A Software Scheme for Multithreading on CGRAs

JARED PAGER, REILEY JEYAPAUL, and AVIRAL SHRIVASTAVA,
Compiler Microarchitecture Lab, Arizona State University

Recent industry trends show a drastic rise in the use of hand-held embedded devices, from everyday applications to medical (e.g., monitoring devices) and critical defense applications (e.g., sensor nodes). The two key requirements in the design of such devices are their processing capabilities and battery life. There is therefore an urgency to build high-performance and power-efficient embedded devices, inspiring researchers to develop novel system designs for the same. The use of a coprocessor (application-specific hardware) to offload power-hungry computations is gaining favor among system designers to suit their power budgets. We propose the use of CGRAs (Coarse-Grained Reconfigurable Arrays) as a power-efficient coprocessor. Though CGRAs have been widely used for streaming applications, the extensive compiler support required limits its applicability and use as a general purpose coprocessor. In addition, a CGRA structure can efficiently execute only one statically scheduled kernel at a time, which is a serious limitation when used as an accelerator to a multithreaded or multitasking processor. In this work, we envision a multithreaded CGRA where multiple schedules (or kernels) can be executed simultaneously on the CGRA (as a coprocessor). We propose a comprehensive software scheme that transforms the traditionally single-threaded CGRA into a multithreaded coprocessor to be used as a power-efficient accelerator for multithreaded embedded processors. Our software scheme includes (1) a compiler framework that integrates with existing CGRA mapping techniques to prepare kernels for execution on the multithreaded CGRA and (2) a runtime mechanism that dynamically schedules multiple kernels (offloaded from the processor) to execute simultaneously on the CGRA coprocessor. Our multithreaded CGRA coprocessor implementation thus makes it possible to achieve improved power-efficient computing in modern multithreaded embedded systems.

Categories and Subject Descriptors: C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors), Parallel processors; C.1.4 [**Processor Architectures**]: Parallel Architectures, Mobile processors; C.3 [**Special-Purpose and Application-Based Systems**]: Microprocessor/Microcomputer Applications; C.4 [**Performance of Systems**]: Design Studies

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: CGRA, multithreading, power efficiency, embedded system, runtime transformation, scheduling, compiler framework

ACM Reference Format:
Jared Pager, Reiley Jeyapaul, and Aviral Shrivastava. 2015. A software scheme for multithreading on CGRAs. ACM Trans. Embedd. Comput. Syst. 14, 1, Article 19 (January 2015), 26 pages.
DOI: http://dx.doi.org/10.1145/2638558

This work was partially supported by funding from National Science Foundation grants CCF-0916652 and CCF-1055094 (CAREER).
Authors' addresses: J. Pager, R. Jeyapaul, and A. Shrivastava, Compiler Microarchitecture Lab, Arizona State University, Tempe, Arizona 85281; emails: {jppager, Reiley.Jeyapaul, Aviral.Shrivastava}@asu.edu.

DOI: http://dx.doi.org/10.1145/2638558

1. INTRODUCTION

In recent years, as the time and power budget to process large applications on embedded devices has decreased, the need for power-efficient computing has become an undeniable reality. Smartphones and tablets today execute multiple applications (e.g., Facebook updates, multiple email clients, weather updates, news updates, music player, etc.) simultaneously, which are all performance- and power-hungry applications that compete for processor, display, GPU, and network resources. In the attempt to enable and enhance modern mobile computing applications, the battery life of such devices becomes the limiting factor. Battery size determines the dimensions, weight, and usability of modern and smart handheld devices. For example, in the *Apple iPhone 4*, the battery alone is 40% of the device weight, occupies 36% of the device volume, and allows only 7 hours (over 3G) of talk time. Here, power efficiency directly translates into system weight, recharge time, and processing frequency of the device.

In an application, specific power-hungry code segments (kernels) can be offloaded into specialized hardware that executes those segments in a power-efficient manner. For example, media-based SIMD tasks on large chunks of data can be offloaded to a GPU for power-efficient graphics processing. Such hardware, called accelerators, improves the performance of the system by allowing the processor core to perform its tasks while executing the offloaded code faster (and on a low-power architecture). The Intel MMX unit (OpenCL) and NVIDIA GPU (CUDA) are examples of such accelerators used in modern computing systems. Coarse-Grained Reconfigurable Arrays (CGRAs) are extremely power-efficient accelerator processors used as accelerator hardware in streaming applications [Liang and Huang 2009].

ADRES, a popular CGRA architecture, operates at an efficiency of up to 40MOPS/mW in the 90nm technology node [Bouwens et al. 2007]. This is compared to the Intel Atom N550, which provides about 4.6GOPS of computation while consuming a maximum 8.5W of power [Intel-N550 2010] at an efficiency of about 0.54MOPS/mW. On the other hand, a generalized CGRA architecture has been estimated to be able to achieve power efficiencies of 10 to 100MOPs/mW [Singh et al. 2000]. In addition, the architecture itself is very flexible, allowing for quick reconfigurability. This power efficiency of around an order of magnitude more than that of general-purpose CPUs and flexible reconfigurability is available only in CGRAs, making them an attractive solution for power-efficient coprocessors to be used in modern embedded systems.

Traditionally, CGRAs have been used for streaming applications in extremely embedded systems (e.g., applications like smart TV, routers, etc.), where computing needs are more deterministic for a very narrow domain of applications [Liang and Huang 2009]. This contrasts from a general-purpose multitasking embedded system where any arbitrary task can be run in conjunction with any number of other tasks that are all dynamically scheduled. In order to apply a CGRA in such multitasking environments, several tools and capabilities must be made available at both the hardware and software layers of design. Current and previous research in CGRAs has focused on the CGRA architecture itself and on compilation techniques to map code (kernels) on a CGRA architecture to be processed efficiently. Mapping refers to the process of taking the operations (of a kernel) and placing them spatially and temporally on individual compute nodes in the CGRA processor. This process is relatively complicated, and ongoing research aims to improve the efficiency and effectiveness of such methodologies, thereby expanding the applicability of CGRAs to the ever-increasing range of applications using smart embedded systems. In this, a pertinent research problem explored has been to improve CGRA utilization by identifying additional parallelization opportunities within the code. Reduced CGRA utilization (effective number of compute

nodes used during kernel computation) translates into power-efficient utilization of the computing power consumed, and thereby reduced overall power consumption for the same application processing.

There still exists a growing need for improved tools and capabilities to ensure widespread use of CGRAs in modern systems. Researchers in the past decade have developed a wide array of application mapping techniques and an arsenal of CGRA processor designs to be used as coprocessors for single-threaded embedded systems. A recent paradigm shift in the embedded computing domain has seen a drift toward multithreaded programming and possible extraction of parallelism at the thread level, as in multithreaded and multicore processors (e.g., ARM A9, etc.). Among coprocessors, the NVIDIA CUDA processor boasts the ability to handle thousands of threads simultaneously and deliver hundreds of gigaflops of computation power [CUDA-fermi 2010]. However, to be able to extract this maximum computing potential in such accelerators, an application mapping framework is required (CUDA framework for NVIDIA GPUs). If CGRAs were to be used as a coprocessor for a multithreaded system, extensive compiler and application mapping support is required. To the best of our knowledge, no such general framework exists that can enable multithreaded kernels to be accelerated simultaneously on a single CGRA coprocessor.

In this work, we envision the use of multithreaded CGRAs as a coprocessor (or accelerator) for general-purpose multithreaded embedded processors and propose a comprehensive software framework to solve the multithreading limitations in CGRAs' utilization. Our software framework includes a compilation technique that integrates with most existing CGRA mapping techniques and a runtime transformation scheme that enables dynamic scheduling of multiple kernels onto a single CGRA. In this, each thread executed on the general-purpose processor (GPP) offloads its respective kernels onto the CGRA for power-efficient execution. Our runtime transformation dynamically schedules these kernels to execute on the single CGRA simultaneously, ensuring power-efficient resource utilization. The implemented software framework is experimented over a cycle-accurate CGRA simulator setup that extracts the system performance and utilization metrics, which are presented in our experiments. In this work, we concentrate specifically on achieving maximum possible performance with minimum CGRA resource utilization (which translates into lower power consumption in the CGRA), thereby achieving power-efficient multithreading in CGRAs. The developed features and contributions of this work are:

(1) *Compiler Framework:* We develop a compiler framework that enables multithreading on CGRAs. Our multithreading compiler methodology is essentially a wrapper over existing application-to-CGRA mapping compilers specific to the target CGRA architecture. Our compiler methodology is therefore independent of the underlying CGRA hardware and is therefore a *completely software-only* scheme with a broad range of applicable CGRA architectures.

(2) *Runtime Transformation:* We develop a fast runtime transformation algorithm that performs dynamic scheduling of kernels (offloaded for acceleration) on the multithreaded CGRA. The runtime transformation is the second part of our software framework that interacts with the compiler instrumentation set by our *compiler framework* during the CGRA compilation phase. Our runtime methodology allows multiple threads executed on the multithreaded embedded processor to extract acceleration from the multithreaded CGRA simultaneously, with increased power efficiency.

