Subgraph Decoupling and Rescheduling for Increased Utilization in CGRA Architecture

Chen Yin, Qin Wang, Jianfei Jiang, Weiguang Sheng, Guanghui He, Zhigang Mao and Naifeng Jing*
Department of Micro/Nano Electronics, Shanghai Jiao Tong University, Shanghai, China
Email: {yinchen, sjtuj}@sjtu.edu.cn

***Abstract*—When coarse-grained reconfigurable array (CGRA) architecture is shifting towards general-purpose, some complex control flows, such as nested loop, conditional branch and data dependence, may embarrass it and reduce the processing element (PE) array utilization by breaking the intact dataflow graph (DFG) into multiple regions with inconsistent control regions. This paper proposes subgraph decoupling and rescheduling, which decouples the inconsistent regions into control-independent subgraphs. Each subgraph can be rescheduled with zero-cost domino context switching and parallelized to fully utilize the PE resources. Then, we propose lightweight hardware changes based on general CGRA architecture to enable our design. The experiment results show that our proposal can improve the performance and energy efficiency by 1.35× and 1.18× over a static-mapped CGRA (Plasticine), and by 1.27× and 1.45× over an instruction-driven CGRA (TIA).**

I. Introduction

For the past decade, CGRA architecture is emerging as a highly programmable and energy-efficient acceleration for domain-specific computing. It eliminates the centralized program counter (PC) and register file with a large number of reconfigurable processing elements (PEs) and interconnects to enable dataflow-driven execution in spatial. Because multiple PEs can fire at the same time whenever their input data is available, CGRA can offer abundant parallelism and high computing throughput via software pipelining.

With such benefits, CGRA tends to get wider adoption in more general-purpose acceleration, but has to face multiple dataflows coupled with complex control flows. To this problem, several prior works [1]–[3] have proposed different techniques with augmented control structures to accommodate the coupled data and control flows in the same array without switching the kernel back-and-forth between the host and accelerator. For example, conditional branch operations having both true and false paths can be mapped in spatial through static single assignment and if-conversion.

Although control flow succeeds in the spatial mapping, it may compress the computing efficiency and utilization of the dataflow-driven execution in the array. As next section elaborates, we identify three typical cases. For example, for kernels with conditional branch, the divergent paths are executed exclusively but may waste PEs in the other path. For kernels with imperfect loops, the different data rates between outer and inner loops may result in lower utilization of outer PEs. For loop carried dependence, the serial execution may break the software pipelining that results in higher iteration interval. In addition, the three cases can be common in many kernels in MachSuite [4], Rodinia [5] and PolyBench [6]. As Fig. 1 reports, the average percentage of the three cases can be over 50%, which implies that at least half of the PEs may be under-utilized during the CGRA execution. There are possible solutions for each case, but unfortunately, none of them work for all.

This work is supported by National Key Research and Development Program of China (Grant No. 2018YFA0701500) and National Natural Science Foundation of China (Grant No. 61772331). Naifeng Jing is the corresponding author.

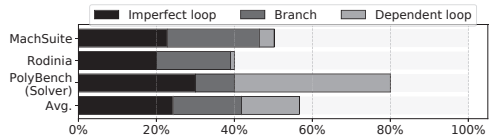

Fig. 1. Percentage of instructions in different inconsistent control regions for several typical benchmark suites. To avoid double counting, if a case exists in more than one inconsistent regions, we only count the closest one.

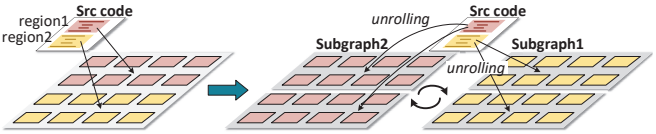

Fig. 2. The basic concept of subgraph decoupling and rescheduling.

In essence, these cases all suffer from the low efficiency and utilization because the control flow may break a DFG into multiple regions which bear inconsistent control, by imposing different dataflow rates in each region and breaking the uniform dataflow execution preferred in CGRA. To address the problem, we propose subgraph decoupling and rescheduling. As illustrated in Fig. 2, instead of mapping the inconsistent control regions in the same array as normal, we split the DFG into multiple subgraphs across the inconsistent control regions. Then, we map only one subgraph at a time but with sufficient unrolling on PEs, and reschedule different subgraphs in serial to remove the inconsistency in one array. We mainly made three contributions in this paper:

- We propose a subgraph decoupling method, and in this way the same subgraphs can execute in the same rate at a time, avoiding wasting the PEs to sustain a high utilization rate throughout the execution.
- To reduce the overhead in subgraph decoupling, we im-

plement a zero-cost domino context switching method by chasing the dataflow propagation.

