To appear in the 55th Design Automation Conference (DAC), 2018

RAMP: Resource-Aware Mapping for CGRAs

Shail Dave
Arizona State University
shail.dave@asu.edu

Mahesh Balasubramanian
Arizona State University
mbalasu2@asu.edu

Aviral Shrivastava
Arizona State University
aviral.shrivastava@asu.edu

ABSTRACT

Coarse-grained reconfigurable array (CGRA) is a promising solution that can accelerate even non-parallel loops. Acceleration achieved through CGRAs critically depends on the goodness of mapping (of loop operations onto the PEs of CGRA), and in particular, the compiler's ability to route the dependencies among operations. Previous works have explored several mechanisms to route data dependencies, including, routing through other PEs, registers, memory, and even re-computation. All these routing options change the graph to be mapped onto PEs (often by adding new operations), and without re-scheduling, it may be impossible to map the new graph. However, existing techniques explore these routing options inside the Place and Route (P&R) phase of the compilation process, which is performed after the scheduling step. As a result, they either may not achieve the mapping or obtain poor results. Our method RAMP, explicitly and intelligently explores the various routing options, before the scheduling step, and makes improve the mapping-ability and mapping quality. Evaluating top performance-critical loops of MiBench benchmarks over 12 architectural configurations, we find that RAMP is able to accelerate loops by 23× over sequential execution, achieving a geomean speedup of 2.13× over state-of-the-art.

CCS CONCEPTS

• **Hardware → Hardware accelerators**; • **Software and its engineering → Compilers**;

ACM Reference Format:
Shail Dave, Mahesh Balasubramanian, and Aviral Shrivastava. 2018. RAMP: Resource-Aware Mapping for CGRAs. In *DAC '18: The 55th Annual Design Automation Conference 2018, June 24–29, 2018, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3195970.3196101

1 INTRODUCTION

The need for ever increasing power-efficiency targets paved the way for multicores as well as hardware 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 their fine-grained management [1].

CGRA is an attractive alternative as a programmable, yet power efficient accelerator, and is quite popular in embedded systems for streaming and multimedia applications [2–7]. 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 neighbors, 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 power-efficient acceleration due to simple hardware and intelligent software techniques.

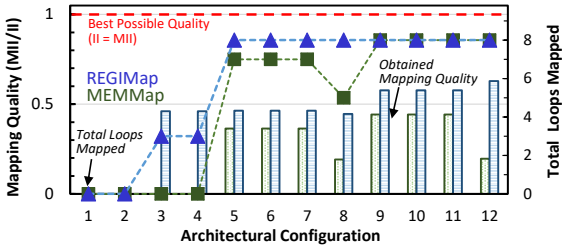

Figure 1: Existing techniques perform poorly and may not even map the loops on CGRAs with modest resources.

A good CGRA compiler should be able to efficiently map all the loop operations onto a CGRA with its limited resources. Most existing CGRA compilation techniques first generate the data dependency graph (DDG) of the loop, and then break down the mapping problem into 2 main parts: i) scheduling, and ii) *place and route or (P&R).* In the scheduling step, the DDG is software pipelined, and a schedule-time is assigned to each operation. This step usually takes into account the resource constraints, e.g., not schedule more operations in a cycle than the number of PEs. In the P&R step, operations are placed onto PEs, and the dependencies are routed. The goodness of the obtained mapping is critically dependent on how efficiently the compiler can route the data dependencies. The best way to route data dependencies is to map the operations to PEs that are directly connected to each other. However, that may not be possible due to resource constraints, e.g., operations are already mapped onto all the PEs that are directly connected. Previous works have explored various methods to route data dependencies between PEs, including using other PEs, local or global registers, memory, and even recomputation [2–4, 8–13].

A major challenge faced by existing schemes is that routing by these methods changes the DDG (typically in the form of new operations, and new data dependencies or new scheduling constraints). For example, routing through memory requires an addition of a new store and a load operation (that must now be mapped to PEs), and a scheduling constraint (that store must happen before the load). However, adding new nodes and constraints can make it impossible to map without re-scheduling. Previous schemes for routing through PEs like [3, 4, 8], explore the routing options *implicitly* inside P&R phase of the compiler, and *do not perform re-scheduling*. Previous approaches for routing through registers – in order to avoid rescheduling – *restrict* themselves to using only the local registers in the PE [10], and prior approach for routing through memory [13] – in order to avoid re-scheduling – decides even before scheduling that it will spill only variables that have a lifetime that is greater than certain threshold, even if those dependencies are actually trivial to map. As a result, previous approaches achieve poor mapping, and sometimes, they are simply unable to generate any mapping!

