Scalable Register File Architecture for CGRA Accelerators

by

Shail Dave

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

Approved November 2016 by the
Graduate Supervisory Committee:

Aviral Shrivastava, Chair
Fengbo Ren
Umit Ogras

ARIZONA STATE UNIVERSITY

December 2016

ABSTRACT

Coarse-grained Reconfigurable Arrays (CGRAs) are promising accelerators capable of accelerating even non-parallel loops and loops with low trip-counts. One challenge in compiling for CGRAs is to manage both recurring and nonrecurring variables in the register file (RF) of the CGRA. Although prior works have managed recurring variables via rotating RF, they access the nonrecurring variables through either a global RF or from a constant memory. The former does not scale well, and the latter degrades the mapping quality. This work proposes a hardware-software codesign approach in order to manage all the variables in a local nonrotating RF. Hardware provides modulo addition based indexing mechanism to enable correct addressing of recurring variables in a nonrotating RF. The compiler determines the number of registers required for each recurring variable and configures the boundary between the registers used for recurring and nonrecurring variables. The compiler also pre-loads the read-only variables and constants into the local registers in the prologue of the schedule. Synthesis and place-and-route results of the previous and the proposed RF design show that proposed solution achieves 17% better cycle time. Experiments of mapping several important and performance-critical loops collected from MiBench show proposed approach improves performance (through better mapping) by 18%, compared to using constant memory.

DEDICATION

To My Parents

for their unconditional love and support

ACKNOWLEDGMENTS

Firstly, I would like to express gratitude towards Prof. Aviral Shrivastava who has been an amazing mentor and teacher throughout the journey of my master's degree. Beginning from my computer architecture course to the end of this work, he has been a very nice source of - knowledge, constructive criticism, and inspiration. Full credit goes to his teaching that stimulated me to pursue the thesis and to join Compiler Microarchitecture Lab (CML), which eventually motivated me to research on the promising technology of Coarse-Grained Reconfigurable Array (CGRA) accelerators.

I am thankful to Dr. Fengbo Ren and Dr. Umit Ogras for guiding me to improve the quality of the thesis and for serving on my committee. VLSI Design classes from Dr. Ogras and Dr. Yu Cao has provided me good insights into the hardware design.

I am especially thankful to Mahesh, Moslem, and Dheeraj for providing me the constructive feedback throughout; nurturing the thought process. I thank other CML colleagues including Mohammadreza, Jian, Yooseong, Edward, Mohammad, Jinn-Pean, my friend Bhavin Nayak, colleagues at Center for Embedded Systems- Shin-Ying, Davesh, Kevin, Jay and Sai Rama for supporting me throughout the journey.

I take this opportunity to thank and appreciate ECEE and CIDSE graduate advisers and the individuals at ASU especially Lynn Pratte, Monica Dugan, and Pamela Dunn. I am also thankful to ASIC team at SanDisk Corporation, Milpitas, CA; which provided me a nice opportunity through my internship to learn and explore new problems. I gratefully acknowledge the support from School of Computing for granting me a teaching/research assistantship. I would also like to thank Design Automation Conference, which awarded me, Richard. A. Newton young student fellowship and an amazing kick-starter experience to the EDA community. Last but not the least, I would like to acknowledge past CML members Mahdi, Shrihari, and Dipal, whose past contributions to CGRA research have been useful resources.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

Need for faster and power-efficient processors has paved the way for multicore and many-core processors along with considerable research in accelerators. Accelerators are special purpose computational units designed to accelerate compute-intensive parts of an application. They can achieve speedup and power-efficiency more than that by multicores alone [1, 2]. Of course, customized accelerators implemented as Application Specific Integrated Circuits (ASICs) can achieve the best power and performance but suffer from poor usability [3]. Field programmable gate arrays (or FPGAs) are reconfigurable and general-purpose but are marred by low power efficiency due to fine-grain management; plus, they are hard to program. [1, 4, 5]. Graphics processing units (GPUs) have made it to the general purpose processor market, accelerating a broad range of parallel applications. Although programmable, their acceleration is limited to parallel loops and loops with higher trip counts [1, 6, 7, 8]. While executing loops with conditionals, GPUs also suffer from extreme performance loss (This well-known problem is referred to as the branch-divergence problem[9]). Hence, although GPUs may provide high acceleration on few kernels [10, 11], many performance-critical loops left unaccelerated, achieving less acceleration at an application level, as per Amdahl's law [12].

Coarse-Grained Reconfigurable Arrays (CGRAs) are an attractive alternative as programmable, yet power efficient accelerators [1], that can accelerate even non-parallel and low trip-count loops. A CGRA is simply an array of processing elements (PEs) interconnected by a 2-D network, as shown in Figure 1.1. Each PE consists of an ALU-like computational unit and a register file (RF). Functional units can

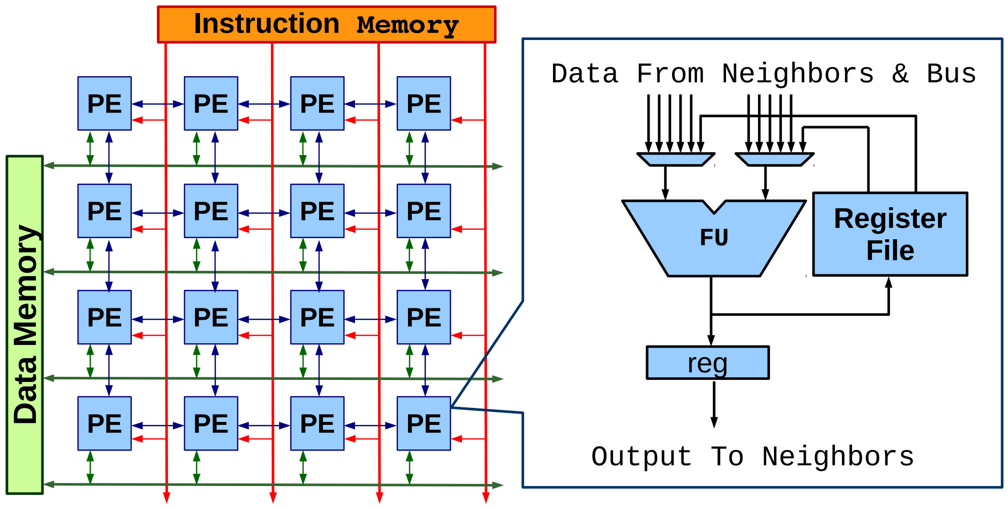

Figure 1.1: A 4×4 CGRA with PEs Connected in a 2-D Mesh. A PE Consists of an ALU and a Register File.

perform arithmetic, logical and comparison operations. At every cycle, contexts are issued from the configuration memory to the PEs, specifying their tasks. Usually, data/address bus are shared either by PEs in the same column or by PEs in the same row. CGRA achieves higher power efficiency due to simpler hardware and intelligent software techniques. CGRAs can achieve power efficiencies of several GOPs per watt [13, 14] and are demonstrated to be power-efficient than even SIMDs [5] for imaging applications. Owing to their power-efficiency, CGRAs are very popular in accelerating applications from multimedia and embedded system domain [15, 16, 17, 18, 19].

One of the key challenges in the efficient use of CGRAs is about managing loop variables using the CGRA registers. There are two kinds of variables in loops: the recurring variables (repeatedly written and read), and the nonrecurring variables (read-only and constants). Previous techniques manage recurring variables in rotating register files [20]. Rotating RF is a specialized hardware which resolves the issue of cross-iteration register overwriting by either rotation of the data through shift registers or by accessing different physical register at each iteration [21]. In addition, the nonrecurring variables are stored and accessed from either from constant memory

[22] or via a global RF [23, 24]. Accessing global/central register file (which may be far from PEs) results in higher cycle time, and accessing constant memory can increase the number of loads, and in turn degrade the performance. [25] manage short and long-lived data through special hardware solutions (shift registers, retiming chain etc.), which can be complex and costly. Although prior works explored different RF architectures [13, 26], they lack in demonstrating the their scalability and in describing the software management of the RF solutions. It is also unclear that how such solutions can be integrated with any register-aware compiler techniques for CGRAs.