- We design a hardware subgraph scheduler and add a lightweight subgraph switching logic inside each PE to support the out-of-order execution on the subgraph-level.

Our experiment results show that our design improves the performance and energy efficiency over a static-mapped CGRA (Plasticine) [7] by $1.35\times$ and $1.18\times$, over an instruction-driven CGRA (TIA) [1] by $1.27\times$ and $1.45\times$.

II. Related Works and Case Studies

In this section, we identify three inconsistent control regions commonly seen in DSA benchmark codes.

Imperfect loop. It represents a nested loop which has computation in outer loop bodies. It can be found in blocked matrix multiple (GEMM) and computational fluid dynamics (CFD) in the scientific computing and finite element analysis.

Fig. 3(a) shows a code snippet in *GEMM*, where the inner loop (in red) executes every $block\_size$ times while the outer loop (in yellow) executes only once. Therefore, if they are mapped in one array, PEs of the outer loop have to wait or idle until the inner loop finishes all its iterations. Because iterations of inner loops are usually much longer than that of the outer loop, the outer PEs are likely to be idle during most of the execution time that results in significant low unitization. To this problem, [8]–[10] applies loop scheduling techniques, such as loop interchange and flattening, to reorganize the inner and outer loop bodies, but cannot eliminate the low utilization of the outer PEs in a complete way.

Branch divergency. It represents conditional operation with both true and false paths. It can be common in many kernels, such as *Sort* and *Merge*, in database and sparse computing.

Fig. 3(b) shows a code snippet in *Sparse Vector Multiply*, where the dataflow in conditional branch will dynamically fork into two (true and false) paths in an exclusive way. If we map both paths onto dedicate PEs in the same array, each data will trigger just one path according to the condition while the other one will become idle. To this problem, 4d-CGRA [11] and Laser [12] fuse the operations of the mutually-exclusive paths into one set of PEs to improve the utilization. However, they fall short in dealing with imbalanced branches, because there could be extra operations in the longer path that cannot be fused into the shorter one.

Loop dependence. For kernels with loop dependence, the correlative produce-consumer relationship are common in numerical analysis, such as *LU/QR/Cholesky decomposition*.

Fig. 3(c) shows a code snippet in *LUD*, where there are two sibling loops modifying the same matrix alternately (by updating $a[i][j]$ and $a[j][i]$) to decompose it into an upper and a lower triangular matrix. However, in each iteration, both of them relies on the values updated by the other one. Therefore, there exists a correlative producer-consumer relationship of the dataflow between these two data dependent loop bodies. If we mapping these dependent loops simultaneously with an explicit barrier for synchronization as Fig. 4(e), both of them have to be idle alternately. To this problem, dMT-CGRA

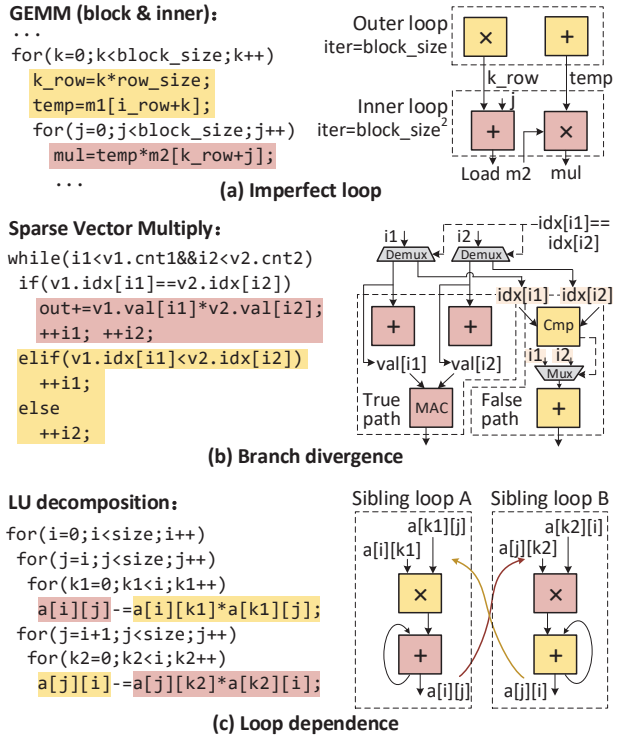

Fig. 3. The code snippets that result in inconsistent control regions.

