electronics

MDPI

Article

Similarity-Aware Architecture/Compiler Co-Designed Context-Reduction Framework for Modulo-Scheduled CGRA

Zhongyuan Zhao [1], Weiguang Sheng [1], Jinchao Li [2], Pengfei Ye [3], Qin Wang [1] and Zhigang Mao [1,*]

[1] Department of Micro/Nano Electronics, Shanghai Jiaotong University, Shanghai 200240, China; zyzhao.sjtu@gmail.com (Z.Z.); wgshenghit@sjtu.edu.cn (W.S.); qinqinwang@sjtu.edu.cn (Q.W.)
[2] Huawei Technologies Shanghai, Shanghai 200299, China; lijinchao@hisilicon.com
[3] Intel Aisa Pacific Development, Shanghai 200241, China; freddy.ye@intel.com
* Correspondence: maozhigang@sjtu.edu.cn

Abstract: Modulo-scheduled coarse-grained reconfigurable array (CGRA) processors have shown their potential for exploiting loop-level parallelism at high energy efficiency. However, these CGRAs need frequent reconfiguration during their execution, which makes them suffer from large area and power overhead for context memory and context-fetching. To tackle this challenge, this paper uses an architecture/compiler co-designed method for context reduction. From an architecture perspective, we carefully partition the context into several subsections and only fetch the subsections that are different to the former context word whenever fetching the new context. We package each different subsection with an opcode and index value to formulate a context-fetching primitive (CFP) and explore the hardware design space by providing the centralized and distributed CFP-fetching CGRA to support this CFP-based context-fetching scheme. From the software side, we develop a similarity-aware tuning algorithm and integrate it into state-of-the-art modulo scheduling and memory access conflict optimization algorithms. The whole compilation flow can efficiently improve the similarities between contexts in each PE for the purpose of reducing both context-fetching latency and context footprint. Experimental results show that our HW/SW co-designed framework can improve the area efficiency and energy efficiency to at most 34% and 21% higher with only 2% performance overhead.

Keywords: CGRA; similarity-aware; context reduction; modulo scheduling; simulated annealing

Citation: Zhao, Z.; Sheng, W.; Li, J.; Ye, P.; Wang, Q.; Mao, Z. Similarity-Aware Architecture/Compiler Co-Designed Context-Reduction Framework for Modulo-Scheduled CGRA. *Electronics* **2021**, *10*, 2210. https://doi.org/10.3390/electronics10182210

Academic Editors: José Teixeira de Sousa and Mário Véstias

Received: 7 August 2021
Accepted: 6 September 2021
Published: 9 September 2021

1. Introduction

CGRA processors are gaining increasing attention from both academia and industry due to their promising energy efficiency and programmability over data- and computation-intensive applications. CGRA is known to be more efficient than FPGA in terms of reconfigurability in computation and interconnection resources [1]. During recent decades, many CGRAs [1–18] have been proposed with different reconfigure schemes. Wijtvliet et al. [19] classified the reconfigure schemes into coarse and fine temporal reconfiguration granularities. Coarse-grained temporal reconfiguration CGRAs can be reconfigured at every code region, nested loop or kernel. However, the processing elements inside the array keep executing the same operation within the code region, nested loop or kernel. Under this reconfigure scheme, no strict reconfiguration speed is required but the spatial architecture of CGRA should match the computing pattern of the target code regions. This class of CGRAs are efficient when accelerating a restricted class of kernels such as convolution and GEMM [14–16] or kernels with specific parallel patterns [1]. On the other hand, because fine-grained temporal reconfiguration CGRAs can be reconfigured at every cycle or operation, the processing elements (PEs) inside the array are frequently reconfigured to execute different operations during the CGRA runtime with low latency. This reconfigure scheme allows the compiler to efficiently deploy a large code region onto the resource-limited spatial array leveraging both temporal and spatial scheduling techniques, which makes

CGRAs able to adapt to kernels with diverse computing patterns. The aforementioned properties of these two reconfigure schemes suffer from different CGRA design challenges.

In this paper, we focus on modulo-scheduled CGRAs, a class of fine-grained temporal reconfiguration CGRAs that is capable of executing diverse loop kernels in software-pipelining scheme [8–10,13,18,20]. In general, the modulo-scheduled CGRAs (Figure 1) consist of a processing element array (PEA), on-chip global context memory (GCM) and on-chip multi-bank shared data memory (SDM). The PEA can be designed in a homogeneous or heterogeneous style. The heterogeneous PEA consists of PE with different functionalities and interconnections. The heterogeneous CGRAs (such as ADRES [8] and SRP [10]) usually have higher area efficiency and resource use due to less redundant components inside PE, but usually bring more restriction to the compilation flow. On the other hand, with moderate sacrifice, homogeneous CGRAs (such as HyCUBE [13], HReA [18] and CCF [20]) are usually able to simplify the compilation problem by providing the PEA with symmetric PE and interconnect networks between each PE. The compilers [21–30] for those CGRAs use a modulo-scheduling-based approach to schedule the operations inside the data flow graph (DFG) of the loop kernel. Modulo scheduling can efficiently deploy a large DFG onto CGRA with a limited number of PEs and fully explore the instruction-level and loop-level parallelism. However, this scheduling scheme needs CGRAs to provide enough bandwidth for feeding the context (context is also referred as configuration or instruction) to PEs with low latency, and enough memory space to buffer the context of every control step for every PE. Therefore, the global context memory is typically implemented as on-chip SRAM thousands of bits wide and several hundred bits deep, thus occupying a significant amount of chip area and power consumption. For example, in Figure 2, HReA [18] uses GCM 1024 bits wide and 128 lines deep feeding the context to PEs and it takes 42% of the entire chip area and 38% of the chip power consumption.