This work proposes a hardware-software approach to manage both read-only and recurring values in a non-rotating local (inside the PE) RF. The hardware has a modulo indexing mechanism to the access the RF. The final register index can be computed by adding the register number and the stage count, to access right register for recurring values. Read-only operands are preloaded into local registers before the loop execution. In the software, the compiler reserves necessary registers in the local RF. In addition to this, the compiler provides a configuration to determine the number of registers inside rotating and non-rotating sections of RF. After synthesis and place-and-route of the previous and the proposed RF architectures, results show that proposed RF design achieves 17% better cycle time. Mapping results of various important and compute-intensive loops collected from MiBench show that proposed approach improves performance by 18%, compared to using constant memory.

Chapter 2

BACKGROUND

2.1 Mapping of Loops on CGRAs

Compute-intensive loops are extracted from the target application, and each loop is converted to a data flow graph (DFG) as shown in Figure 2.1(a). DFG is a directed graph *D=(V,E)* where V and E are vertices and edges respectively. Vertices or nodes represent the operations to be executed by PEs and edges represent data dependencies between the operations. A 2×2 CGRA is shown in Figure 2.1(b). A valid mapping of the given DFG (a) on the CGRA (b) is shown in Figure 2.1(c). This mapping can be generated based on the iterative modulo scheduling [20]. The CGRA compiler explicitly performs software pipelining, mapping consecutive iterations of the loop simultaneously on the PE array.

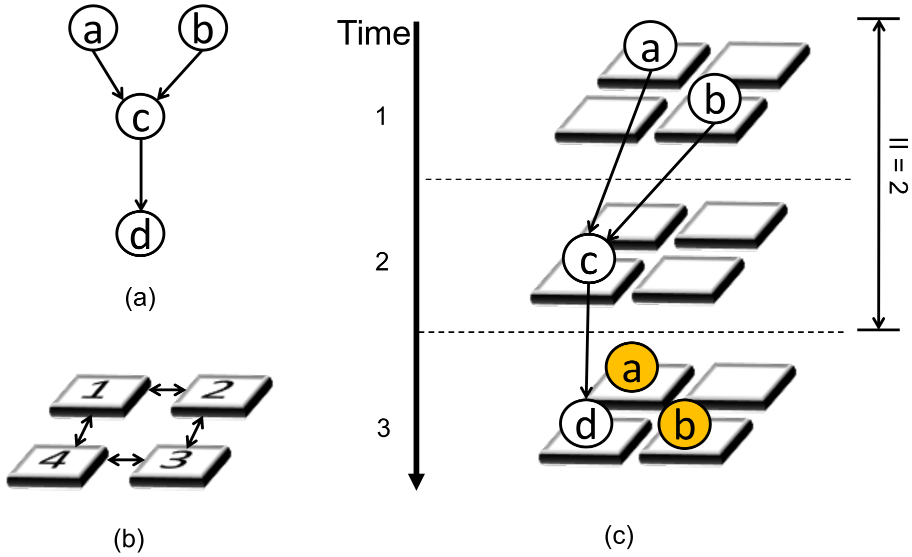

Figure 2.1: (A) DFG of a Simple Loop, (B) a 2×2 CGRA, (C) a Valid Mapping of (A) on (B) with $II = 2$

As shown in Figure 2.1(c), firstly nodes a and b are mapped to PE_1 and PE_3 at time 1, considering consumer node c. Then, nodes c and d are mapped on PE_4 at time 2 and 3, respectively, forming the first iteration of execution. Darker nodes represent the second iteration of the loop, which can start before the completion of the first iteration due to software pipelining. In the iterative modulo schedule, the earliest time at which next iteration can start is called *Initiation Interval* (II) [20], which is an important performance metric. Here, II is 2. There are various techniques to obtain valid mapping for CGRAs [14, 24, 27, 28]. CGRA architecture description should be known a priori to the mapping algorithm to achieve better valid mapping.

2.2 How to Use Registers?

Registers of the CGRA are used to store short-term and long-term values, required by PEs during the loop execution [29]. Since software pipelined schedule is generated, the liveness of the same variable may overlap [27]. Additionally, in accelerating loops with loop-carried dependency, the data values are required across iterations. To address this issue, rotating RF is used to store the values for multiple iterations and nodes can use them, whenever needed. They can be implemented either by rotating the data of the registers through shift registers at every II cycles [25] or by accessing different physical registers through same virtual register index [21, 27]. The latter is implemented through modulo addition of the fixed register index value and stage counter, which increments at every II cycles. Hence, it results in the different physical register index at every II cycles, preserving recurring values.

Figure 2.2(a) represents a DFG of a loop to be mapped on 1×2 CGRA shown in Figure 2.2(b), and Figure 2.2(c) represents a valid mapping with $II = 3$. Each PE has dedicated registers. Here, L computes load address for $a[i]$ with base address $l = \&a[0]$. There is an arc from d to b, with weight 2, indicating the recurrency.

Hence, node b at i^{th} iteration requires the value of d from $(i-2)^{th}$ iteration. Every value of d for at most two iterations should be stored into two different registers to provide values during later iterations. So, we need value of d from different iterations to be managed in different registers. We can see that at time $t+1$, $d^{(i-1)}$ writes its value into register 0 of PE 2. Register 1 of PE 2 contains value of previously computed $d^{(i-2)}$; which can be used by b^i at time $t+2$. Similarly, at time $t+4$, d^i can write resultant value into register 1 of PE 2; preserving value of $d^{(i-1)}$ into register 0 which is needed by $b^{(i+1)}$ at time $t+5$. Hence, recurring values are stored into the rotating RF [24, 27]; different physical registers are accessed at every II cycles with a fixed virtual register index.

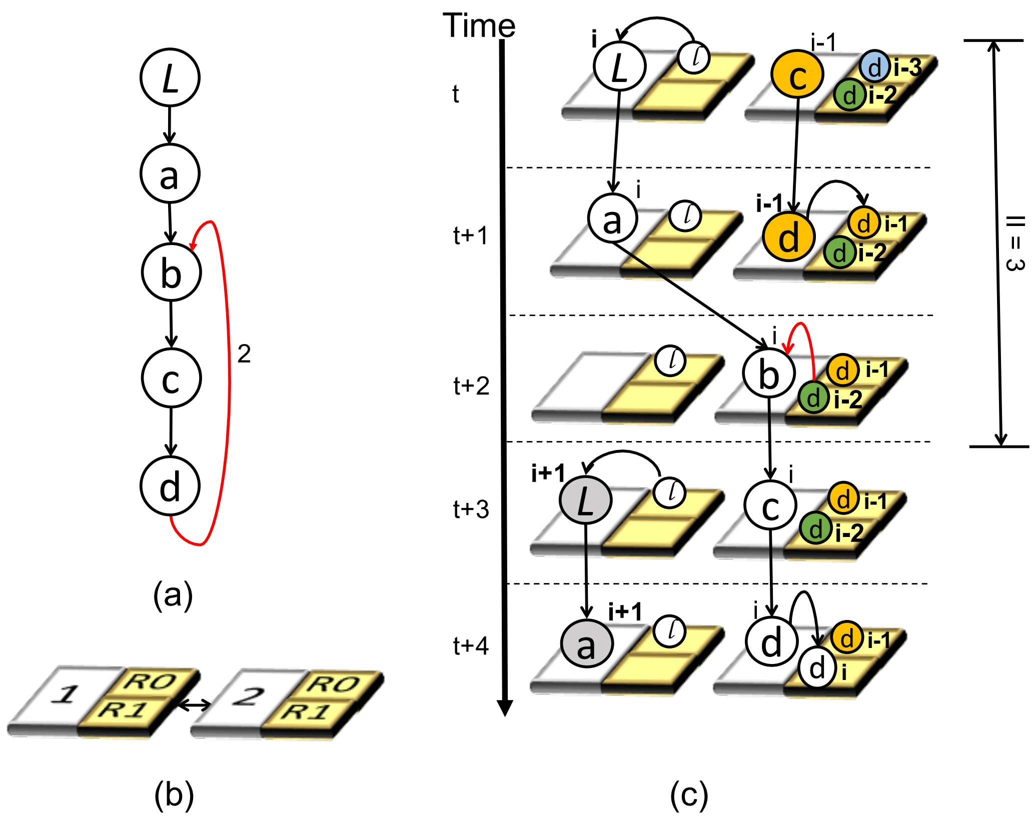

Figure 2.2: (A) DFG of a Loop with Recurrency, (B) a 1×2 CGRA, (C) a Valid Register Aware Mapping of (B) on (C) with $II = 3$