(3) *Performance and Power Analysis of the Multithreaded CGRA:* For our experiments, we design a simulation setup that simulates the execution of the CGRA architecture considered, including the runtime transformation algorithm implemented, thereby simulating its multithreaded operation. An analysis of the performance

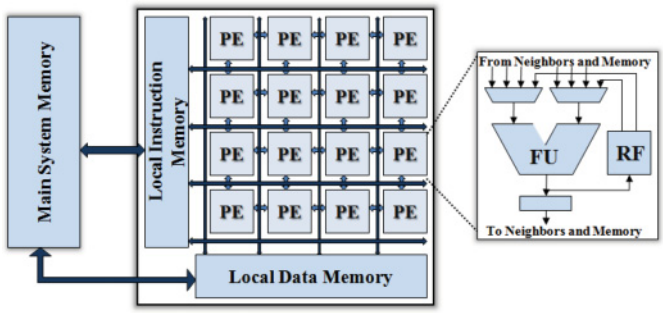

Fig. 1. A basic CGRA architecture, with processing elements (PEs) connected by a mesh fabric, local data memory, and local instruction memory. An expanded view of a PE is displayed, showing the input multiplexers and the register file (RF) structure.

and power consumption in the multithreaded CGRAs is demonstrated through the evaluation of system runtime and *CGRA utilization* (which improves by 2.8 times when multithreading is enabled). CGRA utilization is a metric that measures the effective number of CGRA nodes used per cycle, which is a measure of the amount of useful energy consumed during CGRA operation for that application. Improved CGRA utilization means reduced power consumption.

(4) *Detailed Case Study on CGRA Multithreading:* We perform design space exploration over the parameters involved in the design of a multithreaded CGRA and also present an optimal system design configuration for increased performance and power efficiency. In summary, our results indicate that in a heavily threaded environment, the system with a multithreaded CGRA (1) shows increased performance by almost 3.5times when compared to a single-threaded CGRA and (2) is over 20 times faster than an identical system with only a CPU and no CGRA accelerator.

2. BACKGROUND AND TERMINOLOGY

2.1. CGRA Architecture Is Simple and Power Efficient

The CGRA architecture is best seen as a class of architectures. Several distinct processors, such as MorphoSys [Singh et al. 2000], ADRES [Mei et al. 2007], RSPA [Kim et al. 2005], and KressArray [Hartenstein and Kress 1995], have been presented over the years, all of which are best classed as CGRAs. Hartenstein [2001] gives a comprehensive summary of many different CGRA architectures. 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 meshed communication network (Figure 1). Each PE contains a rotating register file, and some or all of the PEs have access to a bank of local memory, buffered from the main system memory. Each PE can perform simple operations (such as shift, multiply, add/subtract, bitwise operations, etc.), and some architectures allow for more complex operations (such as CGRA Express [Park et al. 2009]). The interconnect network can be described as one in which neighboring PEs (on all four sides) can communicate with each other through a multiplexer at the input of each PE. The memory bus from the local memory to the PEs usually operates in a similar manner. Which of the multiplexed inputs to use in a computation is configured by the instructions mapped to the PEs. 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 a small loop kernel executed multiple times. The CGRA is responsible for copying data and instructions to and from the main memory.

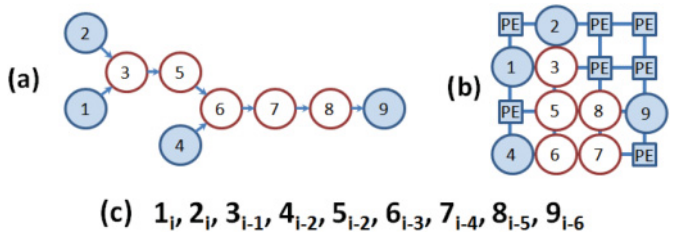

Fig. 2. A basic example of mapping on a CGRA. (a) DFG from a kernel of MPEG2. (b) The DFG mapped onto the CGRA. (c) In order to correctly map the kernel, it must be unrolled and software pipelined.

2.2. Mapping a Kernel onto the CGRA Is Complex

An example of the mapping process is shown in Figure 2. Figure 2(a) describes the DFG (Dataflow Graph) of the MPEG2 kernel (of a loop), where the shaded nodes are load/store operations. In order to extract maximum possible parallelism, the loop is unrolled and software pipelined to allow for one full iteration of the loop kernel to complete execution at the end of every clock cycle. The CGRA mapping process involves two key components: (1) Spatial Mapping (Figure 2(b)), where the nodes to execute on the CGRA are mapped to their respective PEs, based on the data interaction with neighboring nodes and the connectivity allowed by the CGRA network, and (2) Temporal Mapping (Figure 2(c)), where the time that a particular node (of a particular iteration from the unrolled loop kernel) is scheduled to be mapped on its respective location on the CGRA.

2.2.1. Terminology Used. Code eligible for acceleration on a CGRA is referred to as a *kernel* (often the innermost loop code in a nested loop). The process in which instructions of the kernel are assigned specific time slots and PEs on a CGRA is called *mapping*. When mapped onto a CGRA, the *iteration interval (II)* is the number of cycles it takes to complete a single iteration of the kernel. A complete mapping is referred to as a *schedule*. Since a CGRA is statically scheduled and its memory stored in a CGRA memory buffer, performance directly increases as the II decreases.

Optimizing Schedules: The goal of all mapping algorithms should be to minimize the II, which improves performance by reducing the total number of cycles required to execute the *kernel*. An ideal case would be when an n iteration kernel (or loop) mapped onto a CGRA takes exactly n cycles to complete execution; where (*Iteration Interval*) $\text{II} = 1$. The minimum II for a mapping is limited by both resource constraints and recurrence constraints (of the kernel).

Resource Constraint: To define resource constraints, we can state that for a kernel that has x nodes and a CGRA that has y PEs, if $x = 4 \times y$, then the II *cannot* be less than 4. In other words, if the number of nodes in a kernel is n times the number of available PEs, ideally only four nodes of the kernel can be executed in each cycle, and the kernel will require a minimum of y cycles for its execution. In this, the mapping and minimum II are limited by the available PEs in the CGRA. Another resource constraint is the number of data memory ports available to each PE, which limits the data communication bandwidth to the PEs and therefore affects the execution time of the kernel.

Recurrence Constraint: A recurrence constraint is illustrated in Figure 3. A simple DFG is shown in Figure 3(a), along with its mapping to the CGRA in Figure 3(b). In Figure 3(c), the DFG is unrolled once and mapped to the CGRA Figure 3(d). In the first case (Figure 3(b)), a single iteration of the loop is executed every two cycles; in the second case (Figure 3(d)), two iterations are executed every four cycles, for an effective $\text{II} = 2$, for both cases. For this DFG, the lowest achievable II is 2, for *any* CGRA size.

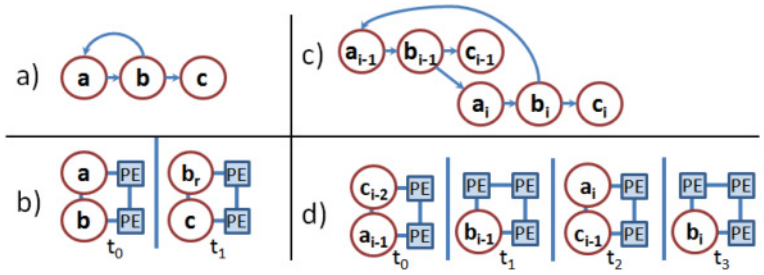

Fig. 3. The DFG in (a) is mapped to a 2 × 2 CGRA in (b) with an II of 2. DFG is unrolled once in (c) and mapped in (d) with an II of 4, for an effective II of 2.

Instructions-per-Clock (IPC): This is a metric to quantify performance of a multithreaded application on a CGRA (and directly correlates to throughput). When multiple threads are executed on a multithreaded CGRA, the utilization of the CGRA becomes an addition of each individual thread, up to at best 100%. The IPC of the CGRA is then additive of the IPC of each individual thread. In this article, we use this metric to demonstrate the effectiveness of the CGRA in its role as a multithreaded coprocessor to improve system performance.

3. MOTIVATION

In this article, the primary objective is to enable multithreading on CGRAs and allow for the use of CGRAs as power-efficient coprocessors to general-purpose processors. On analysis of CGRA usage and kernel mapping methodologies over different programs and different CGRA configurations, we observe a key by-product of enabling multithreading: *reduced CGRA utilization*'. In this section, we illustrate this by-product and motivate the design of our *paging* technique and runtime transformation to enable multithreading in CGRAs, with our eventual motive of power-efficient embedded computing in mind.

Let us consider a loop kernel L, compiled into two different schedules (S^1 and S^2) with an II of 3 each. Schedule S^1 uses 24 PEs to execute in the CGRA, while schedule S^2 uses 12 PEs. Considering the number of PEs used in the computation, we can conclude that schedule S^2 is more power efficient, since in S^1, though the II is the same, the remainder of the PEs will be used for routing data and will also consume active power. For a CGRA in the single-thread mode, no other schedule can execute on the CGRA and therefore all the PEs must remain active. Some modified CGRA architectures (and corresponding mapping techniques) support confining unused PEs into segments such that the unused PEs can be switched off, thus saving on power. In the multithreaded mode, such unused PEs can be used to allow another schedule to execute alongside the first, thereby motivating for the possibility of using the CGRA in the multithreaded mode. In this example, schedule S^1 using 24 PEs has a higher CGRA utilization (effective number of PEs used for execution) than that of schedule S^2 using 12 PEs, where both accomplish the same amount of computation in the same amount of time (same iteration interval).

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 $II \times Utilization \times CGRASize - IPC$, where IPC is equal to the number of operations performed in each iteration of the kernel (for a given kernel, IPC is constant). Multithreading allows decreasing the number of unutilized PEs by reducing CGRA utilization of an individual schedule. The PEs not used by one schedule can be used by other schedules for an overall

higher IPC and throughput. The quantity of useful *CGRA utilization* differs from one schedule to the other, which is the focal point in our motivation for the implementation of our developed software framework.