[13] and REVEL [14] use inter-thread/lane communication to exploit fine-grained parallelism among dependent regions, but it introduces additional data communication overhead.

In fact, there could be other cases bearing inconsistent control regions, such as function calling and submodule (Divider/Cordic) sharing in general computing domain. Mapping multiple control regions in the DFG as a whole onto the array (if possible) may incur considerable inefficiency on PE utilization.

III. Subgraph Decoupling and Rescheduling

To the PE low-efficiency and under-utilization problems, we will propose the subgraph decoupling and rescheduling in this section. We will also show how to decouple the computational DFG into the subgraphs to enable our proposal.

A. Subgraph Decoupling

Based on the basic idea in Fig. 2, we will still take the three cases to explain how the subgraph decoupling works.

Imperfect loop decoupling. By decoupling the imperfect nested outer and inner loops into two subgraphs, we can map each of them respectively. As the example in Fig. 4(b) illustrates, the subgraph of the outer loop can unroll twice to fully occupy the PEs. Then, the two unrolled iterations produce the intermediate data of iterations k and $k+1$ simultaneously. Instead of sending them directly to the PEs of the inner loop as the traditional dataflow mapping(Fig. 4(a)), the produced intermediate values of $k\_row[\cdots]$ and $temp[\cdots]$ can be stored in a buffer, namely decoupling buffer, in serial following the software pipelining manner. Upon subgraph switching, e.g.

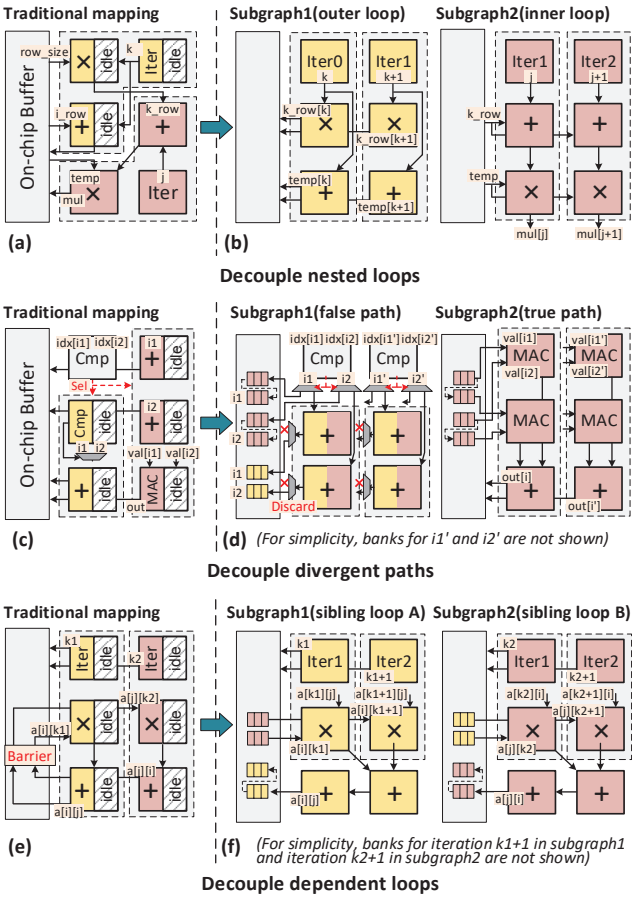

Fig. 4. Mapping (a) imperfect loops, (c) divergent branch paths, (e) dependent sibling loops, entirely on the array; (b) Decoupling the imperfect loop into two subgraphs with unrolling twice; (d) Decouple the branch into true/false path subgraphs and allocate dedicated decoupling buffer for each; (f) Decoupling the dependent sibling loops and alternately execute each with fully parallelization.

when the buffer is full, the subgraph of inner loop will be brought in and unrolled to occupy as many PEs as possible. Then, the inner loops consume the intermediate data from the decoupling buffer on $k\_row[\cdots]$ and $temp[\cdots]$ one-by-one. Once the buffered data runs out, it will switch to the outer loop again. In this way, the two decoupled subgraphs run in turn until both are completed.