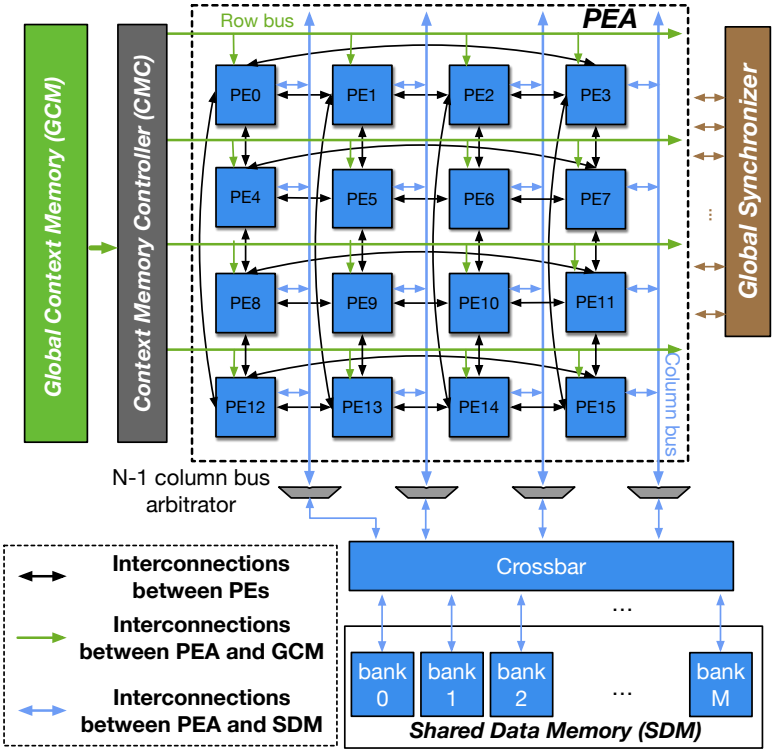


Figure 1. A 4 × 4 CGRA consists of processing element array (PEA), global context memory (GCM) and shared data memory (SDM).

To reduce context-fetching overhead and context-memory footprint, most existing context-reduction frameworks [31–36] rely on the statistical analysis over the context bitstream in the pre-silicon phase, and their compressed context is encoded after the

original context-generation phase. These approaches can be classified as post-context-generation method [37]. However, the post-context generation can barely explore some inherent redundancies due to the correlation of operations and data dependency, which can be captured by the scheduling algorithm in the pre-context-generation phase.

In this work, we propose an architecture/compiler co-designed framework called similarity-aware context reduction to reduce the area and power overhead of the context memory and context-fetching inside CGRA with negligible performance degradation and hardware overhead. Our framework leverages the CGRA compilation flow in the pre-context-generation phase for context reduction and does not rely on statistical analysis over the loop kernels set in the pre-silicon phase. This makes our work able to be adaptive to diverse loop kernels that cover more application domains. The contributions of this paper can be summarized as follows:

1. We provide a context-fetching primitive (CFP)-based fetching scheme by partitioning the context to subsections with appropriate granularity and only fetch the CFP that contains the corresponding different subsections relative to the former context whenever fetching the new context.

2. We explore the hardware design space by providing centralized and distributed CFP-fetching CGRA to support the new context-fetching scheme.

3. For the purpose of reducing context-fetching latency and context-memory footprint, we develop a simulated annealing-based similarity-aware (SA) tuning algorithm which is orthogonal to existing CGRA modulo scheduling and data-placement optimization algorithms. The SA tuning algorithm is integrated into the CGRA compilation flow with a state-of-the-art modulo-scheduling algorithm [38] and data-placement algorithm [39], and realize the goal by orchestrating the spatial mapping of the operations and encoding the inactive bits to improve the similarities between the consecutively scheduled context in each PE.

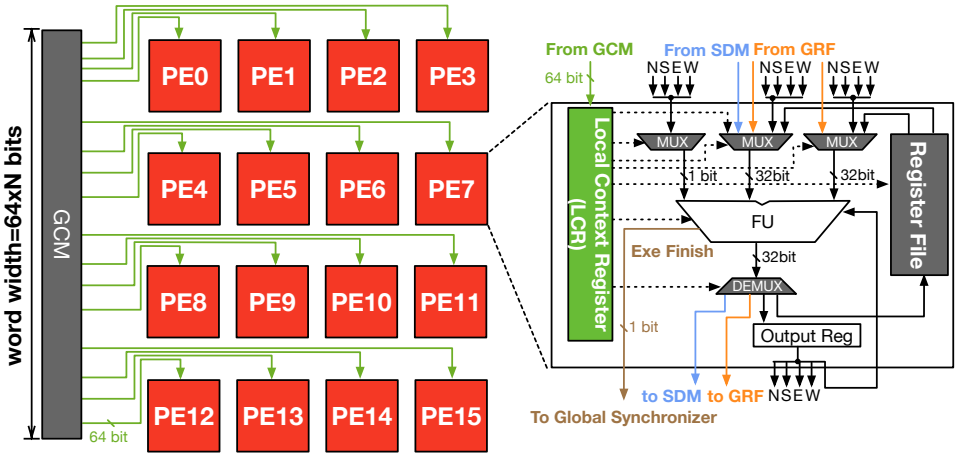


Figure 2. Organization of controlling CGRA.

Experimental results show the energy and area efficiency of our frameworks can reach 21% and 34% higher respectively than the one using the original context.