Additionally, PEs need to access nonrecurring variables like read-only operands, live-in data vital for the loop execution, etc. If they are managed in rotating RF, it can cause the registers to be overwritten, resulting in incorrect output. For example, L computes $\&a[i]$ which needs nonrecurring variable $l = \&a[0]$. l is stored into register 0 of PE 1 and should be available throughout the loop execution. Similarly, some nodes may need constant operands, which can not be supplied as immediate bits due to instruction set architecture (ISA) constraints. Since nonrecurring variables should be accessed from the same register index every time, they should be managed in the nonrotating registers. Nonrecurring variables can be part of on-chip memory (L1 cache or a memory bank in scratch-pad memory) [22], as shown in Figure 3.1(a) and can be loaded during the loop execution. Alternatively, they can be stored in a global RF, which is accessible to all PEs [23, 24], as shown in Figure 3.1(b). But, having both rotating and nonrotating RF is inevitable.

Chapter 3

LIMITATIONS OF PRIOR APPROACHES

Register file architectures can be broadly classified as 1) Global RF 2) Local RF and 3) Shared RF [27, 28]. As the name suggests, global RF is a centralized RF, accessed by all PEs as shown in Figure 3.1(b). Local RFs are RFs dedicated to each PE of the CGRA, as shown in Fig 3.1(c). Shared RFs allow data sharing between neighboring PEs. Depending on the design choice, various PEs can access these structures, introducing heterogeneity. [13, 27].

Mostly, these register files are used to keep the recurring data, generated and needed throughout the loop execution. Consequently, these RFs can be rotating RFs, used for accessing recurring variables. So, one way to access constants can be loading the values from on-chip memory (also referred to as constant memory). Accessing constant memory [22] is simple, but it results in extra load operations, which can degrade the performance. In fact adding more loads can be much more harmful because of 2 main reasons: i) in most CGRAs, only a few of the PEs can perform the memory operations [24], ii) Often the load/store bandwidth in CGRAs is limited, e.g., data and address buses are typically usually shared by PEs in a row or by PEs in a column [30].

Past works like [23, 24] have considered this issue of pressure on the memory. They manage rotating data into local RF and reserve the nonrotating registers into a global RF, to manage live-in (needed for the loop execution) or live-out (to be stored back at the end) values, as shown in Figure 3.1(b). Such global RF is then a non-rotating or regular RF. Managing variables through global RF allows data sharing between PEs without external routing. Although this may save from redundant

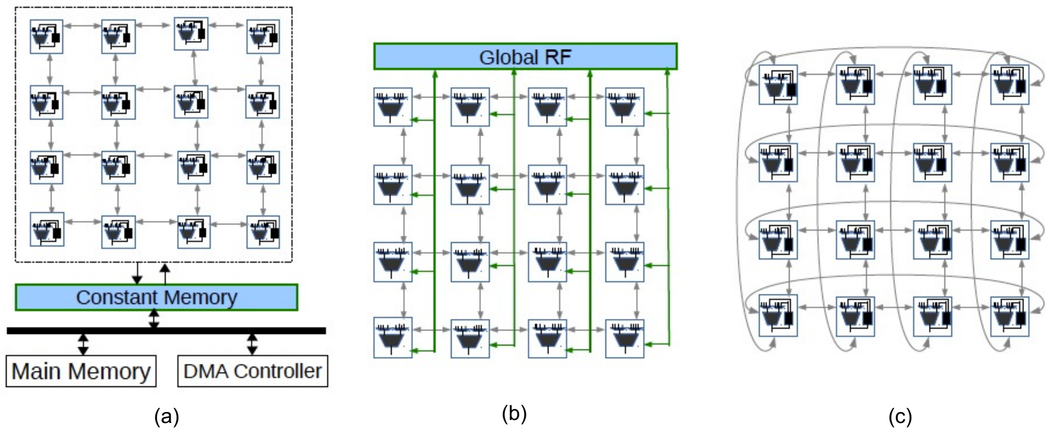

Figure 3.1: (A) CGRA with On-chip Constant Memory. (B) CGRA with Global RF Where Each PE Is Connected to Global RF Through Column-wise Bus Structure. (C) CGRA with Local RF.

register reservations to store single value for different PEs, the global RF design does not scale well. Experiments have shown the need for connecting all the PEs to the global RF [29]. But, increasing read and write ports for more PEs to access global RF can result in performance degradation and increase in total area [29]. Further, the addition of a global RF size burdens ISA, as the number of bits representing read and write registers increases. For example, each PE would require 18 bits to access a 64 register global RF, as both inputs can be from the registers and a PE can write to global RF. Increasing ISA width results in increased memory bus width, increase in context size and in-turn, increase in area and power.

Bouwens et al. [13] proposed a shared RF solution, which is better for data sharing within the neighbors and may scale well, but no application mapping and register allocation scheme has been proposed. Similarly, [26] explored different RF solutions targeting combinations of both rotating and non-rotating RFs. However, how such solutions can reserve and allocate registers for a given kernel, their software management and their integration with a compiler technique, is not proposed. Besides, scalability and the effectiveness of the solutions compared to prior variable management schemes is not demonstrated.

Local RFs are usually smaller and accessing the data through them can provide scalability achieving better performance. Dedicated or local RFs are considered as a better alternative, and it helps to obtain better performance [23, 27]. So, this work focuses on managing both recurring and nonrecurring variables locally within PEs. We can have separate rotating and nonrotating RFs locally, as demonstrated in [25, 26]. But, utilizing registers effectively becomes a challenge. Also, additional complex hardware structures can consume more area and power. It makes design decisions more challenging, and naive architectural choice can deteriorate the performance.

This work proposes to use a local unified non-rotating RF, in which both recurring and nonrecurring variables are managed. The proposed hardware-software approach is scalable and does not burden the ISA. The compiler has to configure the boundary between the registers used for recurring and nonrecurring variables, do necessary register allocations, and preload the constants and read-only variables.

Chapter 4

PROPOSED APPROACH: UNIFIED RF

To make the cut for the demand of efficient management of both recurring and nonrecurring variables locally in a single RF, this work presents local unified RF as a scalable solution. The complexity lies in the hybrid hardware-software approach where the hardware is kept simple, though configurable, with a regular (nonrotating) RF; rotation is implemented through modulo addition with register index [26, 27]. Nonrecurring variables can be preloaded into registers inside nonrotating section and are available throughout the execution. The compiler reserves appropriate registers for RF of fixed size; during mapping, one problem is in the efficient utilization of registers to manage all variables in RF of fixed size. The proposed approach can be integrated with any mapping technique for CGRAs as shown later in section 4.4. Through this solution, it is shown that how compiler takes on the challenge by mapping operations based on the available registers. It enforces efficient register utilization for numerous loops by providing a configuration to decide the number of registers in rotating and nonrotating part of the unified RF.