Divergent path decoupling. Because the true and false paths are exclusive that cannot be activated at the same time, we can decouple the *if*-statement into two subgraphs. As illustrated in Fig. 4(d), the index i_1 and i_2 which satisfy the *if* condition will be collected in the decoupling buffer (in red) and consumed by the true path subgraph with fully unrolling. Due to index i_1 and i_2 need updates in both true and false paths[1], we fuse these operations into the false path subgraph and discards one of the outputs according to the *elseif* condition. To make the branch condition separate and in order, we organize the local buffer as true-path buffers (in red) and false-path buffers (in yellow)

[1]The *elseif* and *else* statements are both regarded as the false path.

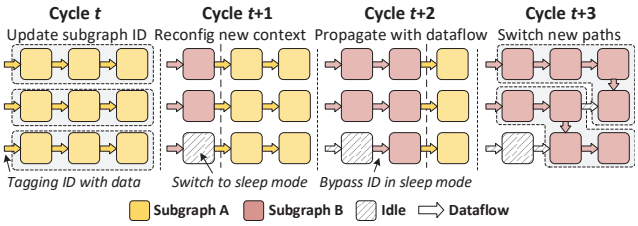

Fig. 5. Dynamic domino context switching.

for the next two subgraphs independently. This method also applies to nested branches.

Dependent loop decoupling. We can decouple the sibling loops in two individual subgraphs and each can run in parallel without synchronization with the other. We also unroll them to occupy as many PEs as possible as illustrated in Fig. 4(f).

In summary, decoupling the inconsistent control regions into several subgraphs of the same dataflow rate can exploit the potential parallelism of each subgraph. Then, unrolled mapping can increase the PE utilization in the array as much as possible. Nevertheless, the subgraph decoupling introduces intermediate data, which has to be stored in the decoupling buffer for later subgraphs. We will propose to leverage the on-chip buffer local to PE array to fulfill this need.

B. Domino Context Switching

Oftentimes, PE execution in CGRA is driven by dataflow. So, we cannot switch the PE contexts all in one go unless all PEs along the dataflow complete. However, this may result in many bubbles between switching due to pipeline draining and refilling. For zero-cost subgraph switching, we propose domino context switching, which can switch the context of each PE along with the dataflow propagation like a domino at runtime.

Fig. 5 illustrates how the domino context switching works. When the current subgraph A completes at cycle t, the subgraph scheduler will swap in a new subgraph B, by tagging the subgraph ID of B with the last valid data of subgraph A. When a following PE detects the ID switching, it will reconfigure itself to subgraph B after the last data of subgraph A has finished its operation in this PE at cycle $t+1$. With the dataflow propagation, more PEs switch contexts like domino while the subgraph B can start just after the reconfiguration in cycle $t+2$. If any PE are not needed in subgraph B, it can be power gated as [15] to save power. A gated PE still connects to a nearby active PE for later wake-up. At cycle $t+3$, the whole array is reconfigured from subgraph A of unrolling three times into subgraph B of unrolling twice.

C. Subgraph Generation

The subgraphs, split by inconsistent control regions, are generated by Algorithm 1. The input contains target DFG and architecture description, mainly including the number of PEs. The output consists of partitioned subgraphs and associated unroll times.

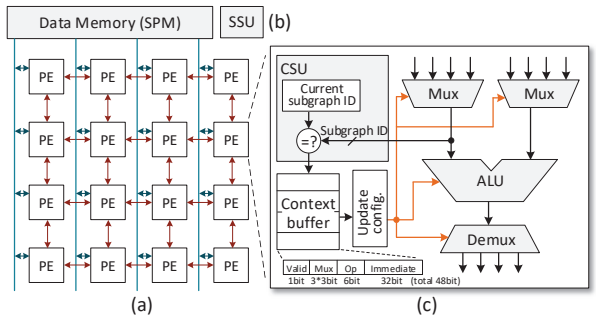

Fig. 6. (a) Overall architecture of a typical CGRA; (b) SSU: subgraph scheduler unit; (c) DPE: PE supporting domino switching.

