the page. b) Page divisions here allow pages to be translated when moving locations. This is illustrated by the straight arrows*. 28

7.3 *Application of the multi-threading framework from mapping to transform. Note that in (c), to illustrate the page mirroring principle more clearly, node text is underlined and the number is suffixed by a* . . 29

7.4 *Shown is a transform of 6 pages to 4 pages. Pages with light text are pages scheduled during the initialization phase. The two pages bracketed were schedules using the tailing method. Pages with the darkest background and no shading behind them are those scheduled using Case 1 described in Section 7.5 on page 26. Pages with a lighter background and italicized are scheduled with Case 3. Those with a white background are scheduled with Case 2*. 30

8.1 *Performance of benchmarks compiled for different CGRA setups and sizes. Performance is the inverse of II. Performance is normalized to that of the original compiler. It can be seen that for a page size of 4 PEs, the constrained compiler can achieve almost equal performance.* . 35

Figure Page

8.2 *Average performance per page for an average of all benchmarks across different size CGRAs. Performance is inversely proportional to total utilization. A larger number indicates a higher quality mapping. This indicates that a page size of 4 PEs/page has the best utilization for this set of benchmarks*. 37

8.3 *Speedup of a multi-threaded CGRA using paging over that of a single threaded CGRA when running at maximum theoretical throughput for different page sizes and CGRA sizes. This indicates that a page size of 4 is expected to achieve the most speedup under multi-threaded loads*. 38

9.1 *Relative run time of systems using either a CPU or a single-threaded CGRA. For each system, run times are normalized to the run time of the CPU-only system of 16 threads. It is seen that a single-threaded CGRA decreases or achieves equal run time for a small number of threads, but begins to suffer as compared to simply running on a CPU when more threads are added*. 47

9.2 *Relative run time of systems using a paging-enabled CGRA for varying page sizes. For each system, run times are normalized to that of 8 PEs/page and 16 threads. It can be seen that a page size of 4 provides the most predictable run time*. 49

9.3 *Relative run time of systems using a single-threaded CGRA of varying sizes. For each system, run times are normalized to that of a 4x4 CGRA and 16 threads. It is seen that in single-threaded mode, run*

time is not decreased by increasing CGRA size. 50

9.4 *Run time of different CGRA sizes for a single-threaded CGRA vs a paging CGRA in a non-DMA-constrained system running 16 threads. The run time is normalized to a single-threaded 4x4 CGRA. It is seen that by allowing multi-threading through paging, larger CGRA structures are more effectively utilized and decrease run time*. 51

9.5 *Relative run time for systems with a single-threaded CGRA vs many CGRAs vs a multi-threaded paging CGRA. For each system, run times are normalized to that of a single-threaded CGRA and 16 threads. It can be seen that for DMA-constrained systems, a paging CGRA achieves equal run time to that of many CGRAs, while near equal run time to that of many CGRAs*. 53

9.6 *Average time of total run time each thread spent stalled as a percent of total run time using an 8x8 CGRA and a page size of 4 PEs. It is seen that a single-threaded CGRA becomes a bottleneck, and not until sufficient DMA bandwidth is available does a paging CGRA become a bottleneck*. 55

9.7 *Relative run time of non-DMA-constrained quad-core system vs optimized system for a benchmark with 16 threads. Run time is normalized to to the optimized system. It is seen that the optimized system provides near-equal run time to the non-DMA-constrained quad-core system while removing bottlenecks and being more efficient*. 55

Figure Page

9.8 *Overview of optimized system run time. Run time is normalized to 16 threads of a 6x6 CGRA. Various trends are present due to the optimization of this system, the most important being a decrease in run time as CGRA size increases*. 56

9.9 *(a) Useful Utilization: A multi-threaded CGRA is able to more effectively use the space it reserves than a single-threaded CGRA (8x8 CGRA). (b) PE Usage: A single-threaded CGRA must use the entire CGRA, leaving many PEs idle. A multi-threaded CGRA can use only a portion of the CGRA, decreasing the number of idle PEs*. 57

9.10 *Relative Power/Energy of the PEs in a single-threaded CGRA verse a multithreaded CGRA. While a multi-threaded CGRA increases the relative power in highly threaded environments because it has more active PEs, the total energy is significantly reduced due to the decrease in run time*. 60

9.11 *Relative energy of a CPU-only system verse a CGRA system is shown. As can be seen, a CGRA provides higher energy efficiency than CPUs, and a multi-threaded CGRA provides additional benefits as the number of threads increase*. 61

Chapter 1

INTRODUCTION

The need for power-efficient computing continues to increase. From the proliferation of smartphones to the desire to lower the development time and cost of application-specific embedded systems, a high performing and power-efficient flexible embedded system provides an attractive solution. The size and weight of a handheld device is largely determined by the battery size. On the other hand, the development of System-on-Chip (SoC) devices is both costly and time consuming. A power-efficient, flexible embedded system can mitigate both of these problems.

The efficiency at which operations are performed is becoming the largest concern. To solve this issue of power-efficiency, both hardware and software must be improved. It has become apparent that the serial execution of code no longer can deliver the computing needs demanded today and in the future. This is evidenced by the proliferation of parallel architectures in production and research. Software, in addition, must be designed and optimized for use on these architectures.

An architecture of interest is the coarse-grained reconfigurable array (CGRA). More specifically, the paradigm of using a CGRA as an accelerator processor in an embedded system is of interest. This co-processor paradigm has seen much research and commercial success, as seen by processor extensions such as MMX and SSE and GPU paradigms for OpenCL and CUDA. Enabling CGRAs to be used as co-processors will create scalable, power-efficient embedded systems.

CGRAs are extremely power efficient. ADRES, a popular CGRA-based architecture, operates at an efficiency of up to 40 MOPS/mW on 90nm technology [4]. This is compared to the Intel Atom N550, which provides about 4.6 GOPS of computation while consuming a maximum 8.5 W of power [16]. This provides a power efficiency of

about 0.54 MOPS/mW. A generalized CGRA architecture has been estimated to be able to achieve power efficiencies of 10-100 MOPs/mW [36]. In addition, the architecture itself is very flexible, allowing for quick reconfiguration every cycle. This power efficiency and flexibility of CGRAs make them an attractive solution for a power-efficient generalized embedded system.

Traditionally, CGRAs have been used as streaming processors in extremely-embedded systems where computing needs were deterministic. This contrasts from a generalized embedded system, where any arbitrary task can be run in conjunction with any number of other tasks. These tasks come and go at random times and greatly increase the complexity of the computing needs for the system.

In order to make effective use of CGRAs in a generalized embedded system, several tools and capabilities must be present. Current and previous research has focused on the CGRA architecture itself and compilation techniques to take arbitrary code and map it to a CGRA. Mapping refers to the process of taking operations and placing them spatially and temporally on individual compute nodes in the CGRA. This process is relatively complicated and research is ongoing to improve the efficiency and effectiveness of such techniques.

There exists still a need for more tools and capabilities. First and foremost must be a more complete compiling approach for CGRAs used in a general purpose system. While research has been done on compiling loop kernels for CGRAs, the ability to take an entire application and map it between a general purpose processor (GPP) and a CGRA is not readily available. This ability must be present if CGRAs are to become a viable solution. This includes the task of identifying kernels to be mapped to the CGRA and automatically directing computation and data flow between the two resources.

In addition to this support, the capability of CGRAs to execute multiple threads at once is necessary. Due to the complexity and the nature of mapping, CGRAs are

able to operate on only a single thread. A pertinent research question has been how to improve CGRA utilization by identifying additional parallelization opportunities. Threading has become a quick and easy way to introduce parallelism in computing. For example, the NVIDIA CUDA architecture boasts the ability to handle thousands of threads and deliver hundreds of gigaflops of computation because of it [6]. Most processors for generalized embedded systems (ie, ARM A9) contain multiple cores, each capable of concurrently executing several threads. By enabling CGRAs to execute multiple threads at once, CGRA utilization and throughput can be increased.

This work helps enable the use of CGRAs as co-processors in general purpose embedded systems. This paradigm for CGRAs is new and little research exists on it. Performance but not power efficiency is presented for work done. Features and contributions of this work are:

• *Software-focused Solution:* This solution focuses on using software as a solution, leaving the hardware unchanged when possible. Additionally, it leverages already existing solutions, including state-of-the-art mapping algorithms.

• *Full System Compiler Framework:* A proof of concept system compiler based off the GNU compiler collection (GCC) is presented. This compiler enables the use of CGRAs as a co-processor by generating code that is able to execute between a GPP and CGRA.

• *Multi-threading Compilation Framework:* A framework that existing compilers can follow is created with minimal performance impact. Results are presented for a large range of embedded application kernels showing less than a 1% cost for enabling this framework.

• *Fast Run-time Transformation Algorithm:* An algorithm used to enable dynamic scheduling is provided to be used with schedules generated by compilers following the framework.

• *Generalizable and Dynamic Multi-threading on CGRAs:* Combining the previous

bullets, a generalizable and dynamic multi-threading is enabled for CGRAs. Very little attention has been given to CGRAs and multi-threading.

• *Performance Analysis and Case Study:* An analysis of the benefits of enabling multi-threading is given showing utilization increases of over 280%. A case study exploring important design aspects of a generalized embedded system using a CGRA is given. System performance is shown to increase by almost 350% compared to a single-threaded CGRA and over 20x faster than an identical system with only a CPU and no CGRA in heavily threaded environments.

Combining these, this work presents a complete end-to-end software solution for CGRAs as co-processors.

Chapter 2

BACKGROUND AND TERMINOLOGY

The CGRA architecture is best seen as a class of architectures. Several distinct architectures, such as MorphoSys [36], ADRES [37], RSPA [18], and KressArray [14], have been presented over the years, all of which are best classed as CGRAs. [13] gives a comprehensive summary of many different CGRA architectures. Additional architectures include PipeRench [12], PADDI [5], XPP [3], REMARC [25], and MATRIX [24].

If a general description for CGRAs were to be given, it would be to describe them as a grid of simple processing elements (PEs) placed on a communication mesh. This is shown in Figure 2.1. Specifically, PEs generally have methods receiving as inputs the outputs of neighboring PEs (which specific neighboring PEs is a design specification). In addition, they contain a rotating register file, and some or all of the PEs have access to a bank of local memory buffered from main system memory. PEs can perform simple operations, such as shift, multiply, add/subtract, bit-wise operations, etc., and some architectures allow for more complex operations (such as CGRA Express [31]).

The interconnect network can be described as the multiplexing of outputs of PEs into one of the inputs for one or more neighboring PEs. The memory bus from the local memory to the PEs usually operates in a similar manner. Which multiplexed input to use is contained in the instruction to the PE. Instruction memory is traversed serially using a counter, and upon reaching the end of the instructions, the counter is reset. This is indicative of the typical operating environment for CGRAs, which is generally small loop kernels executed multiple times.

The use of a CGRA in a generalized embedded system would necessitate its actual placement on a bus. The CGRA would be responsible for copying data and instructions to and from main memory into local memory. Hardware required for this

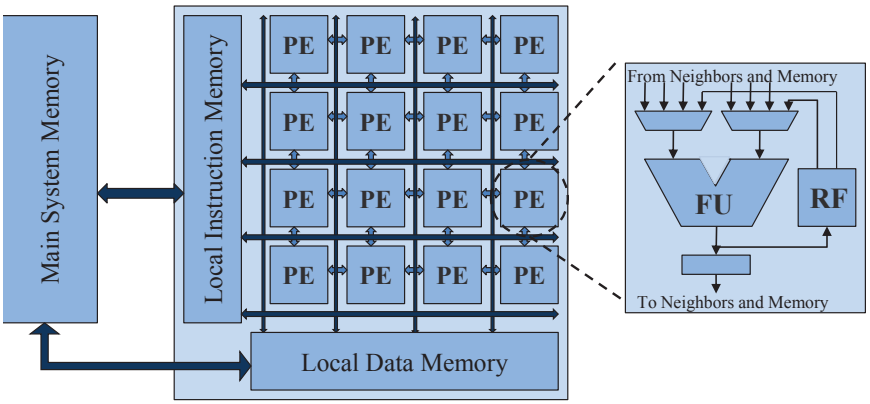

Figure 2.1: *A basic CGRA architecture, with processing elements (PEs) connected by a fabric to each other, local data memory and local instruction memory. An expanded view of a PE is given, showing the input multiplexers and the rotating register file (RF).*

operation is standard and would not be changed for use with a CGRA. In this work, CGRAs are used as co-processors to a general purpose processor (GPP). This means that the GPP is responsible for executing serial code and other code not suitable for execution on a CGRA, and directing code to be executed to the CGRA and preparing the CGRA memory buffers.

This paradigm of having a general purpose processor with a accelerator processor has many benefits. General purpose processors contain specialized hardware (memory controller, register offsets, cache, etc) that is not necessary for all code. A co-processor can be built that is much simpler and more specialized, optimizing for power efficiency. Thus it is with CGRAs, having very simple processing cores on a low power mesh. The caveat is that by removing conventional hardware pieces, more burden is placed on the compiler to correctly use the hardware.

2.1 Kernel Mapping is Complex

Due to the unique CGRA architecture, advanced techniques for performing operations on a CGRA have been developed [30, 38]. The process of taking a piece of code and modifying it to run on a CGRA is known as mapping. This process involves generating a data-flow graph (DFG), software-pipelining using modulo-scheduling to map the

DFG to the CGRA into a kernel [32], and generating a prolog/epilog to prepare/finalize data used for/by the kernel.

Many algorithms exist to assist in mapping code to CGRAs. Since CGRA designs widely vary, mapping algorithms are developed for a particular CGRA. For example, the ADRES [22] architecture uses DRESC [20, 21], the RaPiD [9] uses SPR [11], and CGRA express [30] uses modified EMS [27]. Attempts are being made to increase the independence of the mapping algorithm from the architecture. Instead of designing mapping algorithm for an architecture, algorithms now work for a basic description of the target architecture. However, the joint scheduling and operand routing problems are NP-complete. Therefore, existing compilation techniques, in an attempt to generate a good solution, take a long time [1, 7, 27, 38].

Compilation time is not a major concern for extremely embedded systems, in which the application is compiled once, and then executed indefinitely. The DRESC algorithm [21] uses simulated annealing to simultaneously solve the scheduling and routing problem. Park et al. [27], develop an edge-centric modulo scheduling (EMS) based mapping technique, which repeatedly searches for a route to connect the newly placed node to its predecessors. The runtime of their technique would be exponential, but is limited to polynomial time due to user-defined constraints. Dimitroulakos et al. [7], propose a modulo scheduling based backtracking scheme to generate a good mapping. Compilation techniques by Ahn et al [1, 38] use an ILP inside the compiler to obtain better mappings.

An example of the mapping process is shown in Figure 2.2, which takes a kernel for MPEG2 processing and maps it to a 4x4 CGRA. Light colored nodes are load/store operations, while dark colored nodes are data operations, such as add or multiply. The loop is unrolled and software pipelined to allow a complete iteration of the loop to complete every clock cycle, giving it an iteration interval of 1. This is explained further in the next section.

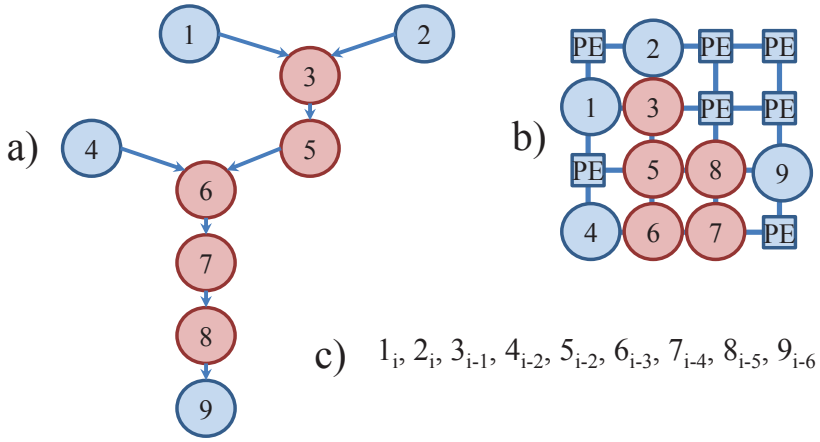

Figure 2.2: *A basic example of mapping on a CGRA. Shown in (a) is a DFG from a kernel for MPEG2. (b) gives the mapping on a the CGRA. In order to correctly map to the device, the kernel must be unrolled and software pipelined. This is illustrated in (c).*

2.2 Mapping Terminology

Due to the uniqueness of CGRAs, they use very specific terminology.

