URECA: A Compiler Solution to Manage Unified Register File for CGRAs

Shail Dave, Mahesh Balasubramanian, Aviral Shrivastava
Compiler Microarchitecture Lab, Arizona State University, Tempe, AZ
Email: {shail.dave, mbalasu2, aviral.shrivastava}@asu.edu

***Abstract*—Coarse-grained reconfigurable array (CGRA) is a promising solution to accelerate loops featuring loop-carried dependencies or low trip-counts. One challenge in compiling for CGRAs is to efficiently manage both *recurring* (repeatedly written and read) and *nonrecurring* (read-only) variables of loops. Although prior works manage recurring variables in rotating register file (RF), they access the nonrecurring variables through the on-chip memory. It increases memory accesses and degrades the performance. Alternatively, both the variables can be managed in separate rotating and nonrotating RFs. But, it increases code size and effective utilization of the registers becomes challenging. Instead, this paper proposes URECA, a compiler solution to manage the variables in a single nonrotating RF. During mapping loop operations on CGRA, the compiler allocates necessary registers and splits RF in rotating and nonrotating parts. It also pre-loads read-only values in the unified RF, which are then directly accessed at run-time. Evaluating compute-intensive benchmarks from MiBench show that URECA provides a geomean speedup of 11.41x over sequential loop execution. It improves the loop acceleration through CGRAs by 1.74x at 32% reduced energy consumption over state-of-the-art.**

I. Introduction

The need for faster and power-efficient processors paved the way for multi-cores along with considerable research in accelerators. ASIC accelerators are efficient but suffer from poor usability. Although popular, acceleration benefits through GPUs are often limited to parallel loops and loops with high trip-counts [1]. Field programmable gate arrays (FPGAs) are reconfigurable and general-purpose but are marred by low power efficiency due to fine-grained management [1].

CGRA is an attractive alternative, as programmable, yet power efficient accelerator that is quite popular in embedded systems for streaming and multimedia applications [2]–[4]. CGRA is simply an array of *processing elements (PEs)* interconnected by a 2-D network. Each PE consists of an ALU-like functional unit and a *register file (RF*). At every cycle, instructions are issued to the PEs. The PE gets the inputs from the neighboring PEs, itself, and registers and executes some operation. Then, it writes the result into RF and to the output register, from which neighboring PEs may read the result in the next cycle. The PE optionally sends/gets the data to/from the data memory. CGRA achieves higher power efficiency due to simpler hardware and intelligent software techniques.

A challenge for CGRA compiler is how to manage loop variables efficiently. Recurring variables are repeatedly read and written throughout the loop execution. Their outcomes across multiple loop iterations need to be managed simultaneously. This is because of i) loop-carried dependencies and ii) in a software pipelined schedule [5], operations from multiple iterations are executed simultaneously. So, prior techniques manage recurring variables in *rotating RF*s [5], preserving the outcome of operations across multiple iterations. Rotation is done in the hardware by either interchanging the data through shift registers or by accessing different physical register at each iteration [2]. However, storing read-only values in rotating RF leads to accessing incorrect values. So, they are usually managed through the memory [6]–[9]. Accessing memory increases the number of loads and degrades the performance. An alternative can be to access read-only values from a separate global RF [3]. But, managing variables in separate RFs requires larger RFs, resulting in poor register utilization. It also increases the instruction width and hence, the code size.

This paper proposes ***URECA*** – a compiler solution to manage all the loop variables in **u**nified **re**gister file for **C**GRA **a**ccelerators. *Unified RF* is a single nonrotating RF, proposed in [10], which is local (i.e. within a PE). URECA enables management of the both recurring and nonrecurring values in the unified RF. Based on register requirements of the operation being mapped on CGRA, the compiler dedicatedly reserves the registers for recurring and/or nonrecurring values. After mapping, it generates a configuration instruction to split the RF into rotating and nonrotating parts. This enables the hardware of the RF to flexibly support a different number of registers for storing both recurring and nonrecurring variables, for mapping of different loops. Moreover, URECA generates machine instructions to pre-load the nonrecurring variables into registers of the RF before loop execution. As unified RF is nonrotating, read-only values are then directly accessed during loop execution. Furthermore, our solution can be easily integrated with any mapping technique for CGRAs.