4.1 Accessing Registers in Unified RF

Designing unified RF with simpler hardware jettisons the use of specialized components including shift registers. So, a regular RF is used and a modulo addition of the register index and stage counter is performed in order to implement the rotation. The stage counter is incremented at the end of every II cycle and given a read/write operation, virtual register index remains constant. An overflow of the stage counter

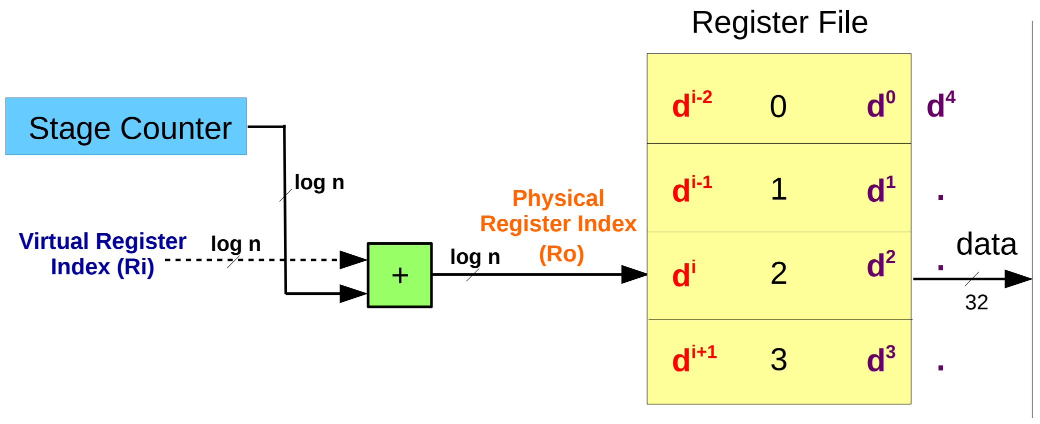

Figure 4.1: Rotating RF with Regular Register File. Rotation Is Implemented Through Modulo Operation on Register Index.

and adder results in modulo operation [27]. In this way, with same register index, different physical registers are accessed at each II cycle. Figure 4.1 shows RF with four registers to manage the value of a variable d across four different iterations. Mapping requires writing d^i in virtual register 0. Hence, for iteration 2, with stage counter as 2, d^2 is written into physical register 2. At the same time, we can access the older value during iteration 0 (d^0) from register 0. In the next II cycle, with stage counter as 3, d^3 is written into physical register 3. Next, stage counter is overflowed with value 0 and d^4 is written in physical register 0 and so on. Such implementation requires the total number of rotating registers as the power of 2 [27].

Figure 4.2 shows local unified RF with both rotating and nonrotating parts. Read reg1, read reg2 and write are register index for the read and write operations, respectively. The control unit provides the value of c; c is the maximum register index inside rotating section and decides the boundary between rotating and nonrotating parts. It gives the flexibility to support different register requirements for different loops. It is explained later in this section, about how to decide the value of c. If the register index is less than or equals to c, then we need to access rotating section. The select signal is generated as 0 and RF can be accessed for recurring values, as described through Figure 4.1. In this case, the addition of the given register index and stage counter is done. Then, the modulo operation is performed by ANDing the result with c, as $c + 1$ is the total number of rotating registers [26]. The output of the AND gate locates correct register number to read/write recurring values. When stage counter reaches the value of c, it is reset to zero for the next iteration. On the other hand, if the register index is greater than c then, the control unit generates a select signal as 1 and register index just bypasses the adder driving the read/write port of the RF. Hence, both recurring and read-only values can be managed in the unified RF by using the simple hardware.

At the beginning of the loop execution, the read-only operands are pre-loaded into local registers of nonrotating part of the RF of corresponding PEs. As the compiler is aware of the PEs that require live-in values or read-only operands, it can generate instructions accordingly (as part of the prologue) to pre-load them.

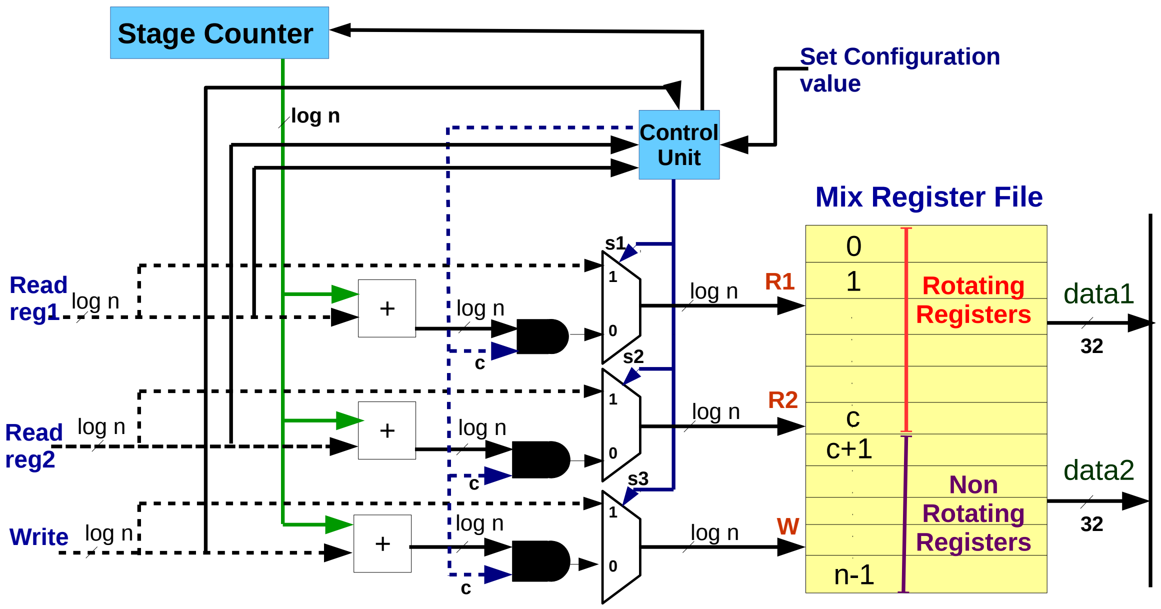

Figure 4.2: Local Unified RF with Regular Register File. Configuration by Compiler Decides Number of Registers Inside Rotating and Nonrotating Sections.

4.2 Determining the Register Requirement During Mapping

By analyzing the mapping of the operations on PEs, it is easy to find the number registers that each node requires to manage the nonrecurring variables in a PE and that for a recurring operation. This analysis would reveal the minimum number of registers required inside the rotating and nonrotating sections of unified RF. Algorithm 1 provides the number of nonrotating registers essential to map an operation. If any of the operands is constant and if its value is larger than maximum value supported by immediate bits in the PE instructions, such nonrecurring variable can be pre-loaded in the reserved nonrotating register. In this way, live-in data can be preloaded and managed in the RF, which can be then used by the operands during the kernel execution.

Algorithm 1: *getNonrotatingRegisters*(Input Node v_i, Input PE p_i)

```
begin
    total_operands ←get_number_of_operands($v_i$);
    operands[total_operands] ←get_operands($v_i$);
    nonrotating_reg ← 0;
    $i \leftarrow 0$;
    while $i < total\_operands$ do
        $o_i \leftarrow operands[i]$;
        if (($is\_constant(o_i)$ && ($is\_greater\_than\_max\_immediate(o_i)$)) then
            nonrotating_reg++;
        $i$++;
    $nonrotating\_reg\_per\_PE[p_i] += nonrotating\_reg$;
    return nonrotating_reg;
```

Algorithm 2: *getRotatingRegisters*(Input Node v_i, Input PE p_i)

```
begin
    total_successors ←get_number_of_successors($v_i$);
    successors[total_successors] ←get_successors($v_i$);
    rotating_reg ← 0;
    $i$ ← 0;
    while $i < total\_successors$ do
        $s_i \leftarrow successors[i]$;
        if ($isMappedMoreThanACycleApart(v_i, s_i)$) then
            $reg\_needed \leftarrow calculate\_distance\_and\_reg\_requirement(v_i, s_i)$;
            if ($reg\_needed > rotating\_reg$) then
                $rotating\_reg \leftarrow reg\_needed$;
        $i$++;
    $rotating\_reg\_per\_PE[p_i] += rotating\_reg$;
    return rotating_reg;
```

Similarly, algorithm 2 provides the number of rotating registers required to preserve recurring values. Given an operation, it checks for the type of the dependency between the operations. With the availability of absolute mapping times of the node and its successor node; the mapping distance can be calculated in terms of II [14]. In this way, correct register requirements can be computed to map each of the operation and its successor pairs for both intra-iteration dependency and loop-carried dependency [14]. Finally, the algorithm provides a total number of rotating registers required to map a node v_i on PE p_i. These both algorithms together enable a mapping technique to reserve corresponding registers during the mapping.