Kernel, Mapping, II, and Schedule: Code eligible for acceleration on a CGRA is referred to as a kernel, often the innermost loop code. When mapped to the CGRA, the iteration interval (II) is the number of cycles it takes to complete a single iteration of the kernel. A completed mapping is referred to as a schedule. Since CGRAs are statically scheduled and memory stored in a buffer, performance directly increases as II decreases.

Optimizing Schedules: The goal of all mapping algorithms should be to minimize II. The minimum II is limited by both resource constraints and recurrence constraints.

Resource Constraint: A resource constraint means that for a kernel that has x nodes and a CGRA that has y PEs, if $x = 4 \times y$, then II *cannot* be less than 4. Other resource constraints exist, such as a limited number of data memory ports available to PEs.

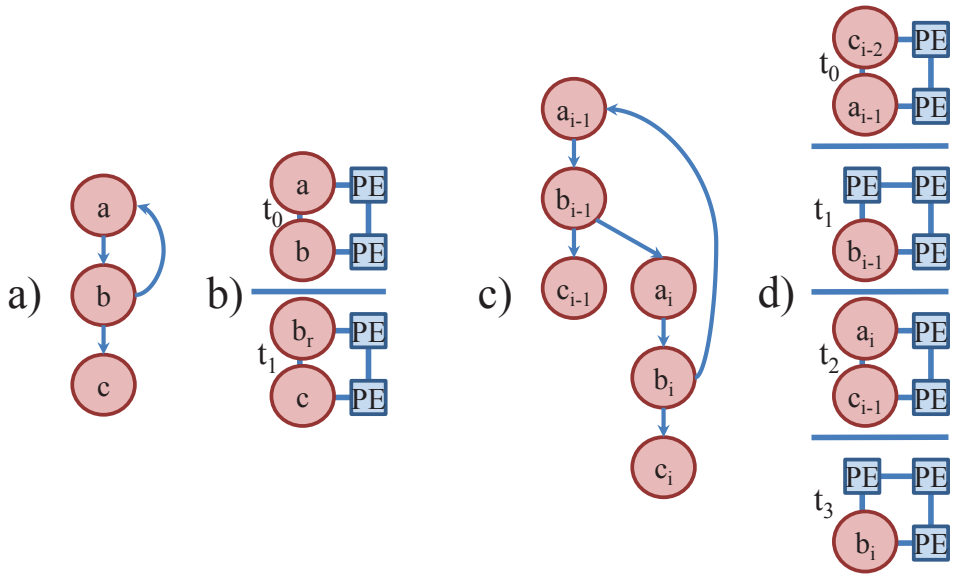

Figure 2.3: *DFG in (a) is mapped to a 2x2 CGRA in (b) with an II of* 2*. DFG is unrolled once in (c) and mapped in (d) with an II of* 4*, for an effective II of* 2*.*

Recurrence Constraint: A recurrence constraint is illustrated in Figure 2.3. A simple DFG is shown in (a), along with its mapping to the CGRA in (b). In (c), the DFG is unrolled once and mapped to the CGRA (d). In the first case, a single iteration of the loop is executed every two cycles, while the second case, two iterations are executed every four cycles, for an effective II of two for both cases. For this graph, the lowest achievable II is two for *any* CGRA size.

Chapter 3

MOTIVATION

3.1 Compiler Increases CGRA Applicability and Usability

In order for CGRAs to truly become a usable co-processor, there must be a large code base that can be ran under that paradigm. This is impractical by hand. Therefore, a compiler toolset should be created that can do this automatically. This will enable more robust research to be performed and enable a useful set of applications that can be run using this paradigm.

3.2 Multi-threading Provides Better CGRA Utilization

II is only a measure of single-threaded performance. When introducing multi-threading, the schedule must be more closely examined. To illustrate this need, a short example is in order:

- If a kernel compiled into two different schedules has an II of 3, but one schedule uses 24 PEs while the other only uses 12, it is clear the second is more efficient. The first schedule may use more PEs for routing data. While these PEs are necessary for the operation of the schedule, they do not perform useful calculations. In single-threading mode, since no other schedule can access the CGRA, unused PEs must necessarily remain unused in either case. However, in multi-threading mode, unused PEs can be used by other schedules if properly set up.

This example shows a discrepancy for definitions for utilization. On the one hand, the schedule using 24 PEs has a higher utilization than the schedule using 12 PEs. However, both accomplish the same amount of computation in the same amount of time. Therefore, the amount of useful utilization is the same. In this work, the difference between the first utilization and the useful utilization is a focal point.

Many works have recognized the low CGRA utilization of individual schedules, especially as CGRA size increases [19, 17]. Multi-threading enables CGRAs to take advantage of thread-level parallelism in addition to instruction level parallelism.

Instructions-per-clock (IPC) is an appropriate metric for a multi-threaded CGRA. When multiple threads are ran on a CGRA, the utilization of the CGRA becomes additive of each individual thread, up to at best 100%. IPC of the CGRA is then additive of the IPC of each individual thread.

Utilization, IPC, and Throughput: The internal details of the mapping for many kernels show that a large number of PEs actually are idle or not usefully utilized much of the time. The number of PEs that are idle or are not usefully utilized for an iteration of a kernel is given by $II \times Utilization \times CGRA\ Size - IPC$, where IPC is equal to the number of operations performed in each iteration of the kernel (for a given kernel, IPC is constant). In a single-threaded CGRA, the only way to decrease the number of idle/poorly utilized PEs is by decreasing II as utilization is necessarily 100%. However, II has a minimum value. Multi-threading allows decreasing the number of idle/poorly utilized PEs by reducing the utilization of an individual schedule. The PEs then not used by the schedule can be used by other schedules for a higher IPC and throughput.

Chapter 4

RELATED WORK

Much research has been done with CGRAs for use in extremely embedded systems. This research has produced the large number of CGRA architectures present, as well as mapping algorithms. Little to no work exists for enabling multi-threading in CGRAs as well as a compiler framework for using CGRAs as a co-processor.

4.1 Compiler Framework

To the best of the author's knowledge, no work presents a complete compiler framework for CGRAs for use as a co-processor in a generalized embedded systems. This makes the work presented the first of its type. This is not to be confused with mapping algorithms, plenty of which have been presented.

4.2 Multi-threading

Much research has been done with CGRAs for use in extremely embedded systems. This research has produced the large number of CGRA architectures present, as well as mapping algorithms. Little work exists for enabling multi-threading in CGRAs.

Multi-threading on CGRAs is a new concept. It is important to contrast what multi-threading can mean. In this work, a general multi-threading ability is enabled, ie, the ability to allow multiple unrelated threads to run on the CGRA. These threads can either be spawns of the same parent process or completely independent processes. However, it would be valid to claim multi-threading if only related threads could be run on the same CGRA, though clearly only a sub-class of the generalized multi-threading ability.

In addition, the dynamism of the technique is an important characteristic. In this work, since the threads do not be related nor do they have to run on any specific

partition of the CGRA, dynamism is unlimited. However, if threads must be related and target specific portions of the CGRA, the dynamism is limited, and oftentimes, statically enabled. Therefore, in this work, a general and dynamic multi-threading framework is said to be enabled.

Polymorphic Pipeline Arrays

One work that allows several kernels to be executed simultaneously is Polymorphic Pipeline Arrays (PPAs) proposed by Park et al. [28]. The CGRA consists of physically separate cores (each containing four PEs with additional customized hardware), which are allocated to the kernels. These kernels, however, must all be compiled together at compile time and are generally 'pipelines' of a greater task. Thus the kernels are related by data dependencies. At runtime, depending on the data set, the individual kernels can be apportioned cores as needed by data demands. This technique enables a thread-related dynamic multi-threading framework.

MT-ADRES

Another work attempts to enable multi-threading on the ADRES architecture. The ADRES architecture seeks to create a complete processor packaged with a CGRA. It is best seen as a CGRA with a subsection able to run as a VLIW processor. While running in VLIW mode, the rest of the CGRA sits idle. However, ADRES can seamlessly switch to a CGRA mode from VLIW mode and back again. The authors recognize that as ADRES increases in size, performance does not scale as well. Therefore, they propose partitioning ADRES to allow multiple threads to run simultaneously. This partitioning is accomplished manually and is run time-static (the partitioning could possibly be done automatically, but would remain run time-static). While the authors only showed enabling running two related threads, the technique could presumably be expanded to enable a run time-static generalized multi-threading.

Chapter 5

CONTRIBUTIONS OF THIS WORK

5.1 Compiler Framework

A compiler framework for CGRAs for use as co-processors is significant. By automatically forming a data-dependency and control-flow graph from code and exporting it, existing mapping algorithms can be used to compile code for the CGRA, while the compiler handles communication between the CGRA and GPP. This is an essential link for expanding the research scope of mapping algorithms and CGRAs.

5.2 Multi-threading Framework

Very little attention has been given to CGRAs and multi-threading. This may partly be due to CGRAs use in extremely embedded systems. However, as CGRAs are used in more generalized systems, multi-threading will become an essential characteristic of CGRAs. The framework presented in this work can be included in future work with CGRAs. This would enable general and dynamic multi-threading on CGRAs.

Chapter 6

CGRA COMPILER FRAMEWORK

Most research on CGRAs has been focused on efficiently mapping data-dependency graphs (DFGs). However, this in itself is incomplete. What is still needed is automated DFG generation and data management.

This work extends these works by integrating with GCC techniques to automate the compilation process for the CGRA. This includes taking source code (written in C, though expandable to other languages) and compiling it into a binary that is able to execute between the general purpose processor and the CGRA. To do this, the process is separated into sections, including: identification, transformation, compilation, and data communication.

In this work, CGRA acceleration is limited to loop kernels. While CGRAs can conceivably accelerate other types of code, the acceleration of loop kernels is a well-established research area. Research has been done on accelerating serial code, including multiple basic blocks; however, these works often assumed a tightly-coupled CGRA to the processor, leaving the bounds of the paradigm used in this work.

6.1 Compiler Operation

Figure 6.1 gives an overview of the compilation process. The compiler is divided into distinct phases, which perform basic operations on the code to generate a complete binary.

Identification

This first step in creating a compilation framework for CGRAs is determining a means of identifying code that should be ran on the CGRA and code for the GPP. The ideal code portions for CGRA execution are innermost loops, though it is not limited to only

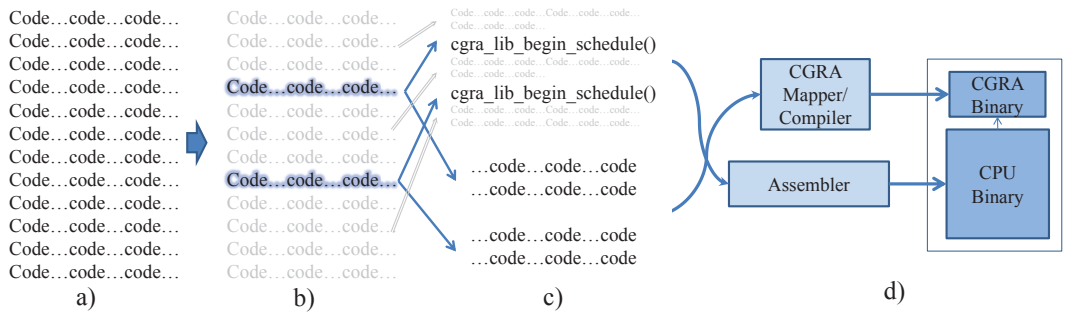

Figure 6.1: *Overview of compiler operation.*

this. During this identification phase, how well a loop can be unrolled is determined, along with possible initialization times compared to expected run times. All code not expressly identified for processing on the CGRA goes through a typical compilation process. This step is illustrated in Figure 6.1 between (a) and (b).

Transformation

Once code sections to be ran on the CGRA are identified, the code must be transformed into a data-dependency graph (as shown in Figure 6.2) and also a data-flow graph. Kernels that do not include branching do not need a data-flow graph. This transformation must take into account array address calculation and memory operations. The generated data-dependency graph must also be adjusted to hide nodes that represent data flow only and not data manipulation (as the concept of intermediate variables is unnecessary in CGRAs, as this is modeled in data flow). This step is illustrated in Figure 6.1 between (b) and (c).

Compilation

After transformation, the resultant data-dependency graph and data-flow graph is passed to the CGRA compiler. This compiler takes as inputs the description of the target CGRA architecture, data-dependency graph (DDG), and DFG. It then uses a mapping algorithm to generate instructions for the CGRA, including a prologue, kernel, and

epilogue. The prologue and epilogue are used to start and stop loops that have been unrolled, as well as initialize other necessary portions of the CGRA. These instructions along with necessary control descriptions for the CGRA are dumped into binary files. This is shown in Figure 6.1 under (d).

Data Communication

At the completion of compilation, a binary file exists assuming a few conditions. The compiler is responsible for setting up these conditions, including preparing CGRA local memory. A library orchestrating these transfers is written, referenced in Figure 6.1(c), which from a high level simply copies data into CGRA memory, reading CGRA binary files from disk into CGRA memory, setting up a control structure, executing the CGRA, and stalling the processor until execution completes and memory is copied back from CGRA memory.

6.2 Implementation Details

The actual implementation of the compiler in this work was done using GCC as the general purpose compiler and the EMS algorithm for mapping code for the CGRA. The modifications in GCC included the addition of an OpenMP-like pragma to identify code to be ran on the CGRA. GCC then generates a DDG and passes it to the EMS-portion of the compiler, which is written as a shared library.

The EMS shared library handles typical mapping operation for the CGRA and generates a control file related to that mapping and a binary instruction dump file to be loaded at run time. GCC generates the necessary code using a shared runtime library to load instructions and data into CGRA memory at runtime and stall the thread until CGRA completion (it is this library that is modified to support multi-threading).

This set up of CGRA using a shared library for the CGRA portion of the code generation is extremely useful. This allows for any mapping algorithm to generate code

for any target CGRA, only needing to generate the control file and instruction dump.

Figure 6.2 gives a representation of the actual implementation created in this work. This figure gives greater detail to what happens between steps (c) and (d) in Figure 6.1. In this case, the code is a kernel to perform matrix-matrix multiplication (a). The code is first gimplified (b), exposing all operations needed to be performed. The compiler then creates a DDG (c), which in general can be divided into two parts: address calculation and data manipulation. For this DDG, the greater portion is dedicated to address calculation (17 nodes), while a small portion performs data manipulation (5 nodes). This DDG is then mapped to the CGRA (d), which process is explained in Section 2.1 on page 6.