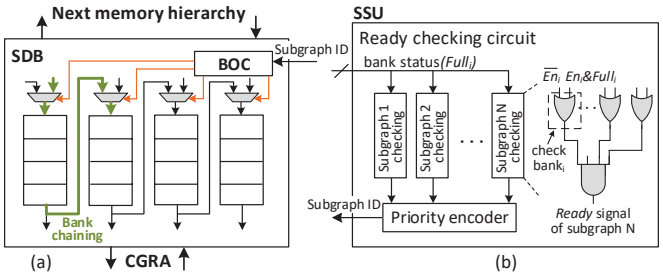

Fig. 7. Implementation details of (a) SDB and (b) SSU.

Initially, we partition the intact DFG into several *basic subgraphs* across the control regions[2] (line 1), as the different colors in Fig. 3. In order to avoid deadlock at runtime, we need to ensure the connection is convex, i.e. there is no cyclic dataflow between any two *basic subgraphs*. Otherwise, we need to cut off the cycle and split one of them into another two *basic subgraphs* (line 3-4). If the convex subgraph is still too large to map in the array, we also need split it into smaller ones (line 6-10). To minimize cut edges, we use the Min-Cut algorithm in subgraph splitting. Then, we figure out the unrolling times for each subgraph according to the hardware resource (line 13). Finally, we have several independent subgraphs that can run individually on the array.

Algorithm 1: Subgraph generation

Input: *DFG, PhyRes* = {*Pe_num, Bank_num*}
Output: {$Subgraph_i$, $unroll\_num_i$}

1 {$basic\_Subgraph_i$} ← Partition(*DFG*);
2 **foreach** $basic\_Subgraph_i$ **do**
3 | **if** *!checkConvex*($basic\_Subgraph_i$) **then**
4 | | {$basic\_Subgraph_i$} ← Convex($basic\_Subgraph_i$);
5 | **end**
6 | **if** *Res*($basic\_Subgraph_i$) > *PhyRes* **then**
7 | | {$Subgraph_i$} ← Split($basic\_Subgraph_i$);
8 | **else**
9 | | {$Subgraph_i$} ← ($basic\_Subgraph_i$);
10 | **end**
11 **end**
12 **foreach** $Subgraph_i$ **do**
13 | $unroll\_num_i = \lfloor PhyRes \,/\, Res(Subgraph_i) \rfloor$;
14 **end**

IV. Proposed CGRA Architecture

Based on a typical CGRA as in Fig. 6(a), we will introduce some lightweight architecture changes, namely the *subgraph decoupling buffer*, *subgraph scheduling unit* (SSU) as in Fig. 6(b) and *domino reconfigured PE* (DPE) as in Fig. 6(c) to support the subgraph rescheduling and switching.

A. Subgraph Decoupling Buffer (SDB)

To support the subgraph decoupling, the proposed CGRA needs to buffer intermediate data between consequently scheduled subgraphs. We propose to leverage the existing on-chip local data buffer attached to each PE array to fulfill this need.

Fig. 7(a) illustrates the details of the proposed subgraph decoupling buffer. In our proposal, the conventional local data buffer can both respond to memory requests and store intermediate data between subgraphs in Fig. 4. As prior studies [16], [17] show, the local buffer often applies a multi-bank structure for paralleled accesses from the large number of PEs. So, when it is configured to store the intermediate data, we allocate different banks to different data (i.e. k_{row} and $temp$ as in Fig. 4). To avoid address computing for each data, the bank should work like a FIFO. Due to the static configuration of each PE, each bank serves one PE in a certain subgraph to avoid bank conflicts. The memory requests work as normal, which communicates between PEs and the next memory hierarchy.

To avoid interference between decoupled intermediate data and normal local buffer accesses, we can allocate them to different banks in a certain subgraph if the number of banks permits. So, we should reduce the number of decoupled dataflows, which in fact is the number of cut edges between subgraphs, when generating the subgraphs. Because keeping the intermediate data will occupy additional buffer, we leverage fast subgraph switching to restrict the amount of intermediate in subgraph execution.

The decoupling buffer will be managed by the *Bank Organization Controller* (BOC). For larger decoupling buffer, several adjacent banks can be chained as a large FIFO for being written serially under one subgraph and read in parallel under another subgraph. In this way, it can support sufficient bank bandwidth for subgraph unrolling.