4. RELATED WORK

4.1. CGRA Mapping

Due to the uniqueness of CGRA architectures, advanced techniques for performing operations on a CGRA have been developed [Park et al. 2009b; Yoon et al. 2008]. The process of taking a piece of code and modifying it to run on a CGRA is known as *mapping*. This process involves (1) generating a DFG and software pipelining using modulo scheduling to map the DFG of the kernel onto the CGRA [Rau 1994] and (2) generating a prologue (and epilogue) to prepare (and finalize) data used by the kernel (executed on the CGRA). Since CGRA designs widely vary, architecture-specific mapping algorithms have been developed; for example, the ADRES [Mei et al. 2005] architecture uses DRESC [Mei et al. 2002, 2003], the RaPiD [Ebeling et al. 1997] uses SPR [Friedman et al. 2009], and CGRA express [Park et al. 2009b] uses modified EMS [Park et al. 2008].

4.2. Multithreading on CGRAs

Multithreading on CGRAs is a relatively new concept. In this work, the ability to allow multiple unrelated thread kernels to execute simultaneously on the CGRA is defined as *multithreading ability of a CGRA*. These threads can be either spawned of the same parent process or completely independent processes. One work that allows several kernels to be executed simultaneously is Polymorphic Pipeline Arrays (PPAs) proposed by Park et al. [2009a]. 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 dataset, the individual kernels can be apportioned cores as needed by data demands. This technique enables a thread-related dynamic multithreading framework.

Over the years, researchers have recognized the low CGRA utilization of individual schedules and developed techniques to either improve the same or introduce hardware mechanisms to reduce active PE energy consumption [Kim et al. 2010]. Since the II has a possible minimum, useful utilization must have a maximum for any given kernel. Multithreading enables CGRAs to take advantage of thread-level parallelism in addition to instruction-level parallelism, and therefore maximize CGRA utilization and eventually the power efficiency of the system. Hamzeh et al. [2012] and Hamzeh et al. [2013] develop a recompilation-based heuristic to increase the applicability of CGRA-based parallel computation on a wide range of applications.

Another closely related work attempts to enable multithreading 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 runtime static (the partitioning could possibly be done automatically but would remain runtime static). While the authors only showed enabling running two related threads, the technique could presumably be expanded to enable a runtime-static generalized multithreading.

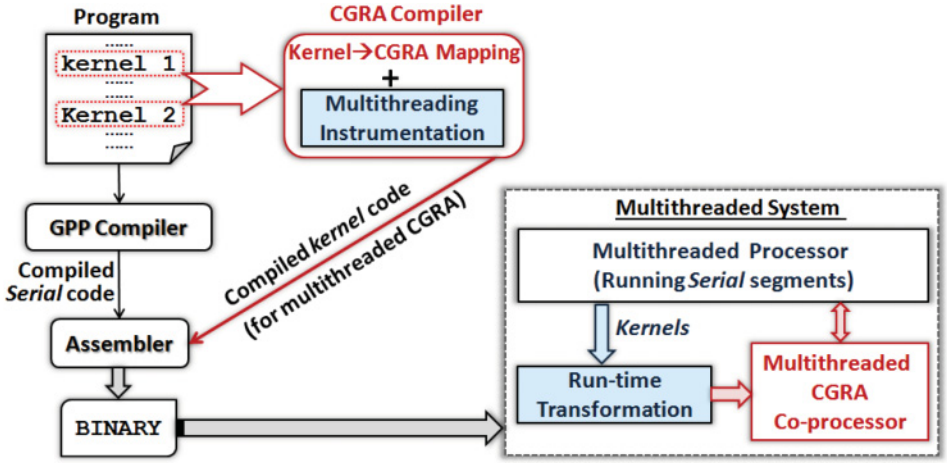

Fig. 4. Overview of our multithreading CGRA framework is described with details on the workflow involved in the application and use of our framework for a multithreaded system coupled with a multithreaded CGRA coprocessor. The components developed as part of our framework are indicated as shaded blocks: (a) multithreading instrumentation wrapper of the CGRA compiler, and (b) runtime transformation in the mapping of kernel code to the multithreaded CGRA coprocessor.

5. OUR MULTITHREADING FRAMEWORK

There are several ways in which multithreading can be enabled in CGRAs without modifying the hardware. However, only a few ways are readily conceivable. Due to the complexity of the CGRA mappings (which can be viewed themselves as DFGs), no naive modifications to the mapping can guarantee a working schedule. It is possible to combine two separate DFGs and map them simultaneously together, but this is impractical at runtime and not flexible enough to be done statically. Instead, a form of hard multiplexing should be executed, either in time or in 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 of our framework is to create schedules that can be quickly transformed at runtime to run on different portions of the CGRA as needed [Shrivastava et al. 2011].

5.1. Overview

Figure 4 describes our multithreading framework with specific details on the internals of the CGRA compiler and the multithreaded system. During the compilation stage of the application, specific portions are identified that are eligible for acceleration on the CGRA (hereby named *kernels*), and the rest is annotated as *serial* code. The *serial* portion of the application is compiled by the general-purpose processor (GPP) compiler specific to the target processor, and the *kernel* portions are compiled by the CGRA compiler. In our framework, the CGRA compiler that maps the kernels to the CGRA coprocessor is coupled with the multithreaded instrumentation setup, which packs the mapping into pages, making it conducive for the runtime transformation for easy scheduling on the multithreaded CGRA. The two compiled portions of the application are `assembled` into a binary to be executed in the multithreaded system. While the application executes on the processor, the *kernels* are offloaded to the coprocessor through the *runtime transformation* setup, which schedules the kernels to execute on available pages of the multithreaded CGRA.

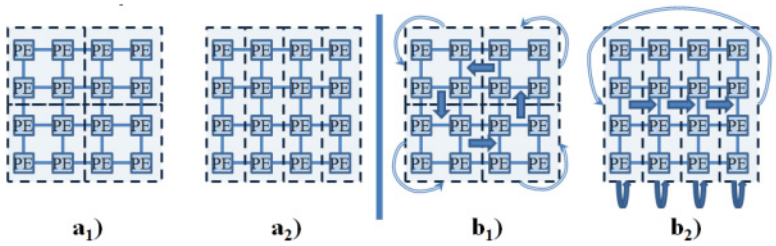

Fig. 5. (a) An example of possible page divisions. Note that transforms for the first division (a_1) are slightly more complex than the second (a_2). This is explained in Section 5.5.2. (b) An illustration of page-level dataflow that the compiler is allowed to generate.

5.2. CGRA Hardware Specification

5.2.1. Connection Topologies. The multithreading framework presented requires a uniform and symmetric connection topology. This can be as simple as the one shown in Figure 5, or perhaps more complex. This requirement exists because the CGRA will be conceptually divided into sections called pages by the compiler, and connections between pages must by identical (thus ensuring truly dynamic threading). These pages will then become the objects of transformation.

5.2.2. Register Requirements. In order to fully utilize our multithreading framework, registers must be present and reserved only for multithreading use. A limited form of multithreading is possible without registers, but this work does not present results for this architecture configuration.

5.3. Compilation Constraints

The majority of the preparation for our multithreading process happens during the compilation phase. This is because the compiler is responsible for producing schedules that can quickly be transformed at runtime and also maintain high performance (low *II*) and power efficiency (high CGRA 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 5(a).

5.3.1. Dataflow 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 dataflow at the page level to ensure a fast transformation. This is done by allowing data to flow to either the same or the next page in the next time (cycle), 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 (cycle). This restriction is illustrated in Figure 5(b). It will be shown later in Section 6.1 that this restriction does not degrade kernel performance. This dataflow constraint can be implemented by modifying the connection topology the compiler assumes the CGRA has.

5.3.2. Register Constraints. If the CGRA provides registers, the compiler must not use these registers during mapping, as they will be used for the runtime transformation. If a register is needed, the compiler can use global registers (often implemented simply as local memory) for these cases. Many mapping algorithms [Dimitroulakos et al. 2009, 2005; Hatanaka and Bagherzadeh 2007; Park et al. 2006, 2008; Yoon et al. 2008] do not use local registers well in practice.

5.3.3. Two-Hop Constraints. The PE connectivity within the CGRA, based on the hardware constraints established, can be translated into the two-hop constraint in the compiler. For example, in the page-divided CGRA (as in Figure 5(b)), page $P2$ can transfer

data to $P1$, $P2$, or $P3$. Conversely, $P2$ can only receive data from these pages, which is the hardware restriction on the CGRA paging mechanism. Our software restriction is to remove one of these connections such that $P2$ can send/receive data to/from $P2$ and $P3$ only. Therefore, if nodes mapped to page $P2$ in time t are placed on page $P4$ in hardware, its dependent nodes must be placed in $P3$ or $P4$ of time $t-1$. In this, the number of hops between $P2$ and $P4$ is two, which defines the *two-hop constraint*. This constraint relates only to the correctness of the computation upon scheduling on the pages and does not affect the performance of the system. There is a negligible performance impact owing to the software restriction, which we show is negligible in our experimental results.

5.4. 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.

5.4.1. Input. Schedule P for a CGRA with N pages. Suppose the II 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 $p_{(n,t-1)}$.

5.4.2. 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 operations that will be performed on page n' at time t'.

5.4.3. Constraints. The first constraint is that no two pages in P must be mapped to the same page in Q in the same time index t. 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 as follows:

(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) $x_1, x_2, x_3 < M$

5.4.4. 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.

5.5. 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 serves the following purposes:

(1) First, having the ability to shrink schedules allows schedules to be sized to fit in unused portions of the CGRA at runtime.

(2) The second is less obvious and involves how recurrence constraints can be mapped using this framework. They can be mapped either 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 four pages can handle schedules compiled for a ring topology of four, three, two, or one page(s). This allows the ring to be sized according to the size of the recurrence cycle. However, a transform of N pages to N 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 Transformation*(PT), which runs in time linear to the number of pages in the transformed schedule.

5.5.1. Transforming a Schedule. The PT 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 (i.e., 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 (i.e., $(N)mod(M) \neq 0$), pages are scheduled 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.