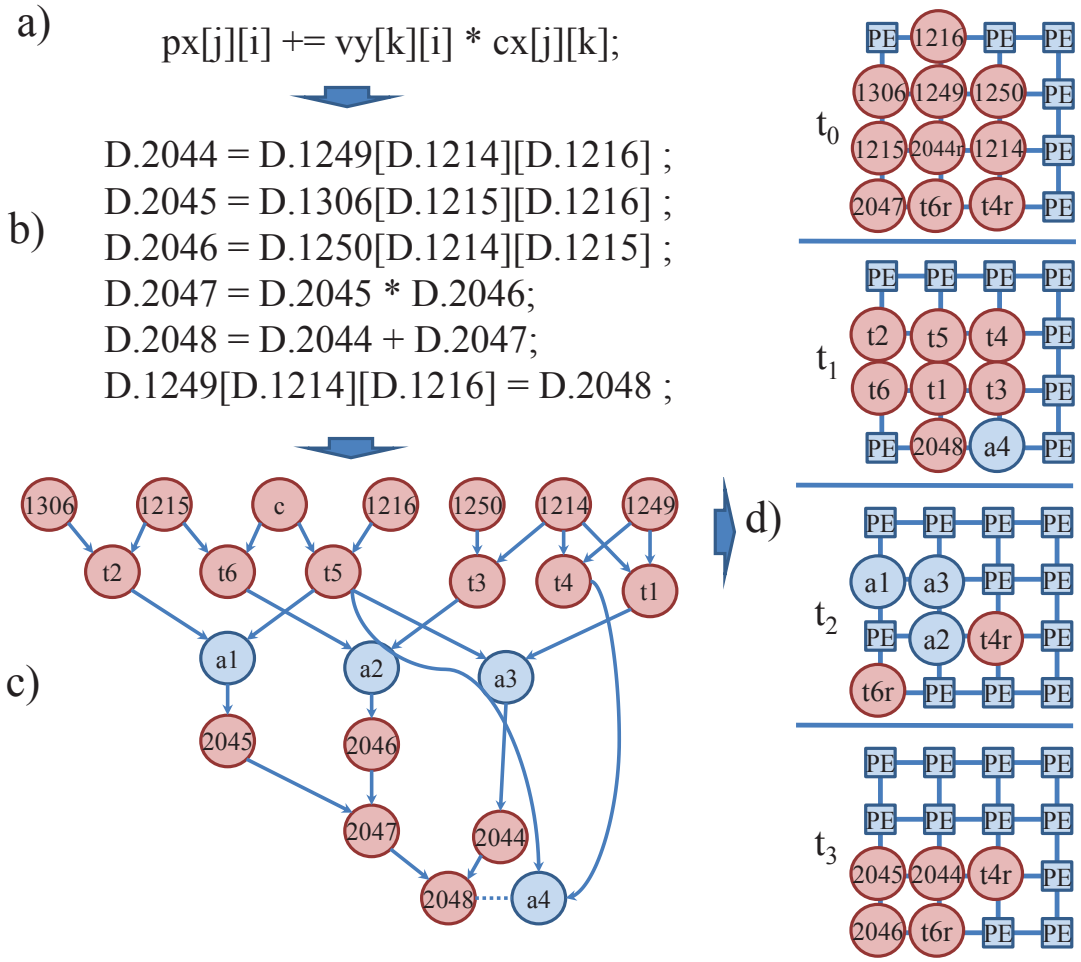

Figure 6.2: *An example of mapping basic matrix multiplication to a CGRA. First, the code (a) is gimplified (b) to expose needed operations, a DFG is formed (c), and the operations are mapped in time and space on the CGRA (d). Note that in (c) the dotted lines between 2048 and a4. This is a target-specific DFG, which requires a store node's data and address to be ready at the same time. This extra information is stored in the DFG for the CGRA compiler to use.*

Chapter 7

MULTI-THREADING FRAMEWORK

As explained in Section 2.2 on page 8, II has a lower bound for a given kernel. Conversely, useful utilization for a given CGRA has an upper bound for this same reason. It is this upper bound that provides a motivation for enabling multi-threading in CGRAs. By fully utilizing a CGRA, performance increases in the form of higher IPC.

There are several ways in which multi-threading can be achieved in CGRAs. Without modifying the hardware, however, only a few ways are readily conceivable. Due to the complexity of CGRA mappings (which can be viewed themselves as DFGs), no naive modifications can guarantee a working schedule. It is possible to combine two separate DFGs and map them simultaneously together, but this is impractical at run time and not flexible enough to be done statically.

Instead, a form of hard multiplexing should exist, either in time or space. Little, if any, performance gains can be gleaned from time multiplexing (this is true because a schedule uses all cycles to execute, but not all PEs), so this work creates a method of space multiplexing. This allows for a software-focused solution that does not require specialized hardware.

The key idea contained in this framework is to create schedules that can be quickly transformed at run time to run on different portions of the CGRA as needed. These transformations are done at such a granularity that they are quick but also able to allow fine grained targeting of portions of the CGRA. By transforming schedules to target portions of the CGRA, multiple schedules can be ran simultaneously on the CGRA for increased IPC. Additionally, if the actual utilization of the schedule is minimized (approaching the value for useful utilization), then more schedules can be ran simultaneously, increasing IPC further.

To form the framework, a few basic CGRA specifications are presented, as well as a compiler framework. After giving these specifications and framework, a problem definition will be formed. Finally, an algorithm to solve this problem as well as a few practical implementation details are given.

7.1 Overview of Framework

Figure 6.1 gives an overview of the responsibilities of a compiler for using a CGRA in a generalized embedded system. This is explained fully in Chapter 6 on page 15 but a quick overview is given here. The compiler takes code (a) and identifies portions that are eligible for acceleration on the CGRA (b). It splits this code, inserting the necessary library calls in the portion to be ran on the CPU and separating CGRA-eligible code from the rest (c). Between (c) and (d), the compiler generates a DFG for the mapping algorithm. In (d), binaries are generated and assembled together. When a thread is run, the CPU executes portions targeted for it, and directs execution to the CGRA when appropriate. It is in (d) that the multi-threading framework resides. In addition, the framework works during run time to enable dynamic scheduling.

7.2 CGRA Hardware Specification

Due to the vagueness of the definition of CGRA architectures, it is necessary to give a more complete definition of the CGRA architecture for which the framework is targeting. The requirements listed are not in themselves special or unique to this work, but not all CGRAs are designed with all the necessary components. This work uses an ADRES-like CGRA.

Connection Topologies

The multi-threading framework presented requires a uniform and symmetric connection topology. This can be as simple as the one shown in Figure 7.1, or perhaps more complex. This requirement exists because the CGRA will be conceptually divided into

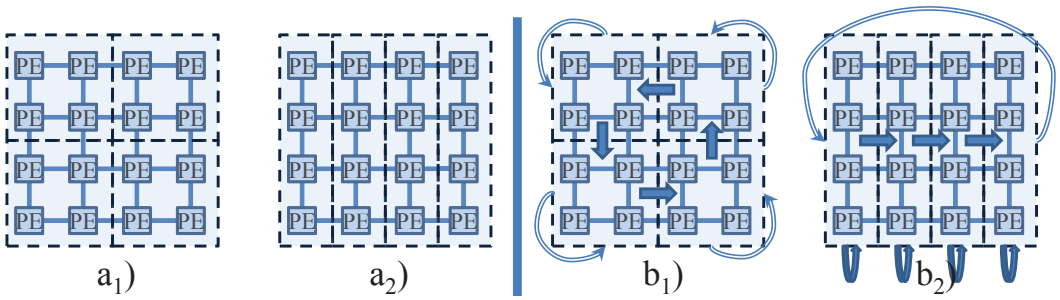

Figure 7.1: *a) An example of possible page divisions. Note that transforms for the first division (a_1) are slightly more complex than the second (a_2). This is explained in Section 7.5 on page 27. b) An illustration of page-level data flow that the compiler is allowed to generate.*

sections called pages by the compiler, and connections between pages must by identical (thus assuring truly dynamic threading). These pages will then become the objects of transformation.

Register Requirements

In order to use the full multi-threading framework, registers must be present and reserved only for multi-threading use. A limited form of multi-threading is possible without registers, but this work does not present results for this.

7.3 Compilation Constraints

Most of the magic for multi-threading happens during the compilation phase. This is because the compiler is responsible for producing schedules that can quickly be transformed at run time, but maintain high performance and efficiency (low II and utilization). The compiler targets a CGRA, but views it as pages. These pages are identical to each other and possible arrangements are shown in Figure 7.1(a).

Data Flow Constraints

At a high level, data can be seen to flow between and within pages. Since transformations will be performed at the granularity of pages, the compiler need only limit data

flow at the page level to assure a fast transform. This is done by allowing data to flow either to the same or next page in the next time, forming a ring of pages. This is the same as saying a page can only have dependencies from either the same or previous page of the previous time. This restriction is illustrated in 7.1(b). It will be shown later in Section 8.2 on page 33 that this restriction does not degrade kernel performance. This data flow constraint can be accomplished by modifying the connection topology the compiler assumes the CGRA has.

Register Constraints

If the CGRA provides registers, the compiler must not use these registers during mapping, as they will be used for the run time transformation. If a register is needed, the compiler can use global registers (often implemented simply as local memory) for these cases. Many mapping algorithms [7, 8, 10, 15, 21, 23, 27, 29, 30, 38] do not use local registers well in practice.

7.4 Problem Definition

Given a schedule P mapped to the CGRA structure with the compile-time constraints, reschedule the application at page-level granularity to a CGRA with equal or fewer number of pages.

Input: Schedule P for a CGRA with N pages. Suppose the Iteration Interval of the mapping is II_p. The mapping is specified as:

$$P = \{p_{(n,t)} : 0 \leq n < N, 0 \leq t < II_p\}$$

where $p_{(n,t)}$ represents the set of operations that will be performed on page n at time t. The constraint on the mapping is that the operations in $p_{(n,t)}$ are dependent through the interconnect (ring topology) from $p_{(n-1,t-1)}$ or the same page $p_{(n,t-1)}$.

Output: Schedule Q onto M pages of the CGRA. The new schedule can be specified as:

$$Q = \{q_{(n',t')} : 0 \leq n' < M, 0 \leq t' < II_q\}$$

where $q_{(n',t')}$ represents the set of operations that will be performed on page n' at time t'.

Constraints:

The first constraint is that no two pages in P must be mapped to the same page in Q. If $p_{(n,t)} \in P$ is mapped to $q_{(x',t')} \in Q$, we denote it by $p_{(n,t)} \rightarrow q_{(x',t')}$. Thus if $p_{(n_1,t_1)} \rightarrow q_{(x'_1,t'_1)}$, and $p_{(n_2,t_2)} \rightarrow q_{(x'_2,t'_2)}$, then if $n_1 \neq n_2$ and $t_1 \neq t_2$, then $x'_1 \neq x'_2$ and $t'_1 \neq t'_2$.

The other constraint is that the mapping Q must not break any of the dependencies in P. Thus, for each n,t, if $p_{(n,t)} \rightarrow q_{(x_1,t_1)}$, $p_{(n-1,t-1)} \rightarrow q_{(x_2,t_2)}$, and $p_{(n,t-1)} \rightarrow q_{(x_3,t_3)}$, the constraints are:

1. $(x_2 - 1 \leq x_1 \leq x_2 + 1)$ & $t_1 > t_2$
2. $(x_3 - 1 \leq x_1 \leq x_3 + 1)$ & $t_1 > t_3$
3. $x_1, x_2, x_3 < M$

Objective: Clearly the objective of the mapping is to minimize the II of the schedule Q, II_q. If the II of the original mapping P is II_p, then $II_q \geq II_p \times \lfloor \frac{N}{M} \rfloor$, by resource constraints.

7.5 Fast Runtime Transformation

It can be verified that the constraints placed on the compiler will produce a schedule P that meets the problem definition. In addition, it can also be verified that the hardware

requirements listed can execute a schedule Q. This transformation is useful for many reasons:

First is that having the ability to shrink schedules allows schedules to be sized to fit in unused portions of the CGRA during run time.

Second is less obvious and involves how recurrence constraints can be mapped using this framework. They can either be mapped entirely within a single page, or along the page ring. The ring topology presented to the compiler is adjustable. For example, a CGRA with only 4 pages can easily handle schedules compiled for a ring topology of 4, 3, 2, or 1 page(s) (while it could technically handle ring topologies of greater sizes than 4, it is not generalizable and not done in this work). This allows the ring to be sized according to the size of the recurrence cycle. However, if a ring is sized for 3 pages and scheduled to a CGRA of 4 pages, a transform of 3 pages to 3 pages must be performed.

What is left is an algorithm to perform this transformation, given as $T(P) \rightarrow Q$. The given algorithm is the Pagemaster algorithm which runs in time linear to the number of pages in the transformed schedule.

Transforming a Schedule

The Pagemaster algorithm handles transforming a schedule in two distinct stages. The first stage is an initialization stage in which the first iteration of all pages in P are placed in Q. The second stage involves scheduling the remaining pages from P in Q.

Initialization Stage For any schedule, any arbitrary page must be placed in $q_{0,0}$. If this page is given by $p_{n,0}$, then $p_{n,1}$ and $p_{(n+1)mod(N),1}$ are the two successor pages that have dependencies on $p_{n,0}$. These two pages have two other dependencies $p_{(n-1)mod(N),0}$ and $p_{(n+1)mod(N),0}$. These two pages must be placed next in Q in order to maintain the

two hop constraint from the problem definition. This will produce the following schedule:

- $p_{n,0} \rightarrow q_{0,0}$
- $p_{(n-1)mod(N),0} \rightarrow q_{1,0}$
- $p_{(n+1)mod(N),0} \rightarrow q_{2,0}$

Using this same argument, the remaining portion of P can be scheduled in Q. Since Q can have fewer pages than P, not all pages of P may be scheduled in the first iteration of Q. In the case where an entire iteration of Q can be filled with pages from P (ie, the second iteration of Q can be filled entirely if $2 \times M \leq N$), the same pattern established as before is followed, wrapping at the edges of the schedule. However, in the case where an entire iteration of Q cannot be filled with the remaining pages from the first iteration of P (ie, $(N)mod(M) \neq 0$), pages are schedule tailing vertically ascending numerically along the edge of the structure.

Filling Remaining Schedule To place the remaining pages in P, Algorithm 1 is used. The algorithm is called for each page in an iteration of P and then for subsequent iterations in P. This is done so that `findDependencyColumns()` can be optimized to run in constant time, as dependency columns can be estimated. Since dependencies can be at most two hops apart, there exist three possible cases for dependencies.

1. **Case 1:** *The dependencies are two hops apart*

   This is the most common case, and there exists only one page column that $p_{n,t}$ can be scheduled, $(d1+d2)/2$.

2. **Case 2:** *The dependencies are a single hop apart*

   This case can only happen if one of the dependencies are on the edge. Since Case 1 cannot schedule any pages on the edge, in this case the page is scheduled on

Data: d_1 is the column location of $p_{n-1,t-1}$
d_2 is the column location of $p_{n,t-1}$
t_1 is the next available time in the column being placed in after the time $p_{n-1,t-1}$ and $p_{n,t-1}$ have executed in Q

$d_1, d_2 \leftarrow findDependencyColumns()$
switch d_1 **do**
  case *($d_2 \pm 2$)* /* Two hops apart */
    $p_{n,t} \rightarrow q_{(d1+d2)/2,t1}$
  case *($d_2 \pm 1$)* /* One hop apart */
    if *(d_1 = 0 or d_2 = 0)* **then**
      $p_{n,t} \rightarrow q_{0,t1}$
    else if *(d_1 = M-1 or d_2 = M-1)* **then**
      $p_{n,t} \rightarrow q_{M-1,t1}$
    end
  case *(d_2)* /* Zero hops apart */
    if *(d_1 - 1 has less pages scheduled in it)* **then**
      $p_{n,t} \rightarrow q_{d1-1,t1}$
    else if *(d_1 + 1 has less pages scheduled)* **then**
      $p_{n,t} \rightarrow q_{d1+1,t1}$
    end
  endsw
endsw

Algorithm 1: PlacePage($p_{n,t}, Q$) outputs Q

the edge.

3. **Case 3:** *The dependencies are in the same page column*

   This case is created when scheduling pages tailing in the initiation phase (if not, then Case 2 becomes a fall back). If a page from P in one iteration is scheduled with this case, all pages from subsequent cases will be scheduled with this case. Therefore, the page is scheduled in the page column in Q that has the fewest already scheduled pages, or, in other words, where the previous times of the current page have not already been scheduled.