2. Background

2.1. Architecture Overview

Typically, the modulo-scheduled CGRAs [8,13,18,20] consist of PEA, GCM and SDM. Both GCM and SDM are implemented as on-chip SRAM and designed to have enough space to buffer all the context as well as the input, temporal and output data of at least one loop kernel. When CGRA is running a large application which contains multiple loop kernels and either GCM or SDM is not able to buffer the context or data of all the loop kernels, the host CPU needs to perform multiple transfers sending context or data from main memory to on-chip memories in CGRA. To reduce the context-switching overhead

between kernels, some work [40] provides a prefetching technique allowing CPU to fetch the context and data of the next loop kernel from main memory to GCM and SDM while CGRA is executing the current loop kernel. The PEs inside PEA are connected through specific topology such as torus or mesh. This allows CGRA to efficiently transfer the data between the local buffers inside PEs without writing back to the remote global shared data memory.

The modulo-scheduled CGRA is implemented as a two-stage pipeline composed of context fetch and execute stage, and it adopts a centralized control scheme in which all PE execution and context-fetching occur in lockstep. The global synchronizer acquires the finish signal from PE to control the start of context-fetching and PE execution.

The control step (CS) is the minimal time period that guarantees all the PEs can finish executing one operation or fetching their new context. Figure 3 shows the example when PE0 and PE1 are executing 3 consecutive operations. In PE0, the fetch stage takes one cycle and execution stage takes three cycles. However, in PE1, the fetch stage takes 2 cycles and execution stage takes one cycle. One observation from this example is that the cycle number within each control step depends on the stage who takes the longest time of all the PEs. In this example, the longest latency happens during the execution stage for operations in PE0 (3 cycles). This situation may happen when PE0 is performing the memory access operation and conflicts with the other concurrent memory access operations performed by the other PEs. In this example, PE1 takes 2 cycles to fetch its context. Even though this is not the case in existing modulo-scheduled CGRAs which provide wide GCM for single cycle context-fetching, it may happen in our framework of this paper. This is one of the motivations for which we propose the similarity-aware tuning algorithm in the compilation flow.

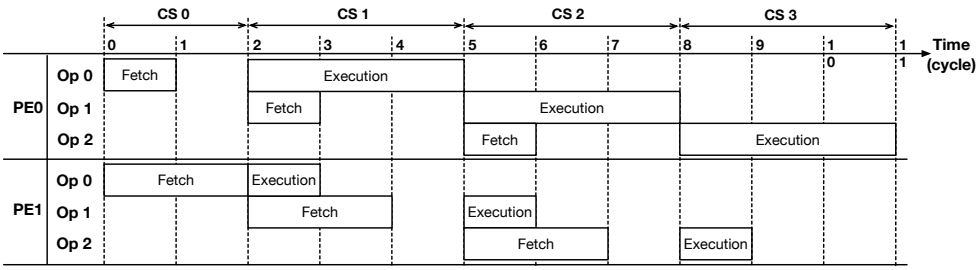


Figure 3. Pipeline and synchronization example. CS is the control step; within each control step, each PE can perform at most one operation or fetch at most one new context.

2.2. Modulo-Scheduling Mapping Scheme

Modulo scheduling is a promising way of deploying loops onto CGRAs [21,24–27,30,41]. Previous works have formulated the CGRA modulo-scheduling problem as mapping the data flow graph of a loop body onto the modulo time-extended abstracted hardware resource graph [24,42] and this mapping problem has been proved to be NP-Complete [23,24]

In 2002, researchers began to explore the efficient heuristics [21–30], and their main goals are to generate the schedule that minimizes the initiation interval (II) between consecutive iterations of the loop and less compilation time. These algorithms may use different mapping strategies by leveraging the different properties of CGRA. For example, modulo schedulers such as DRESC [21], Recurrence-aware [43] and Bimodal scheduler [44] tried to manipulate the routing cost functions to tackle the different routing properties between different function units inside heterogeneous CGRAs. The cost function inside the Bimodal scheduler also considers the routing between the constant memory and function units. However, EPIMap [24], REGIMap [25], RAMP [26] and [38] mainly target homogeneous CGRAs. In this paper, our similarity-aware tuning algorithm cooperates with the state-of-the-art modulo-scheduling algorithm [38], which will be illustrated in Section 5.

2.3. Multi-Bank Shared Data Memory Access Conflict

Even though modulo scheduling can exploit massive parallelism for loops, it also introduces large amounts of parallel memory access operations between SDM and PEA. Multi-banked SDM is a promising solution for providing a good fulfillment for parallel memory accessing [45]. The blue arrows in Figure 1 plots the network between SDM and PEA. Each PE can access any memory bank. To reduce the interconnection overhead and keep the scalability of PEA, existing CGRAs [46–48] use full crossbar fabric to connect SDM banks to any column bus, instead of connecting SDM banks to any PE. PEs in the same column are connected to the column bus. Memory access conflicts may take place in two situations:

Bank conflict: When PEs of different column buses access the same SDM bank at the same time, the arbitrator allows only one PE to access SDM at each cycle.

Column bus conflict: When PEs of the same column access SDM at the same time, they must access sequentially.

It should be noticed that the column bus conflict is not a general case for all the existing CGRAs. In some heterogeneous CGRA such as ADRES [8] and SRP [10], their PEs which can access the SDM have a dedicated data path to SDM. This is because in heterogeneous CGRA, not all the PEs are able to access the SDM, which can help control the interconnection overhead between PEA and SDM. However, the scope we discuss in this paper mainly focuses on homogeneous CGRA, inside which every PE can access SDM.