B. Subgraph Schedule Unit (SSU)

The SSU is in charge of the PE execution. It reschedules the subgraphs according to the subgraph status, which mainly depends on the availability of its incoming and outgoing data in the decoupling buffer.

In each cycle, SSU checks whether the working subgraph has to suspend based on the status of the decoupling buffer. That is, if the incoming data runs out or the outgoing data overfills the decoupling buffer (or any bank inside) corresponding to this subgraph, this subgraph suspends. At the same time, it immediately signals PEs in the array to switch their contexts by adding one bit signal, namely *last* flag, on the last dataflow

[2]Namely, a *basic subgraph* is consist of *basic blocks* in a same control region.

of the old subgraph. This signal will trigger PEs for domino context switching with zero-cost. Note that due to the limited size of the decoupling buffer, a loop with a large number of iterations need to be sliced into several chunks. At the end of each chunk and the last data in an ending chunk, the *last* signal takes effect for context switching.

If any subgraph is suspended, SSU will select a new subgraph among the ready ones by checking whether its incoming data is available in the decoupling buffer. The checking can be done by the *ready checking circuit* of each subgraph and a priority encoder as illustrated in Fig. 7(b). The priority encoder is configured according to the dependence (if exist) among all the subgraphs, which can be determined in the subgraph generation stage (III-C). If there are multiple ready subgraphs, any of them can be picked and out-of-order executed.

C. Domino Reconfigured PE (DPE)

To support the rapid domino context switching, we add a *Context Switching Unit* (CSU) in each PE. Fig. 6(c) shows the CSU built upon a general reconfigurable PE structure.

Whenever a new subgraph is rescheduled by SSU and switched into a heading PE, the new subgraph ID bind with the last data of the old subgraph will update the PE's old configuration. That is, once the CSU detects the ID has changed, it will send the new ID to the context buffer and fetch the new configuration of this PE. The entry number of the context buffer equals to the maximum number of subgraphs supported in CGRA, and each entry stores the configuration of a PE. The new configuration will instruct each PE to reconfigure itself in a pipeline manner by chasing the last dataflow of the old subgraph. When the last dataflow reaches the next PE along the previously configured data path, the new subgraph ID will trigger the CSU in the next PE for reconfiguration. In this way, PEs can be dynamically reconfigured along with the dataflow like a domino as illustrated in section III-B.

To enable the domino context switching on PEs, the decoupled buffer will also be reorganized as soon as the SSU suspends the old subgraph and picks out the new one. The decoupling buffer, managed by the BOC unit will also be reconfigured to start fetching the intermediate data generated by previous subgraphs, so PEs in the new subgraph can get new data immediately to start computing under this new subgraph.

V. Experimental Results

A. Experiment Setup

We select several workloads from three widely used benchmark suites in domain specific computing because they consist of the three cases for inconsistent control regions. Table I lists the selected kernels.

For comparison, we use two typical CGRA architectures, i.e. Plasticine [7] and an improved TIA [18]. Plasticine features by dedicated PEs and TIA features by instruction-scheduled PEs. The parameter settings of the three architectures are shown in Table II, where we align them with similar area budget.

To model their performance, we leverage the open source implementations for Plasticine [7] and TIA [18]. We modify the PE in Plasticine by adding CSU to model our PE structure. We implement the BOC and SSU based on Buffets [19], which provides a flexible on-chip storage management. We synthesize our design using Synopsys Design Compiler with a 40nm technology library at a 800MHz frequency. We profile the power of each single PE using Synopsys PrimeTime with RTL traces and estimate the RAM base on CACTI [20].