Page Mirroring

While performing transformations, when placing pages in new locations, they cannot be simply translated. Instead, they must be mirrored. This mirroring is illustrated in Figure 7.2. This mirroring is only necessary when a page is translated across an axis

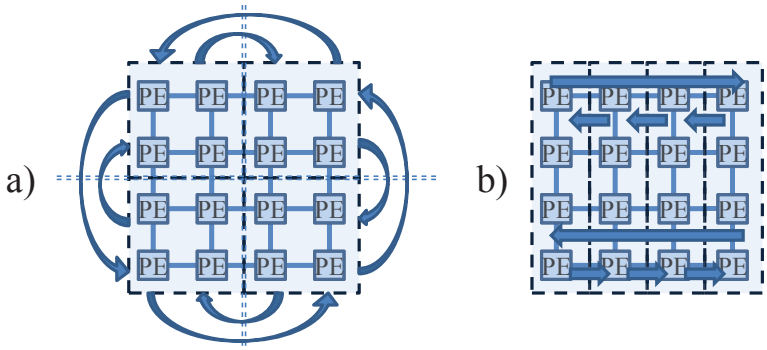

Figure 7.2: *a) Due to the page divisions, page mirroring must be performed. Pages must be 'folded' along dotted axises when moving locations of the page. b) Page divisions here allow pages to be translated when moving locations. This is illustrated by the straight arrows.*

and the page width (in number of PEs) is greater than 1. Thus it is not necessary to perform mirroring on any pages in Figure 7.1(a_2).

7.6 Example Transforms

Two examples are given to illustrate both the complete framework and the Pagemaster Algorithm.

Example One: Mapping to Transform

The first example shows the process of mapping a DFG according to the compiler restrictions and then transforming that same mapping to run on a single page. For this example, the mapping done in Section 2.1 on page 6, shown in Figure 2.2 is used. The first step in the framework is to apply the compiler data flow restrictions. This is shown in (a) of Figure 7.3. While not always the case, in this example, the mapping already satisfies the restrictions, as shown in (b) of Figure 7.3. This shows the completed mapping, and becomes Schedule P from the problem definition. The next step is to transform to a Schedule Q using the Pagemaster Algorithm.

In this example, the transform performed is from 3 pages (as the mapping only needs 3 pages) to 1 page. This is shown in (c) of Figure 7.3. Here the pages are mirrored

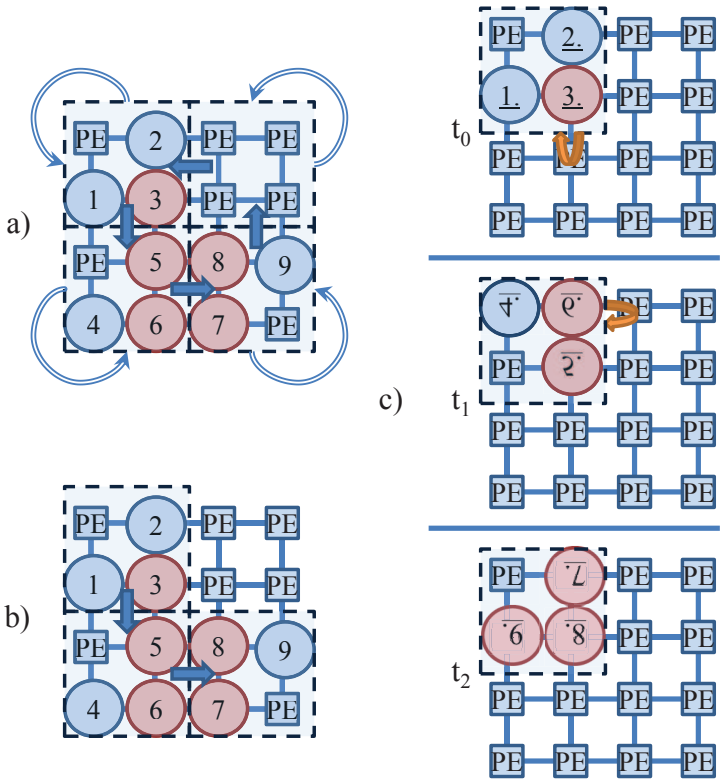

Figure 7.3: *Application of the multi-threading framework from mapping to transform. Note that in (c), to illustrate the page mirroring principle more clearly, node text is underlined and the number is suffixed by a decimal and then mirrored appropriately.*

in order to maintain correct internal page mappings. In addition to correctly placing the pages, data transfers must be modified. Arrows in (c) indicate data flow that would originally leave the page and flow to the next is modified to return to the same page. For example, node 6 normally passed data to the right to node 7. However, node 7 will now execute on the same page as node 6, so data just needs to be passed through an output buffer to node 7.

Not shown in the Figure 7.3 is the additional register usage. For example, normally node 1 passed data to node 3 in the next time iteration. However, node 3 will not execute in the next time iteration, so the data must be stored in a register. This is the case for all such situations.

Figure 7.3(c) clearly shows the general nature of the multi-threading that is

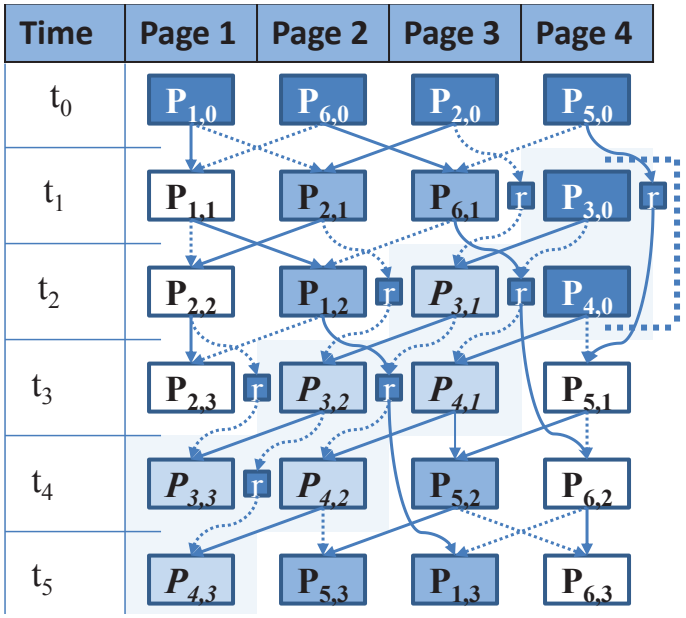

Figure 7.4: *Shown is a transform of 6 pages to 4 pages. Pages with light text are pages scheduled during the initialization phase. The two pages bracketed were schedules using the tailing method. Pages with the darkest background and no shading behind them are those scheduled using Case 1 described in Section 7.5 on page 26. Pages with a lighter background and italicized are scheduled with Case 3. Those with a white background are scheduled with Case 2.*

enabled. This schedule is completely unaffected by whatever else is scheduled to the other sections of the CGRA. It runs independently in its allocated space.

Example Two: Advanced Transform

The second example shows a generalized application of the transformation algorithm. Here, the precise internal page mappings are not exposed. Part of the strength of the framework is that the transform works regardless of what the internal page mappings are. In this example, a schedule P of 6 pages is transformed to a schedule Q of 4 pages. This is shown in Figure 7.4.

How all page dependencies are filled is shown. Dotted arrows indicate how previous page dependencies are fulfilled, while solid arrows indicate how same page dependencies are fulfilled. Small boxes with an 'r' indicate registers that must be used.

As can be seen, at no time are more than two register sets needed for a page at any time. Generalized, the most register sets needed by any set of pages is equal to the number of pages in P minus the number of pages in Q. Bracketed in the figure are pages scheduled during the initialization phase using the tailing method described. These pages are given a light shade behind them. When another page serious crosses this boundary, registers must be used to fulfill a page dependency. Excluding this intersection, data flow follows a regular pattern, as can be seen in the figure. Note that the shown schedule is not fully complete. The pattern must be continued for the looping nature of the kernel to work correctly.

An exception to Case 2/Case 3 described in Section 7.5 on page 26 is seen with $p_{5,1}$. While it appears that this page should be scheduled following Case 3, it is scheduled using Case 2. This is because this page is not a tailing page.

7.7 Implementation Details

In this work, the mapping algorithm used was EMS. CGRAs were divided as shown in a_2 in Figure 7.1. The caveat of this division is that a transform must always be performed when scheduling to the CGRA, whether the schedule is to be shrunk or not. A transform of $T(P) \rightarrow Q$, where Q has as many pages as P works just as any other transform. However, no registers are required. This requirement, however, is not expected to cause performance degradations, as a transform would have to be performed for any schedule compiled using fewer pages than the original structure (ie, a CGRA of 4 pages can have a schedule that uses only 2 pages). These transforms are necessary to remove the necessity of a hardware connection from the last page to the first page that the compiler assumed was present.

The advantage to using this division method is that transforms themselves become simpler. As explained in Section 7.5 on page 27, page mirroring is a technique to assure correct mappings. However, mirroring along the vertical axis a vertical division

of a single PE in width produces the same output as the original. In addition to this advantage, the CGRA used divided load buses by PE column. This is important because the hardware only supports one load/store operation on each bus each clock cycle. By mapping pages to a single load bus, the process of compilation becomes simpler.

Chapter 8

ANALYSIS AND RESULTS

8.1 Compiler Framework

The compiler framework presented in this work is not analyzed for performance. Since the use of CGRAs as co-processors is new, little work exists to compare against. Performance related issues are to be addressed in future work. By using GCC as the base compiler, this work benefits from an open source but mature project.

8.2 Multi-threading Framework

To analyze the multi-threading framework, a few metrics need to be identified. First, the mode of operation needs to be identified. There are two cases: one where only a single thread accesses the CGRA and one where multiple threads access the CGRA. In the case where only a single thread accesses the CGRA, such as when the user is only running only a single thread on the system, the performance of a schedule can easily be compared for the original, unmodified case and the case when the schedule is compiled following the restrictions in the multi-threading framework by comparing the II of each schedule. This is done for all schedules. In the case of multiple threads accessing the CGRA, a more complete analysis is done. The cost of the framework is any performance lost for single-threaded environments and the benefits of the framework are any performance improvements for multi-threaded environments.

Small Cost in Single-threaded Performance

To determine single-threaded performance, the II of a schedule just needs to be compared. An identical kernel with a lower II has a higher performance when compared to one of a higher II. This is true for all kernels. This work used 20 benchmarks (those listed in Figure 8.1, refer to Table 8.1 for abbreviations used) to analyze the effect of the restrictions imposed in the multi-threading framework. Shown in Figure 8.1 are the

Benchmark	Abbreviation
Banded Linear Equations	BLE
First Difference	First Dif
General Linear Recurrence Equations 1	GLRE 1
General Linear Recurrence Equations 2	GLRE 2
General Linear Recurrence Equations 3	GLRE 3
Hydro Fragment	HF
Matrix-Matrix Multiplication	M-M Mul
MPEG2 Form Pred	MPEG2 FP
Swim Calc 1	Swim 1
Swim Calc 2	Swim 2
Tri-Diagonal Elimination	TDE

Table 8.1: *Abbreviations for benchmarks used in this work. Refer to Figure 8.1.*

compilation results for the benchmark set normalized to the performance of the original, unmodified compiler. Three separate graphs are shown for three CGRA sizes of 4x4, 6x6, and 8x8. The average performance for the benchmarks for different CGRA sizes indicate that the restrictions have little effect on II (less than 1% difference on average) in the general case for a correctly chosen PE page size. It should be noted that the CGRA architecture targeted requires two PEs in the same column at the same time in order to perform a store operation. Therefore the page size of 2 PEs was unable to obtain a solution for every benchmark.

It can be concluded that the multi-threading framework has minimal cost for a correctly chosen page size.

Multi-threading Increases Utilization and Benefits Performance

Before presenting the multi-threading case study, an analysis of multi-threading and the predicted benefits is given, followed by an analysis of important factors that can affect performance. Afterwards, an overview of the system benchmark is given and the results of the case study.

To analyze multi-threading performance, a metric besides II must be used.

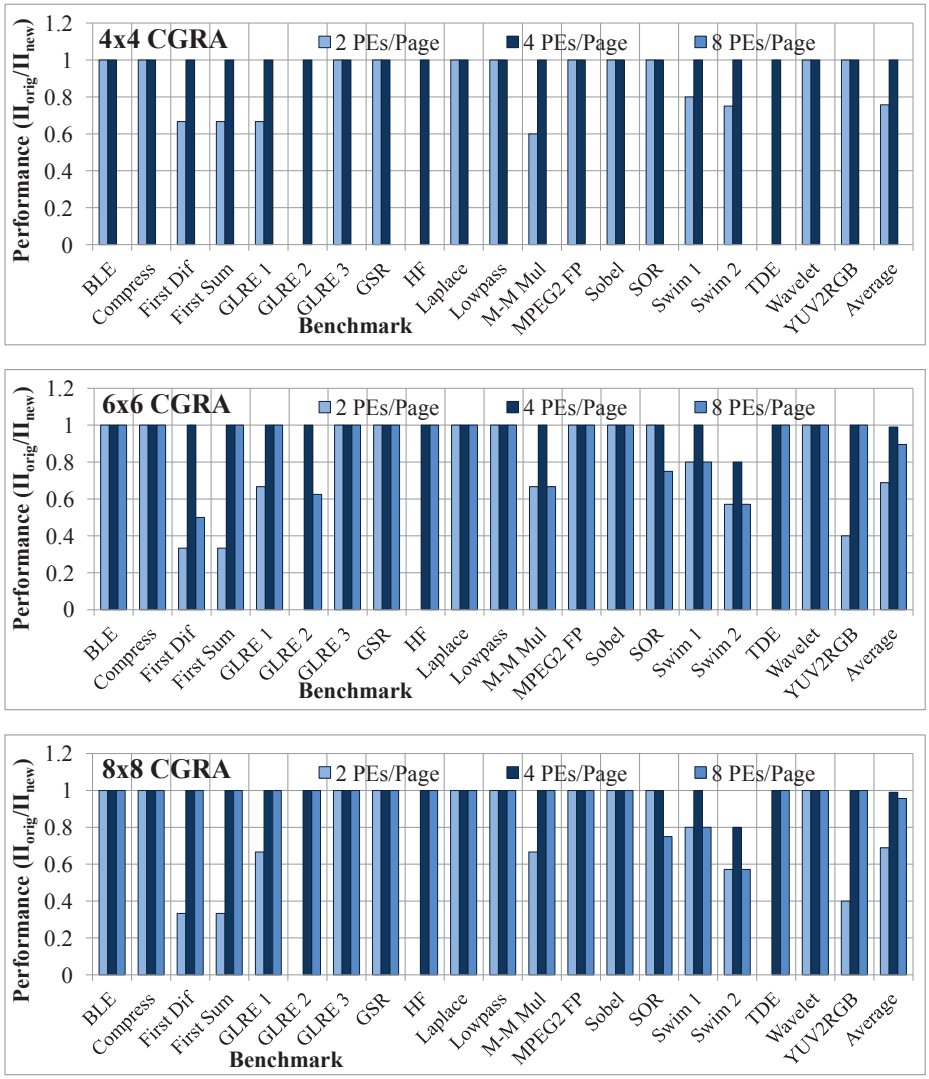

Figure 8.1: *Performance of benchmarks compiled for different CGRA setups and sizes. Performance is the inverse of II. Performance is normalized to that of the original compiler. It can be seen that for a page size of 4 PEs, the constrained compiler can achieve almost equal performance to that of the original compiler.*

While II indicates an individual schedule's performance, it does not take into account PE utilization within a CGRA. This is okay for single-threaded applications, but if the CGRA is to be shared, efficient use of PEs is crucial. This efficiency is manifest in multi-threading performance.

In this work, a new metric is created, referred to as useful utilization. A given kernel has a fixed minimum II, as explained in Section 2.2 on page 8. In addition, this

kernel has a minimum number of nodes, each of which must be mapped to a unique PE. A mapping algorithm necessarily (sometimes) introduces false nodes which occupy additional PEs. This number of false nodes represents a level of inefficiency, either due to the mapping algorithm's weakness or simply because some branches of a DFG cannot be mapped to the existing CGRA fabric without adding nodes. Before, these false nodes, so long as they did not increase II, did not impact performance. In multi-threading, this is no longer true.