Many optimization techniques are proposed to optimize the memory-accessing conflict problem. For example, approaches such as memory partitioning [49], multi-bank mapping [47], conflict-free mapping [45], and data-placement optimization [39] leverage carefully designed memory-partitioning schemes and data-placement strategies to guarantee the concurrent accessed data locate in different memory banks. CASCADE [50] decouples data access from PE by developing a stream engine for optimizing the memory-accessing conflict issue. However, the memory-accessing conflict is still an inevitable source that causes pipeline stall. First, due to the limited interconnection budget and scalability considerations [47,48], the hardware cannot provide enough connectivity between SDM and PEA by, for example, using a column bus in the aforementioned CGRAs. For some data-intensive applications, the number of the parallel memory access operations can easily exceed the bandwidth of the on-chip SDM. Second, existing memory access conflict optimizations are not able to capture the dynamic memory access behavior such as the indirect memory access. Therefore, it is non-trivial to completely eliminate the memory conflicts between the parallel memory access.

2.4. The Context-Reduction Framework Overview

Figure 4 shows the overview of the hardware/software co-designed context-reduction framework. The similarity-aware compilation flow first adopts the modulo scheduling [38] (MS) to optimize the II in terms of control step and data-placement algorithm [39] (DP) to reduce the execution cycles within the control step. The following similarity-aware tuning algorithm is used to improve the similarity between the context of consecutive scheduled operations of each PE for the purpose of reducing the CFP context-fetching latency and reducing the context footprint. The CFP context generation finally compresses and generates the CFP-based context and feeds it to CFP context-fetching CGRA. Section 4 will introduce the CFP-based context-fetching scheme and two micro-architecture design styles to support this fetching scheme. Section 5 will introduce the similarity-aware tuning algorithm. Table 1 shows the commonly used acronyms and their corresponding meaning in this paper.

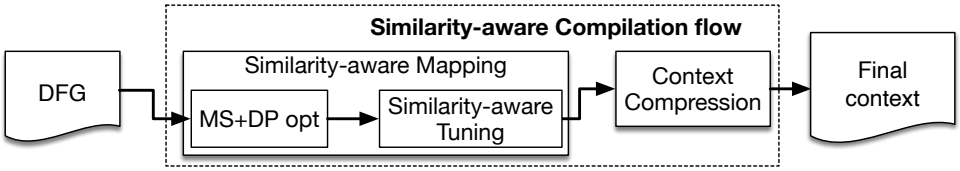


Figure 4. The overview of the context-reduction framework.

Table 1. Commonly used acronyms list.

List of the Commonly Used Acronyms in the Text	
GCM: Global Context Memory	SDM: Shared Data Memory
CFP: Context-Fetching Primitive	PEA: Processing Element Array
GCFP: Global context-etching primitive	LCFP: Local Context-Fetching Primitive

3. Related Works

Existing context-reduction works for modulo-scheduled CGRAs mainly focus on reducing the context-storage and context-fetching overhead. Typically, they adopt different high-level methodologies to tackle these issues.

Execution Model-level optimization—Some works borrow from the classic instruction-fetching overhead-reduction techniques by enabling the CGRA to support SIMD execution model. For example, in Morphosys [9], PEs in the same row work in SIMD mode, and the single context is broadcast to the entire row which helps reduce the context storage. Research such as [51,52] use context sharing and pipelining technique, only the PEs in the first column fetch the context from context memory and the context is later shared by the other PEs by forwarding the context from ith column to the $i+1$th column. [53] further reduce the context-storage overhead by support context broadcasting to the PEs in the same column by setting the CGRA to half/full mode. However, it only works better for the applications that match this execution model. Therefore, these classes of approaches are facing challenges on supporting loop kernels that cover diverse applications.

Hardware decompression level optimization—A straightforward method for context reduction is to invent a compression format and compress the context, and then use the hardware decompressor to dynamically decompress the context. The most commonly used one is the dictionary-based compression techniques which puts frequently used bitstream patterns in a dictionary and uses a dictionary index to encode them. During decompression, they are fetched from a dictionary according to the index and decompressed to the original context. Therefore, most optimizations focus on reducing the dictionary footprint and reducing the overhead of the decompressor. Aslam et al. [31] uses state-of-the-art dictionary methods and reorganized the PEs to improve the compression in the dictionary. The approach taken by Chung et al. [32,33] exploits the spatial and temporal redundancy from the configuration stream and saves the most frequently occurring values in a dictionary. The latency of their decompressor is two cycles and can be pipelined, but the overhead of the decompressor is not shown in their papers. The dictionary-based approach usually suffers from the trade-off between the dictionary size and decompression speed. It is still a challenge to design a dictionary that does not consume too much additional energy and allow for good compression, while at the same time, the decompression logic does not reduce the circuit frequency or has low energy or area overhead.

Another class of compression techniques focus on compressing the inactive bit sections inside the context. The dynamic compression [34] redesigns the context-memory architecture so that it can dynamically fetch the active section of the context. The context compression techniques designed for SRP [35,36] partition the entities inside the long instructions to different groups based on the statistical analysis over the kernels of the target application domain in the pre-silicon phase. Then, they use the ASAP-ALAP encoding approach to encode the inactive bits within each partition to minimize the redundancy.

Typically, this class of approaches first needs to decide the application domains that their CGRAs target and perform a series of algorithms to analyze the kernels of the applications and search for the best pattern to compress the context. However, this makes these pre-silicon approaches kernel-sensitive, and the efficiency of these approaches of running the applications out of their analyzed dataset is still questionable.

The approaches mentioned above either belong to the *post-context-generation* approach or are based on analyzing the statistical characteristics of the context bitstream over a set of loop kernels during the pre-silicon phase. However, our work uses the *pre-context-generation* approach and is not based on statistical analysis. REMUS [37] also uses the *pre-context-generation* method; however, their CGRA is not executed in a modulo-scheduled manner; therefore, the compilation problem is different with our work.