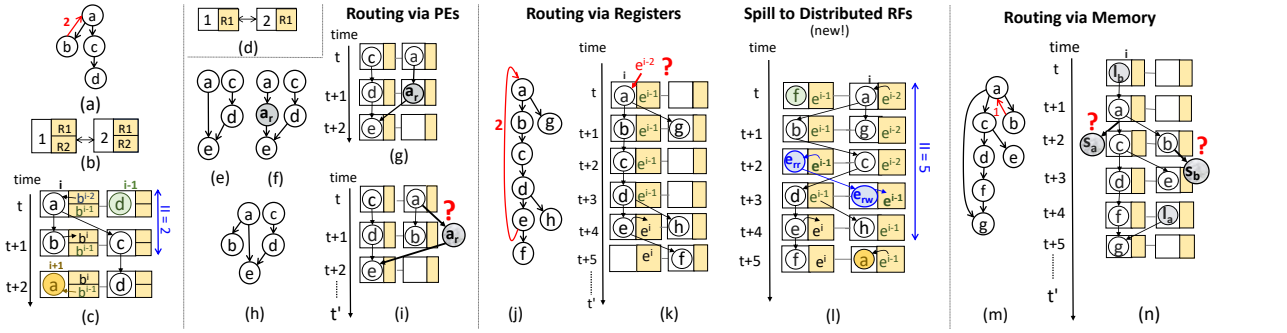

Figure 2: (a) DDG of a loop with loop-carried dependence (b) a 1×2 CGRA (c) a register aware mapping of a) on b) with II=2 (d) Target CGRA. A P&R attempt is shown for ad-hoc routing via (e)–(i) PEs (j)–(k) Registers (m)–(n) Memory.

Fig. 1 shows the mapping ability and mapping quality obtained by the state-of-the-art register-aware (REGIMap[10]) and memory-aware (MEMMap[13]) techniques across different CGRA configurations (Table 1) that vary in the number of PEs, registers, and memory-bus configurations. It shows that for the top performance-critical loops from 8 MiBench [14] benchmarks, previous techniques failed to obtain mappings for almost all loops (triangles and squares in Fig. 1) when highly constrained by the resources. Also, the mapping quality obtained (bars in Fig. 1) is far from the best possible mapping, even when the target CGRA has higher resources.

To ramp-up acceleration achieved through CGRAs, this paper introduces RAMP – a **R**esource-**A**ware **Map**ping. The major idea behind RAMP is to *explicitly* model various routing choices. So, RAMP explores various routing choices *systematically* with *re-scheduling*, without any *restrictions* (allows for unrestricted use of all the registers in the CGRA), and *adaptively* (can decide which dependencies must be routed through memory depending on the mapping). RAMP partitions the mapping problem into 3 steps: i) Systematic exploration of routing strategies ii) (Re)scheduling iii) P&R. When RAMP fails to P&R a dependency due to resource constraints, it analyzes reasons for failure and intelligently applies routing alternatives. If multiple strategies can successfully route the dependency, RAMP selects the option which utilizes the least resources. Thus, RAMP maps dependencies efficiently while exploiting resources.

Evaluating top performance-critical loops of MiBench over various CGRA configurations, we find that RAMP can map majority of the loops even onto CGRAs with modest resources. RAMP achieves the mappings of nearly best possible quality (MII/II = 0.97). RAMP outperforms state-of-the-art with a geomean speedup of 2.13×.

2 BACKGROUND

To accelerate loops on CGRA, a target application is profiled and compute-intensive loops are extracted. For each loop, a DDG is generated [15] after parsing the intermediate representation [16]. 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 [17] is generated for DDG and operations are mapped on PEs in a software pipelined manner. For example, a mapping of DDG of Fig. 2(a) on a 1×2 CGRA of Fig. 2(b) is shown in Fig. 2(c). Nodes a and b of i^{th} iteration are mapped to PE_1 at time t and $t+1$. c and d are mapped to PE_2 at $t+1$ and $t+2$, honoring the data dependencies. A *loop-carried dependence* is indicated through an arc $b \rightarrow a$, with weight of 2. Hence, node a of i^{th} iteration (a^i) needs data from

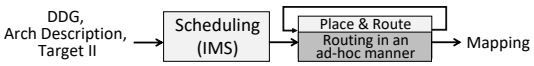

Figure 3: Previous approaches implement the routing strategies inside the P&R step, and do not re-schedule the graph.

previously executed node b^{i-2}, which it obtains from the registers. In an iterative modulo schedule, the constant interval between the start of successive iterations is referred as *Initiation Interval (II)* [17], which is the performance metric. In this example, the operation 'a' can execute after every 2 cycles and hence, II obtained is 2 cycles. Moreover, it takes at least 2 cycles to map a total of 4 operations onto 2 PEs. Therefore, obtained II is *Minimum II (MII)* [17].