4.3 Register Reservation for Efficient Usage

A natural choice to design local unified RF is through the fixed hardware, where it can have pre-decided numbers of registers inside the rotating and nonrotating sections [25]. Once the number of rotating and nonrotating registers required are known, the RF size can be fixed. Although making such design choices for target loop(s) is one alternative, it lacks the support for the general purpose computing and may not always work. For example, different loops in the target application(s) can require the different number of rotating and nonrotating registers. In such scenarios, the RF size should be increased to accommodate the requirement; else a valid mapping may not be achieved. An important fact is that it is not the issue of local unified RF, but is, in general, an issue of managing both types of variables through any RF architecture.

Algorithm 3 shows register reservation inside the rotating and nonrotating sections of local unified RF. It keeps track of register allocation per PE for mapped operations. Calculation of the number of nonrotating and rotating registers, required by the current node is provided by the functions of Algorithm 1 and 2, respectively. Before the register reservation, Algorithm 3 ensures register availability, based on the past allocation. $reg[p_i]$ indicates total registers utilized by PE p_i. For a PE p_i, based on the reservation of registers inside rotating and nonrotating sections, we can configure boundary between two sections, for each loop execution. This configuration boundary can vary for different mappings of different loops as we obtain different combinations of registers inside rotating and nonrotating section with a total of N registers. With this unique feature, the mapping technique is capable of efficient register usage due to reserving any number of registers inside nonrotating and rotating section, finding a valid mapping for various loops, without changing the hardware. The effectiveness of such reservation is demonstrated later in the section 5.3.

Algorithm 3: *Reserve_Registers*(Input PE p_i, Input RF size N,Input node v_i)

```
begin
    $n \leftarrow reg[p_i]$ ;                                   // Total reserved registers
    $r_1 \leftarrow$get_number_of_nonrotating_registers($v_i, p_i$);
    $r_2 \leftarrow$get_number_of_rotating_registers($v_i, p_i$);
    $r \leftarrow$get_nearest_power_of_two(configuration[$p_i$] $+ r_2$);
    $nr \leftarrow r_1 + (n - configuration[p_i])$ ; // Nonrotating registers required
    $n' \leftarrow r + nr$ ;                                     // Total registers needed
    $n \leftarrow n + r_1 + r_2$ ;                          // Total registers to be reserved
    if $n' \leqslant N$ then
        $reg[p_i] \leftarrow n$ ;                           // Update actual registers needed
        $configuration[p_i] \leftarrow$ configuration[$p_i$] $+ r_2$ ; // Update configuration
        $configuration\_power\_of\_two[p_i] \leftarrow r$;
        return true;
    return false;
```

Moreover, Algorithm 3 ensures that size of the rotating section is equal to the nearest power of 2, calculated as r, satisfying the constraint due to modulo addition implementation. Total reserved registers n' should be less than RF of size N. Once the register requirements are met, currently reserved registers $reg[p_i]$ and current configuration boundary $configuration[p_i]$ is updated accordingly. $configuration\_power\_of\_two[p_i]$ is the final configuration, aligned to the nearest power of 2. At the end of the mapping, instructions for configuring the RF of each PE can be generated based on $configuration\_power\_of\_two[p_i]$ and fed to control unit of Figure 4.2. Control unit generates c as $configuration\_power\_of\_two[p_i]$ - 1, enforcing a boundary between the rotating and nonrotating sections which aids to calculate the modulo register index.

4.4 Integration with a CGRA Mapping Technique

Figure 4.3 shows a flowchart with a high-level overview of an integration of the proposed methodology with a CGRA mapping technique. Firstly, it takes a DFG as an input, generates a modulo schedule and forms set of edges/clusters/cliques, followed by initial cost calculation or initial resource reservation. Then, it tries to map an operation from the set on a PE. If it finds a PE slot for that operation, it checks for register availability else finds another PE. If no other PE is available, it increases II by 1. Register reservation can be made through the algorithm 3. If II value crosses the preset limit, it terminates mapping, resulting in failure. Upon successfully mapping all the operations, a valid mapping is generated, ensuring that nodes are mapped on PEs targeting the register availability within unified RF. In this way, proposed solution can be combined with any CGRA compiler technique.

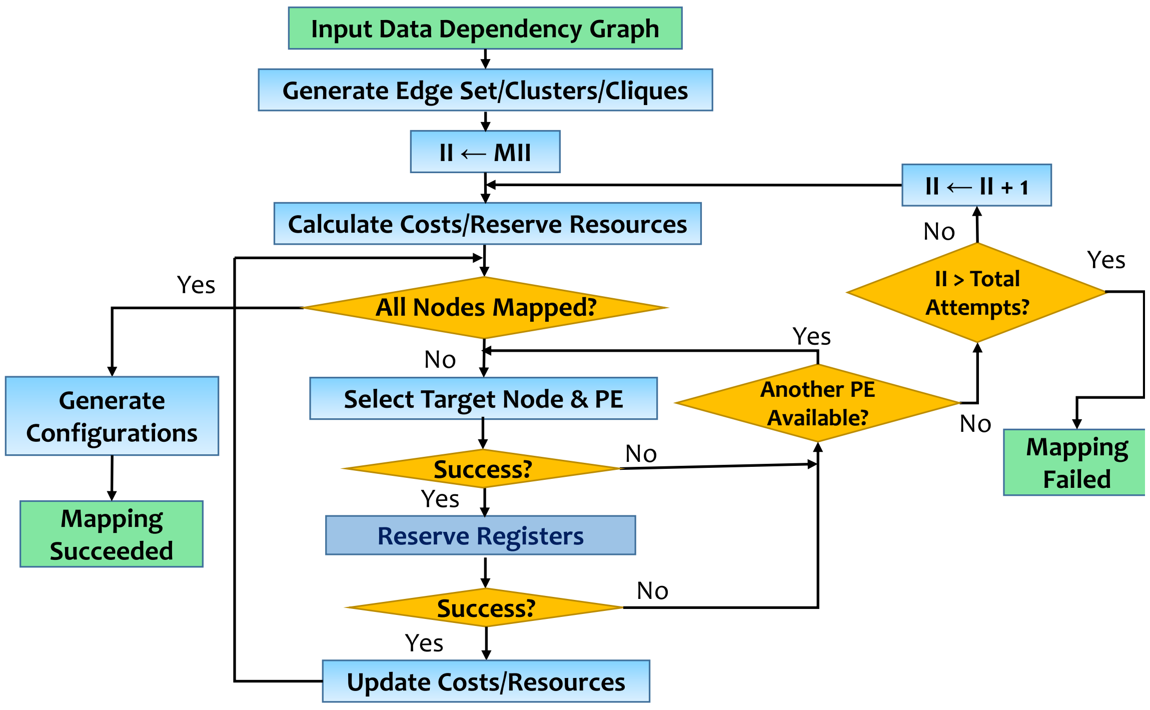

Figure 4.3: High Level View of Register Reservation Function Integrated with a CGRA Mapping Technique

Chapter 5

EXPERIMENTAL RESULTS

For the mapping experiments throughout, the baseline target architecture is a 4×4 homogeneous CGRA, connected in a 2D torus as shown in Figure 3.1(c). PEs are capable of performing fixed-point operations with the latency of 1-cycle. The memory bus is shared among PEs in a row. So, only one PE from a row can access the data memory during each cycle. For the load and store operations, two instructions are executed; one generates the address and other loads/stores the data. Load or store of the data takes place in the same row where the corresponding address is generated, and the bus is asserted. In the target architecture, each PE can access RF of four 32-bit registers. For evaluation of the approach, state-of-the-art register aware compiler technique for CGRAs REGIMap [14] is used. Several innermost loops are extracted from MiBench [31] benchmarks, which are computationally critical or important to represent a variety of application kernels.

As the incorporated CGRA ISA has a 12-bit immediate field, read-only data up to 12-bits can be provided as an immediate value. It is assumed that all the approaches compared can manage up to 12-bit value as immediate. In the proposed approach, for preloading 32-bit value, three cycles are required to load a nonrecurring variable (due to a provision of a 12-bit immediate field with ISA). During the kernel execution, some intermediate values are generated which are read-only and live within II cycles (a shorter period than even the execution of one loop iteration). To be fair in comparison, it is assumed that prior works also store them in either local rotating registers or global RF, instead of using constant memory. The proposed approach can store them into local registers inside nonrotating part.