4. Architecture Support for Context Compression Frameworks

4.1. Context Architecture Analysis

CCF [20] applies 32-bit context architecture (shown in Figure 5). It uses 1 bit for instruction format, 5 bits for opcode, 5 bits for each data input (left and right) and one predication input, and 3 bits for output. Within the 5 bits, 3 bits are for controlling the MUX to get data from at most 8 different sources and 2 bits are for indexing local register files with at most 4 registers. For the 3-bit output section, it uses 1 bit to register file writing and 2 bits to register indexing. The Morphosys [9] context is similar to CCF except that it does not support prediction; the prediction input section is replaced with the 5-bit shift control section.

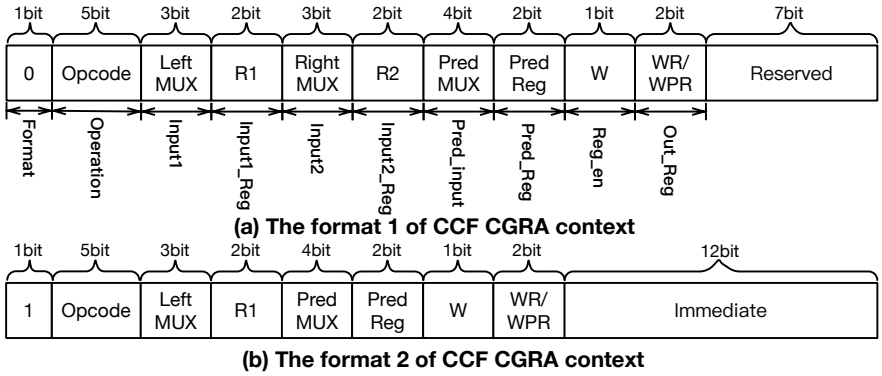


Figure 5. Two context formats of CCF [20].

HReA [18], which targets 13-Dwarfs [54] adopts 64 bits to enhance the PE control. Specifically, there are three multiplexers, two (input 1 and input 2) for selecting data and the third one (input 3) for selecting a 1-bit prediction signal that enables CGRA to support loops with if/else branches. HReA provides five different kinds of data buffers, including local register file, global register file, output register in PE, on-chip data memory and constant. This means the multiplexer should choose data from 5 different types of buffers, which needs 3 bits to encode them. Moreover, within each buffer, there is an additional bit field to specify the index of the data inside the buffer. For example, the size of a global or local register file is from 2 to 16 data words, which means it needs 4 bits to index the data placed in the register file. As HReA uses 16KB on-chip SRAM, it uses 12 bits for direct SDM accessing and 32 bits for constant.

Even though the 64-bit context is better than the 32-bit context in terms of generality and scalability, it brings more redundancy. For example, more inactive bits appear in the 64-bit context whereas the 32-bit context solves this by reusing the bit-width for different formats. This motivates us to propose a new context-fetching scheme to reduce this redundancy.

Figure 6 shows the average toggling rate of the context of 4×4 CGRA running over 19 kernels. The data are calculated by $Avg(A_0, A_1, \ldots, A_{N-1})$, where A_i is the average

toggling rate of PE_i. The average toggling rate of running all the kernels is 11.7%, which means 8 out of 64 context bits are different from the previous context on average, and other bits are either inactive or remain the same for the new context. Thus, fetching the whole context during context-switching causes the waste of both memory footprint and energy consumption when fetching the context. An intuitive idea to tackle this problem is to completely fetch the first context, and partially fetch the following context by updating the different parts relative to the previous context.

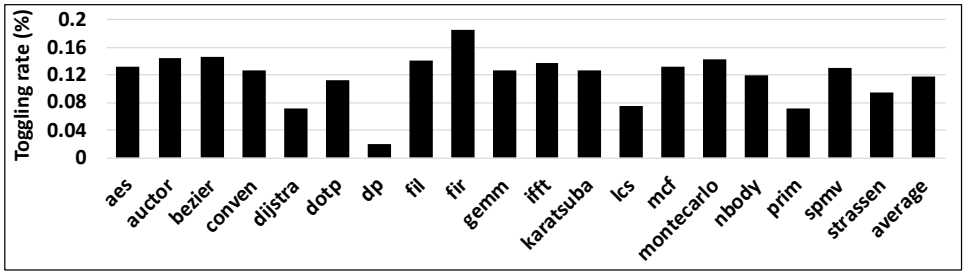
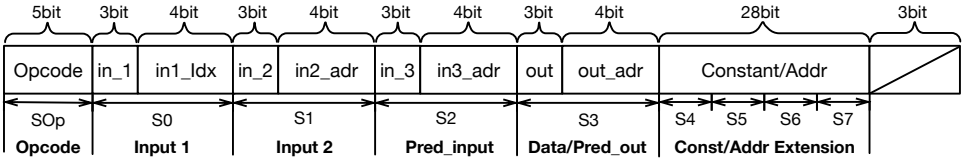
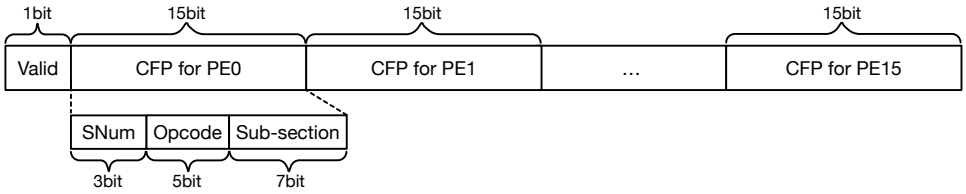


Figure 6. The average toggling rate of 4x4 CGRA running over 19 applications.