TABLE I
Workloads for Evaluation

Workload	Characteristic	Benchmark suite
GEMM, Viterbi Sort, FFT	Imperfect loop Branch divergency	MachSuite
CFD HotSpot LUD, GE	Imperfect loop Branch divergency Loop dependency	Rodinia
Gesummv Cholesky	Imperfect loop Loop dependency	PolyBench

TABLE II
Architecture Parameter Setting

		Plasticine [7]	TIA [18]	Ours
PE	Instr./context buffer size	\	16×128bit	8×48bit
	Area(um^2) @800MHz	10327	31090	13745
On-chip buffer	Bank num	16		
	RAM size	64KB		64KB+BOC+SSU
	Area(um^2)	RAM: 266160 (97%)		BOC+SSU: 7186 (3%)
Overall system	Array Size	8×8	4×5	6×8
	Area(mm^2)	0.927	0.887	0.933

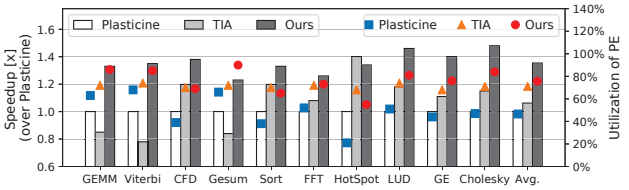

Fig. 8. PE utilization (in marker) and performance (in bar) comparisons among the three architectures normalized to Plasticine.

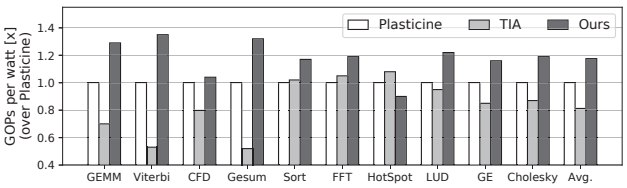

Fig. 9. Energy efficiency comparisons among the three architectures normalized to Plasticine.

B. Results and Analysis

Utilization and performance. Fig. 8 reports the performance and PE utilization on the three architectures. Different kernels behave differently. For imperfect loop like GEMM, Viterbi and Gesummv, the inner and outer loops are almost equal in size. Thus nearly 50% PEs are idle in Plasticine because it applies dedicated PEs. TIA and our design can leverage flexible instruction/subgraph rescheduling to improve the PE utilization up to ∼80%. So our design can gain ∼1.3× speedup compared with Plasticine. Due to the area overhead of a large instruction buffer and control logic, the array size of TIA is much smaller, which limits its performance to only 0.8× compared with Plasticine. For CFD, this kernel is too large to be mapped in a single configuration for all these three architectures. Thus in Plasticine, PE utilization reduces to 42% suffering from pipeline refilling and draining on context switching. TIA leverages its large instruction buffer to reduce reconfiguration times and our design use domino context switching to relieve this problem.

For kernels with branches such as Sort, FFT and HotSpot, the PE utilization is significantly degraded in Plasticine, especially in HotSpot (only 21%), which has plenty of *elseif* statements. TIA fuses the operations of different branch paths to maintain PE utilization around 75% and archives a ∼1.2× speedup. Our design decouples the branch paths into independent subgraphs. Due to the context buffer size limitation, it is hard to decouple each branch path as a single subgraph, so our PE utilization reduces to around 70%, but it still achieves a speedup of ∼1.3× due to more PE resources.

For workloads as LUD, GE and Cholesky, data dependence reduces the PE utilization of Plasticine by limiting inter sibling loops parallelism. Explicit data barrier also exacerbates the problem and reduces the utilization down to ∼45%. However, TIA and our design maintain the utilization around 75% by PE rescheduling.

Energy efficiency. Fig. 9 reports the energy efficiency evaluation of the three architectures. Although TIA achieves a relative high PE utilization and performance, the instruction scheduler consumes a considerable fraction of energy because it works each cycle, which significantly reduce its energy efficiency compared with Plasticine. For our design, it is efficient in the most of cases by rescheduling at subgraph level. But for HotSpot and CFD, our design consumes more energy on decoupling buffer accesses due to the frequently subgraph switching.

In conclusion, with the subgraph decoupling and PE rescheduling, our design archives an average performance speedup of 1.35× and 1.27× and an average energy efficiency gain of 1.18× and 1.45× with PE utilization improvement of 1.62× and 1.06× compared with Plasticine and TIA.

VI. Conclusion

This paper presents a subgraph decoupling and rescheduling when CGRA dealing with kernels bearing inconsistent controls. By decoupling the entire DFG into multiple subgraphs, our design can parallelize each subgraph on CGRAs to fully utilize PE resources with a lightweight hardware modification. The evaluation shows that our proposal gains enhancements on both performance and energy efficiency. And this mechanism can be extended to support other inconsistent controls as function calling and submodule sharing during CGRA execution in future work.