To show the need for managing all variables within scalable local RF, proposed solution is compared with the approach of accessing data from global RF. Global RF yields mappings of equal quality in most of the cases. Hence,the evaluations are done on CGRAs of different sizes, from 4×4 to 32×32, demonstrating the scalability directly at RTL level. Then, it takes on comparing with accessing data through constant memory. In this case, cycle time does not get affected and consequently, comparison take place in terms of the mapping quality only. All of these experiments demonstrate the efficacy of the claims about how prior approaches can degrade the performance. Results are validated on a cycle-accurate simulator gem5 [32].

5.1 Local Unified RF Achieves Better Cycle Time

To compare against state-of-art approaches that manage nonrecurring variables into global RF, the RTL of CGRA with local unified RF and that of CGRA with global RF shown in Figure 3.1(b), is modeled. RTL is synthesized and taken through ASIC flow using the Cadence RTL compiler with 32 nm standard cell library. The functionality of the CGRA implementation is verified at every step of the ASIC flow. To demonstrate the effectiveness of the RTL implementation using Verilog, the measurements are taken at 250 MHz, as previous works targeted frequencies in the range of 100-200 MHz [14, 15, 16, 27]. Power estimations are done using

Table 5.1: Results for 4x4 CGRA @ 250 MHz for 32 nm CMOS Technology

Parameter	Local Unified RF	Global RF
Number of Registers	4 per PE	64
Area (sq. um)	463915	541822
Power (mW)	127	125

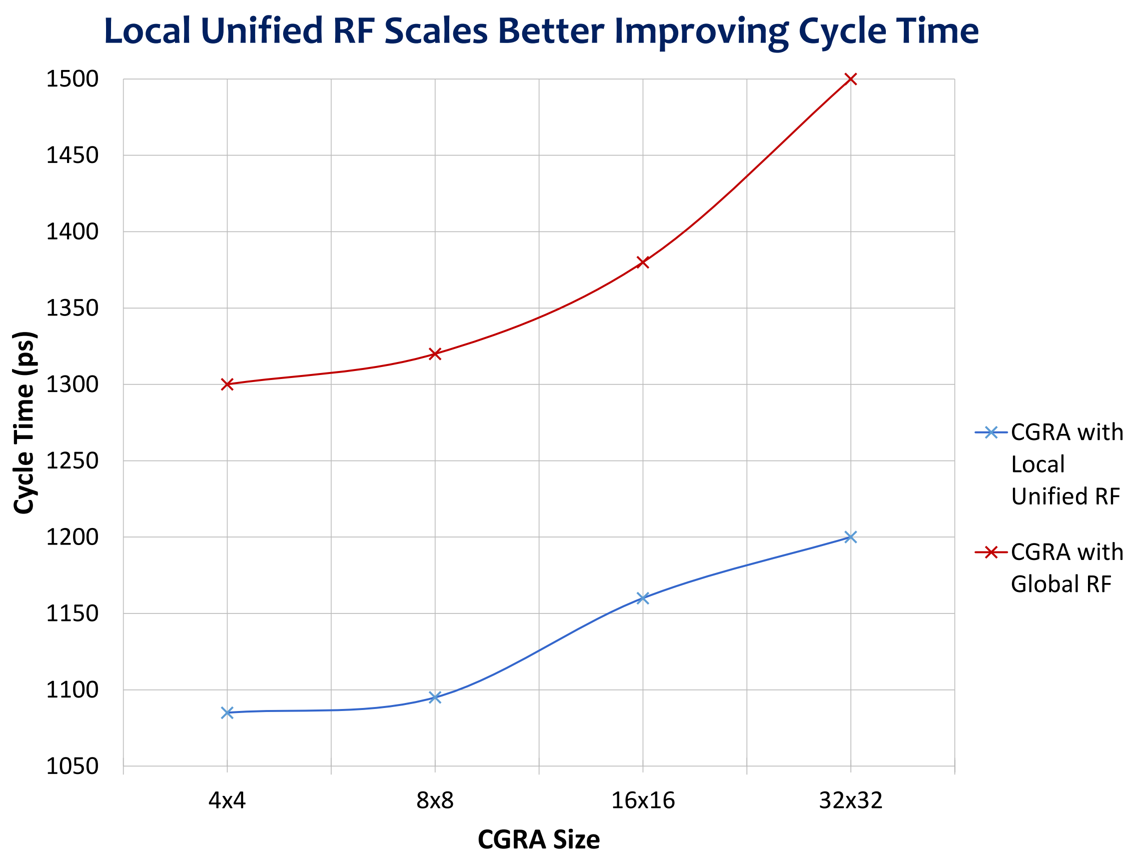

Figure 5.1: Comparison of Latency for CGRA with Local Unified RF & Global RF

Synopsys PrimeTime. Measurements in Table 5.1 show that implemented 4×4 CGRA architecture can achieve the maximum power efficiency of 32 GOPS/W and on average that of 21.5 GOPS per watt due to IPC of 10.75 for REGIMap [14], which is in the expected range.

For both global RF and local RF, the cycle time and read-write access latency to RFs is compared, at their best frequencies. Global RF structure chosen is one shown in Figure 3.1(b) and described in backgrounds section. Results are demonstrated for various CGRA sizes, to highlight the issue of the scalability. From Figure 5.1, we can see that for the different size, cycle time increases rapidly for CGRA with global RF, as compared to the proposed solution. On average, use of local unified RF reduces cycle time by 17.38% and, read access latency by 23.38%. These results ensure the need for managing nonrecurring and recurring variables locally. It reduces the cycle time and in-turn, total execution time, compared to past approaches. Hence, unified local RF turns out to be scalable solution improving the performance.

5.2 Local Unified RF Improves Performance

Another way to access read-only operands is through on-chip constant memory, as discussed in section 3. The proposed solution of local unified RF is compared to the CGRA with constant memory and it is shown that how we can improve performance by eliminating additional loads, through the proposed approach. To access the nonrecurring variables from constant memory, load operations are performed by using 4 KB on-chip memory.

Figure 5.2 compares the II between the proposed solution and data management

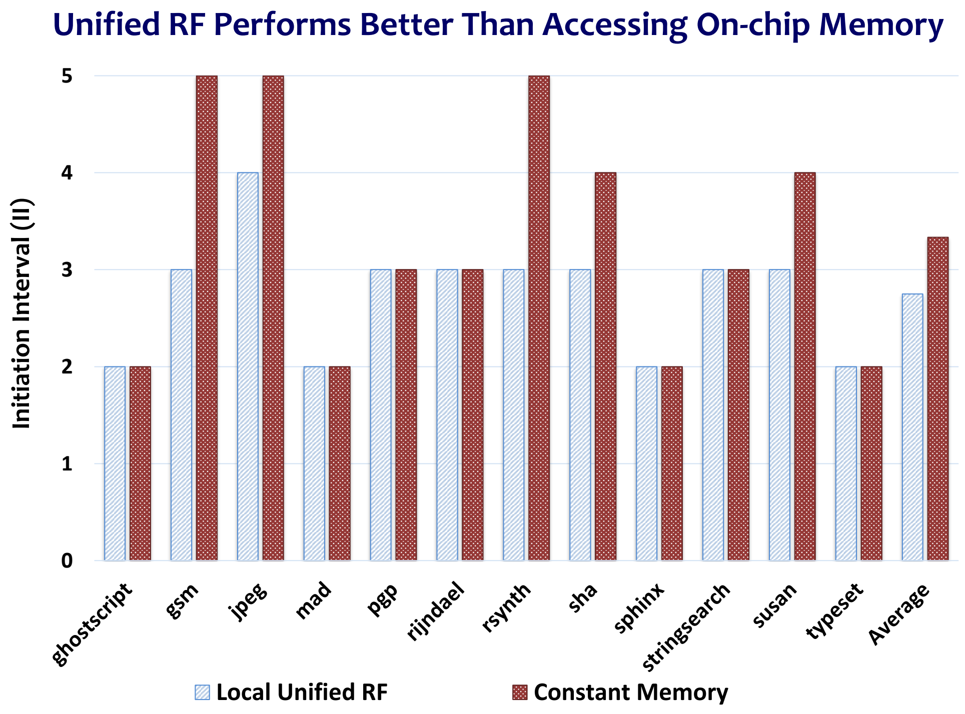

Figure 5.2: Managing All the Variables Within Local Unified RF Reduces II as Compared to CGRA Accessing On-chip Constant Memory.