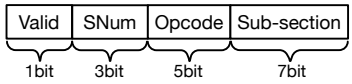
Typically, the changed bits are mainly distributed in sections that control the opcode, input and output sources, constant or direct memory-addressing section. Thus, the original context can be partitioned into 9 subsections which include one opcode subsection (SOp), three input subsections ($S0$, $S1$ and $S2$), one output subsection ($S3$) and four subsections for the extension of constant and direct memory addressing ($S4$ to $S7$). The partitioned context architecture is shown in Figure 7a. Except for the 5-bit opcode subsection, all the other subsections are set to 7 bits. Inside each input or output subsection, 3 bits indicates the buffer type (*in_1*, *in_2*, *in_3* and *out*) and 4 bits specifies the index of the corresponding buffer (*in1_adr*, *in2_adr*, *in3_adr* and *out_adr*).



(a) The PE context architecture after reorganization and partition



(b) The 15 bit CFP of N PEs and 1 bit valid signal formulates a GCFP for GCM in centralized CFP fetching CGRA



(c) The 16 bit LCFP of each PE in distributed CFP fetching CGRA

Figure 7. The reorganized context format of HReA [18] and two different CFP formats for two CGRA architectures proposed in this paper.

4.2. Context-Fetching Primitive (CFP)

We conceptually refer to the *context-fetching primitive* (CFP) as the atomic context-fetching unit that updates over the context of each PE, and fetching each CFP takes one cycle. The CFP is a 15-bit section that consists of three basic bit subsections (Figure 7b): the 5-bit opcode subsection, the 7-bit subsection of the new context which is different to the

corresponding subsection of the old context, and the 3-bit index of that subsection (*SNum*). Based on the newly organized and partitioned context architecture, the context-fetching scheme is that when PE is fetching a new context, it updates the old context by sequentially fetching the CFP or CFPs and updating the opcode subsection and the corresponding different 7-bit subsection or subsections over the old context. Fetching each CFP takes one cycle. If we package more 7-bit subsections into CFP (coarser granularity), there will be more redundancies and this will lead to a larger context footprint, even though it may take fewer cycles to update the new context. For example, when there is only one subsection of the new context that is different from the older one, the CFP still needs to fetch the other same subsections. Another reason is that the similarity-aware tuning algorithm will help improve the similarity between the consecutive contexts and reduce the context-fetching latency. We provide two architecture solutions to support the CFP-based context-fetching scheme which we refer to as the centralized and distributed CFP fetching CGRA.

4.3. Centralized CFP-Fetching CGRA

4.3.1. Global CFP (GCFP)

Figure 8 shows the architecture of the centralized CFP-fetching CGRA. It uses the same PE architecture with the original CGRA and is also implemented as a two-stage pipeline; one is a context-fetching stage which fetches the CFPs from GCM to PE to update the context, and another one is PE execution. The centralized CFP-fetching CGRA packages the 15-bit CFP of each PE and a 1-bit valid bit into a 241-bit global CFP (GCFP). During each cycle of the context-fetching process, the GCFP fetching unit fetches one GCFP from GCM. The *Valid* bit is sent to the GCFP fetching unit synchronizer indicating whether all the different subsections of the previous context have been updated to the new context. Among the PEs, if there is any PE whose new context has multiple different subsections with its old context, the GCFP fetching unit takes additional cycles to fetch the GCFPs for all the PEs, until all the PEs are updated to the new context.

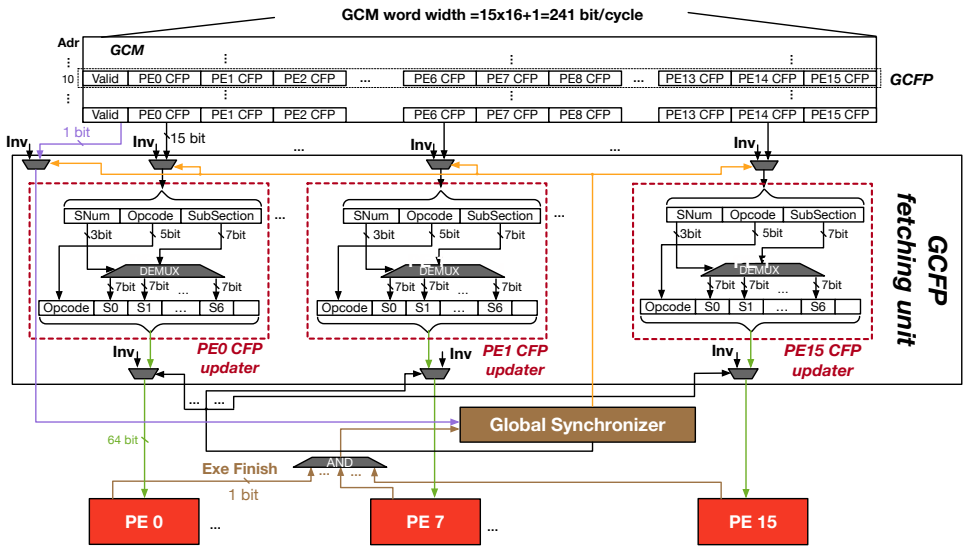


Figure 8. Architecture support for the centralized CFP-fetching CGRA.

Figure 9a shows one example of the original 64-bit context of two operations in PE0 and PE1. The subsections tagged with red circles are the different subsections between the context of two consecutively executed operations in each PE. Except for the opcode section, both PE0 and PE1 have one different 7-bit subsection (*S*0) between their two context words (*S*0) Therefore, only one GCFP is enough to update the context of the second operation for these two PEs. However, Figure 9b shows the second example of the original 64-bit context of two operations in PE0 and PE1. In this example, PE0 has one different 7-bit subsection but PE1 has two. In this case, two GCFPs are needed to update the context of the second

operation for these two PEs and Figure 9c is the example of the corresponding two GCFPs. According to Figure 9c, there are two sources of redundancies:

The first one is the redundant fetching of the CFP of PE0 (the CFP filled with blue in Figure 9c), the same CFP of PE0has to be fetched even though there is only one different subsection between contexts in PE0.