By introducing the concept of space multiplexing to allow multi-threading and the ability to compile for sub-portions of the CGRA, useful utilization becomes an important target metric. Therefore, in addition to minimizing II to approach the theoretical minimum, it is important to minimize utilization to approach the theoretical minimum. In this work, utilization is measured in units of pages, as this is the level of granularity a transform can be performed. This is to say that a schedule that uses fewer pages and achieves equivalent II as another schedule for the same kernel is preferable (whether it is preferable for a schedule that has a higher II but uses fewer pages than another schedule is application specific; in this work it is assumed that minimizing II is the primary goal and minimizing page utilization is secondary).

A page size of 4 PEs minimizes utilization

The number of pages used by each schedule for the 20 benchmarks listed was measured for each PE size. This number alone does not capture the performance gain, as different PE sizes have a different number of pages available for a given CGRA structure. In addition, not all benchmarks were able to achieve the same II across all page sizes. To capture the effectiveness of a mapping, performance per page was measured. For example, a given CGRA has a set number of 'page executions' per cycle. In the case of a 4x4 CGRA with a page size of 2 PEs, this is 8 page executions per cycle. That is to say 8 different pages of instructions can be executed each cycle. If it were assumed

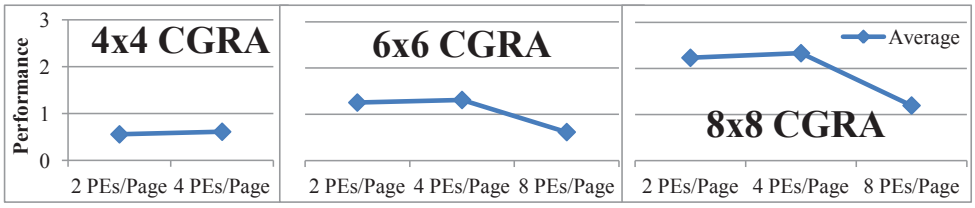

Figure 8.2: *Average performance per page for an average of all benchmarks across different size CGRAs. Performance is inversely proportional to total utilization. A larger number indicates a higher quality mapping. This indicates that a page size of 4 PEs/page has the best utilization for this set of benchmarks.*

that a schedule could be expanded instead of shrunk, the time it takes to complete one iteration of the kernel is equal to the number of pages the mapping used divided by the number of page executions per cycle multiplied by the II. The inverse of this is performance per page. The results of these calculations are shown in Figure 8.2. As can be seen, the average performance per page is slightly higher for a page size of 4 PEs compared to that of 2 PEs and significantly greater when compared to a page size of 8 PEs.

A lower utilization increases throughput (IPC)

Once efficiency is determined, throughput performance becomes proportional to the number of threads being accelerated. There exists a case where the CGRA is executing as many threads as possible, and threads appear only as other threads complete. In this case, the CGRA is running at maximum efficiency and throughput performance. This case is determined by the schedules running on the CGRA, the CGRA size, and the page size used in the CGRA. In cases where more threads arrive than can be executed, the CGRA is still running at maximum efficiency and throughput performance. Analysis is done to estimate what this throughput performance is.

At this point, it is possible to obtain a CGRA-side acceleration factor for enabling multi-threading for this set of benchmarks, disregarding any other factors. These speed-ups compared to a single-threaded CGRA are shown in Figure 8.3. In the best

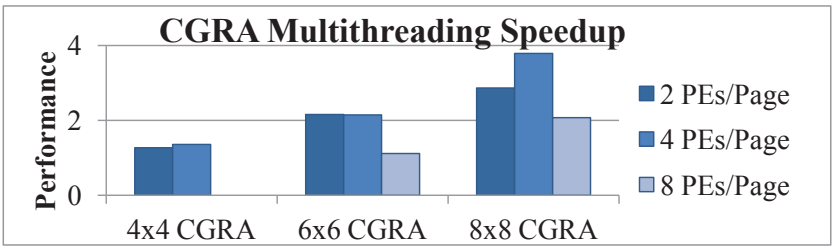

Figure 8.3: *Speedup of a multi-threaded CGRA using paging over that of a single-threaded CGRA when running at maximum theoretical throughput for different page sizes and CGRA sizes. This indicates that a page size of 4 is expected to achieve the most speedup under multi-threaded loads.*

case, performance increases by over 280%. Note the similarities in trends between Figure 8.2 and Figure 8.3. This trend indicates that in a system simulation, a page size of 4 PEs/page should be expected to perform the best in multi-threading mode. In addition, as seen in Figure 8.1, a page size of 4 PEs/page is expected to perform the best in a single-threaded environment also. This makes a page size of 4 PEs the preferred size. These conclusions will be again validated in the system benchmarks presented later in Section 9.3 on page 50.

Chapter 9

MULTI-THREADING CASE STUDY

Up to this point, only the CGRA-side of performance has been analyzed. This is not the same as whole system performance. There are many factors that can effect the system performance for a system with a CGRA. This analysis is explained.

9.1 Multi-threading Performance Factors

Multi-threading vs Single-threading

As already discussed, CGRA utilization is a major factor in multi-threading performance. Also shown is that utilization for CGRAs, especially larger CGRAs, is low. Thus, by enabling multiple threads to run on the CGRA, utilization is increased. It is therefore reasonable to expect an increase in performance by enabling multiple threads access to a CGRA.

The performance benefits of adding a CGRA in a single-threaded environment can be analytically described. Since in this work only loop kernels are accelerated, the remainder of the thread remains unchanged. Thus, speedup will be described by Ahmdal's Law, which says that speedup of the entire thread is equal to the original run time divided by the sum of the run time of the non-accelerated portion of the thread and the run time of the accelerated portion of the thread. The run time of the accelerated portion of the thread when ran on a CGRA is given by Equation 9.1.

$$RunTime = init + max(DMANeed, II) \times iterations + resultDMA \tag{9.1}$$

In Equation 9.1, *init* is the time it takes to DMA the instructions and the initial data buffer, *DMANeed* is the number of cycles it takes to DMA an iteration of data, *iterations* is the number of iterations the kernel is ran for, and *resultDMA* is the number

of cycles it takes to DMA results back into memory. All of these values can be determined at compile time (since the technique targets an embedded system, specifications such as DMA and CGRA speed should already known). So long as this *RunTime* is less than the time it takes to execute that portion of the thread on the CPU, there is performance benefit to running the thread on the CGRA (power benefit is different and is explained later in Section 9.5). The time it takes to execute the thread on the CPU is possible to estimate, and many works address such techniques [35, 33].

Direct Memory Access (DMA) Issues

An issue of extreme importance is whether the time to set up an initial DMA buffer for the CGRA, execute the schedule on the CGRA, and then execute a DMA memory transfer back will be less than simply executing on a general purpose processor. In addition, a single DMA channel (one in each direction, to and from the CGRA) may preclude the ability of multi-threading. This is because if the time to transfer the initial data buffer to the CGRA is greater than the time to execute on the CGRA, no two threads will ever need to be executed on the CGRA simultaneously. This must be modeled in a simulation; in this work, it is closely examined.

Transform Time

When scheduling kernels to a CGRA in multi-threading mode, there is a time associated if the kernel needs to be shrunk before execution. This transform must be performed on the CPU and is linear in relation to the length of the schedule. Therefore a scheduling policy should be chosen that does not unduly burden the CPU with unnecessary transformations, either of to-be-ran schedules or currently running schedules. CPU time required by transformations can easily be hid during DMA times.

9.2 Multi-threading System Benchmark

Based off this analysis, 3 different systems were modeled and benchmarked. In each system, the CGRA clock speed, DMA bandwidth, and CPU clock speed are modified.

1. *DMA-constrained Dual CPU System:* System with a CGRA clock speed of 350 MHz, dual CPUs running at 800 MHz and a DMA bandwidth of 300 MB/s in each direction.

2. *DMA-constrained Quad-core CPU System:* System with a CGRA clock speed of 500 MHz, 4 CPUs running at 1.8 GHz and a DMA bandwidth of 1 GB/s in each direction.

3. *Non-DMA-constrained Quad-core CPU System:* System with a CGRA clock speed of 500 MHz, 4 CPUs running at 2.5 GHz and a DMA bandwidth of 8 GB/s in each direction.

4. *Optimized Dual-core CPU System:* System with a CGRA clock speed of 600 MHz, dual CPUs running at 800 MHz and a DMA bandwidth of 4 GB/s in each direction. This system is discussed in Section 9.4.

Benchmarking was performed on these systems, modifying the following variables:

1. *Threads*: How many threads are being ran on the system (*1, 2, 4, 8,* or *16*). By varying this metric, the effect of sharing a single CGRA will be illustrated.

2. *Page Size*: The number of PEs/page (*2, 4,* or *8*). By varying this metric, it can be determined what an optimal page size for this set of benchmarks.

3. *CGRA Size*: The size of the CGRA used (*4x4, 6x6,* or *8x8*). By varying this metric, it will be shown the limits of current mapping implementations and the need for multi-threading.

Threads are composed of a random generation of sections of serial and loop-kernel code, each of random lengths/iterations. The following system classes are modeled:

1. *CPU-only*: This system has no CGRAs. Therefore all code must be ran on the CPUs.

2. *Single CGRA System*: This system has a single CGRA with a single DMA channel and 64 KB of data memory.

3. *Many CGRA System*: This system has as many CGRAs as threads, but only a single DMA channel and 64 KB of shared data memory. This system models the expected maximum performance potential for a CGRA system.

4. *Paging CGRA System*: This system has a single CGRA with the ability to multi-thread using paging and a single DMA channel and 64 KB of data memory. Schedule transform times are modeled.

The simulation models active/idle time/usage along with the time spent stalled waiting for each of the following resources:

1. *CGRA*: This refers to the execution structure of the CGRA.

2. *CPU*: This refers to the central processing unit of the system.

3. *DMA*: This is the mechanism by which data is transferred between CGRA memory and system memory.

4. *Data Memory (DMEM)*: This is the local CGRA data memory buffer.

From this, relative performance of each of the system classes is obtained by comparing run times of these systems. A conclusion can then be drawn in which situation a paging CGRA is desirable, along with other system characteristics that are

important. Performance of the implementation of the Pagemaster Algorithm is also judged and other system bottlenecks that inhibit better performance/lower run times.

To perform the simulation, run time characteristics were taken from various systems and system resources. For example, the CPU execution time for each individual benchmark was taken when running a thousand iterations of the benchmark. While this under-estimates the CPU execution time, this is acceptable. Care was taken not to over-estimate any performance, thus assuring results were not skewed in favor of the techniques being tested. In other words, previous established techniques were assumed to be the ideal best case, while tested techniques were modeled in the worst case.

In this light, the average time a CPU took to execute an iteration of a kernel was underestimated, DMA performance was modeled according to specifications, CGRA data needs were overestimated, and transformations in the paging case were always ran on the CPU, whether needed or not. Threads which had received the fewest execution resources at the given time of scheduling were given highest priority.

To understand the system benchmark, the execution of a single thread is explained. A thread is composed of either code that is to be executed on the CPU, or code that must be executed on the CGRA (the proportion breaks down to about 25% serial execution and 75% loop kernel execution). It is executed as follows:

If the next portion of the thread to be executed is to be executed on the CPU, the thread is either scheduled to an available CPU or stalled until a CPU becomes available.

If the next portion of the thread to be executed is targeted for the CGRA, multiple checks must be performed before the thread can be scheduled.

Data Memory (DMEM) Check: An initial buffer must be set up and room for the results of the kernel reserved in the data memory. If there is not sufficient space, the thread must be stalled. The initial buffer size is determined by examining the current DMA load and sized in such a way that the schedule can begin execution on the CGRA

while DMA is still active. This allows overlapping of both DMA and CGRA execution times. This calculation is as simple as determining the data needs of the schedule per iteration and the amount of data that can be DMA'd per iteration on average.

DMA Check: Kernel instructions must also be DMA'd to instruction memory. If there is not enough available DMA bandwidth, the thread is stalled until enough bandwidth is available. For a single-threaded CGRA, DMA of a thread's data cannot begin until the currently running thread completes. It is during these stall times that transforms can be performed in the case of a paging CGRA.

CGRA Check: Once a thread's initial buffer and instructions are DMA'd, execution can begin on the CGRA. If no pages are open for execution in the case of a paging CGRA, the thread is stalled (calculations are done such that the thread is stalled before DMA begins).

At this point, a thread can be scheduled for the CGRA. Once execution completes on the CGRA, CGRA pages and data memory used for buffering are released, but result memory used is held until data can be DMA'd back. If multiple threads complete around the same time, threads DMA data back in a serial fashion on a first come, first serve basis. Once this DMA completes, all data memory for the thread is released.

The same set of randomly generated threads are used in all benchmarks, assuring a fair comparison for all results. This is to say, if it were reasonable, data could be normalized to a single benchmark for all systems and benchmarks and presented in a single graph and be correct.

9.3 Multi-threading Performance Results

To present every case is not practical (there are over *1000* sets). A selected portion of results will be shown to illustrate trends. In each case, the result run times are normalized to the worst case for the subset of results being presented for each system.

For example, a set of graphs showing run time of each system based off the number of threads being executed, the results would be normalized to the case with 16 threads for each of the systems, as this has the longest run time. Once these results are presented, system bottlenecks will be identified and a case study will be performed on an optimized system.

The following results are presented to illustrate the following trends:

1. *CPU-only Results*: A single-threaded CGRA is compared to CPU-only execution of threads. This illustrates the limitations of a single-threaded CGRA compared to a system able to run multiple threads.

2. *Page Size Variation*: A paging CGRA is compared with different page sizes. The trends explored in the previous sections will be shown to support the preferred page size of 4 PEs.

3. *CGRA Size Variation*: A single-threaded CGRA is compared for different size CGRAs along with a single-threaded CGRA compared to a paging CGRA of different CGRA sizes. The first set of results show the meager gains of increasing CGRA size due to constraints that limit II. The second results support the assertion that IPC is additive of the running schedules. Therefore larger CGRA sizes provide lower run times than smaller CGRA sizes.

4. *Threading*: The different CGRA setups are compared against each other as the number of threads are varied. These results help identify resource limitations, whether run time is constrained by the execution ability of the CGRA or some other resource.

5. *Bottlenecks*: These results are an extension of the threading results. The stall times are shown for each resource, explicitly identifying bottlenecks in the system.

6. *System Optimization*: An optimized system is created from trends observed in previous sections. These results indicate the necessary system design points to assure optimal run time.

CPU-only Results Show Benefits of a Multi-threading Ability

There are many difficulties when trying to compare CPU results to CGRA results. While CGRA execution time is entirely deterministic, the general purpose CPU is much more dynamic. Factors such as thread scheduling policy, benchmark optimizations, cache misses, etc, all can greatly change run time. As mentioned, CPU execution times were underestimated in order to not bias results in favor of the presented technique. Therefore, results presented here show only expected trends.

Figure 9.1 shows run time differences between CPU-only execution and execution using a single single-threaded CGRA as the number of threads increases. These graphs illustrate the limitations of a single-threaded CGRA. While run time for a single-threaded CGRA is less in a dual CPU DMA-constrained system, the benefits of multiple CPUs and therefore their multi-threading ability are seen in the DMA-constrained and non-DMA-constrained quad-core systems. As process technology improves, finding a quad-core CPU in an embedded system will not be uncommon [2]. It can also be seen that as DMA bandwidth increases, CGRA usage is able to increase. This trend will prove important, which will be shown in later tests.

A few trends are important to identify here. First, it should be noticed that total CPU active time for a CGRA system is minimal (less than 5%) for all systems. This indicates that the CPU does not greatly change run time in these cases. Instead, DMA bandwidth and CGRA accessibility are more crucial. It will be seen in later sections that a multi-threaded paging CGRA significantly increases accessibility.