References

[1] A. Parashar, M. Pellauer, M. Adler, B. Ahsan, N. Crago, D. Lustig, V. Pavlov, A. Zhai, M. Gambhir, A. Jaleel *et al.*, "Triggered instructions: a control paradigm for spatially-programmed architectures," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3, pp. 142–153, 2013.

[2] M. Gao and C. Kozyrakis, "HRL: Efficient and flexible reconfigurable logic for near-data processing," in *Proc. HPCA (2016)*, pp. 126–137.

[3] D. Voitsechov and Y. Etsion, "Single-graph multiple flows: Energy efficient design alternative for GPGPUs," *ACM SIGARCH computer architecture news*, vol. 42, no. 3, pp. 205–216, 2014.

[4] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks, "MachSuite: Benchmarks for accelerator design and customized architectures," in *Proc. IISWC (2014)*, pp. 110–119.

[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. IISWC (2009)*, pp. 44–54.

[6] L.-N. Pouchet *et al.*, "PolyBench: The polyhedral benchmark suite," *URL: http://www. cs. ucla. edu/pouchet/software/polybench*, 2012.

[7] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, "Plasticine: A reconfigurable architecture for parallel patterns," in *Proc. ISCA (2017)*, pp. 389–402.

[8] X. Lin, S. Yin, L. Liu, and S. Wei, "Exploiting parallelism of imperfect nested loops with sibling inner loops on coarse-grained reconfigurable architectures," in *Proc. ASP-DAC (2014)*, pp. 456–461.

[9] H. Rong, Z. Tang, R. Govindarajan, A. Douillet, and G. R. Gao, "Single-dimension software pipelining for multidimensional loops," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 4, no. 1, pp. 7–es, 2007.

[10] J. Lee, S. Seo, H. Lee, and H. U. Sim, "Flattening-based mapping of imperfect loop nests for CGRAs," in *Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis*, 2014, pp. 1–10.

[11] M. Karunaratne, D. Wijerathne, T. Mitra, and L.-S. Peh, "4D-CGRA: Introducing branch dimension to spatio-temporal application mapping on CGRAs." in *Proc. ICCAD (2019)*, pp. 1–8.

[12] M. Balasubramanian, S. Dave, A. Shrivastava, and R. Jeyapaul, "LASER: A hardware/software approach to accelerate complicated loops on CGRAs," in *Proc. DATE (2018)*, pp. 1069–1074.

[13] D. Voitsechov, O. Port, and Y. Etsion, "Inter-thread communication in multithreaded, reconfigurable coarse-grain arrays," in *Proc. MICRO (2018)*, pp. 42–54.

[14] J. Weng, S. Liu, Z. Wang, V. Dadu, and T. Nowatzki, "A hybrid systolic-dataflow architecture for inductive matrix algorithms," in *Proc. HPCA (2020)*, pp. 703–716.

[15] A. Nayak, K. Zhang, R. Setaluri, A. Carsello, M. Mann, S. Richardson, R. Bahr, P. Hanrahan, M. Horowitz, and P. Raina, "A framework for adding low-overhead, fine-grained power domains to CGRAs," in *Proc. DATE (2020)*, pp. 846–851.

[16] Y. Kim, J. Lee, A. Shrivastava, J. W. Yoon, D. Cho, and Y. Paek, "High throughput data mapping for coarse-grained reconfigurable architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 11, pp. 1599–1609, 2011.

[17] S. Yin, X. Yao, T. Lu, D. Liu, J. Gu, L. Liu, and S. Wei, "Conflict-free loop mapping for coarse-grained reconfigurable architecture with multi-bank memory," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 9, pp. 2471–2485, 2017.

[18] T. J. Repetti, J. P. Cerqueira, M. A. Kim, and M. Seok, "Pipelining a triggered processing element," in *Proc. MICRO (2017)*, pp. 96–108.

[19] M. Pellauer, Y. S. Shao, J. Clemons, N. Crago, K. Hegde, R. Venkatesan, S. W. Keckler, C. W. Fletcher, and J. Emer, "Buffets: An efficient and composable storage idiom for explicit decoupled data orchestration," in *Proc. ASPLOS (2019)*, pp. 137–151.

[20] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "CACTI-P: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques," in *Proc. ICCAD (2011)*, pp. 694–701.