3 LIMITATIONS OF EXISTING HEURISTICS

One of the challenging and unique aspects of a CGRA compiler is the problem of routing the data dependencies among the PEs. If all dependent operations can be placed on PEs that are directly connected to each other, then the application mapping problem would be trivial. However, due to resource constraints, this may not be possible. Previous works have explored many different ways of routing the data dependencies, including other PEs [2, 3, 8], register files [2, 9, 10, 12], memory [13], and even recomputation [8]!

Works like [2], formulate one problem for the whole mapping (scheduling, placement, and routing), and attempt to solve it in one shot. However, this makes the problem very complex, and they solve it by simulated annealing based approaches, that not only take a long time, but also generate the poor mapping, and offer no insights. EMS [3] and EPIMap [8] developed schemes to route dependencies via PEs. To do so, they modify DDG by inserting the routing operations, which can be mapped on spare PEs. Fig. 2(e) shows a dependency graph that cannot be mapped on a 1×2 CGRA because it is not possible to route the data dependency $a \rightarrow e$, as PEs on which they are mapped are not directly connected. This problem can be solved by adding an additional node a_r, which transforms the DDG into the one shown in Fig. 2(f), and now it can be mapped as shown in Fig. 2(g). Now, if we try routing for the edge $a \rightarrow e$ in Fig. 2(h) by adding an additional routing node a_r, it cannot be routed (without rescheduling) because of resource constraints (no more PEs are available at time $t+1$). The problem is that previous approaches implement this routing *implicitly* inside the P&R stage of the compilation (see figure 3), and *do not perform re-scheduling* and therefore are unable to generate the good mapping.

GraphMinor [9] and REGIMap [10] allow routing dependencies via registers. They allow using local rotating registers of the PEs

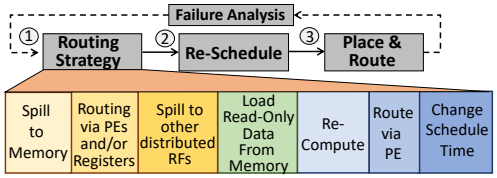

Figure 4: A High-Level Overview of RAMP.

to map the recurring values. As a result, both the predecessor and the successor of the dependency must be placed on same PE, so that they can access the same RF. If there are not enough rotating registers in one PE, then the dependency cannot be routed even if, there are more rotating registers in RFs of rest of the PEs. Fig. 2(k) shows a P&R attempt to map DDG of Fig. 2(j) onto 1×2 CGRA. Routing loop-carried dependence $e \rightarrow a$ via RF requires 2 registers. Even though CGRA has total 2 registers, a single RF does not have sufficient registers. So, the dependence cannot be routed via registers. However, it is possible to map this DDG using the register of other PE, as shown in Fig. 2(l). In this valid mapping, e^i writes output to the register of PE1 (in cycle $t+4$), and then it is transferred to PE2 (in cycle $t+2$ and $t+3$, for previous iteration e^{i-1}), and eventually (in cycle $t+5$), it is used by operation a. As seen in this example, a generalized approach to routing via distributed registers requires changing the DDG (adding new nodes e_{rr}, and e_{rw}), and *re-scheduling*. This is hard to do in previous approaches since they implement routing *implicitly* in P&R phase of the compiler, and *avoid the need to re-schedule* the DDG by restricting their solution.

MEMMap [13] allows routing dependencies via memory. However, at the scheduling stage, they pre-decide on the dependencies that will be routed through memory. They route the dependencies with length (difference in schedule times) greater than or equal to 3 via memory. For DDG of Fig. 2(m), since dependencies $a \rightarrow g$ and $b \rightarrow a$ have the length more than 3, they are slated to be routed through memory. MEMMap replaces them with additional store and load operations (S_a, S_b and L_a, L_b in Fig. 2(n)), and then runs the P&R on it. However, adding these nodes without re-scheduling can make the graph un-mappable (Fig. 2(n)). Furthermore, the decision of routing the dependencies with the longest length through memory may not be right. For example, if there are enough registers or PEs to route, then it may be better to route the dependency through registers or PEs (requires adding fewer nodes to the DDG). In particular, in this example, the dependencies $a \rightarrow g$ and $b \rightarrow a$ can be easily routed via registers. In this sense, MEMMap has a rigid strategy to route via memory and does *not adapt* to the needs of the application. This is because they implement their routing scheme *implictly* in the P&R stage, and *avoid re-scheduling*.