Evaluating compute-intensive applications from MiBench [11] shows that URECA improves CGRA's loop acceleration capability by 1.74x with 32% reduction in energy consumption as compared to CGRA accessing constant memory. It also reduces the number of registers needed by 39% in comparison with CGRA managing variables through two separate RFs.

II. Managing Loop Variables in CGRA Execution

To accelerate loops on a CGRA, a target application is profiled and compute-intensive loops are extracted. For each loop, a *data dependency graph (DDG)* is generated by parsing the intermediate representation (IR) [12]. As shown in

978-3-9819263-0-9/DATE18/©2018 EDAA

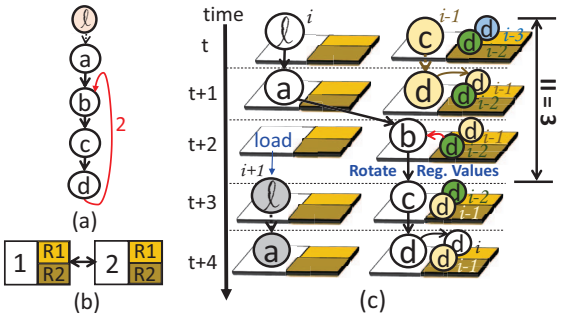

Fig. 1. (a) DDG of a critical loop with loop-carried dependence, (b) a 1×2 CGRA. (c) a register aware mapping of (a) on (b) with II=3

Fig. 1(a), DDG is a directed graph *D=(V,E)*; nodes *V* represent the operations to be executed by PEs and edges *E* represent data dependencies among the operations. An iterative modulo schedule [5] is generated for DDG and operations are mapped on PEs in a software pipelined manner. For example, a valid mapping of DDG of Fig. 1(a) on a 1×2 CGRA of Fig. 1(b) is shown in Fig. 1(c). Node a of the i^{th} iteration is mapped to PE_1 at time t+1. Nodes b, c and d are mapped to PE_2 at consequent timings, honoring the data dependencies. Node l is a live-in value and is always loaded by PE_1. In an iterative modulo schedule, the constant interval between the start of successive iterations is referred as *Initiation Interval (II)* [5], which is the performance metric. In this example, operation 'a' executes after every 3 cycles and hence, II is 3 cycles. Mapping 5 operations on 2 PEs requires at least 3 cycles. Thus, obtained II is *Minimum II (MII)* [5]. There are various techniques to obtain the mapping [2], [3], [9].

Need to Manage Recurring and Nonrecurring Variables: Mapping loop operations to CGRA PEs require management of two kinds of variables – i) recurring and ii) nonrecurring. Recurring values are repeatedly read and written throughout the loop execution. For example, in a software pipelined schedule, outcomes of a node execution across few iterations are stored in the registers and as a result, the liveness of the same variable may overlap [2], [5]. Additionally, in accelerating loops with *loop-carried dependency*, the data values are required across iterations [5]. For example, Fig. 1(a) indicates a *recurrence* through an arc $d \rightarrow b$, with weight of 2. Hence, node b of i^{th} iteration (b^i) needs data from previously executed node d^{i-2}. This implies that every value of d for 2 iterations must be stored in 2 different registers of the RF.

To address the issue of overwriting recurring values, rotating RF is used [2], [5], [13]. For example, Fig. 1(b) shows that each CGRA PE has a rotating register with a depth of 2 (total 2 registers R1 and R2 to hold 2 different values of a variable). In the mapping of Fig. 1(c), operation d executes on PE_2 and always writes to R1 and b reads from R2. At time t+1, d^{i-1} writes its value into R1 of PE_2. R2 of PE_2 contains the value of previously computed d^{i-2}, which is read later at time t+2 to compute b^i. For correct execution, rotation of the register values occurs at every II cycles (shown by exchanging the values of R1 and R2 at the beginning of t+3). After rotation, R1 of PE_2 contains unwanted value d^{i-2} which is overwritten by new value d^i at time t+4. Thus, rotation helps preserving d^{i-1} into R2 which is needed by b^{i+1} at t+5 (not shown).

CGRA PEs also need to access nonrecurring variables like read-only operands, live-in data (values needed for loop execution) etc. They are frequently accessed throughout the loop execution and should be stored in registers. But, if managed in rotating RF, they undergo the rotation. It causes either the register value to be overwritten or PEs access incorrect values. This results in incorrect execution. For example, operation a needs to access live-in value l which cannot be stored in the rotating registers. Hence, PE_1 always loads l from the memory throughout the loop execution. Alternatively, such nonrecurring values can be managed in a separate nonrotating RF. But, managing both recurring and nonrecurring values separately has been inevitable.