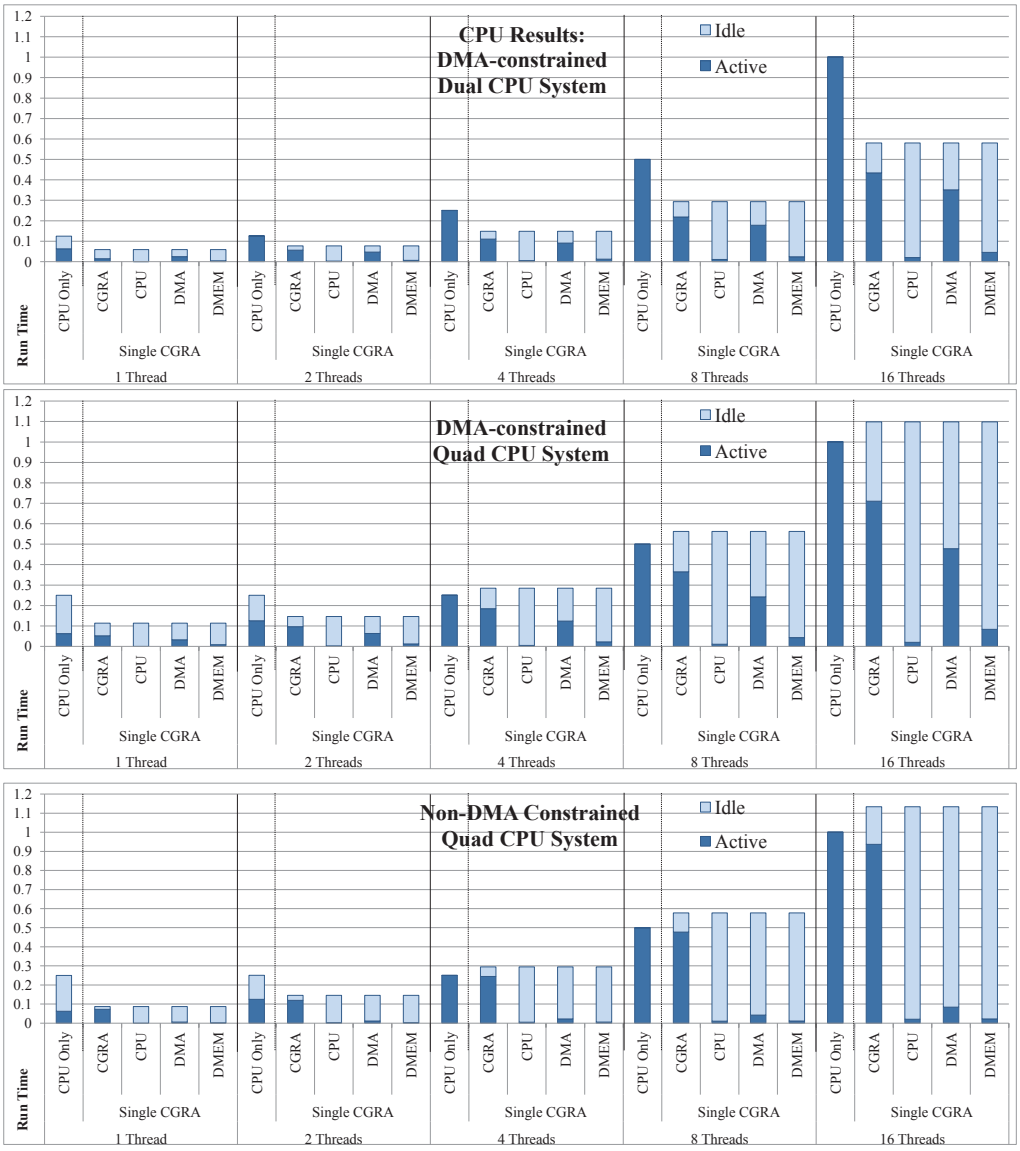

Figure 9.1: *Relative run time of systems using either a CPU or a single-threaded CGRA. For each system, run times are normalized to the run time of the CPU-only system of 16 threads. It is seen that a single-threaded CGRA decreases or achieves equal run time for a small number of threads, but begins to suffer as compared to simply running on a CPU when more threads are added.*

Page Size Greatly Effects Overall Performance

Since not all kernels could be mapped using a page size of 2, page size is difficult to compare. Figure 9.2 shows the effects of non-uniform resource use. A page size of 2 benefits from an inherent division of resources, while the other page sizes must always execute on the CGRA. This is especially evident in the non-DMA-constrained quad-core system, where CPU utilization is nearly maximized and providing better run time than the other page sizes. However, since the goal of this work is to enable better generalized embedded system performance, a quad-core 2.5 GHz processor would be too power consuming. Results presented later will show that nearly the same run time present in the non-DMA-constrained quad-core system can be achieved using less DMA bandwidth and a dual core 800 MHz processor. Important to note also is that in the case of a single thread running on the system, performance (inverse to run time) follows the same trend as shown in Figure 8.1.

For remaining sections, a page size of 4 will be used when illustrating trends. This page size is what is predicted in Section 8.2 in both Figure 8.2 and Figure 8.3 to perform the best.

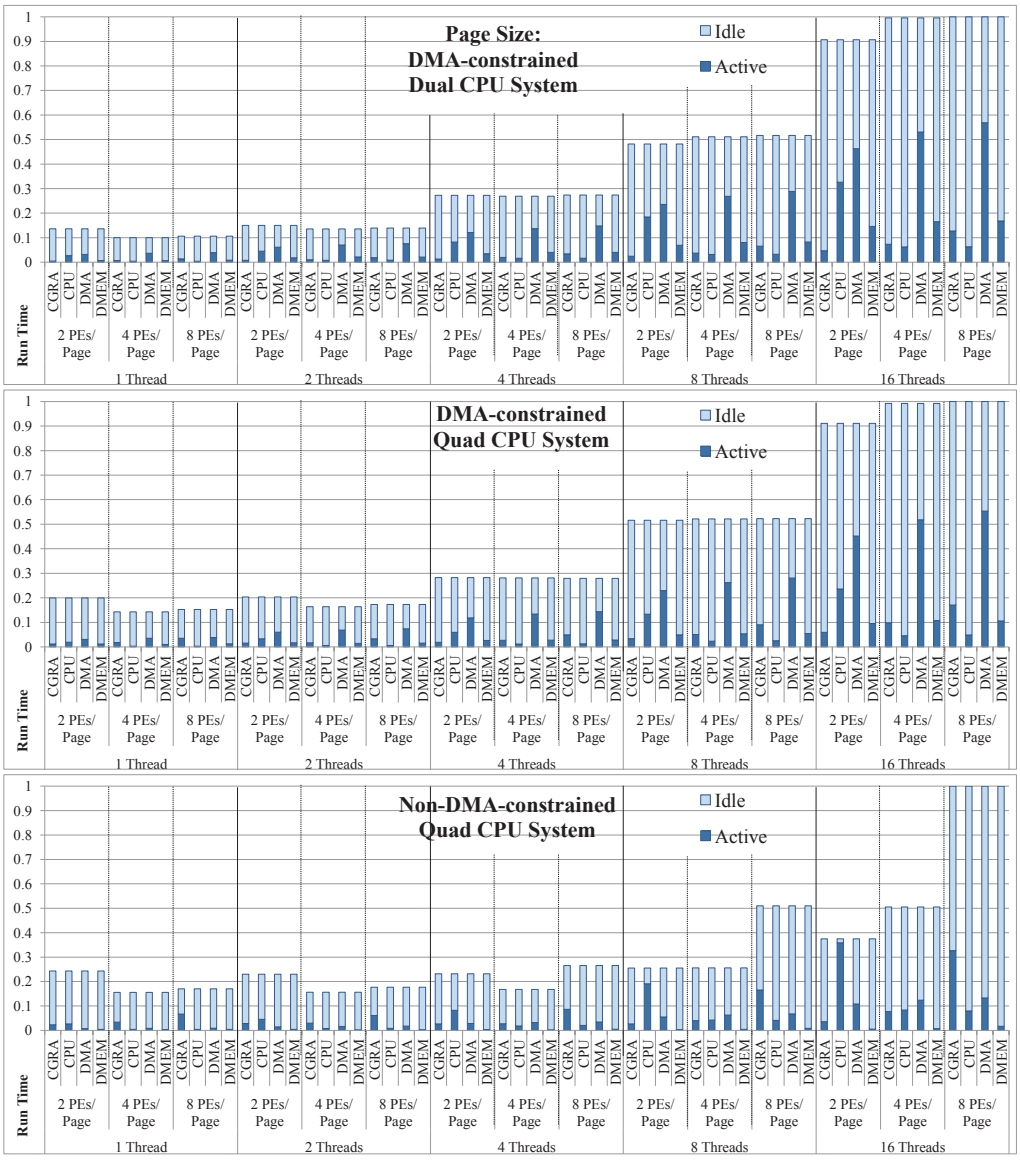

Figure 9.2: *Relative run time of systems using a paging-enabled CGRA for varying page sizes. For each system, run times are normalized to that of 8 PEs/page and 16 threads. It can be seen that a page size of 4 provides the most predictable run time.*

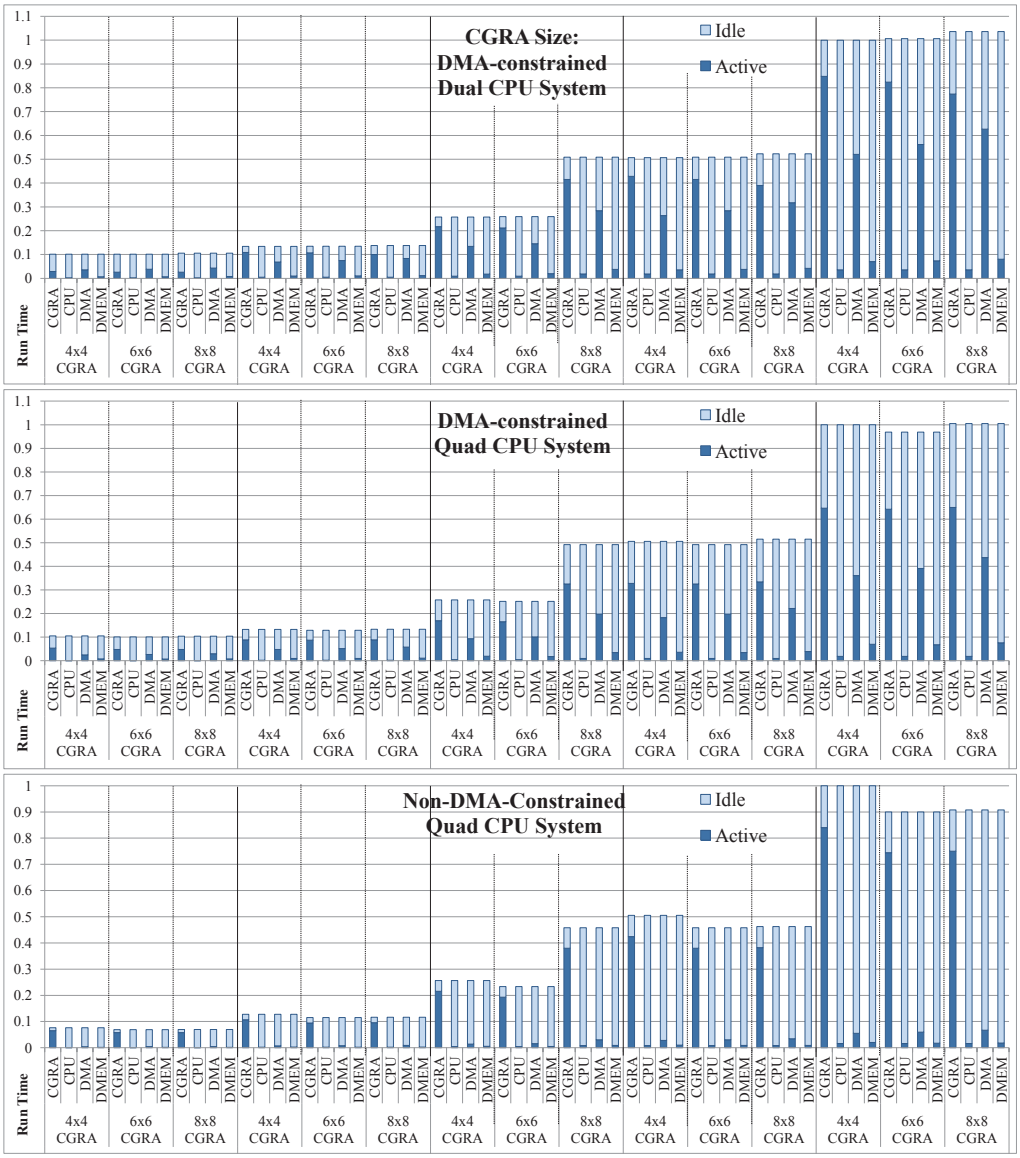

Figure 9.3: *Relative run time of systems using a single-threaded CGRA of varying sizes. For each system, run times are normalized to that of a 4x4 CGRA and 16 threads. It is seen that in single-threaded mode, run time is not decreased by increasing CGRA size.*

Increasing CGRA Size Benefits only Multi-threaded CGRAs

Results presented in this section provide the motivation to enable multi-threading in CGRAs. As can be seen, as long as the CGRA is single-threaded, increasing CGRA size does not decrease run time significantly and, in some cases, can even increase run time due to the increase in instruction size. This is shown in Figure 9.3.

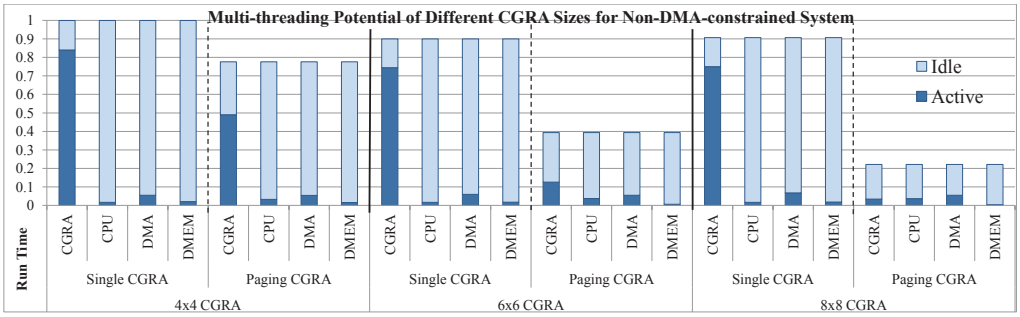

Figure 9.4: *Run time of different CGRA sizes for a single-threaded CGRA vs a paging CGRA in a non-DMA-constrained system running 16 threads. The run time is normalized to a single-threaded 4x4 CGRA. It is seen that by allowing multi-threading through paging, larger CGRA structures are more effectively utilized and decrease run time.*

Figure 9.4 compares a single-threaded CGRA to that of a paging CGRA for different sizes in a non-DMA-constrained system. 16 threads are used. Performance (inverse to run time) for a paging CGRA follows almost exactly the trend seen in Figure 8.3. This illustrates much of the wasted computation potential by not allowing multi-threading.

Increasing Threads Highlight DMA Needs of CGRA

To illustrate the effect the number of threads trying to access the CGRA has, a CGRA size of 8x8 was used, allowing for the most multi-threading potential when paging is enabled.

It can be seen in DMA-constrained systems, a paging CGRA is able to achieve the same run time of many CGRAs (An unintended benefit of paging for DMA-constrained systems is that by shrinking the instruction size by not loading instructions for unused pages, DMA time is decreased). In the non-DMA-constrained system paging provides near-ideal run time improvements as that of many CGRAs.