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, $(d1 + d2)/2$.

Case 2: The dependencies are a single hop apart: This case can only happen if one of the dependencies is on the edge. In this case, the page is scheduled on the edge based on the availability, in such a way that the two-hop constraint is not violated.

Case 3: The dependencies are in the same page column: This case is created when scheduling pages are tailing in the initiation phase (if not, then Case 2 becomes a fallback). This becomes a recurring case. Therefore, the page is scheduled where the previous times of the current page have not already been scheduled.

5.5.2. Page Mirroring. While performing transformations (or when placing pages in the new locations), they cannot be simply translated. Instead, they must be *mirrored*. This mirroring is illustrated in Figure 6. This mirroring is only necessary when a page is

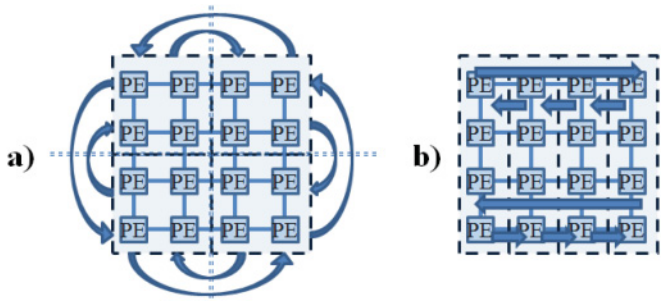

Fig. 6. (a) Due to the page divisions, page mirroring must be performed. Pages must be "folded" along dotted axes when moving locations of the page. (b) Page divisions here allow pages to be translated when moving locations. This is illustrated by the straight arrows.

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

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 thetime $p_{n-1,t-1}$ 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

translated across an axis 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 5(a_2).

5.6. Example Transforms

Two examples are shown to illustrate the nuances in their implementation of both the complete framework and our *Pagemaster Algorithm*.

5.6.1. Example One: Mapping to Transform. The first example shows the process of mapping a DFG with the compiler restrictions and then transforming that same mapping to run on a single page. For this example, the mapping (described in Section 2.2 and shown in Figure 2) is used. The first step in the framework is to apply the compiler dataflow restrictions. This is shown in Figure 7(a). While not always the case, in this example, the mapping already satisfies the restrictions, as shown in Figure 7(b). 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*.

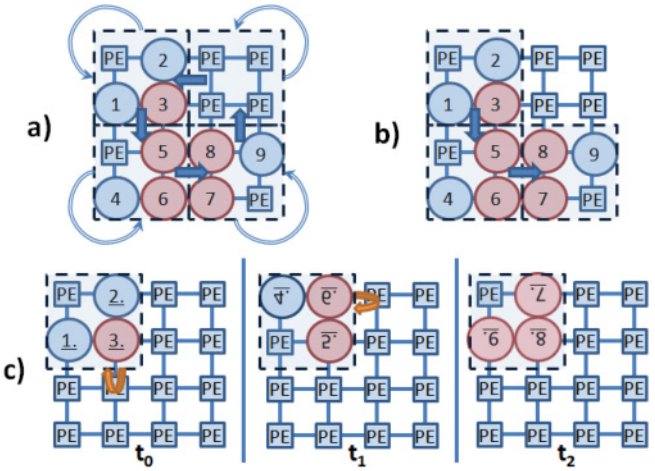

Fig. 7. Application of the multithreading framework from mapping to transform. To illustrate the page mirroring principle more clearly, node text is underlined and the number is suffixed by a decimal and then mirrored appropriately.

In this example, the transform performed is from three pages (as the mapping only needs three pages) to a one-page configuration. This is shown in Figure 7. Here, the pages are mirrored in order to maintain correct internal page mappings. Arrows in Figure 7 indicate how the original dataflow (that was to the next page) is modified to return to the same page. For example, node 6 normally passed data to the right—node 7. However, node 7 will now execute on the same page as that in node 6, so data just needs to be passed through an output buffer to node 7.

Not shown in Figure 7 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 clearly shows the general nature of the multithreading that is 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 on the CGRA.

5.6.2. 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 six pages is transformed to a schedule Q of four pages. This is shown in Figure 8.

We will now see how all the page dependencies are filled. Dotted arrows indicate how previous page dependencies are fulfilled, while solid arrows indicate how the same page dependencies are fulfilled. Small boxes with an "r" indicate registers that must be used. As can be seen in Figure 8, no more than two register sets are needed for a page at any cycle of the mapping. 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 series crosses this boundary, registers must be used to fulfill a page dependency. Excluding this intersection, dataflow follows a regular pattern, as can be seen in the figure. The pattern must be continued for the looping nature of the kernel to work correctly.

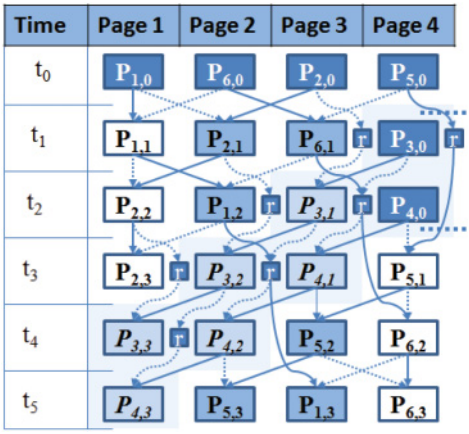

Fig. 8. Shown is a transform of six pages to four pages. Pages with light text are pages scheduled during the initialization phase. The two pages bracketed were schedules using the tailing method. Pages with the darkest background and no shading behind them are those scheduled using Case 1 described in Section 5.5.1. Pages with a lighter background and italicized are scheduled with Case 3. Those with a white background are scheduled with Case 2.

An exception to Case 2/Case 3 described in Section 5.5.1 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.

5.7. Implementation Details

In this article, the underlying mapping algorithm used in our experiments is EMS (Edge-centric Modulo Scheduling) [Park et al. 2008]. The CGRA PEs were divided as shown in Figure 5. The caveat of this division is that a transform must always be performed when scheduling onto 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 except no registers are required. This is not expected to cause any performance degradation, as a transform would have to be performed for any schedule compiled using fewer pages than the original structure. These transforms are necessary to remove the necessity of a hardware connection from the last page to the first page, which the compiler assumed was present.

The advantage to using this division method is that transforms themselves become simpler, as explained in Section 5.5.2. In this, 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.

6. ANALYSIS AND RESULTS

To analyze the multithreading framework, a few quantifying 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 running only a single thread on the system, the performance of the modified schedule can easily be compared with the original unmodified case by comparing the II of each. In the case of multiple threads accessing the CGRA, a more complete analysis that incorporates the effective CGRA utilization and system throughput in the multithreaded mode is performed. The cost of the framework is any performance lost for

Table I. Benchmark List

Benchmark	Abbreviation
Banded Linear Equations	BLEs
First Difference	First Dif
General Linear Recurrence Equations 1	GLREs 1
General Linear Recurrence Equations 2	GLREs 2
General Linear Recurrence Equations 3	GLREs 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

Abbreviations for benchmarks used in this work and in the graphs presented.

single-threaded environments and the benefits of the framework are any performance improvements for multithreaded environments.

6.1. Small Cost in Single-Threaded Performance

To determine single-threaded performance, the *Iteration Interval* of a schedule needs to be compared. An identical kernel with a lower II has a higher throughput when compared to one of a higher II. This is true for all kernels. We experimented with over 20 benchmarks (refer to Table I for abbreviations used) to analyze the effect of the restrictions imposed in the multithreading framework. Shown in Figure 9 are the compilation results for the benchmark set normalized to the performance of the original, unmodified compiler. Three CGRA sizes of 4×4, 6×6, and 8×8 are shown. The average performance for the benchmarks for different CGRA sizes indicate that the restrictions have little effect on the II (less than 1% difference on average) in the general case for a correctly chosen PE page size. It should be noted here that the CGRA architecture targeted requires two PEs in the same column at the same time in order to perform a store operation, preventing a page size of two from obtaining a solution for every benchmark.

6.2. Multithreading Increases Overall CGRA Utilization and Benefits Performance

To analyze multithreading performance, a metric besides the II must be used. Useful utilization and limitations of single-threaded mapping are explained in Section 2.2 and Section 3. By introducing the concept of space multiplexing to allow multithreading and the ability to compile for subportions of the CGRA, useful utilization becomes an important target metric. Therefore, minimizing II remains the primary goal, but maximizing utilization becomes an additional goal. By minimizing the number of active PEs used for computations in the mapping of a kernel (for the same II), CGRA hardware utilization is maximized, and power-efficient use of the CGRA is achieved, and thereby reduced system power consumption.