The second one is the Opcode subsection in the CFP of both PE0 and PE1 (specified with dash circles in Figure 9c). Therefore, the compiler technique is extremely important to guarantee the similarities between the switched contexts which will be introduced in Section 5. Improving the similarity between contexts can reduce context redundancy and context-fetching latency at the same time.

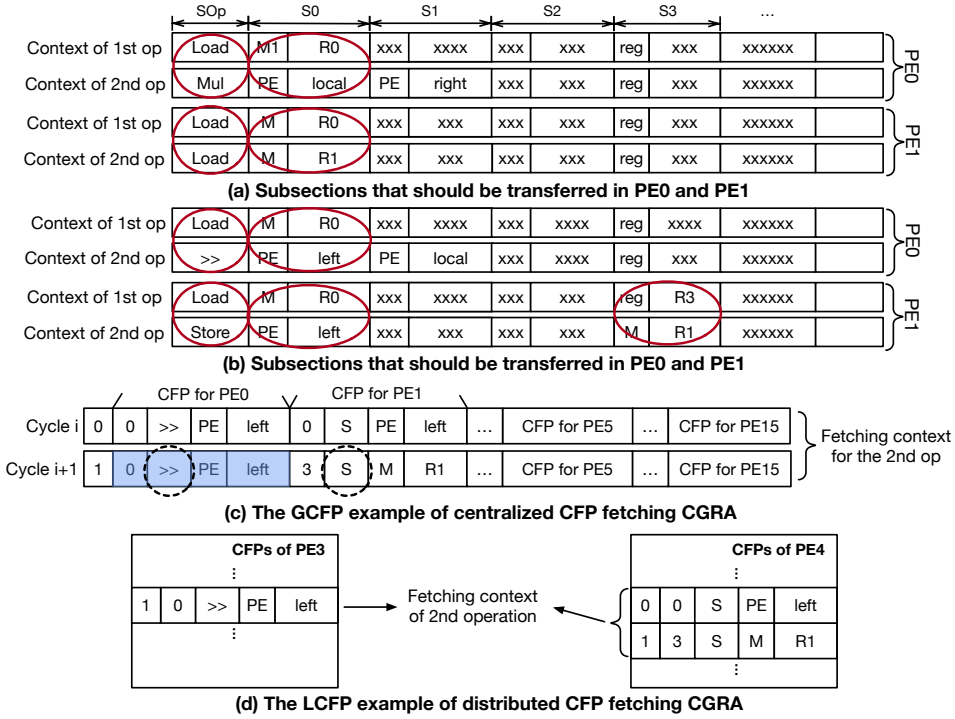


Figure 9. The compressed context-fetching example.

4.3.2. The GCFP Fetching Unit

The GCFP fetching unit controls the fetching of the GCFP from GCM to PEA. The architecture of the GCFP fetching unit for 4 × 4 CGRA is shown in Figure 8. The fetching unit consists of 16 CFP updaters for 16 PEs and 33 on/off switches. These switches act like a gate that controls the fetching of new GCFP from GCM and the dispatching of all the updated context to all the PEs. The on/off switch is controlled by the enable signal sent by the global synchronizer. The global synchronizer gets the 1-bit *Valid* signal from GCFP and the *Exe Finish* signals from all the PEs, and sends the enable signal to those switches according to the finite state machine (FSM) logic. The updater directly forwards the opcode of CFP to the context register, and uses demultiplexer to update the subsection in CFP to the corresponding subsection in the original context according to the 3-bit section number index in CFP. Each time the GCFP fetching unit receives the GCFP from GCM, the 16 updaters concurrently update the corresponding subsections of the previous contexts of each PE. If the valid bit is 0, it means current GCFP cannot update the context for all the PEs. When the valid bit is 1 and all the PEs finish computing the old operations, the global synchronizer sends the enable signal to all the on/off switches, which means the contexts of the current operation are ready to be executed, and the GCFP fetching unit is ready to fetch the GCFPs for the new context in GCM.

4.4. Distributed CFP-Fetching CGRA

Recall the example of using GCFP to update the context in Figure 9c, the CFP of PE0 is redundantly transferred in the second GCFP; also, this CFP is redundantly stored in memory. To further eliminate this redundancy, we use Local CFP to update the context of each PE in a distribution manner. Figure 10 shows the distributed CFP-fetching CGRA. In this CGRA framework, instead of using 1 bit inside the GCFP to control the fetching of CFPs of all the PEs, we package each CFP with 1-bit *Valid* signal to formulate a 16-bit Local CFP (LCFP) for each PE. The LCFP is stored in the compressed local context memory (CLCM) and placed closer to PE. During the CGRA execution phase, each PE fetches its own LCFP from CLCM and the local CFP updater updates the CFP over the previous context. In addition to the global synchronizer, each PE has a local synchronizer to synchronize the context-fetching and operation execution.