III. Limitations of Prior Approaches

Majority of prior compiler solutions manage recurring variables in local registers of PEs and nonrecurring variables through memory. Alternatively, variables can be managed in separate RFs, recurring in local rotating RFs, nonrecurring in a global nonrotating RF. Fig. 2(a) shows CGRA managing nonrecurring values in constant memory (L1 cache or a memory bank in scratch-pad memory) [6]–[9]. In some CGRA designs, only specific PEs can access constant memory; a constant is placed and routed through a PE [14]. Accessing memory is simple, but it results in extra load operations during each loop iteration, which can degrade the performance. In fact, adding more loads can be much more harmful because of 2 reasons: i) in most CGRAs, only a few PEs can perform memory operations [2], [3], ii) Often load/store bandwidth is limited, e.g., data and address buses are usually shared by PEs in row [8], [9]. Such restrictions along with more operations to be mapped and executed on CGRA PEs result in higher II i.e. more execution cycles. It also increases the code size as we need to manage more CGRA instructions with the increased II. Besides, managing nonrecurring variables in memory require larger data memory throughout the execution.

CGRA mapping technique proposed by Oh et. al. [3] considered the issue of increased memory accesses. They suggested reserving the nonrecurring values into a separate global RF. As shown in Fig. 2(b), global RF is accessed by all PEs, allowing data sharing between PEs without external

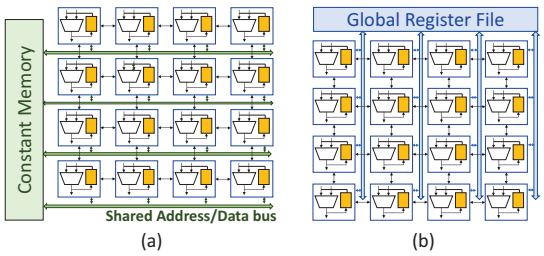

Fig. 2. CGRAs manage recurring variables in local rotating RFs. For nonrecurring values, CGRA accesses (a) on-chip memory (b) a global RF; each PE is connected to global RF through column-wise bus structure

routing. Experiments have shown that it is crucial to connect all PEs to global RF [15], which requires many R/W ports, resulting in performance degradation and increased total area [15]. Furthermore, accessing separate RFs burdens instruction set architecture (ISA) and increases instruction width. For example, a 32-bit instruction for a PE requires – 5 bits for opcode field, 2×3 bits for selecting input through 2 multiplexers for two operands and 3×2 bits for indexing register number to access a local RF with 2 read and 1 write port; RF contains 4 registers. Moreover, 1 bit is needed for each of – indicating RF write, asserting address bus and asserting data bus. Finally, the immediate field consists of 12 bits. Now, consider CGRA of Fig. 2(b) that manages recurring values in local RF of 4 registers and read-only operands in a global RF of 64 registers. Then, we need 18 bits just to index registers of the RF; PE can get 2 inputs from RF and writes back to the RF (i.e. 3×6). Managing 2 separate RFs requires selection between RFs for each register index field (i.e. 3 more bits). Hence, such approach increases instruction width to 47-bits from 32-bits when compared to CGRA with a local RF of 4 registers. It increases memory bus width and the code size.

True that we can have separate rotating and nonrotating RFs to manage recurring and nonrecurring variables [3], [13]. But, efficient utilization of registers becomes a challenge. This is because different loops in target application(s) require the different number of rotating and nonrotating registers. For example, some loops may feature loop-carried dependencies with larger distance whereas, operations of some other loops may need many live-in values. So, if any of the rotating and nonrotating RF is of smaller size, then a mapping may not be achieved. CGRA with larger rotating and nonrotating RFs not only end up with poor utilization of registers but also consume more area and power. Plus, managing separate larger RFs increases the instruction width and hence, the code size.

Instead, this paper proposes to manage all the variables in the unified RF. *Unified RF* [10] is a regular (i.e. nonrotating) RF, which is configurable. Our solution targets local unified RF, as local RFs are smaller, scalable and help to obtain the better performance [3]. The compiler allocates necessary registers and configures the RF dynamically, splitting it into rotating and nonrotating parts.