6.2.1. A Page Size of Four PEs Minimizes Individual Thread Utilization. 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 4×4 CGRA with a page size of two PEs, this value is eight. That is to say, eight different pages of instructions can be executed each cycle. If it were assumed 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

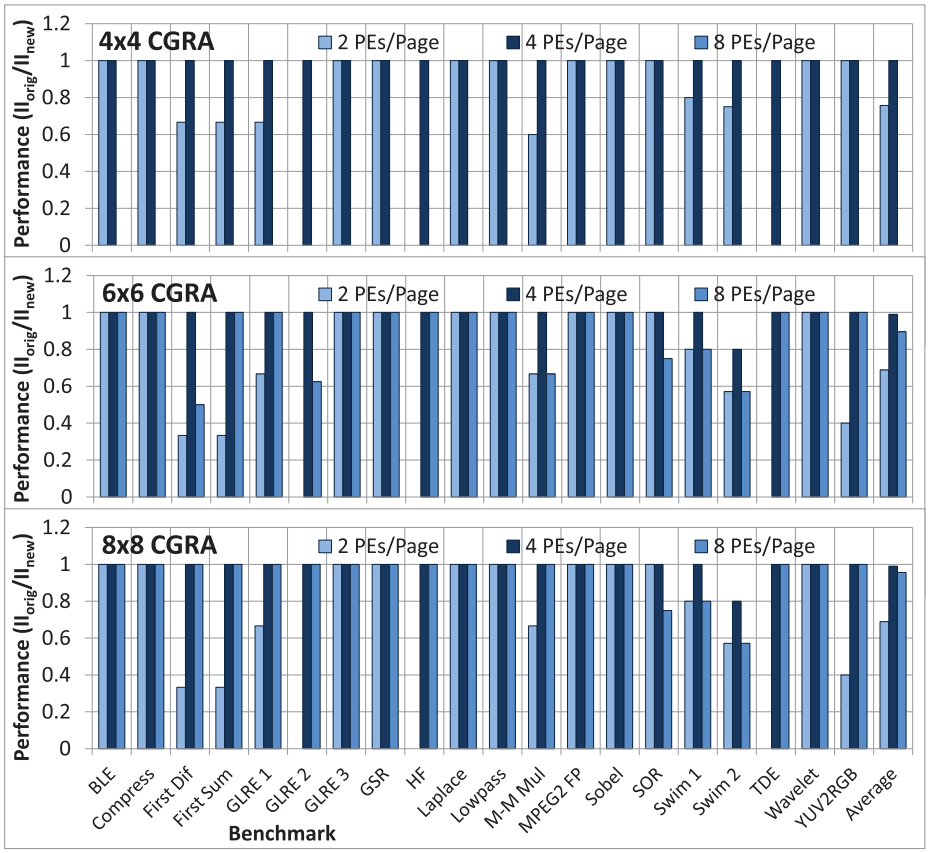

Fig. 9. Performance of benchmarks compiled for different CGRA configurations and sizes. Performance is the inverse of the II. Performance is normalized to that of the original compiler. It can be seen that for a page size of four PEs, the constrained compiler can achieve almost equal performance to that of the original compiler.

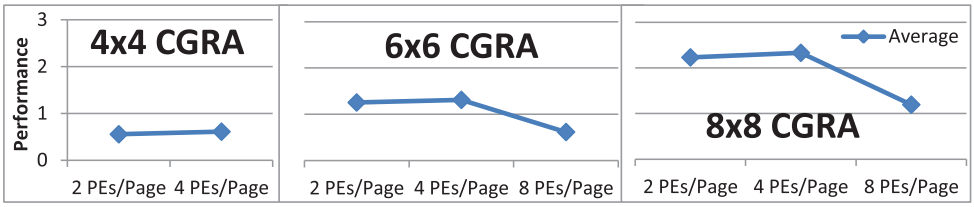

Fig. 10. Average performance per page for an average of all benchmarks across different-size CGRAs. Performance is inversely proportional to total utilization. A larger number indicates a higher-quality mapping. This indicates that a page size of four PEs/page has the best utilization for this set of benchmarks.

cycle multiplied by the II. The inverse of this is *performance per page*. The results of these calculations are shown in Figure 10. As can be seen, the average performance per page (for the set of benchmarks experimented) is slightly higher for a page size of four PEs compared to that of two PEs and significantly greater when compared to a page size of eight PEs.

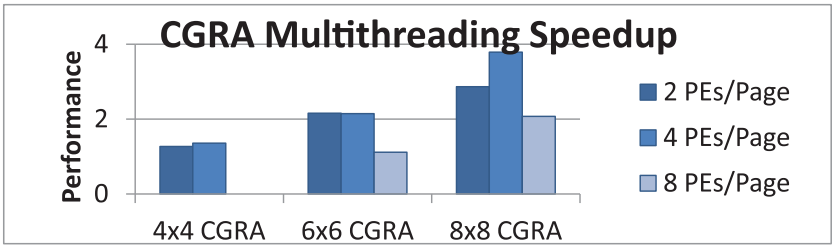

Fig. 11. Speedup of a multithreaded CGRA using paging over that of a single-threaded CGRA when running at maximum theoretical throughput for different page sizes and CGRA sizes. This indicates that a page size of four is expected to achieve the most speedup under multithreaded loads.

6.2.2. Improved CGRA Utilization of an Individual Thread 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. At this point, it is possible to obtain the CGRA-side acceleration factor for enabling multithreading for this set of benchmarks, disregarding any other factors. These speedups compared to a single-threaded CGRA are shown in Figure 11. In the best case, performance increases by over 280%. Note the similarities in trends to Figure 10. This trend indicates that in a system, a page size of four PEs/page should be expected to perform the best in multithreading mode. In addition, as seen in Figure 9, a page size of four PEs/page is expected to perform the best in a single-threaded environment also. This makes a page size of four PEs the preferred size and is used during the case study.

7. CASE STUDY: MULTITHREADED CGRA COPROCESSOR IN AN EMBEDDED SYSTEM

7.1. Influential System Parameters

7.1.1. Direct Memory Access (DMA) Issues. An issue of extreme importance is whether the time to set up an initial DMA buffer for the CGRA, to execute the schedule on the CGRA, and then execute a DMA memory transfer back to the processor, will be less than that of the code executing on the processor alone. In addition, a single DMA channel (one in each direction, to and from the CGRA) may inhibit the efficiency of such multithreading on the CGRA. This is because if the time to transfer the initial data from the buffer to the CGRA is greater than the time to execute the kernels on the CGRA, no two threads can ever be executed on the CGRA simultaneously. In our simulation setup, we model the impact of such DMA architecture configurations accurately and study its system-level impact on a multithreaded CGRA coprocessor-based system.

7.1.2. Transform Time. When scheduling kernels to a CGRA in multithreading mode, there is a finite time involved if the kernel needs to be shrunk before execution when the number of available pages is less than that required by the schedule at hand. This transformation must be performed in 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-run (future) schedules or currently running schedules. CPU time required for such transformations can easily be hidden by scheduling them in parallel with DMA transfers.

7.2. Experimental Setup

7.2.1. Multithreaded CGRA Simulation Overview. To analyze and experiment on our proposed multithreaded CGRA framework, we model through simulation the ADRES-like CGRA architecture (Figure 1) together with the associated hardware blocks to execute a mapped kernel on the CGRA coprocessor (using Gem5 to model the CPU). To simulate the multithreaded mode, multiple instances of a random selection of CGRA benchmarks are composed to execute together, where the proportion of code run on the CGRA versus that on the CPU is a predefined 75% CGRA and 25% CPU. Figure 4 describes an overview of the compiler framework that performs this functionality.

A thread is to run on either the CPU or the CGRA. For our experiments, we perform actual simulations of the code executing on the CGRA. When on the CPU, the execution is as that of a general Intel multicore architecture configuration. While on the CGRA, a series of checks are performed to determine if the mapping for the kernel can actually be scheduled on the CGRA by considering the current state of the CGRA and the nature of the threads currently mapped onto the CGRA. The CGRA configuration files, instruction and data memory, are loaded onto the CGRA buffers through the DMA by means of CPU subroutines. The evaluated performance is relative to a CPU-only execution of 16-thread configuration, where the resource and memory constraints are modeled accurately (as they are deterministic systems). The CGRA execution time is determined by the difference between initial DMA transfer (to the CGRA from CPU) and DMA transfer back (CGRA to CPU).

7.2.2. System Configurations. Based onf this analysis, three 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 350MHz, two CPUs running at 800MHz, and a DMA bandwidth of 300MB/s in each direction
(2) *DMA-constrained quad-core CPU system:* System with a CGRA clock speed of 500MHz, four CPUs running at 1.8GHz, and a DMA bandwidth of 1GB/s in each direction
(3) *Non-DMA-constrained quad-core CPU system:* System with a CGRA clock speed of 500MHz, four CPUs running at 2.5GHz, and a DMA bandwidth of 8GB/s in each direction
(4) *Optimized dual-core CPU system:* System with a CGRA clock speed of 600MHz, dual CPUs running at 800MHz, and a DMA bandwidth of 4GB/s in each direction. This system is discussed in Section 7.6.

In our experiments, the key parameters that are varied to study their impact on the system-level performance are as follows:

(1) *Thread Count*: By varying the number of threads run on the system (*one, two, four, eight,* or *16*), the effect of sharing a single CGRA resource can be identified. These results help identify resource limitations, whether runtime is constrained by the execution ability of the CGRA or some other resource.
(2) *Kernel Need*: How often a thread is executing a loop kernel (chosen from the 20 benchmarked) compared to serial code (low, medium, or high, defined as 50%, 75%, or 87.5% likely to execute on the CGRA).
(3) *Page Size*: The number of PEs/page (two, four, or eight). By varying this metric, it can be determined whether a preferred page size of four PEs is optimal for this set of benchmarks.
(4) *CGRA Size*: The size of the CGRA used (4×4, 6×6, or 8×8). By varying this metric, it will be seen that current mapping implementations are able to provide

meager gains by increasing CGRA size, and therefore the need for multithreading will be seen. The variation in the CGRA utilization across CGRA sizes for the compiler framework can establish the impact on power efficiency achievable.
(5) *Pipelining*: Whether successive kernels can operate on the previous kernel's data (either *enabled* or *disabled*).

7.3. Setup for Case Study Analysis

To perform case study analysis, various interdependent system parameters are grouped and certain classifications are made to help study their system-level impacts.

7.3.1. System-Level Modeling. In a system, based on the availability of hardware resources, system stalls and performance penalties will be incurred. We model the (active and stalled) usage of the hardware resources accurately based on their individual configurations. Based on the coprocessor usage of the applications, the following classifications are made:

(1) *CPU Only*: This system has no CGRAs. Therefore, all code must be run on the CPUs.
(2) *Single-CGRA System*: This system has a single CGRA with a single DMA channel and 64KB of data memory.
(3) *Many-CGRA System*: This system has as many CGRAs as threads, but only a single DMA channel and 64KB 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 multithread using paging and a single DMA channel and 64KB of data memory. Schedule transform times are modeled.