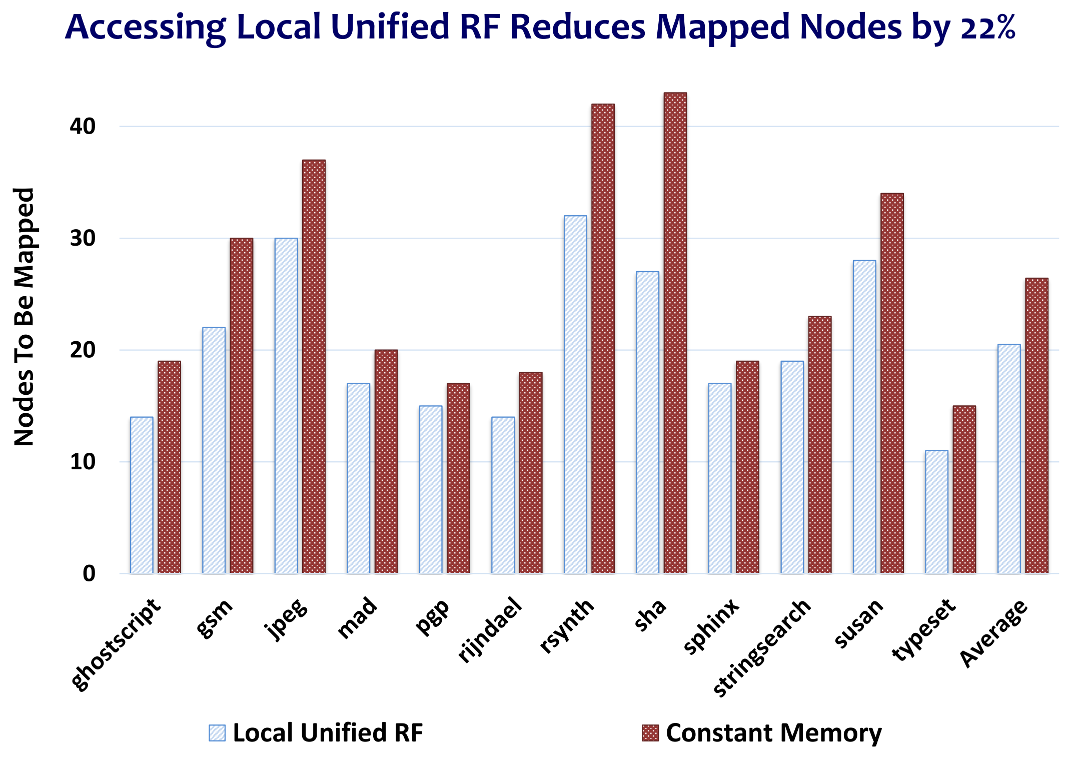

Figure 5.3: Managing Data with Local RF Can Eliminate Additional Load Operations, Compared to Accessing Constant Memory

through constant memory. On average, the proposed approach reduces II is reduced by 17.5%. Figure 5.3 shows a reduction in mapped nodes by 22.40%, due to the elimination of additional load-cycles required in the case of accessing data from constant memory. We can see the effectiveness of using local RF, which eliminates 10 and 16 load operations respectively, in the case of *rsynth* (office) and *sha* (security), as compared to accessing constant memory. Hence, it enforces a reduction in II, improving the performance. On the other hand, loops from *rijndael* (security) and *stringsearch* (office) achieve same II, but reduces additional load nodes by 4 and 2, respectively. Consequently, proposed approach reduces the pressure on memory and saves PE resources from consuming unnecessary power.

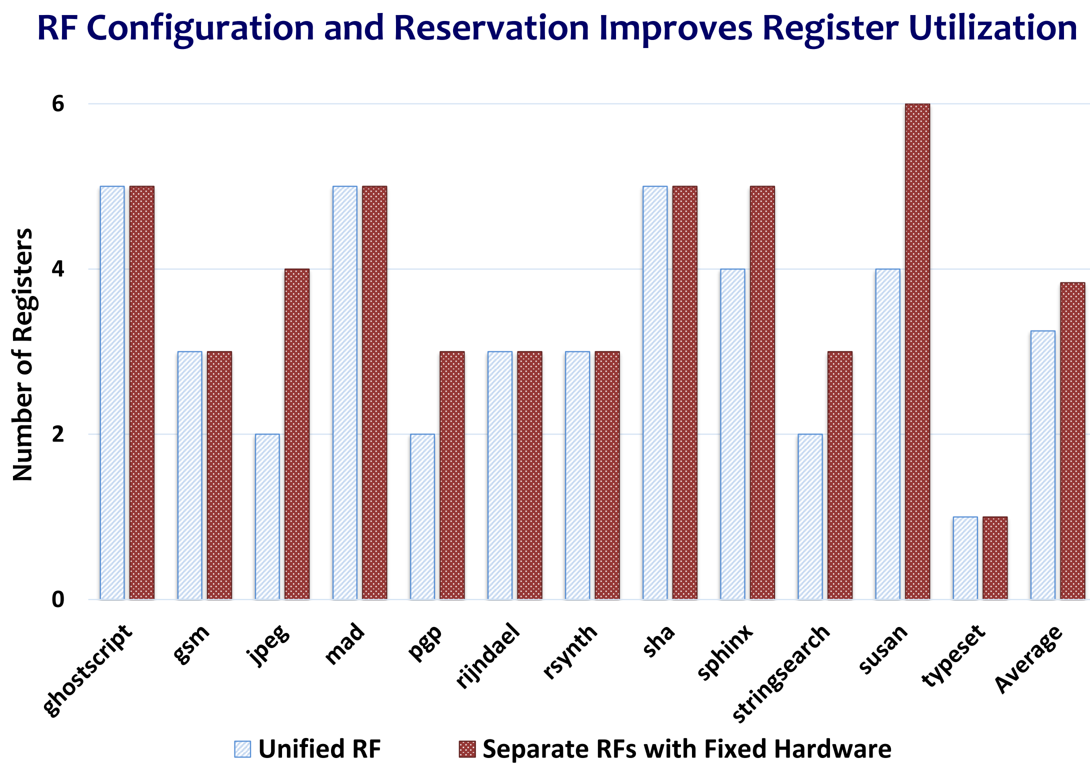

Figure 5.4: For a given Mapping, Provision of RF Configuration Results in Reduced RF Size, as Compared to RF with Fixed Numbers of Registers.

Moreover, on average pre-loading cycles are 3.5, varying between 3-6 for different loops. This means that the pre-loading cycles are negligible compared to the gain of II reduction by 18%. Otherwise, executing a performance-critical loop in 100,000 cycles will still take additional 1800 CGRA cycles even with the assumed load latency of 1 cycle. This emphasizes the requirement of accessing data within local RF instead of doing so from on-chip memory.

5.3 Unified RF Reduces Register Requirements

To show the effectiveness of the proposed register reservation approach, the registers required for various loops are compared, as shown in Figure 5.4. Evaluations show that for a given mapping, use of RF configuration reduces RF size by 14.26%, as compared to RF with fixed number of rotating and nonrotating registers. For example, *susan* requires two nonrotating and four rotating registers for different PEs. So, in the case of fixed number of registers, we need RF of size six for all PEs. But through the proposed approach, we can have local RF of four registers for all PEs; one PE can configure all four registers as rotating and other can opt for RF as a nonrotating section only. Thus, proposed approach results into better register usage. Similarly, at the architectural level, prior approaches of fixed RF hardware needs 4 registers PE, totaling 64 registers for a 4×4 array. However, proposed approach needs on average about 3 registers, totaling just 48 registers. In this way, proposed RF configuration and register management can result in efficient register utilization.