IV. URECA: Efficiently Manage All Variables in Single Reconfigurable Register File

To manage both recurring and nonrecurring variables in an efficient manner, this paper presents **URECA**, a compiler solution to manage all the variables in the unified RF, with the least number of registers. During mapping of each loop operation on a CGRA PE, compiler analyzes the number of registers needed to store recurring and/or nonrecurring values in RF within a PE. An operation is mapped only if registers are available. After register allocation, RF configuration is generated to split the RF in rotating and nonrotating section. While the hardware of the unified RF implements rotation through modulo addition and allows the access to correct recurring values, our compiler pre-loads the nonrecurring variables into registers of the nonrotating section. These nonrecurring variables are then directly accessed from the RF throughout the loop execution. During allocating registers for nonrecurring values, compiler employs data reuse analysis to avoid duplication; same value can be used by numerous operations mapped on a PE.

A. Accessing Right Registers in Unified RF

The unified RF [10] is shown in Fig. 3, which can be split into the rotating and nonrotating parts with the *configuration value* c. URECA generates the configuration value and a machine instruction to dynamically set the value c prior the loop execution. It provides CGRA compiler the flexibility to support different register requirements for different loops.

Accessing Nonrotating Section: The register index for the read and write operations is indicated by $readReg1$, $readReg2$ and $write$, respectively. If the register index is greater than c then, the control unit generates a *select* signal as 1. Then, a PE directly accesses the register inside the nonrotating section. For example, if RF has a total $n = 6$ registers, the value of c = 3 implies that there are 4 registers in rotating section and 2 in a nonrotating section. In this case, a read operation with index $readReg1$ = 5 enables accesses to the register 5 inside nonrotating part. Register 5 contains a nonrecurring value, that directly drives the read port. Note that for the unified RF of each PE, such nonrecurring variables are pre-loaded into corresponding registers through machine instructions, at the beginning of the loop execution.

Accessing Rotating Section: If the register index is less than or equals to c, we need to access the rotating section. The RF is nonrotating and hence, it eliminates the use of complex structures such as shift registers. Instead, the rotation is implemented by a modulo addition of the register index with a stage counter [2]. The *stage counter (SC)* is incremented at the end of every II cycles and is reset to 0 when it reaches the value of c. The outcome of the adder is ANDed with c to get the modulo addition. This mechanism helps to access different physical registers at each loop iteration. For example, we want to access values of a variable d across 4 different iterations. We read the value of d^{i-3} through index $readReg2$ = 0 and we overwrite the new value of d^i with the index $write$ = 3. In such scenario, a $select$ signal is always 0. For SC = 2 (*iteration i = 10*), the summation of SC with $readReg2$ is

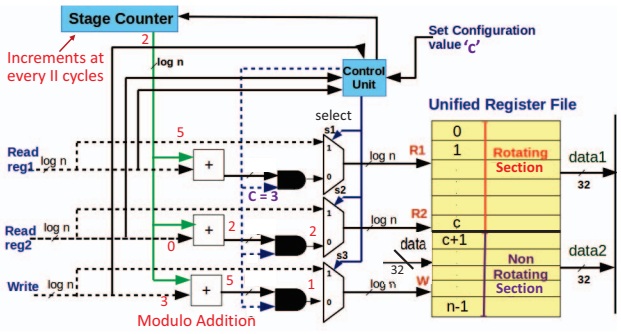

Fig. 3. Unified RF [10] can be split into rotating and nonrotating sections. The compiler configures the RF through a machine instruction.

2. Then, modulo operation through ANDing yields access to the physical register of index 2 that contains d^7. Similarly, physical register 1 is selected to overwrite with the newest value d^{10}. And, physical registers with index 3 and 0 still preserve the older values d^8 and d^9, respectively. Thus, we can manage both recurring and nonrecurring variables in a single RF; RF configuration splits RF into two parts.

B. How Compiler Determines Register Requirements?

In CGRA compiler, register allocation is integrated with a place and route stage of the mapping and the operation is placed on the PE only if the required number of registers are available. So, during mapping an operation on a PE, our compiler analysis determines the number of registers needed for both, i) storing nonrecurring variables ii) managing recurring values. It reveals the total number of registers required inside rotating and nonrotating section of unified RF. The number of nonrotating registers needed is easily determined from the live-in operands in DDG (with a liveness analysis through use-definition chains [12]). Plus, if the value of a constant operand is larger than the maximum value supported by immediate bits in the CGRA instruction then, it is also a nonrecurring variable. During register allocation, data reuse analysis is employed to avoid duplicating the nonrecurring value in the RF. For example, multiple operations often require the same live-in value. So, when they are mapped on the same PE, reuse analysis avoids storing redundant values in the RF.