It should also be noted that the CPU time used by transformation of threads is trivial and do not produce visibly significant use (it accounts for less than 2 percent of total CPU time, or a 60 percent increase in CPU active time). Thus by slightly increasing CPU use, run time equal to that of 16 CGRAs is able to be obtained using only a single CGRA with paging enabled. This indicates both the limitations of DMA and the effectiveness of paging.

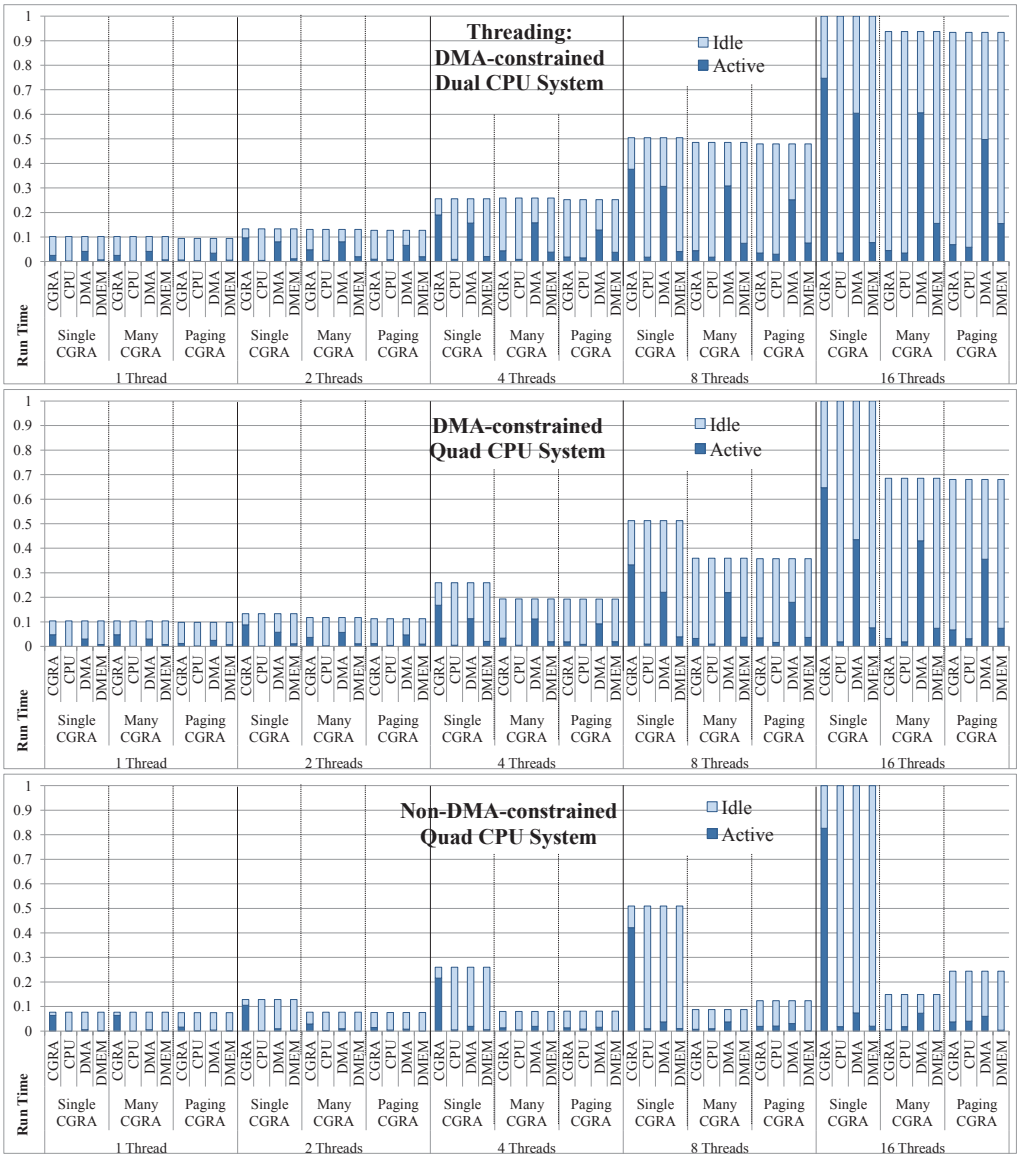

Figure 9.5: *Relative run time for systems with a single-threaded CGRA vs many CGRAs vs a multi-threaded paging CGRA. For each system, run times are normalized to that of a single-threaded CGRA and 16 threads. It can be seen that for DMA-constrained systems, a paging CGRA achieves equal run time to that of many CGRAs, while near equal run time to that of many CGRAs.*

9.4 Designing an Optimized System

To design an optimized system, first current system bottlenecks must be identified then resources strategically apportioned.

Bottlenecks

Figure 9.6 shows the average stall times of each thread for given resources in different scenarios. The stall times are from the benchmarks shown in Figure 9.5 using 16 threads.

A few important characteristics are seen of DMA-constrained systems. Limiting DMA bandwidth has a two-fold effect for multi-threaded systems: the initial buffer size must be larger and completed schedules' data reside in data memory longer. These two effects cause data memory to become full and requires threads to be stalled more often. As seen in the non-DMA-constrained quad-core system, this problem is mitigated by increasing DMA bandwidth. These stall times confirm the results seen in Figure 9.5, where a many-CGRA system and a paging CGRA system provide equal run time in DMA-constrained systems, as both of these systems stall on the resources identical in both systems. In the non-DMA-constrained system, it is seen that a paging CGRA is able to achieve maximum throughput and threads begin to stall waiting for CGRA availability.

An Optimal System Needs Sufficient DMA Bandwidth

Based off the above analysis, a fourth system was designed. This system attempted to achieve similar run times to that of the non-DMA-constrained quad-core system for 16 threads while removing the bottlenecks. This system has an 8x8 CGRA clocked at 600 MHz with dual CPUs at 800 MHz and a DMA bandwidth of 4 GB/s in each direction. A page size of 4 PEs was used.

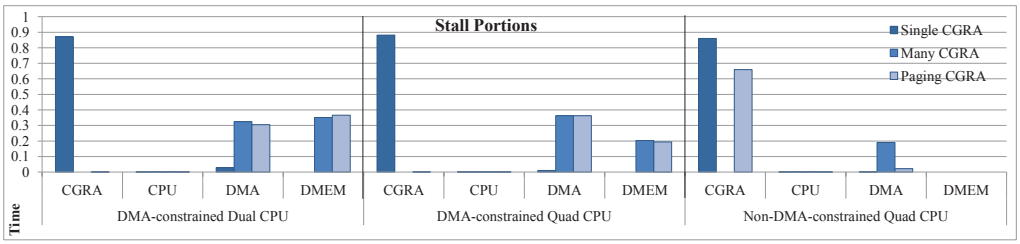

Figure 9.6: *Average time of total run time each thread spent stalled as a percent of total run time using an 8x8 CGRA and a page size of 4 PEs. It is seen that a single-threaded CGRA becomes a bottleneck, and not until sufficient DMA bandwidth is available does a paging CGRA become a bottleneck.*

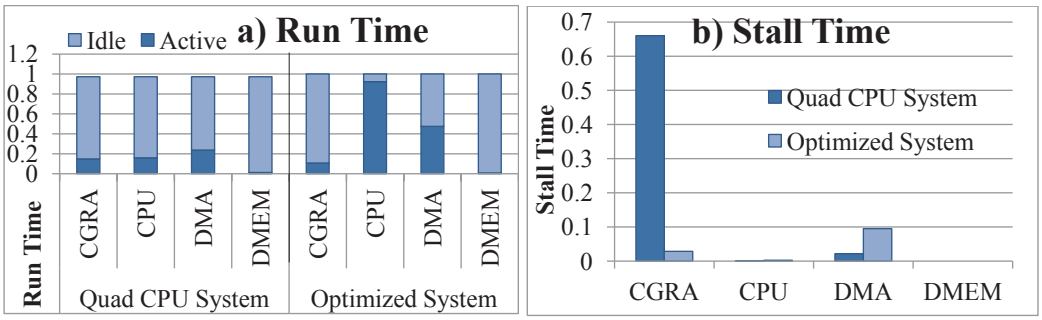

Figure 9.7: *Relative run time of non-DMA-constrained quad-core system vs optimized system for a benchmark with 16 threads. Run time is normalized to to the optimized system. It is seen that the optimized system provides near-equal run time to the non-DMA-constrained quad-core system while removing bottlenecks and being more efficient.*

The results of this simulation are shown in Figure 9.7. The run time for both systems are within 3% of each other, shown in Figure 9.7(a). Additionally, the stall times for each resource are shown in Figure 9.7(b). As can be seen, this fourth system has minimal distributed stall times across all resources.

Optimal System shows expected trends

The overall system statistics for the optimized system are shown. Since this system is optimized, trends are easier to identify. This is shown in Figure 9.8. A few trends need to be noted. First, as threads increase, it becomes clear that a larger CGRA size is beneficial (as noted by decreased run times). As stated before, thread-level parallelism

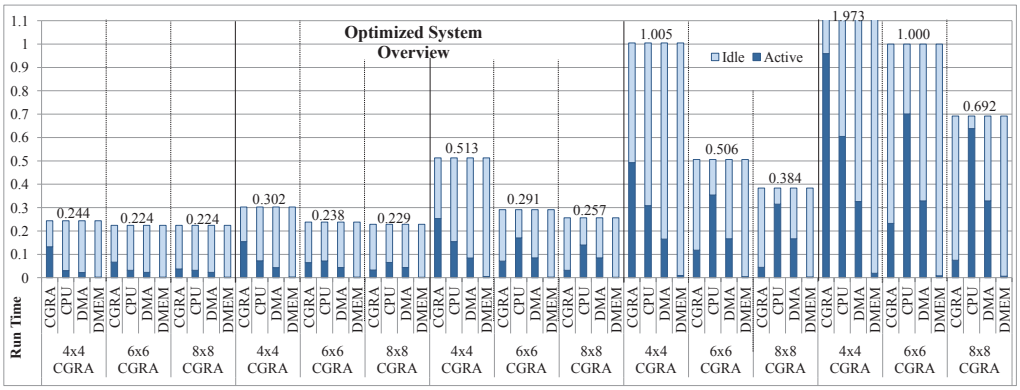

Figure 9.8: *Overview of optimized system run time. Run time is normalized to 16 threads of a 6x6 CGRA. Various trends are present due to the optimization of this system, the most important being a decrease in run time as CGRA size increases.*

is much easier to exploit than instruction-level parallelism. The second trend seen is when each CGRA size reaches its maximum throughput. When the number of threads is doubled, if run time is less than double, the CGRA is not at maximum throughput. Therefore, between 4 threads and 8 threads, a CGRA size of 4x4 reaches its maximum throughput. Between 8 threads and 16 threads, a CGRA size of 6x6 reaches its maximum throughput. These are the same trends identified in Section 8.2.

This system shows a decrease in run time using a multi-threaded CGRA compared to a single-threaded CGRA for a highly threaded environment by almost 350%. In addition, this system is nearly 20x faster than the same system running the workload using only the CPU.

9.5 Power Reduction Analysis

An analysis for power reduction is given. While research has indicated that CGRAs obtain significantly higher power-efficiencies than today's general purpose processors (see Section 1, also [4, 36]), that is under the assumption the CGRA is well utilized. This analysis will approach this from two angles: first showing that CGRA utilization is maximized by enabling multi-threading and that the power consumption for the set of benchmarks in the optimized system for a multi-threaded CGRA is less than a

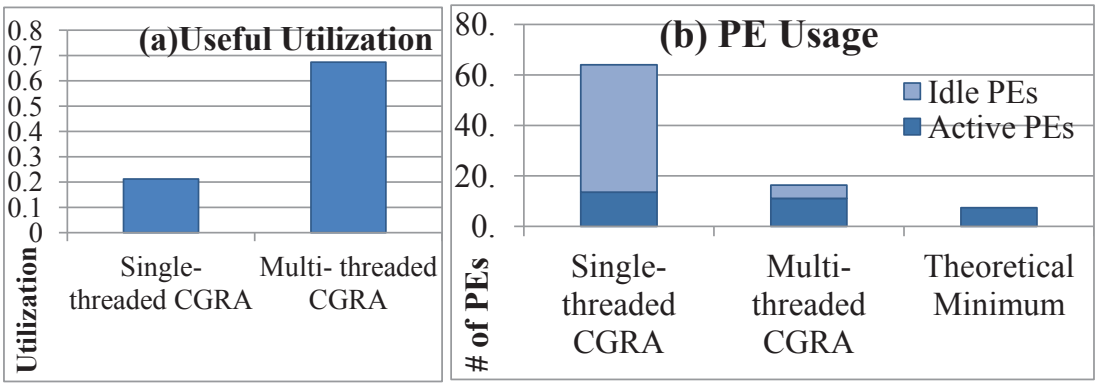

Figure 9.9: *(a) Useful Utilization: A multi-threaded CGRA is able to more effectively use the space it reserves than a single-threaded CGRA (8x8 CGRA). (b) PE Usage: A single-threaded CGRA must use the entire CGRA, leaving many PEs idle. A multi-threaded CGRA can use only a portion of the CGRA, decreasing the number of idle PEs.*

single-threaded CGRA. The analysis is done on an 8x8 CGRA in the optimized system discussed earlier. Execution statistics are extracted from the threads used in the case study just discussed.

Utilization is Maximized

As mentioned in Section 8.2 on page 34, utilization can be measured in multiple ways. In this section, useful utilization is used. Therefore, utilization should be maximized. Shown in Figure 9.9(a) is useful utilization on average for all 20 tested benchmarks; a multi-threaded CGRA using paging maximizes utilization. Utilization is a measure of PEs used in the allotted CGRA space, as given in Equation 9.2. For a single-threaded CGRA, $TotalNumberOfPEs$ is the entire CGRA, or 64 PEs. For a multi-threaded CGRA, $TotalNumberOfPEs$ is however many pages the schedule is compiled for multiplied by the number of PEs per page. The reasons for such low utilization for a single-threaded CGRA is discussed in Section 2.2 on page 8.

$$Utilization = \frac{NumberOfActivePEs}{TotalNumberOfPEs} \tag{9.2}$$

Figure 9.9(b) shows the number of PEs used on average for the 20 benchmarks. Since a single-threaded CGRA is forced to use the entire CGRA (64 PEs) for a schedule, but is limited in both the size of the DDG it is mapping to the CGRA and the II of that DDG, many of the PEs remain idle. For a multi-threaded CGRA, a custom number of pages can be used; so while both CGRAs use nearly the same number of active PEs, a multi-threaded CGRA can greatly reduce the number of idle PEs (note that if only a single thread is ever ran on the CGRA, the remaining PEs in a multi-threaded CGRA will necessarily remain idle). Also shown is the theoretical minimum number of PEs a schedule can use for this set of benchmarks. This assumes that no routing PEs are ever added. It also assumes that the achieved II is the minimum. Therefore, the theoretical minimum number of active PEs is given by Equation 9.3. The average theoretical minimum number of active PEs is the average across all benchmarks.

$$TheoreticalMinimum = \frac{NumberOfNodes}{II} \tag{9.3}$$

As an example, the Tri-Diagonal Elimination benchmark will be used, which has 12 nodes. The theoretical minimum number of PEs for this benchmark is 6, as an II of 2 is achieved. The method for determining utilization for a single-threaded and multi-threaded CGRA is as follows:

Single-Threaded CGRA: The benchmark is scheduled to the CGRA. This requires a total 21 nodes (an additional 9 routing nodes) and achieves an II of 2 while using the entire CGRA (64 PEs). Therefore, there are 128 total PE executions available, but only 21 of them are used. This creates a utilization of about 16.4%.

Multi-Threaded CGRA: The benchmark is scheduled to the CGRA. This requires a total 17 nodes (an additional 5 routing nodes) and achieves an II of 2 while using 3 pages (12 PEs). Therefore, there are 24 total PE executions available, but only 17 of them are used. This creates a utilization of about 70.8%.