Chapter 6

SUMMARY

This work advocates for a scalable solution to manage both recurring and non-recurring variables effectively in a single nonrotating RF. Firstly, the issues related to state-of-the-art techniques of data management i.e. accessing data via global RF and from constant memory are discussed. The former increases cycle time and does not scale and the latter increases II due to additional load operations. Then, local unified RF, a novel approach to managing all variables exclusively through single RF is proposed, which is an efficient solution. This increases complexity at compiler level but, results in better performance with efficient register utilization. Finally, the advantages of the technique are presented by comparing it with prior approaches of accessing variables, regarding scalability and performance of the CGRA. From the experiments done, it can be concluded that local unified RF is better than other existing solutions to manage variables efficiently on-chip.

Chapter 7

FUTURE WORK

Some of the future work can be visualized in the purview of both CGRA compilers and register file architectures and can be encapsulated as follows.

- One problem in CGRA design is an effective use of registers across various RF structures. For example, if the array design must use both local and global RFs, which registers should be allocated by the compiler first while mapping? Such decisions critically affect the performance and design costs, leaving a scope for such optimization techniques. The well-designed scheme can achieve much higher power-efficiency in such scenarios.

- CGRA register files do contain enough registers for a kernel acceleration. However, in many cases, few operations (say one with a large inter-iteration dependency) may require a higher number of registers than what is accessible to the processing elements. In such cases, break-down of such register requirement and distributed allocation of the registers can be unique and can certainly improve the design and performance of these promising accelerators.

REFERENCES

[1] Allan Carroll, Stephen Friedman, Brian Van Essen, Aaron Wood, Benjamin Ylvisaker, Carl Ebeling, and Scott Hauck. Designing a coarse-grained reconfigurable architecture for power efficiency. In *Department of Energy NA-22 University Information Technical Interchange Review Meeting*, 2007.

[2] George Theodoridis, Dimitrios Soudris, and Stamatis Vassiliadis. A survey of coarse-grain reconfigurable architectures and cad tools. In *Fine-and Coarse-Grain Reconfigurable Computing*, pages 89–149. Springer, 2007.

[3] David Koeplinger, Christina Delimitrou, Raghu Prabhakar, Christos Kozyrakis, Yaqi Zhang, and Kunle Olukotun. Automatic generation of efficient accelerators for reconfigurable hardware. 2016.

[4] Shuai Che, Jie Li, Jeremy W Sheaffer, Kevin Skadron, and John Lach. Accelerating compute-intensive applications with gpus and fpgas. In *Application Specific Processors, 2008. SASP 2008. Symposium on*, pages 101–107. IEEE, 2008.

[5] Artem Vasilyev, Nikhil Bhagdikar, Ardavan Pedram, Stephen Richardson, Shahar Kvatinsky, and Mark Horowitz. Evaluating programmable architectures for imaging and vision applications. 2012.

[6] Opencl optimization case study: Simple reductions. `http://developer.amd.com/resources/articles-whitepapers/opencl-optimization-case-study-simple-reductions/`, 2010. Accessed: 2010-08-24.

[7] Mehrzad Samadi, Amir Hormati, Janghaeng Lee, and Scott Mahlke. Paragon: collaborative speculative loop execution on gpu and cpu. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, pages 64–73. ACM, 2012.

[8] Shri Hari Rajendran Radhika, Aviral Shrivastava, and Mahdi Hamzeh. Path selection based acceleration of conditionals in cgras. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015*, pages 121–126. IEEE, 2015.

[9] Tianyi David Han and Tarek S Abdelrahman. Reducing branch divergence in gpu programs. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, page 3. ACM, 2011.

[10] Victor W Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, et al. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. *ACM SIGARCH Computer Architecture News*, 38(3):451–460, 2010.

[11] Richard Vuduc, Aparna Chandramowlishwaran, Jee Choi, Murat Guney, and Aashay Shringarpure. On the limits of gpu acceleration. In *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, pages 13–13. USENIX Association, 2010.

[12] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[13] Frank Bouwens, Mladen Berekovic, Bjorn De Sutter, and Georgi Gaydadjiev. Architecture enhancements for the adres coarse-grained reconfigurable array. In *High Performance Embedded Architectures and Compilers*, pages 66–81. Springer, 2008.

[14] Mahdi Hamzeh, Aviral Shrivastava, and Sarma Vrudhula. Regimap: Register-aware application mapping on coarse-grained reconfigurable architectures (cgras). In *Proceedings of the 50th Annual Design Automation Conference*, page 18. ACM, 2013.

[15] Seth Copen Goldstein, Herman Schmit, Matthew Moe, Mihai Budiu, Srihari Cadambi, R Reed Taylor, and Ronald Laufer. Piperench: a co/processor for streaming multimedia acceleration. *ACM SIGARCH Computer Architecture News*, 27(2):28–39, 1999.

[16] Hertej Singh, Ming-Hau Lee, Guangming Lu, Fadi J Kurdahi, Nader Bagherzadeh, and M Chaves Eliseu Filho. Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications. *Computers, IEEE Transactions on*, 49(5):465–481, 2000.

[17] Changmoo Kim, Mookyoung Chung, Yeongon Cho, Mario Konijnenburg, Soojung Ryu, and Jeongwook Kim. Ulp-srp: Ultra low power samsung reconfigurable processor for biomedical applications. In *Field-Programmable Technology (FPT), 2012 International Conference on*, pages 329–334. IEEE, 2012.

[18] Venkatraman Govindaraju, Chen-Han Ho, Tony Nowatzki, Jatin Chhugani, Nadathur Satish, Karthikeyan Sankaralingam, and Changkyu Kim. Dyser: Unifying functionality and parallelism specialization for energy-efficient computing. *IEEE Micro*, 5(32):38–51, 2012.

[19] Yu-Hsin Chen, Tushar Krishna, Joel Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. In *IEEE International Solid-State Circuits Conference, ISSCC 2016, Digest of Technical Papers*, pages 262–263, 2016.

[20] B Ramakrishna Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the 27th annual international symposium on Microarchitecture*, pages 63–74. ACM, 1994.

[21] James C Dehnert and Ross A Towle. Compiling for the cydra 5. In *Instruction-Level Parallelism*, pages 181–227. Springer, 1993.

[22] Hee-Seok Kim, Minwook Ahn, John A Stratton, and Wen-mei W Hwu. Design evaluation of opencl compiler framework for coarse-grained reconfigurable arrays. In *Field-Programmable Technology (FPT), 2012 International Conference on*, pages 313–320. IEEE, 2012.

[23] Taewook Oh, Bernhard Egger, Hyunchul Park, and Scott Mahlke. Recurrence cycle aware modulo scheduling for coarse-grained reconfigurable architectures. In *ACM Sigplan Notices*, volume 44, pages 21–30. ACM, 2009.

[24] Hyunchul Park, Kevin Fan, Scott A Mahlke, Taewook Oh, Heeseok Kim, and Hong-seok Kim. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 166–176. ACM, 2008.

[25] Brian Van Essen, Robin Panda, Aaron Wood, Carl Ebeling, and Scott Hauck. Managing short-lived and long-lived values in coarse-grained reconfigurable arrays. In *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, pages 380–387. IEEE, 2010.

[26] Dipal Saluja. Register file organization for coarse-grained reconfigurable architectures: Compiler-microarchitecture perspective. Master's thesis, Arizona State University, 2014.

[27] B Mei, M Berekovic, and JY Mignolet. Adres & dresc: Architecture and compiler for coarse-grain reconfigurable processors. In *Fine-and coarse-grain reconfigurable computing*, pages 255–297. Springer, 2007.

[28] Hyunchul Park, Kevin Fan, Manjunath Kudlur, and Scott 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*, pages 136–146. ACM, 2006.

[29] Zion Kwok and Steven JE Wilton. Register file architecture optimization in a coarse-grained reconfigurable architecture. In *Field-Programmable Custom Computing Machines, 2005. FCCM 2005. 13th Annual IEEE Symposium on*, pages 35–44. IEEE, 2005.

[30] Mahdi Hamzeh, Aviral Shrivastava, and Sarma Vrudhula. Epimap: using epimorphism to map applications on cgras. In *Proceedings of the 49th Annual Design Automation Conference*, pages 1284–1291. ACM, 2012.

[31] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3–14. IEEE, 2001.

[32] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.