Algorithm 1 shows how compiler analyzes registers for recurring values. First, it finds out the information about successor nodes that access the outcome through registers, due to either intra-iteration dependency or a loop-carried dependence. Based on the scheduling and mapping information, the compiler calculates a difference of absolute mapping times of a node and its successor (in terms of II cycles). Finally, with calculated mapping distance, it computes the number of rotating registers required to map a node v_i on PE p_i.

C. URECA Ensures Efficient Management in Single RF

Upon determining the total registers needed to manage both the variables, the compiler ensures availability of registers prior to their reservation. For mapped operations, it keeps track of register allocation per PE, as shown in Algorithm 2. For a

Algorithm 1: $getRotatingReg$(Input Node v_i, PE p_i)

```
1  (successors, total_successors) ←get_successors(v_i);
2  while i < total_successors do
3  |   s_i ← successors[i];
4  |   if (isMoreThanACycleApart(v_i, s_i)) then
5  |   |   distance ← calculate_distance(v_i, s_i);
6  |   |   reg_needed ← distance + 1;
7  |   |   if (reg_needed > rotating_reg) then
8  |   |   |   rotating_reg ← reg_needed;
9  |   i++;
10 return rotating_reg;
```

Algorithm 2: $allocateRegs$(Input PE p_i, Size N,Node v_i)

```
1  r_1 ← getRotatingReg(v_i, p_i);
2  r_2 ← get_number_of_nonrotating_registers(v_i, p_i);
3  r'_1 ← get_nearest_power_of_two(rotating[p_i] + r_1);
4  total ← r'_1 + (nonrotating[p_i] + r_2);
5  if total ⩽ N then
6  |   rotating[p_i] += r_1; nonrotating[p_i] += r_2;
7  |   configuration[p_i] ← r'_1; return true;
8  return false;
```

PE p_i, rotating[p_i] indicates the number of registers allocated previously in the rotating section. The compiler ensures that new size of rotating section r'_1 is equal to the nearest power of 2, satisfying constraint due to implementing modulo addition. If enough registers are available to map a new operation v_i, then the allocation is done and the function returns success (lines 5–7). Once all operations are mapped, instructions to configure RFs of PEs is generated based on the value of configuration[p_i], and fed to control unit of Fig. 3 at run-time.

Unlike prior compilation approaches for existing CGRA RF designs, URECA manages the variable number of the recurring and/or nonrecurring values in the unified RF, supporting different mappings of different loops. Thus, URECA enhances the capability of the compiler to efficiently and flexibly manage the variables in a single RF of limited size, promoting general-purpose computing on CGRAs.

D. Integration with CGRA Mapping Techniques

Fig. 4 shows a high-level overview of the compilation flow. Input DDG is translated to a set of clusters/cliques. Then, the compiler tries to map the operation on a PE. If it finds a PE slot, it checks for register availability inside the unified RF else, it finds another PE. If no other PE is available, it increases II by 1. Register reservation is done through Algorithm 2. If II value crosses the preset limit, it terminates the mapping, resulting in failure. Upon successfully mapping all the operations, a valid mapping is generated along with machine instructions to configure the unified RFs and to preload nonrecurring values. In this way, our solution can be easily integrated with any CGRA mapping technique.

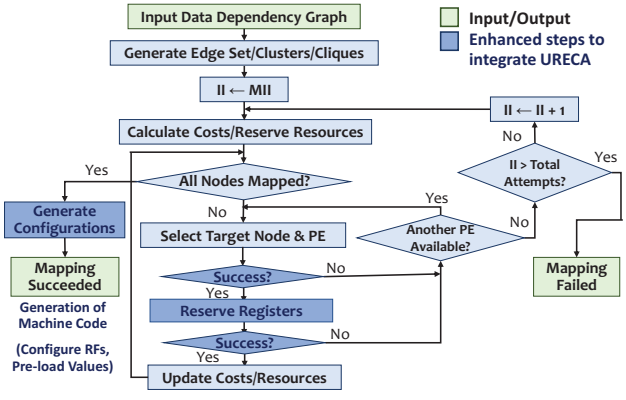

Fig. 4. Integrating Register Reservation Function with a Mapping Technique

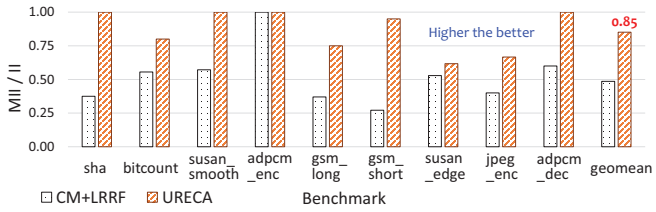

Fig. 5. URECA manages all variables within local unified RF, achieving II close to Minimum II (MII) as compared to CGRA accessing constant memory.