4 RAMP: RESOURCE-AWARE MAPPING

The key to systematically and intelligently exploring various routing choices is to explore them in an explicit manner. As opposed to previous approaches (see Fig. 3), RAMP partitions the mapping problem in 3 sub-problems (see Fig. 4): **i) Systematic Exploration of Routing Strategies ii) Re-Scheduling iii) P&R**. Fig. 4 reveals the high-level overview of RAMP. To explore all the resources flexibly, RAMP models the various routing strategies such as routing via PEs/registers, spilling data to memory, spilling to distributed RFs, re-computation etc., which are selectively applied during mapping attempt at an II. For each strategy, RAMP modifies the DDG

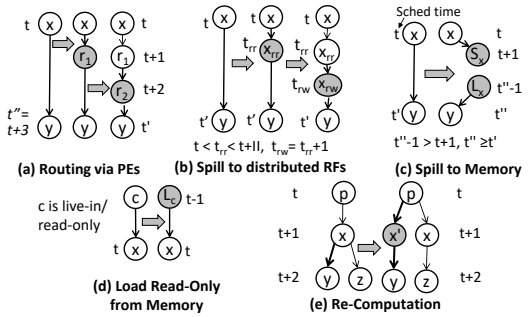

Figure 5: Graph Modification and Re-Scheduling.

accordingly to route the target dependency. The modified DDG is modulo scheduled (with additional scheduling constraints for some of the routing strategies), which takes care of the potential resource constraints. After scheduling the modified graph, RAMP tries to find the mapping for adopted routing strategy. After trying several routing options, if multiple strategies successfully map the operation/dependency at an II, RAMP selects the one which requires the least resources. Such approach enables RAMP to route a dependency in the best possible manner while exploiting resources. Then, with the corresponding modified (and rescheduled) graph, RAMP moves forward to P&R remaining unmapped operations and dependencies. Thus, RAMP can adapt to the loop characteristics, accommodating the routing requirements of the DDG of any loop.

As RAMP models routing phase explicitly and re-schedules the modified graph, it can flexibly explore new routing solutions. For example, RAMP employs the novel approach of spilling to distributed RFs and enables the spilling to memory. Moreover, RAMP benefits from different CGRA resources (e.g. different memory bus bandwidth [10, 13], diverse RFs i.e. various rotating and nonrotating RFs [2, 12, 18] and RFs of local/shared/centralized structure [9], heterogeneous PEs [3, 19, 20] etc.) and succeeds in achieving mappings of better quality even when modest resources are available.

4.1 Routing Options

Fig. 5 shows that for each routing strategy, how a DDG is modified and re-scheduled (sometimes, with additional scheduling criteria).

a. Routing data via PEs: If the dependent operations are scheduled at a distance of d cycles, DDG is modified by inserting $d-1$ routing nodes to route the dependency via PEs. Each of these routing operations can be then placed on spare PEs to satisfy the dependency. In example of Fig. 5(a), operations x and y are scheduled $d=3$ cycles far. So first, dependence $x \rightarrow y$ is routed by inserting 1 routing operation r_1. Then, the graph is re-scheduled with IMS and a P&R attempt checks whether the dependency can be routed by PEs. In the next iteration, 2^{nd} routing operation r_2 is added and the dependency can be successfully routed. In total, up to d-1 routing operations are added iteratively to route the dependency.

b. Spilling to distributed RFs is a novel solution to efficiently route a dependency in the presence of distributed RFs. In the example of Fig. 2(k), routing the dependence $e \rightarrow a$ required 2 registers and was not routed because the distributed RF had 1 register. But, RAMP finds that the total CGRA registers are 2, which can accommodate the requirement. To benefit from such availability of distributed registers, RAMP splits the register requirement. From previous mapping attempt, it determines the maximum number

of free registers (here 1, $R1$) inside an RF. Then, the graph is modified by inserting a *RF-read* operation (e_{rr}), as shown in Fig. 2(l). Operation e_{rr} reads the value of the needed data e^{i-1} and must be scheduled before the register(s) is over-written with a new value of variable e^i (i.e. between time t and $t+3$, for II=5). If successor(s) a^{i+1} is scheduled at a distance from e_{rr} then, RAMP inserts a *RF-write* operation (e_{wr}) which stores the value in free registers of some other distributed RF (e.g. in $R1$ of PE_2). RF-write operation e_{wr} should be immediately scheduled after an RF-read operation e_{rr}. During insertion of these operations, the compiler should ensure to update corresponding dependencies. For example, the distant successor a^{i+1} now obtains the needed variable e^{i-1} from e_{wr} through the RF. Plus, during placement, the spilling operations e_{rr} and e_{wr} should be mapped onto those PEs which do not share the RF.
Fig. 5(b) reflects the corresponding graph modifications and scheduling criteria, when generalized for some data dependency $x \rightarrow y$; operations x and y are scheduled at time t and t'.