7.3.2. CPU - Coprocessor Simulation Procedure. To understand the simulation procedure, the execution of a single thread (containing both serial and CGRA code) is explained as follows. If the next code section is to be executed on the CPU, it is either scheduled to an available CPU or stalled until one becomes available. On the other hand, if it is to be offloaded to the coprocessor, the following steps are performed:

1. Data Memory (DMEM) Check: An initial buffer must be set up such that room for the results of the kernel are reserved in the data memory buffer. If there is not sufficient space on the buffer, the thread must be stalled. The initial buffer size is determined by examining the current DMA load (by running thread kernels) and sized in such a way that the schedule (to be mapped) can begin execution on the CGRA while the DMA is still active. This is made possible by overlapped execution (double-buffering). This calculation involves allocating space on the DMA buffer proportional to the number of pages allotted on the CGRA and the data needs of the kernel mapped.

2. 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. Schedule transforms can be performed during this DMA time.

3. CGRA Check: Once a thread's initial buffer and instructions are loaded, execution can begin on the CGRA. If no pages are open for execution in the case of a paging CGRA, the thread is stalled before DMA begins.

4. CGRA Scheduling: A thread can now be scheduled on the CGRA. Once execution completes on the CGRA, CGRA pages and data memory used for buffering are released, but the result memory used is held until data can be DMA'd back. If multiple threads

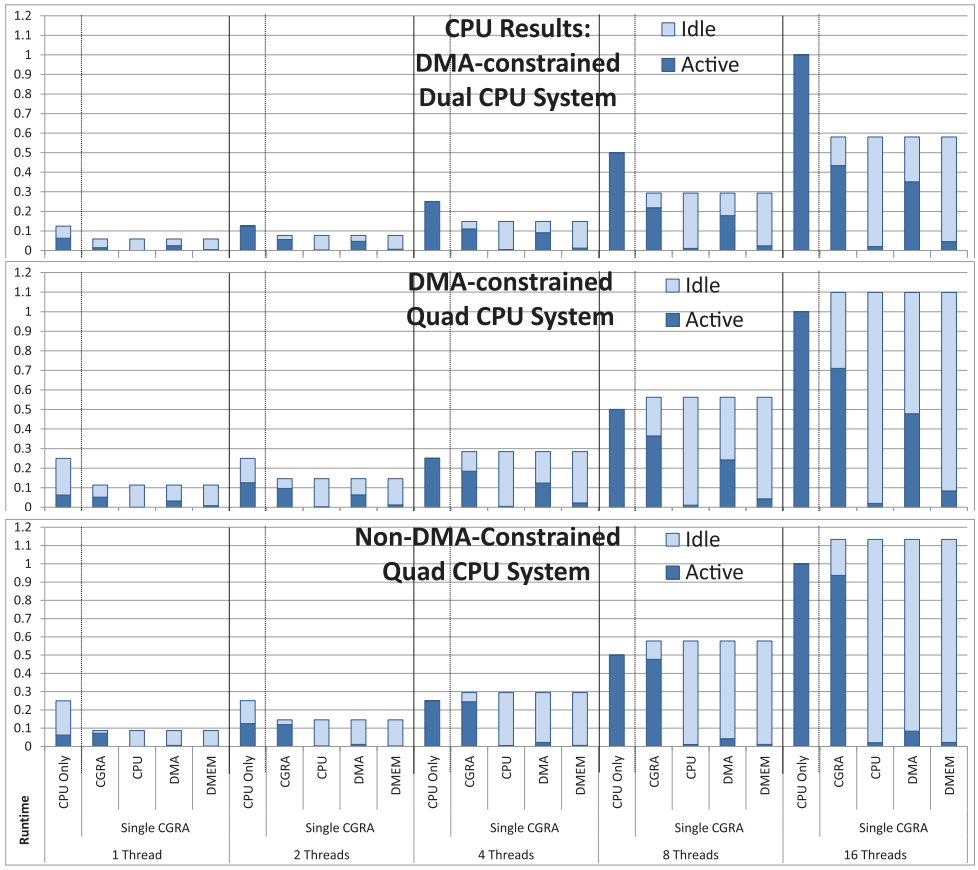

Fig. 12. Relative runtime of systems using either a CPU or a single-threaded CGRA. For each system, runtimes are normalized to the runtime of the CPU-only system of 16 threads. It is seen that a single-threaded CGRA decreases or achieves equal runtime for a small number of threads but begins to suffer as compared to simply running on a CPU when more threads are added.

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.

7.4. Multithreading Performance Results

7.4.1. CPU-Only Results Show the Benefits of Multithreading. While CGRA execution time is entirely deterministic, the general-purpose processor is rather nondeterministic. Factors such as thread scheduling policy, benchmark optimizations, cache misses, and so forth can greatly change runtime on the CPU. In our experiments, worst-case CPU execution times are considered in order to not bias the results in favor of the proposed technique.

Figure 12 shows runtime differences between CPU-only execution and that using a single single-threaded CGRA for varying the number of threads. These graphs illustrate the limitations of a single-threaded CGRA. While runtime for a single-threaded CGRA is less in a dual CPU DMA-constrained system, the benefits of multiple CPUs and therefore their multithreading ability are seen in the DMA-constrained and non-DMA-constrained quad-core systems. As process technology improves, finding a

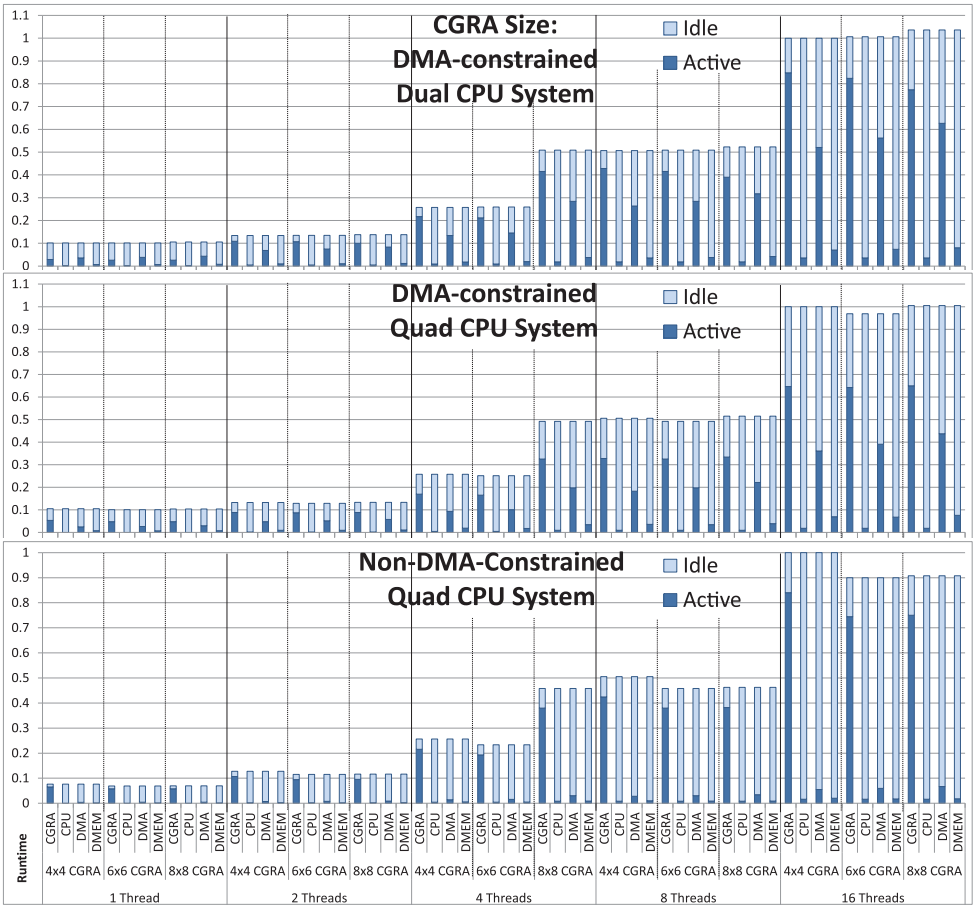

Fig. 13. Relative runtime of systems using a single-threaded CGRA of varying sizes. For each system, runtimes are normalized to that of a 4 × 4 CGRA and 16 threads. It is seen that in single-threaded mode, runtime is not decreased by increasing CGRA size.

quad-core CPU in an embedded system will not be uncommon [ARM-A9 2009]. 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 to be noted 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 runtime in these cases. Instead, DMA bandwidth and CGRA accessibility are more crucial. It will be seen in later sections that a multithreaded paging CGRA significantly increases accessibility.

7.4.2. Increasing CGRA Size Benefits Only Multithreaded CGRAs. As long as the CGRA is single-threaded, increasing CGRA size does not decrease runtime significantly and, in some cases, it can even increase runtime due to the increase in instruction size (Figure 13).