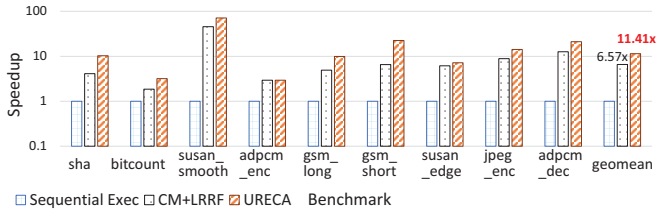

Fig. 6. URECA helps CGRA to accelerate loop execution cycles by 11.41x over sequential execution on ARM Cortex-like core with ARMv7a profile.

V. Experimental Setup

Benchmarks: We profile MiBench benchmark suite [11] with Alinea Map and Intel Parallel Studio XE tools and determine the top non-vectorizable performance-critical loops in compute-intensive applications. These benchmarks represent the workloads in the fields of security, telecom, automotive etc. and can benefit from acceleration through CGRAs.

Compilation: The mapping is obtained through REGIMap [9], which maps operations with a clique based approach; the corresponding rotating RFs of the PEs should have enough rotating registers. Instead, we modify the register allocation constraint (with Algo. 2) to target unified RF, accommodating both recurring and nonrecurring values for the operation. Our CGRA compiler is implemented in LLVM 4.0 [12] as a pass. We use optimization level 3 and also consider loops accessing sub-words/pointers or loops with dynamic trip-counts.

Simulation: Techniques are evaluated on popular cycle-accurate simulator gem5 [16] in system emulation mode; we modeled CGRA as a separate core coupled to ARM Cortex-like processor core with ARMv7a profile. In a 4x4 homogeneous CGRA, PEs are connected in a 2D torus, performing fixed-point operations with 1-cycle latency. PEs access data and instruction memories of 4 kB; memory bus is shared among PEs in a row. For load/store operation, 2 instructions are executed; 1^{st} generates address and 2^{nd} loads/stores data.

Techniques Evaluated: In evaluating prior works, CGRA manages recurring variables in *local rotating RF (LRRF)* of 4 32-bit registers [2]. With 12-bit immediate in our ISA, constants greater than 4095 are treated as nonrecurring variables. In state-of-the-art approach (i.e. *CM+LRRF*), CGRA manages nonrecurring values by accessing 4 kB of data memory *(CM)*. We also evaluate an alternative of managing them in separate *global nonrotating RF (GNRRF)* of 64 registers (i.e. *GNRRF+LRRF*). However, URECA is evaluated with a unified RF [10] for each PE, with 4 registers. RF configuration takes 1 cycle and a variable is preloaded in 3 cycles. We implement RTL for CGRA, mapping it to Synopsys 32nm process and synthesize it with Cadence RTL compiler (Table I).

VI. Results and Analysis

A. URECA Improves CGRA's Loop Acceleration Capability by 1.74× over CGRA Accessing Constant Memory

Fig. 5 shows that employing URECA achieves the mapping of nearly ideal quality. For each performance-critical loop, we measure mapping quality as a ratio of MII to II, since values of II span over a larger range. In CM+LRRF, the nonrecurring values were accessed from memory, increasing nodes by 50%. It increased II by 1.75x due to bandwidth restrictions as only 1 PE among 4 PEs in a row can access memory bus at a time. For example, the critical loop of sha translated to a DDG with 30 nodes (including 10 load/store nodes) and 9 nonrecurring values. This resulted in additional 18 load nodes and in II of 8 for CM+LRRF, while URECA achieved MII of 3 honoring the resource constraints. However, the critical loop of $adpcm\ encoder$ featured a loop-carried dependence with distance 1 and about a delay of 20 operations in the path. So, all approaches easily obtained mapping at higher II (recurrence-bounded). Fig. 6 shows that better mapping provided acceleration of 1.74x in terms of loop execution cycles (including RF configuration/pre-loading cycles). URECA promoted variables to registers from memory, helping CGRA to accelerate loops by 11.41x over sequential execution.