c. Spilling Data to Memory modifies the dependence $x \rightarrow y$ by inserting a store and a load operation S_x and L_x, as shown in Fig. 5(c). $\forall$ (x,y), $\exists$ (S_x,L_x) such that $x \rightarrow S_x$ and $L_x \rightarrow y$. After modification, entire graph is modulo scheduled with additional constraint: load operation (L_x) must be scheduled after a store (S_x).

d. Loading a read-only variable from memory requires to insert a load operation as a predecessor, as shown in Fig. 5(d). When a live-in value 'c' cannot be accommodated in the nonrotating registers, it is loaded from the memory through an operation 'L_c'.

e. Re-computation allows to compute the predecessor operation again as a different node [8, 9]. For example, to route the dependency $x \rightarrow y$, graph in Fig. 5(e) is modified by inserting a new node x' which performs the same operation as the predecessor node x. Input edge(s) to x from p is also copied. Successor y now obtains value from the re-computed node x'. Re-computation is highly beneficial when the PE corresponding to x is connected to the PE on which z is mapped but, it is not connected to the PE on which y is mapped.

During the graph modification for a strategy, the path sharing [9] is also ensured so that all common successors of the predecessor operation can benefit from the selected routing strategy.

4.2 Failure Analysis

After a P&R attempt, several dependencies may not be mapped due to the resource constraints and RAMP should selectively apply some routing strategies, to map these dependencies. RAMP iterates on each of these unmapped dependencies/operations, and determines the potential challenge in mapping it. Based on the schedule timing of the predecessor and successor, as well as their compatibility in the obtained mapping, RAMP diagnoses 1 out of 3 P&R challenges:

i) Dependent operations are scheduled at the distant timings; routing the dependency via single RF is not possible. In such scenario, RAMP applies routing strategies such as routing via PEs, spilling the data to distributed RFs and spilling the data to the memory. In fact, with enough spare PEs, routing via PEs would successfully route the dependency. Similarly, spilling data to the distributed RFs will help to achieve the mapping if RFs in CGRA has enough free registers to accommodate the routing requirement of the data dependency. Otherwise, the dependency would be successfully routed via memory. It is possible that all of these strategies would successfully route the data dependency; the routing strategy utilizing the least resources (to map the DDG) is finally selected.

ii) Predecessor operation is a live-in/read-only value Such routing failure is possible because, either the CGRA does not have a nonrotating RF or variables of the loop accesses more live-in/read-only values than total nonrotating registers available. In either case, the read-only data should be loaded from the data memory.

iii) Dependent operations are scheduled at the consequent timing; the dependency cannot be routed because of the limited interconnect and/or less free PEs. Out-degree problem [8] is one such example of the interconnect restrictions; a node has more successors scheduled at the consequent cycle, as compared to number of PEs that are interconnected with a single PE. To P&R such operation/dependency, RAMP either re-computes the predecessor operation or, routes data via a PE, or re-schedules operations.

Algorithm 1 shows how RAMP analyzes the failures and selectively applies different resource exploration strategies, to route the dependencies (lines 8–23). If a strategy successfully routes the unmapped dependency at a given II, it is considered as a valid option. If no option is valid then it implies that no routing strategies can map the dependency due to resource constraints. So, RAMP increments II by 1 and tries again with the original DDG. Often multiple strategies can be successful to route the dependency. Then, RAMP selects the strategy that maps maximum operations while utilizing least resources (PEs and registers). Then, it keeps the corresponding modified (and rescheduled) graph and proceeds further to map remaining unmapped dependencies, at an II. As a result, RAMP produces a mapping where different data dependencies are efficiently routed through a combination of the various strategies.

Algorithm 1: *RAMP*(Input DDG D, CGRA Architecture CA)

```
1  D ← check_Support_Nonrotating_RF(D, CA), D' ← D;
2  MII ← calculateMII(D', CA), II ← MII;
3  while mapping_attempt < threshold do
4      D_S ← schedule(D', CA), R_II ← getResourceGraph(C, II);
5      CG ← Compat_Graph(D_S, R_II), C ← findMaxClique(CG);
6      if |V_C| = |V_Ds| then return C ;
7      while some node n is not mapped do
8          if (check_failure_dueTo_largeSchedDist(n, II, D_S)) then
9              valid_r, II_r, Util_r, D_r, C_r ← routeViaPEs(n);
10             valid_m, II_m, Util_m, D_m, C_m ← spillToMemory(n);
11             valid_rf, II_rf, Util_rf, D_rf, C_rf ← spillToRFs(n);
12         else if (isReadOnly(get_pred(n))) then
13             valid_ro, II_ro, Util_ro, D_ro, C_ro ← load_ReadOnly();
14         else
15             valid_rec, II_rec, Util_rec, D_rec, C_rec ← reComp(n);
16             valid_r, II_r, Util_r, D_r, C_r ← routeViaPEs(n);
17             valid_re, II_re, Util_re, D_re, C_re ← reSchedTime(n);
18         foreach valid option valid_i do
19             D_best, C_best ← best_option(II_i,Util_i,D_i,C_i);
20         if (no valid option is found) then break;
21         else
22             D' ← D_best, C' ← C_best;
23             if total_left_nodes == 0 then return C';
24     II ← II + 1;
25 return failure;
```