This analysis indicate that by enabling multi-threading through space-multiplexing, a CGRA is more effectively utilized. Therefore the CGRA itself is running at a higher power efficiency (MOPs/mW).

Overall Power is Reduced

When analyzing power/energy consumption of a multi-threaded verse a single-threaded CGRA, a few factors need to be explored. First, since both CGRAs execute the same code, data needs are identical. In fact, a multi-threaded CGRA will transfer the same or less data than a single-threaded CGRA by DMA (due to the decreased instruction size). Therefore, the DMA energy needs will be equal or less for a multi-threaded CGRA. While slightly more processor time is used by the multi-threaded CGRA, the total execution time is reduced enough for this to be more than canceled out. The same number of loads and stores to data memory are performed in either case. Therefore, the only significant power difference between a multi-threaded and a single-threaded CGRA is PE usage. This will be explored.

As mentioned in Section 9.5, useful utilization is increased for a multi-threaded CGRA. In addition, for this set of benchmarks, the number of active PEs per iteration is nearly the same on average. It is therefore safe to conclude, at a high level, that a multi-threaded CGRA will not increase energy usage. This is because the multi-threaded CGRA executes in the same or less time than a single-threaded CGRA and performs nearly the same operations during that time. Therefore, in both cases, nearly the same amount of power is consumed by active PEs, while the single-threaded CGRA has additional power consumed by additional idle PEs.

In this work, McPAT [34] was used to model the CGRA PE power. A PE is assumed to contain an Integer ALU (add, subtract, shift, etc), a complex ALU (multiply, divide), and a register file. These components were modeled using a 32 nm technology. To determine total power, the average use of the listed components was determined in

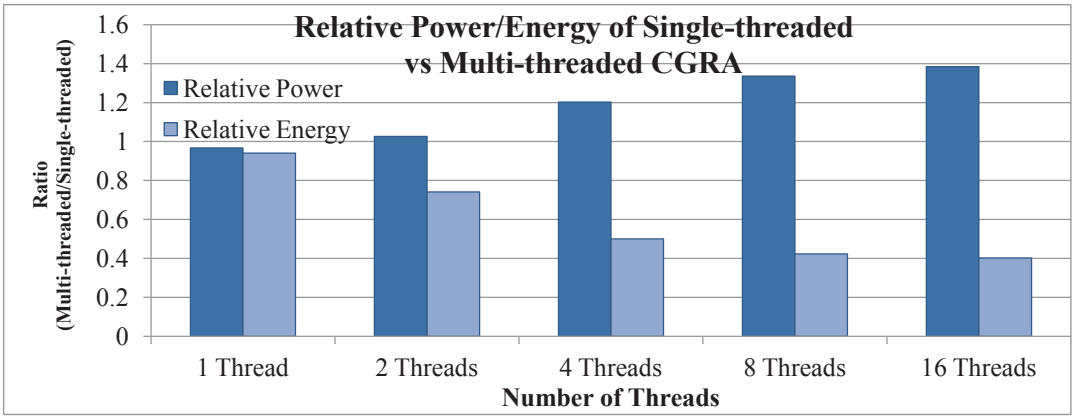

Figure 9.10: *Relative Power/Energy of the PEs in a single-threaded CGRA verse a multi-threaded CGRA. While a multi-threaded CGRA increases the relative power in highly threaded environments because it has more active PEs, the total energy is significantly reduced due to the decrease in run time.*

both the single-threaded and multi-threaded CGRA. This gives the average power of the CGRA PEs. This is then multiplied by execution time to determine the energy use of the CGRA PEs. Since both CGRAs execute the same number of useful operations, the CGRA using less energy is more power efficient.

Figure 9.10 shows both the relative power and relative energy used by the PEs in a single-threaded verse a multi-threaded CGRA on average for different number of threads. As can be seen, the actual power consumption of the multi-threaded CGRA is increased at any given moment for multiple threads, but at the same time, it is accomplishing on average almost $4x$ as much work. Therefore, it is more power efficient. This efficiency can be also seen in Figure 9.10, which also shows the energy consumption of the PEs through the entire execution time. Note that enabling multi-threading does not increase relative energy consumption in any case.

To model a CGRA system verse a CPU-only system, a few factors have to be explored. In this work, only a rough sketch of energy is done. The data memory power of the CGRA was estimated using CACTI [26]. The CPU core (excluding the cache/scratchpad memory) was modeled using McPAT. These values were used to

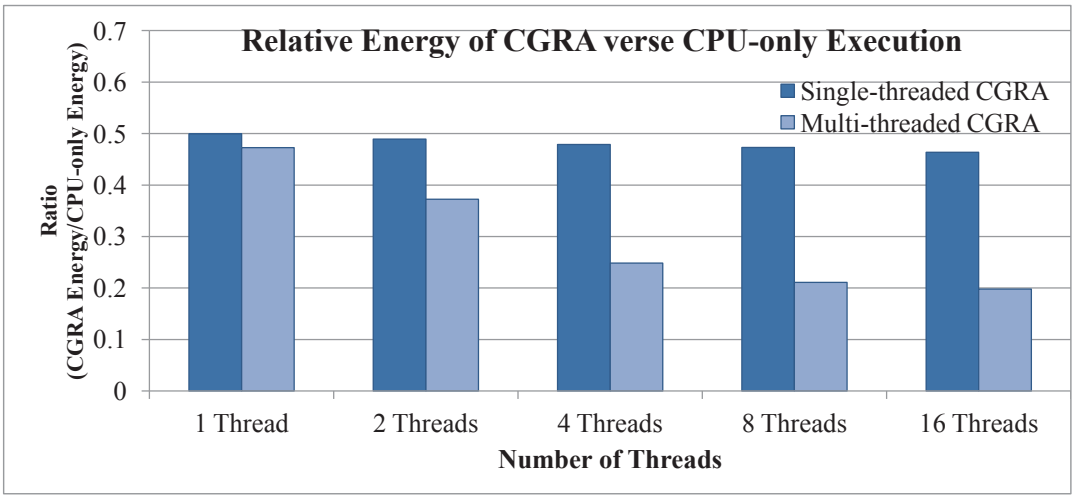

Figure 9.11: *Relative energy of a CPU-only system verse a CGRA system is shown. As can be seen, a CGRA provides higher energy efficiency than CPUs, and a multi-threaded CGRA provides additional benefits as the number of threads increase.*

model active and idle power usage. This was multiplied by execution time in order to obtain energy. Care was taken to assure CPU power is not over-estimated for the CPU-only system. Therefore CGRA power benefits are under-estimated. The results of this analysis is shown in Figure 9.11. While this is a rough analysis, significant energy benefits are still expected for using the CGRA architecture.

Chapter 10

FUTURE WORK

There is still much work that can be done for CGRAs. The research field is young, especially in relation to use of CGRAs as co-processors in general purpose systems. In addition to obtaining exact power statistics for the tested systems, there exists still more work to be done with each of the presented frameworks.

10.1 Compiler Framework

The compiler framework provided with this work has much room for improvement. As stated before, the framework provided focused on functionality, but not on features or optimization. It stood as a proof of concept.

Automatic Identification of Eligible Loops: In this work, an OpenMP-like implementation was created. This meant identifying the desired loops to be compiled on the CGRA by the user. The identified loops were not checked for eligibility for CGRA acceleration effectiveness. However, if a cost-benefit analysis were created, then loops could automatically be identified for CGRA acceleration (GCC already has a structure identifying all loops in the code). Such a feature would then allow for seamless integration of CGRAs into systems without user knowledge.

Code Optimizations Related to CGRA Architecture: No optimizations on generated gimple code by GCC were made in this work. However, certain concepts are not necessary in CGRAs. An example are temporary variables. CGRAs store variables in the communication fabric while routing for data-dependency reasons. If temporary variables were identified and eliminated, the data-dependency graph would be simplified, allowing for more optimal mappings. In addition, many of the DDG branches used for array address calculation can be merged were appropriate, further decreasing DDG size.

Predication: This work did not address control-flow graphs. However, CGRAs have hardware that allows for predication, and mapping techniques can model this control flow in the data-dependency graph. This would expand the scope of loops that can be accelerated by CGRAs.

Automated Identification of Read-only and Modified Data: The simulator used in this work had the ability of copying only a portion of the data back from the CGRA. However, since no mechanism was used to identify which data was written to, the user specified this. Time and energy could be saved by decreasing the amount of data copied without relying on user input.

10.2 Multi-threading Framework

Additional testing and case studies can be performed on the multi-threading framework:

Use with Additional Mapping Algorithms: This work only used EMS as an algorithm. However, any algorithm that allows for custom interconnect topologies can work with the multi-threading framework. It is a pertinent research question to know the effect of the this framework on these algorithms.

Runtime Scheduling Algorithm: A simple, almost naive, scheduling algorithm was used for determining how to schedule additional kernels on a CGRA. More complex and well-thought algorithms can be used (such as analyzing expected kernel run times before transforming the kernel), providing additional performance improvement. Such a scheduler should be integrated into the operating system kernel or device driver.

System Testing: Still more system testing can be performed. The effect of having threads that communicate between each other should be performed. Different ways in which kernel types are scheduled should be explored. For example, trying to schedule DMA-intensive kernels with compute-intensive kernels should provide maximal benefit when compared to DMA-intensive only or compute-intensive only kernels.

Chapter 11

CONCLUSION

By providing a compiler framework, CGRAs can be more thoroughly tested for general purpose use. The paradigm used in this work is promising, and has proven successful in many other similar architectures. The lack of a complete compiler framework has limited this success for CGRAs. However, this work shows that it is not only possible, but practical, and much of the same research that has been used in general purpose compilers can be done in relation to CGRAs.

The addition of the ability of multi-threading on CGRAs proves to be extremely effective in all systems. Performance improvements of almost 350% were shown in system performance when performance was not DMA-constrained. In DMA-constrained systems, performance benefits are still seen, though not of the same magnitude. In an embedded system, however, the CGRA can be integrated into a SoC design, providing the necessary DMA bandwidth.

This work shows CGRAs use as co-processors to general purpose processors. This is a desirable paradigm, providing performance and power-efficiency that promises scalability. A compiler framework allows testing of large, already existing code basis for use in this paradigm, while multi-threading boosts CGRA utilization and therefore performance.

BIBLIOGRAPHY

[1] M. Ahn et al. A spatial mapping algorithm for heterogeneous coarse-grained reconfigurable architectures. In *DATE '06*, 2006.

[2] ARM-A9 Datasheet.

[3] J. Becker and M. Vorbach. Architecture, memory and interface technology integration of an industrial/academic configurable system-on-chip (csoc). In *ISVLSI'03*, 2003.

[4] F. Bouwens, M. Berekovic, and et. al. Architectural exploration of the adres coarse-grained reconfigurable array. 2007.

[5] D. C. Chen. *Programmable Arithmetic Devices for High Speed Digital Signal Processing*. PhD thesis, EECS Department, University of California, Berkeley, 1992.

[6] Tesla S2050 Gpu Computing System.

[7] G. Dimitroulakos et al. Resource aware mapping on coarse grained reconfigurable arrays. *Microprocess. Microsyst.*, 2009.

[8] G. Dimitroulakos, M. Galanis, and C. Goutis. A compiler method for memory-conscious mapping of applications on coarse-grained reconfigurable architectures. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, page 4 pp., 2005.

[9] C. Ebeling et al. Mapping applications to the rapid configurable architecture. In *FCCM '97*. IEEE Computer Society, 1997.

[10] J. eun Lee et al. Compilation approach for coarse-grained reconfigurable architectures. *Design Test of Computers, IEEE*, 2003.

[11] S. Friedman et al. Spr: an architecture-adaptive cgra mapping tool. In *FPGA '09*. ACM, 2009.

[12] S. C. Goldstein et al. Piperench: a co/processor for streaming multimedia acceleration. In *ISCA '99*, 1999.

[13] R. Hartenstein. A decade of reconfigurable computing: a visionary retrospective. In *DATE '01*. IEEE Press, 2001. 65

[14] R. Hartenstein and R. Kress. A datapath synthesis system for the reconfigurable datapath architecture. In *ASP-DAC '95*. ACM, 1995.

[15] A. Hatanaka and N. Bagherzadeh. A modulo scheduling algorithm for a coarse-grain reconfigurable array template. In *Parallel and Distributed Processing Symposium,2007. IPDPS 2007. IEEE International*, pages 1 –8, 2007.

[16] Intel N550 Datasheet.

[17] W. Kehuai, J. Madsen, A. Kanstein, and M. Berekovic. Mt-adres: Multithreading on coarse-grained reconfigurable architecture. 2006.

[18] Y. Kim et al. Resource sharing and pipelining in coarse-grained reconfigurable architecture for domain-specific optimization. In *DATE '05*. IEEE Computer Society, 2005.

[19] Y. Kim, R. Mahapatra, and K. Choi. Design space exploration for efficient resource utilization in coarse-grained reconfigurable architecture. In *Transactions on VLSI Systems*. IEEE Press, 2010.

[20] B. Mei et al. Dresc: a retargetable compiler for coarse-grained reconfigurable architectures. 2002.

[21] B. Mei et al. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. In *DATE '03*. IEEE Computer Society, 2003.

[22] B. Mei et al. Mapping an h.264/avc decoder onto the adres reconfigurable architecture. 2005.

[23] B. Mei, S. Vernalde, D. Verkest, and R. Lauwereins. Design methodology for a tightly coupled vliw/reconfigurable matrix architecture: A case study. In *Proceedings of the conference on Design, automation and test in Europe - Volume 2*, DATE '04, pages 21224–,Washington, DC, USA, 2004. IEEE Computer Society.

[24] E. Mirsky and A. DeHon. MATRIX: A reconfigurable computing architecture with configurable instruction distribution and deployable resources. In *IEEE Symposium on FPGAs for Custom Computing Machines*, 1996.

[25] T. Miyamori and K. Olukotun. REMARC: Reconfigurable multimedia array coprocessor (abstract). In *FPGA*, 1998. 66

[26] N. J. N. Muralimanohar, R. Balasubramonian. Cacti 6.0: A tool to model large caches. In *International Symposium on Microarchitecture*, 2009.

[27] H. Park et al. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In *PACT '08*. ACM, 2008.

[28] H. Park et al. Polymorphic pipeline array: a flexible multicore accelerator with virtualized execution for mobile multimedia applications. In *MICRO 42*. ACM, 2009.

[29] H. Park, K. Fan, M. Kudlur, and S. Mahlke. Modulo graph embedding: mapping applications onto coarse-grained reconfigurable architectures. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, CASES '06, pages 136–146, New York, NY, USA, 2006. ACM.

[30] Y. Park et al. Cgra express: accelerating execution using dynamic operation fusion. In *CASES '09*. ACM, 2009.

[31] Y. Park, H. Park, and S. A. Mahlke. Cgra express: accelerating execution using dynamic operation fusion. In *CASES'09*, pages 271–280, 2009.

[32] B. R. Rau. Iterative modulo scheduling: an algorithm for software pipelining loops. In *MICRO 27*. ACM, 1994.

[33] B. Reistad and D. K. Gifford. Static dependent costs for estimating execution time. In *Proceedings of the 1994 ACM conference on LISP and functional programming*, LFP '94, pages 65–78, New York, NY, USA, 1994. ACM.

[34] e. a. S. Li, J. Ahn. Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *MICRO'09*. ACM, 2009.

[35] M. C. e. a. S. Mera, P. Lopez-Garcia. Towrads execution time estimation in abstract machine-based languages. In *PPDP'08*. ACM, 2008.

[36] H. Singh et al. Morphosys: An integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Trans. Comput.*, 2000.

[37] S. Vassiliadis et al. Adres & dresc: Architecture and compiler for coarse-grain reconfigurable processors. In D. S. Stamatis Vassiliadis, editor, *Fine- and coarsegrain reconfigurable computing*. 2007. 67

[38] J. Yoon et al. Spkm : A novel graph drawing based algorithm for application mapping onto coarse-grained reconfigurable architectures. 2008.