B. URECA Reduces Energy Consumption by 32% in Comparison with CGRA Accessing Constant Memory

With critical path delay (D), power (P) (Table I), we compute energy E as P×C×D [17], for loop execution cycles C. With a significant reduction in execution time, URECA reduces energy consumption by 32% as compared to CM+LRRF. Compiler solutions managing separate RFs e.g. GNRRF+LRRF consume 5% higher energy than URECA and increases code size by 50%, requiring 8 kB of instruction memory. However, the code is partitioned for evaluation.

Here, an exception is the benchmarks where the performance with unified or global RF [1] is similar to the CGRA accessing memory, e.g. $adpcm\ encoder$. Compared to LRRF, unified RF has little higher cycle time and it takes some more cycles for pre-loading constants. So, URECA consumes little higher energy than CM+LRRF. In such scenario, employing global RF is even worst (1.2x) as it consumes higher power and yields larger cycle time because of all PEs accessing a separate larger global RF via many R/W ports.

[1]Although GNRRF+LRRF yields performance at par with URECA, it increases the code size and results in poor utilization of the registers.

TABLE I
Hardware specifications of CGRA with different RFs and 4 kB of data and instruction memories for 32nm process

RF Architecture	Delay (ns)	Area (mm^2)	Power(mW)
LRRF	1.94	1.062	365.26
Unified RF	2.10	1.097	366.69
GNRRF & LRRF	2.15	1.287	382.86

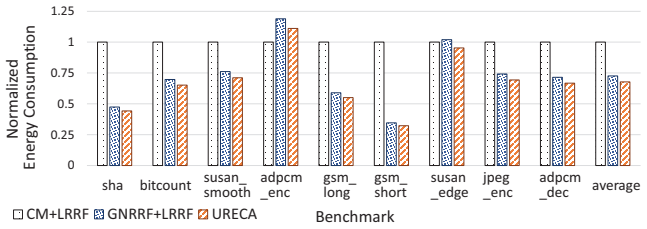

Fig. 7. Normalized energy consumption for URECA and for CGRA managing nonrecurring values in global RF when compared to the approach of accessing nonrecurring variables from memory.

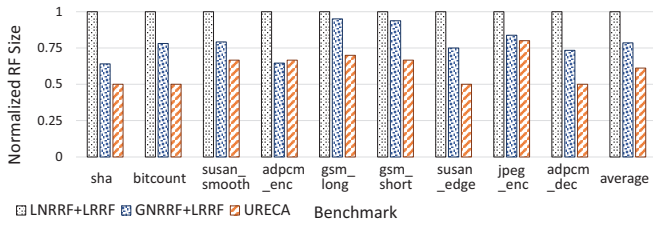

Fig. 8. URECA reduces register requirement significantly as compared to CGRA managing the variables in separate RFs.

C. URECA Reduces Register Requirement by 39% as Compared to CGRA Managing Two Separate Register Files

To demonstrate the impact of variable management by URECA, we analyze RF size requirements for various approaches. A simple way can be to keep nonrecurring and recurring values in 2 separate local RFs, *local nonrotating RF (LNRRF)* and LRRF (i.e. *LNRRF+LRRF*). Other possible solutions are GNRRF+LRRF and URECA. For each approach, we obtain mapping with no constraint on RF size and determine the total number of registers needed. Our analysis reveals that managing 2 separate local RFs is worst as there is no data sharing. Although a GNRRF can help to share the data among PEs, still many rotating registers in separate LRRFs are left unutilized. In contrast, URECA provides CGRA compiler the flexibility to allocate registers for both recurring and/or nonrecurring values in the single RF and reduces total registers required. For example, $Susan\ smoothing$ requires 4 rotating registers. Hence, we need LRRF of at least 4 registers for all 16 PEs. It also needs to manage 12 live-in/live-out values; some PEs require 2 live-in values. Hence, LNRRF of 2 registers is needed (total 32 + 64 registers for LNRRF + LRRF). Alternatively, we need total 12 + 64 registers for accessing GNRRF + LRRF. On the other hand, having URECA with just 4 registers in the unified RF (total 64) is enough. URECA easily manages both types of variables in the unified RF, reducing the register requirements. This greatly helps CGRA compiler to generate the needed mapping. Furthermore, data reuse analysis in managing nonrecurring values in RF also reduces register requirements (especially for $adpcm$ and gsm).