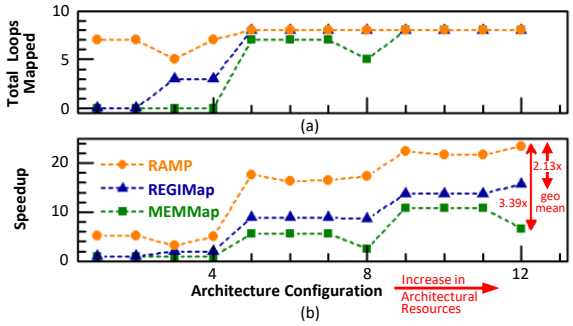

Figure 6: (a) With exhaustive resource exploration, RAMP generates code for majority of loops when compiled for CGRAs with modest resources (b) RAMP enables CGRA to accelerate loops by 23× over sequential execution.

5 EXPERIMENTAL SETUP

Benchmarks: We profiled MiBench benchmark suite [14] and determined the top-most non-vectorizable performance-critical loops for the compute-intensive applications. These benchmarks represent important workloads in the fields of security, telecom, automotive etc. and can benefit from acceleration through CGRAs.

Compilation: We implemented RAMP as a set of passes using CCF [15] – CGRA Compilation Framework (LLVM 4.0 [16] based); various routing strategies were explicitly modeled. We used optimization level 3 and also targeted complex loops consisting accesses to sub-words/pointers. The loops with conditionals were mapped using partial predication [21]. We did not target the loop consisting system calls as it cannot be accelerated by CGRA.

Simulation: Techniques were evaluated on the popular cycle-accurate simulator gem5 [22] in system emulation mode; we modeled CGRA as a separate core coupled to an ARM Cortex-like processor with ARMv7a profile. In our setup, PEs are connected in 2D torus, performing fixed-point operations with 1-cycle latency [4, 10, 13]. The memory bus is shared among PEs in a row unless specified otherwise. For a load/store operation, 2 instructions are executed; one generates address and second loads/stores data. CGRA accesses data and instruction memories of 4 kB.

Techniques Evaluated: We evaluated RAMP against state-of-the-art register-aware technique REGIMap [10] and recent memory-inclined technique MEMMap [13]. All evaluations were over a wide range of CGRA architectures (Table 1).

Table 1: Specifications of CGRA architecture configurations

Config. #	Size	RF	Reg. in RF	Memory Units (PEs)	Sharing of Memory Bus
1	2x2	Centralized	16	3, 4	dedicated
2	2x2	Centralized	16	Homo-geneous PEs (All)	shared among PEs of a row
3	2x2	Local	2		
4	2x2	Local	4		
5	4x4	Centralized	64		
6	4x4	Local	2		
7	4x4	Local	4		
8	4x4	Local	4	2,4,6,8	dedicated
9	8x8	Centralized	128	Homo-geneous PEs	shared among PEs of a row
10	8x8	Local	4		
11	8x8	Local	8		
12	8x8	Local	8	1,3,5,7,9,11, 13,15,19,21	dedicated

6 RESULTS AND ANALYSIS

6.1 With holistic approach RAMP can generate valid mappings even with limited resources

Fig. 6(a) shows a total number of performance-critical loops mapped by each technique, for each CGRA architecture configuration. With limited architectural resources, both MEMMap and REGIMap were not able to generate the code for most of the loops. Hence, these loops had to be executed on the CPU, losing acceleration benefits. For smaller CGRAs, II is often resource-bounded with a higher value; some intra-iteration dependence can be easily routed just through few registers. However, MEMMap routed them through memory by inserting additional load/store nodes. It was not possible for MEMMap to place and route excessive nodes – especially memory operations – due to the limited number of PEs and limited bandwidth. In absence of a re-scheduling scheme, MEMMap failed to place and route all operations, even at higher II. For example, *bitcount* contains 13 operations to be mapped on a 2×2 CGRA, with 4 loop-carried dependencies with weight 1. After scheduling DDG, MEMMap opted to route them through memory i.e. with additional 4 loads and 4 stores. This not only increased total operations by 62% but, required 6 operations to be scheduled and mapped on 4 PEs at the same time. Due to lack of an efficient re-schedule mechanism, MEMMap failed to map these operations, even at unbounded II.