Figure 14 compares a single-threaded CGRA to that of a paging CGRA for different sizes in a non-DMA-constrained system. Sixteen threads are used. Performance (inverse to runtime) for a paging CGRA follows almost exactly the trend seen in

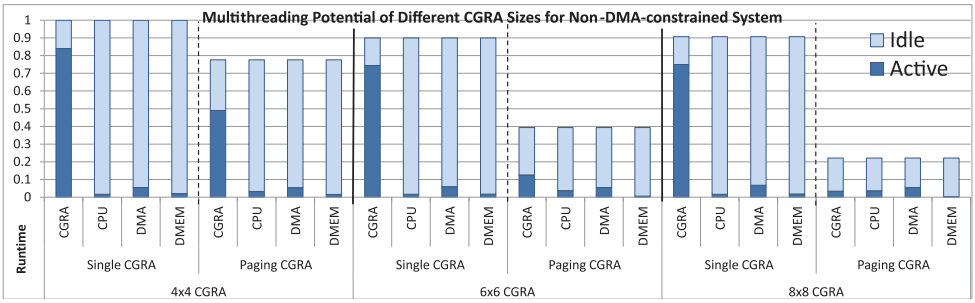

Fig. 14. Runtime of different CGRA sizes for a single-threaded CGRA versus a paging CGRA in a non-DMA-constrained system running 16 threads. The runtime is normalized to a single-threaded 4 × 4 CGRA. It is seen that by allowing multithreading, larger CGRA structures are more effectively utilized and decrease runtime.

Figure 11. This illustrates much of the wasted computation potential by not allowing multithreading.

7.4.3. Increasing Threads Highlight DMA Needs of CGRA. To illustrate the impact of the number of threads on CGRA accesses, a CGRA size of 8 × 8 was used, allowing for the most multithreading potential when paging is enabled (Figure 15). In DMA-constrained systems, a paging CGRA can achieve the same runtime as that of many-CGRA systems. 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 runtime improvements compared to that of many-CGRA systems. It should also be noted that the CPU time used during the transformation of threads is insignificant (it accounts for less than 2% of total CPU time, or a 60% increase in CPU active time). Thus, by slightly increasing CPU usage, runtime equal to that of 16 CGRAs is obtained using only a single page-enabled CGRA.

7.5. Summary of Case Study Analysis

Figure 16 shows the average stall times of each thread for given resources in different scenarios. The stall times are from the benchmarks shown in Figure 15 using 16 threads. A few important characteristics are seen of DMA-constrained systems. Limiting DMA bandwidth has a twofold effect for multithreaded systems: (1) the initial buffer size must be larger, and (2) completed schedules' data reside in data memory longer. These two effects cause data memory to become full and require 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 15, where a many-CGRA system and a paging CGRA system provide equal runtime 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.

7.6. Optimal System Design: Sufficient DMA Bandwidth Is Required

Based onf the previous case study analysis, an optimal system is designed. The goal here is to configure an embedded system, with a multithreaded CGRA as a coprocessor, in such a way that maximum system power efficiency is achieved. In this, the limitations and bottlenecks identified from our analysis have been optimized for a best-case scenario. This gives the system designer information on possible implementation of

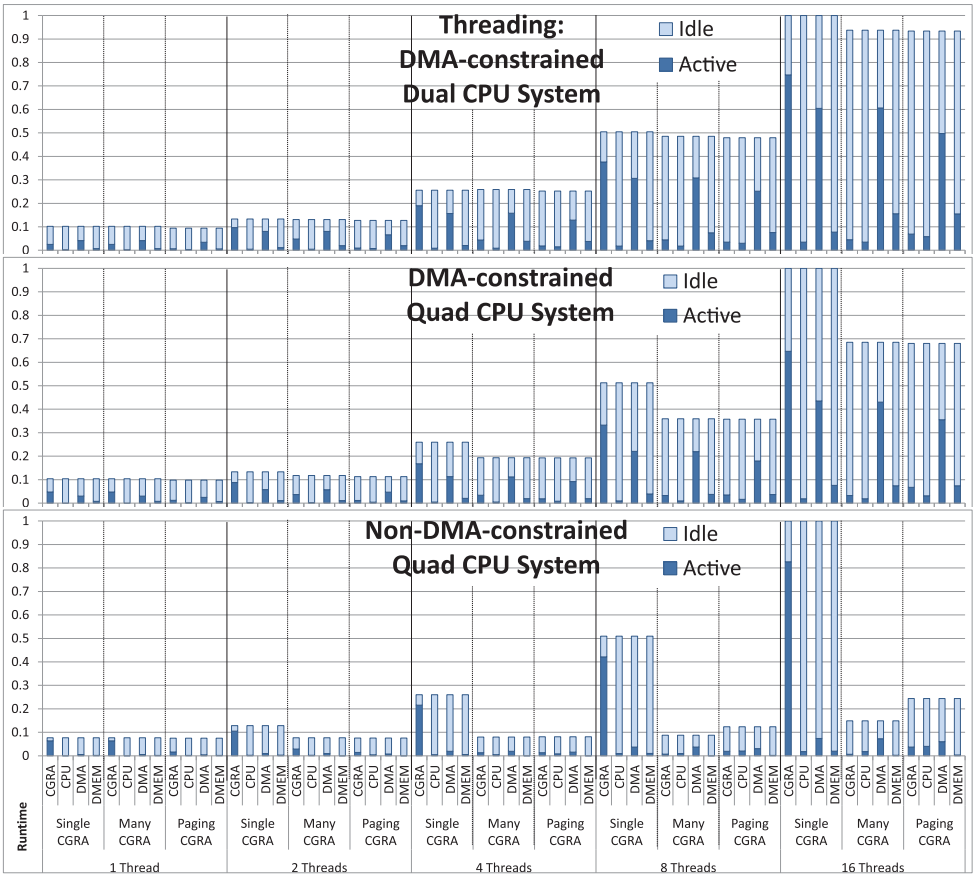

Fig. 15. Relative runtime for systems with a single-threaded CGRA versus many CGRAs versus a multi-threaded paging CGRA. For each system, runtimes are normalized to that of a single-threaded CGRA and 16 threads. It can be seen that for DMA-constrained systems, a paging CGRA achieves equal runtime to that of many CGRAs and near-equal runtime to that of many CGRAs.

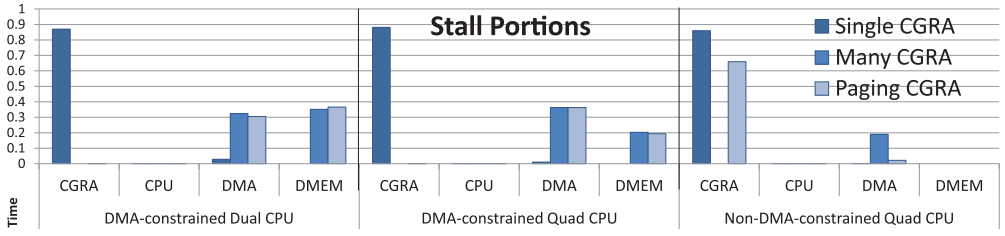

Fig. 16. Average time of total runtime each thread spent stalled as a percent of total runtime using an 8×8 CGRA. It is seen that a single-threaded CGRA becomes a bottleneck, and not until sufficient DMA bandwidth is available does a paging CGRA become a bottleneck.

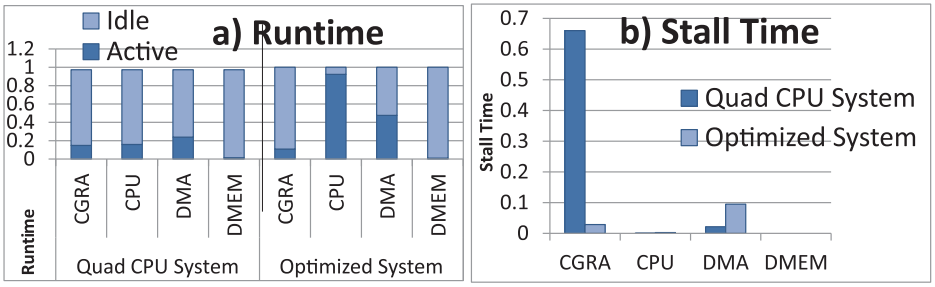

Fig. 17. Relative runtime of non-DMA-constrained quad-core system versus optimized system for a benchmark with 16 threads. Runtime is normalized to the optimized system. It is seen that the optimized system provides near-equal runtime to the non-DMA-constrained quad-core system while removing bottlenecks and being more efficient.

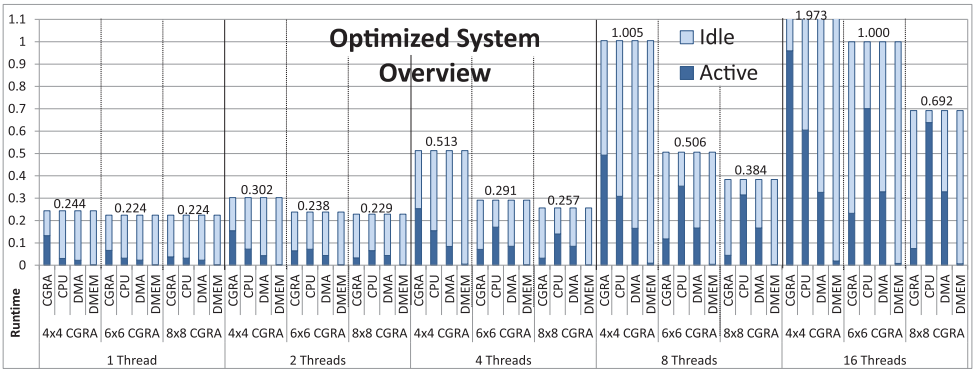

Fig. 18. Overview of optimized system runtime. Runtime is normalized to 16 threads of a 6 × 6 CGRA. Various trends are present due to the optimization of this system, the most important being a decrease in runtime as CGRA size increases.

our multithreaded CGRA framework. The proposed system attempts to achieve similar runtimes to that of the non-DMA-constrained quad-core system for 16 threads. This system has an 8 × 8 CGRA clocked at 600MHz with dual CPUs at 800MHz and a DMA bandwidth of 4GB/s in each direction.