VII. Summary

This paper presents challenges in the traditional approach of the managing nonrecurring values through memory and shows how it degrades the performance. The alternative of managing variables through separate RFs results in poor register utilization and increases the code size. This paper advocates for managing them in a unified RF and presents URECA as an efficient solution. URECA generates the configuration for the unified RF and allows it storing a different number of recurring and nonrecurring values; register reservation is done by the compiler unique to the loop requirement. After evaluating the technique along with prior works, we conclude that URECA improves acceleration capability of CGRAs by 1.74x at 32% reduced energy usage. Our solution manages variables efficiently than other existing solutions.

Acknowledgment

This work was partially supported by funding from the NSF grants CCF 1723476, 1055094 (CAREER), and CNS 1525855.

References

[1] S. Che, J. Li, J. W. Sheaffer, K. Skadron, and J. Lach, "Accelerating compute-intensive applications with gpus and fpgas," in *Application Specific Processors, 2008. SASP 2008. Symposium on*. IEEE, 2008.

[2] B. Mei, M. Berekovic, and J. Mignolet, "Adres & dresc: Architecture and compiler for coarse-grain reconfigurable processors," *Fine-and coarse-grain reconfigurable computing*, pp. 255–297, 2007.

[3] T. Oh, B. Egger, H. Park, and S. Mahlke, "Recurrence cycle aware modulo scheduling for coarse-grained reconfigurable architectures," in *ACM Sigplan Notices*, vol. 44, no. 7. ACM, 2009, pp. 21–30.

[4] B. Egger *et al.*, "A space-and energy-efficient code compression/decompression technique for coarse-grained reconfigurable architectures," in *Proceedings of the 2017 International Symposium on Code Generation and Optimization*. IEEE Press, 2017, pp. 197–209.

[5] B. R. Rau, "Iterative modulo scheduling: An algorithm for software pipelining loops," in *Proceedings of the 27th annual international symposium on Microarchitecture*. ACM, 1994, pp. 63–74.

[6] H.-S. Kim, M. Ahn, J. A. Stratton, and W.-m. W. Hwu, "Design evaluation of opencl compiler framework for coarse-grained reconfigurable arrays," in *Field-Programmable Technology (FPT), 2012 International Conference on*. IEEE, 2012, pp. 313–320.

[7] H. Lee, D. Nguyen, and J. Lee, "Optimizing stream program performance on cgra-based systems," in *Proceedings of the 52nd Annual Design Automation Conference*. ACM, 2015, p. 110.

[8] G. Dimitroulakos, N. Kostaras, M. D. Galanis, and C. E. Goutis, "Compiler assisted architectural exploration framework for coarse grained reconfigurable arrays," *The Journal of Supercomputing*, vol. 48, 2009.

[9] M. Hamzeh, A. Shrivastava, and S. Vrudhula, "Regimap: register-aware application mapping on coarse-grained reconfigurable architectures (cgras)," in *Proceedings of the 50th Annual Design Automation Conference*. ACM, 2013, p. 18.

[10] M. Hamzeh, *Compiler and Architecture Design for Coarse-Grained Programmable Accelerators*. Arizona State University, 2015.

[11] M. Guthaus *et al.*, "Mibench: A free, commercially representative embedded benchmark suite," in *WWC*, 2001.

[12] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *CGO*, 2004, pp. 75–86.

[13] B. Van Essen, R. Panda, A. Wood, C. Ebeling, and S. Hauck, "Managing short-lived and long-lived values in coarse-grained reconfigurable arrays," in *Field Programmable Logic and Applications (FPL), 2010 International Conference on*. IEEE, 2010, pp. 380–387.

[14] P. Theocharis and B. D. Sutter, "A bimodal scheduler for coarse-grained reconfigurable arrays," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 13, no. 2, p. 15, 2016.

[15] Z. Kwok and S. J. 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*. IEEE, 2005, pp. 35–44.

[16] N. Binkert *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[17] M. Karunaratne, A. K. Mohite, T. Mitra, and L.-S. Peh, "Hycube: A cgra with reconfigurable single-cycle multi-hop interconnect," in *Proceedings of the 54th Annual Design Automation Conference 2017*. ACM, 2017.