On the other hand, REGIMap failed to map operations when limited number of registers was available. For example, performance-critical loop of *gsm* contains 200+ operations and 20+ intra/inter-dependencies. REGIMap was not able to spill the data to memory when registers were insufficient. So, it had to route the values through PEs, once registers were already utilized. The small number of total PEs made it impossible to route all such dependencies. In contrast, RAMP flexibly explored various routing alternatives to map such unmapped nodes/dependencies (by spilling to memory or to other distributed RFs). So, RAMP was able to map – and accelerate – the majority of the loops with modest CGRA resources. In fact, with the least resources (Config. 1), RAMP achieved a geomean MII/II of 0.75 and accelerated loops by 4×.

6.2 RAMP Scales Well with Resources

Fig. 6(b) compares the performance of techniques over a sequential execution on the processor core. For total 8 (benchmarks) × 12 (configurations) × 3 (techniques) evaluations, we determined performance in terms of execution cycles and computed speedup of a technique over a sequential execution. Then, we report a geomean of speedup for all the benchmarks. For CGRAs of a particular size, both REGIMap and MEMMap did not benefit much from the variation of resources and their definitions restricted mappings benefiting from available heterogeneous/custom architecture features such as global RF, nonrotating registers (to manage live values), sharing of the memory bus (bandwidth) etc.

In contrast, RAMP's comprehensive problem formulation exploited the architectural resources available, outperforming these techniques. For example, for configurations 7–12, RAMP achieved II closer to MII (MII/II = 0.90–0.97). With the higher resources, RAMP accelerated loops by 23× as compared to the sequential execution. Most of the loops are resource-bounded where RAMP's exhaustive exploration of routing strategies allowed it to consistently perform

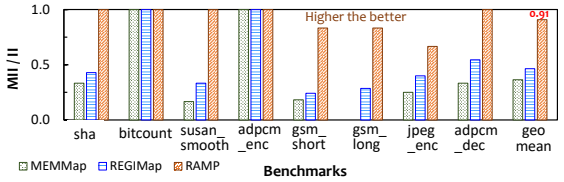

Figure 7: Performance of techniques when loops are mapped to a 4×4 CGRA with 4 local registers (configuration 7).

better. The only exception is the benchmarks where MII of a loop is recurrence-bounded (Fig. 7). For example, a critical path of *adpcm decoder* consists of 22 operations with a loop-carried dependence of weight 1. So, all the three techniques were able to easily achieve a mapping at higher II. Overall, RAMP achieved 2.13× better performance than REGIMap and 3.39× than MEMMap, over 12 different target architectures.

6.3 Spilling to Memory and to Distributed RFs is Very Effective

Detailed analysis of the mappings obtained by RAMP reveals that new strategies of spilling data to distributed RFs or to memory are quite effective. When there were fewer registers (e.g. configurations 1-4), RAMP accessed some live values from memory or spilled to memory some dependencies with large distance. In fact, RAMP spilled data to the memory only after exploiting the available registers (just like how a regular compiler spills!). This ensured relatively less (memory) operations and least routing while utilizing available registers. For example, with limited resources, spilling to memory successfully mapped some operations for *jpeg* and *gsm*. Similarly, when the RFs were distributed, RAMP benefited from the novel solution of spilling values to other RFs, finding better mappings for the loops of *susan*, *adpcm* etc. We also observed that present-day compilation strategy of implicitly routing the dependencies in an ad-hoc manner does not work well. Instead, letting the compiler to re-schedule the graph and systematically determine an efficient way of mapping the dependency resulted in better performance.

Moreover, we implemented RTL for CGRA, mapping it to Synopsys 32nm process and synthesized it with Cadence RTL compiler. With obtained critical path delay (D) and power (P), we computed energy E as P×C×D [7], if execution cycles achieved for a critical loop is C. With a substantial reduction in execution time, RAMP significantly reduced energy consumption. For example, for a 4×4 CGRA with 4 local registers, RAMP reduced energy consumption by 48% as compared to MEMMap and by 34% as compared to REGIMap.

6.4 Computational Complexity

After scheduling n nodes at some II, RAMP constructs operation-PE product graph of size mn; CGRA consists of m PEs. Then, RAMP constructs a compatibility graph [10] of size $N = (mn)^2$, determining whether an operation-PE pair can co-exist with other pairs in the search space. To P&R n nodes, RAMP checks for a clique of size n in the compatibility graph of size N. Our clique search is based on the algorithm of [23], which searches for a clique in polynomial time in at most N^8 steps. Therefore, just like other clique-based mapping heuristics REGIMap and MEMMap, the computational complexity of RAMP is $O(N^8)$.