7.6.1. Performance Results. Results in Figure 17(a) show that the runtimes for both systems are within 3% of each other, where the stall times for each resource are also indicated. This system thus has minimal distributed stall times across all resources.

7.6.2. Overall System-Level Trends. The overall system statistics for the optimized system are shown in Figure 18. In this we observe the following trends:

(1) As threads increase, it becomes clear that a larger CGRA size is beneficial (as noted by decreased runtimes). In this case, thread-level parallelism is much easier to exploit than instruction-level parallelism.
(2) When the number of threads is doubled, if the runtime is less than two times, the CGRA is not at maximum throughput. Therefore, between thread counts of four and eight, a CGRA size of 4 × 4 reaches maximum throughput, while for thread counts of eight and 16, a CGRA size of 6 × 6 reaches maximum throughput.
(3) When the maximum throughput is reached, the runtimes between different thread counts become close to the ratio of number of pages each CGRA size has. For example, a 4 × 4 CGRA has four pages and a 6 × 6 CGRA has nine pages. Therefore,

a 6 × 6 CGRA should have more than double the multithreading ability of a 4 × 4 CGRA, which is close to simulation results. These are the same trends identified in Section 6.2.2.

(4) This system shows a decrease in runtime using a multithreaded CGRA compared to a single-threaded CGRA for a highly threaded environment of almost 350%.
(5) This system is nearly 20 times faster than the same system running the workload using only the CPU.

8. CONCLUSION

The use of power-efficient coprocessors lowers the overall energy consumption of the system. We identify CGRA as an attractive candidate for a coprocessor, which has a power efficiency of around two orders of magnitude greater than that of a CPU (40MOPS/mW for a CGRA compared to 0.54MOPS/mW for an Intel chip). A key limitation in the use of such CGRAs is the lack of a means to offload multiple kernels simultaneously from multiple threads, as most modern processors in embedded devices are multithreaded. We propose a novel paging-based compiler framework that integrates with most CGRA mapping techniques to enable multithreading in the CGRA. A fast runtime transformation is developed that uses the compiled application and allows for a flexible means to allow a varied number of schedules to be simultaneously executed on the CGRA. This method involves a mirroring and page-shrinking technique that allows for efficient utilization of the CGRA hardware, and thus achieves power-efficient application acceleration using the coprocessor. In this work, we analyze the influence of various design parameters on an embedded system implemented with a multithreaded CGRA coprocessor through a detailed case study. With the help of analysis from our study, we configure an optimal system that efficiently utilizes such a CGRA coprocessor and thereby demonstrates improved power-efficient embedded computing.

REFERENCES

ARM-A9. 2009. ARM-A9 Datasheet. Retrieved from http://www.arm.com/files/pdf/ARMCortexA-9Processors.pdf.

F. Bouwens, M. Berekovic, A. Kanstein, and G. Gaydadjiev. 2007. Architectural exploration of the ADRES coarse-grained reconfigurable array. In *ARC'07*. 1–13. http://dl.acm.org/citation.cfm?id=1764631.1764633.

CUDA-fermi 2010. Tesla S2050 GPU Computing System. Retrieved from http://www.nvidia.com/docs/IO/43395/NV-DS-Tesla-S2050-june10-final-LORES.pdf.

G. Dimitroulakos, S. Georgiopoulos, M. D. Galanis, and C. E. Goutis. 2009. Resource aware mapping on coarse grained reconfigurable arrays. *Microprocess. Microsyst.* 33, 2 (2009), 91–105. DOI:http://dx.doi.org/10.1016/j.micpro.2008.07.002

G. Dimitroulakos, M. D. Galanis, and C. E. Goutis. 2005. A compiler method for memory-conscious mapping of applications on coarse-grained reconfigurable architectures. In *19th IEEE International Parallel and Distributed Processing Symposium*. IEEE Computer Society, Washington, DC, USA, 4. DOI:http://dx.doi.org/10.1109/IPDPS.2005.8

C. Ebeling, D. C. Cronquist, P. Franklin, J. Secosky, and S. G. Berg. 1997. Mapping applications to the RaPiD configurable architecture. In *FCCM'97*. IEEE Computer Society, 106–115. DOI:http://dx.doi.org/10.1109/FPGA.1997.624610

S. Friedman, A. Carroll, B. Van Essen, B. Ylvisaker, C. Ebeling, and S. Hauck. 2009. SPR: An architecture-adaptive CGRA mapping tool. In *FPGA'09*. ACM, New York, NY, USA, 191–200. DOI:http://dx.doi.org/10.1145/1508128.1508158

M. Hamzeh, A. Shrivastava, and S. Vrudhula. 2012. EPIMap: Using epimorphism to map applications on CGRAs. In *DAC'12*. ACM, 1284–1291. DOI:http://dx.doi.org/10.1145/2228360.2228600

M. Hamzeh, A. Shrivastava, and S. Vrudhula. 2013. REGIMap: Register-aware application mapping on coarse-grained reconfigurable architectures (CGRAs). In *Proceedings of the 50th Annual Design Automation Conference (DAC'13)*. ACM, New York, NY, USA, Article 18, 10 pages. DOI:http://dx.doi.org/10.1145/2463209.2488756

R. Hartenstein. 2001. A decade of reconfigurable computing: A visionary retrospective. In *DATE'01*. IEEE Press.

R. W. Hartenstein and R. Kress. 1995. A datapath synthesis system for the reconfigurable datapath architecture. In *ASP-DAC'95*. ACM, New York, NY, USA, Article 77. DOI:http://dx.doi.org/10.1145/224818.224959

A. Hatanaka and N. Bagherzadeh. 2007. A modulo scheduling algorithm for a coarse-grain reconfigurable array template. In *IPDPS'07*. 1–8. DOI:http://dx.doi.org/10.1109/IPDPS.2007.370371

Intel-N550. 2010. Intel N550 Datasheet. Retrieved from http://ark.intel.com/products/50154/Intel-Atom-Processor-N550-(1M-Cache-1_50-GHz).

Y. Kim, M. Kiemb, C. Park, J. Jung, and K. Choi. 2005. Resource sharing and pipelining in coarse-grained reconfigurable architecture for domain-specific optimization. In *DATE'05*. IEEE Computer Society, Washington, DC, USA, 12–17. DOI:http://dx.doi.org/10.1109/DATE.2005.260

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

C. Liang and X. Huang. 2009. SmartCell: An energy efficient coarse-grained reconfigurable architecture for stream-based applications. *EURASIP J. Embedded Syst.* 2009, Article 1 (Jan. 2009), [15] pages. DOI:http://dx.doi.org/10.1155/2009/518659

B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. 2002. DRESC: A retargetable compiler for coarse-grained reconfigurable architectures. In *FTP'02*. 166–173. DOI:http://dx.doi.org/10.1109/FPT.2002.1188678

B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. 2003. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. In *DATE'03*. IEEE Computer Society. 296–301. DOI:http://dx.doi.org/10.1109/DATE.2003.1253623

B. Mei, F.-J. Veredas, and B. Masschelein. 2005. Mapping an H.264/AVC decoder onto the ADRES reconfigurable architecture. In *International Conference on Field Programmable Logic and Applications, 2005.* 622–625. DOI:http://dx.doi.org/10.1109/FPL.2005.1515799

B. Mei, M. Berekovic, and J.-Y. Mignolet. 2007. ADRES & DRESC: Architecture and compiler for coarse-grain reconfigurable processors. In *Fine- and Coarse-Grain Reconfigurable Computing*, S. Vassiliadis and D. Soudris (Eds.). Springer Netherlands, 255–297. DOI:http://dx.doi.org/10.1007/978-1-4020-6505-76

H. Park, K. Fan, S. A. Mahlke, T. Oh, H. Kim, and H.-S Kim. 2008. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In *PACT'08*. ACM, New York, NY, USA, 166–176. DOI:http://dx.doi.org/10.1145/1454115.1454140

H. Park, Y. Park, and S. Mahlke. 2009a. Polymorphic pipeline array: A flexible multicore accelerator with virtualized execution for mobile multimedia applications. In *MICRO 42*. ACM, New York, NY, USA, 370–380. DOI:http://dx.doi.org/10.1145/1669112.1669160

H. Park, K. Fan, M. Kudlur, and S. Mahlke. 2006. Modulo graph embedding: Mapping applications onto coarse-grained reconfigurable architectures. In *CASES'06*. ACM, 136–146.

Y. Park, H. Park, and S. Mahlke. 2009b. CGRA express: Accelerating execution using dynamic operation fusion. In *CASES'09*. ACM, New York, NY, USA, 271–280. DOI:http://dx.doi.org/10.1145/1629395.1629433

Y. Park, H. Park, and S. A. Mahlke. 2009. CGRA express: Accelerating execution using dynamic operation fusion. In *CASES'09*. 271–280.

B. Ramakrishna Rau. 1994. Iterative modulo scheduling: An algorithm for software pipelining loops. In *MICRO 27*. ACM.

A. Shrivastava, J. Pager, R. Jeyapaul, M. H., and S. Vrudhula. 2011. Enabling multithreading on CGRAs. In *ICPP'11*. IEEE Computer Society, 255–264. DOI:http://dx.doi.org/10.1109/ICPP.2011.77

H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. Chaves Filho. 2000. MorphoSys: An integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Trans. Comput.* 49, 5 (May 2000), 465–481. DOI:http://dx.doi.org/10.1109/12.859540

J. W. Yoon, J. W. Yoon, A. Shrivastava, S. Park, M. Ahn, R. Jeyapaul, and Y. Paek. 2008. SPKM: A novel graph drawing based algorithm for application mapping onto coarse-grained reconfigurable architectures. In *DAC'08*. 776–782. DOI:http://dx.doi.org/10.1109/ASPDAC.2008.4484056

Received December 2011; revised April 2014; accepted June 2014