However, computation time of RAMP is comparable to REGIMap and MEMMap (in order of seconds), if not always better. Essentially, this stems from higher mapping quality (fewer iterations due to 2× better II) and far less nodes to be mapped (i.e., smaller n) in any of the attempts. For example, both REGIMap and MEMMap load the live-in data from the memory [10, 13, 18]. Plus, REGIMap cannot spill the data and requires many routing operations, when constrained by the availability of few local registers. Similarly, MEMMap often routes data via memory, even if enough registers are available. Thus, they have to map 1.5×-2× nodes than RAMP.

7 SUMMARY

This paper presents challenges with existing mapping techniques, which are unable to make good use of the routing resources. They first schedule the DDG and then attempt the P&R; routing is *internal* to P&R and is carried out in an *ad-hoc* manner. As a result, the operations may not be mapped due to resource constraints. This paper introduces RAMP which models various routing strategies *explicitly* and *flexibly* explore various ways to map the data dependencies while exploiting the CGRA resources. RAMP accelerates the top performance-critical loops of MiBench by 23× over a sequential execution and by 2.13× over state-of-the-art techniques.

ACKNOWLEDGMENTS

This work was partially supported by funding from NSF grants CNS 1525855 and CCF 172346 - NSF/Intel joint research center for Computer Assisted Programming for Heterogeneous Architectures (CAPA).

REFERENCES

[1] Shuai Che, Jie Li, Jeremy W Sheaffer, Kevin Skadron, and John Lach. Accelerating compute-intensive applications with gpus and fpgas. In *SASP*, 2008.
[2] Bingfeng Mei, M Berekovic, and JY Mignolet. Adres & dresc: Architecture and compiler for coarse-grain reconfigurable processors. Springer, 2007.
[3] Hyunchul Park et al. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In *PACT*, 2008.
[4] Taewook Oh et al. Recurrence cycle aware modulo scheduling for coarse-grained reconfigurable architectures. In *ACM Sigplan Notices*, 2009.
[5] Hongsik Lee, Dong Nguyen, and Jongeun Lee. Optimizing stream program performance on cgra-based systems. In *DAC*, 2015.
[6] Zhongyuan Zhao et al. Optimizing the data placement and transformation for multi-bank cgra computing system. In *DATE*, 2018.
[7] Manupa Karunaratne et al. Hycube: A cgra with reconfigurable single-cycle multi-hop interconnect. In *DAC*, 2017.
[8] Mahdi Hamzeh, Aviral Shrivastava, and Sarma Vrudhula. Epimap: using epimorphism to map applications on cgras. In *DAC*, 2012.
[9] Liang Chen and Tulika Mitra. Graph minor approach for application mapping on cgras. *ACM TRETS*, 2014.
[10] Mahdi Hamzeh, Aviral Shrivastava, and Sarma Vrudhula. Regimap: Register-aware application mapping on cgras. In *DAC*, 2013.
[11] Panagiotis Theocharis and Bjorn De Sutter. A bimodal scheduler for coarse-grained reconfigurable arrays. *ACM TACO*, 2016.
[12] Bjorn De Sutter et al. Placement-and-routing-based register allocation for coarse-grained reconfigurable arrays. In *ACM Sigplan Notices*, 2008.
[13] Shouyi Yin et al. Memory-aware loop mapping on coarse-grained reconfigurable architectures. *IEEE TVLSI*, 2016.
[14] Matthew Guthaus et al. Mibench: A free, commercially representative embedded benchmark suite. In *WWC*, 2001.
[15] Shail Dave and Aviral Shrivastava. Ccf: A cgra compilation framework. 2018.
[16] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, 2004.
[17] B Ramakrishna Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *MICRO*, 1994.
[18] Shail Dave, Mahesh Balasubramanian, and Aviral Shrivastava. Ureca: A compiler solution to manage unified register file for cgras. In *DATE*, 2018.
[19] Giovanni Ansaloni, Paolo Bonzini, and Laura Pozzi. Egra: A coarse grained reconfigurable architectural template. *IEEE TVLSI*, 2011.
[20] S Alexander Chin et al. Architecture exploration of standard-cell and fpga-overlay cgras using the open-source cgra-me framework. In *ISPD*, 2018.
[21] Kyuseung Han, Junwhan Ahn, and Kiyoung Choi. Power-efficient predication techniques for acceleration of control flow execution on cgra. *ACM TACO*, 2013.
[22] Nathan Binkert et al. The gem5 simulator. 2011.
[23] Ashay Dharwadker. The clique algorithm, 2006.