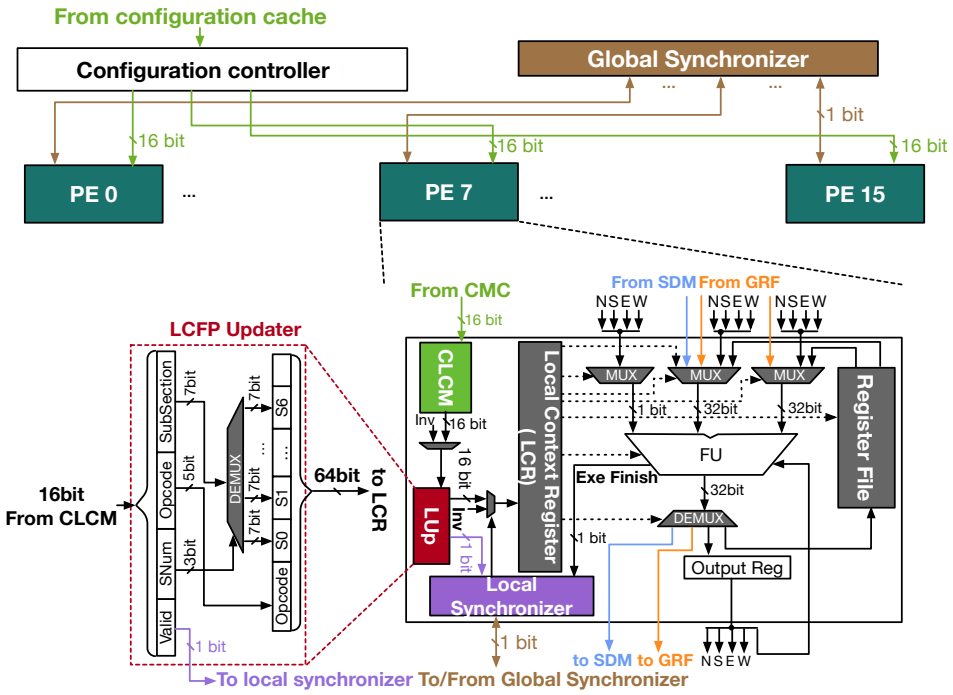


Figure 10. Architecture support for the distributed CFP-fetching CGRA.

The distributed CFP-fetching CGRA does not change the synchronous manner of the execution. However, it removes some context-footprint redundancies caused by the unbalanced fetching of the CFPs between different PEs. For the same example in Figure 9c, PE0 must repeatedly fetch the same CFP (blue color part) in cycle $i + 1$ to align with the second CFP of PE1 even though it has only one different subsection to update. This design style eliminates the redundancy by adding *Valid* into the CFP for every PE and using the local synchronizer to manage the LCFP fetching and PE execution locally inside each PE. In this way, each PE buffers its own LCFP according to the number of different subsections between its own contexts as in Figure 9d.

As this architecture, design style does not change the synchronization scheme; its context-fetching latency is the same with the centralized CFP-fetching CGRA. The efficiency of the distributed CFP-fetching CGRA still critically hinges upon the compilation flow that targets mapping operations onto each PE with high similarities between contexts.

5. Similarity-Aware Compilation Flow

The similarity-aware compilation flow consists of 4 algorithms. It first leverages the modulo-scheduling algorithm in [38] to temporarily and spatially schedule the operations inside DFG to minimize the initiation interval in terms of control step. Then, it applies

the data-placement algorithm [39] to organize the input and output data on multi-banked SDM to reduce the memory access conflict and adjust the scheduling to reduce the column bus access conflict which aims at reducing the PE execution cycles within each control step. The context generated from these two algorithms may suffer from large context differences between the consecutive scheduled operations. This may cause the context-fetching latency to exceed the PE execution when using the CFP-fetching CGRA to execute the corresponding CFP context and degrade the performance of the CFP-fetching CGRA relative to the original CGRA. Furthermore, it also enlarges the footprint of the CFP context since more CFPs must be stored and transferred during execution to update the context. The goal of the similarity-aware algorithm is to prevent this issue. The algorithm gets the scheduling results of the former algorithms as the input and improves the similarity of the context of the consecutive operations of every PE to reduce the fetching overhead in CFP context-fetching CGRA. Finally, the original context is converted to the CFP context in the CFP context-generation pass. We focus on introducing the similarity-aware tuning and CFP context generation in this section and the former two algorithms can be referred from corresponding papers [38,39].

5.1. Similarity-Aware Tuning

5.1.1. Similarity-Tuning Example

Figure 11c is a dataflow graph (DFG) example of a loop body; each node inside the DFG represents an operation. These operations are mapped onto 3 × 2 CGRA in Figure 11d. There are three column data buses which are shared by (PE0, PE3), (PE1, PE4) and (PE2, PE5). In this example, we assume that there is no bank conflict among the parallel memory accessing of PEs in different column buses, but the memory access of PEs in the same column bus should be performed sequentially. For example, inside control step 0 (CS0) in Figure 11a, PE0 and PE3 are accessing SDM at the same time (L0 in PE0 and L4 in PE3). But in each cycle, the column bus gives access to only one memory-accessing operation. Therefore, L1 takes one cycle but L4 takes 2. Under the modulo-scheduling scheme, $II = 2$ in terms of control step, each operation in PE will be assigned with a unique control step (CS); The modulo schedule time of operation v can be calculated as $m(v) = CS(v) \bmod II$, where $CS(v)$ is the scheduled control step of operation v. In example Figure 11c, operation"L1","L", "+", "L4" and "S" belong to modulo time 0 (M0) and other operations including "×", "L3" and "≫" belong to M1. Operations scheduled in the same modulo time indicate these operations from the same or different iterations are executed in parallel. Figure 11a shows the execution scheme of our CFP-based context-fetching CGRA under the compilation flow without the similarity-aware tuning algorithm. Figure 11e shows the CFP-fetching latency of each PE under the scheduling of Figure 11a. The weight of the edge from node v to v' represents the number of cycles to fetch the CFP for updating the context of v'. In CS1, PE4 takes 2 cycles to fetch the CFPs to update the context for "S_{i-1}", whereas all the other PEs also take one cycle to finish updating the context. We assume the latency of all the operations are one cycle. In this way, fetching the context of PE4 will become a bottleneck in the pipeline stage, which needs two cycles, because in CS1, the execution of all the PEs only takes one cycle.

If we use the similarity-aware tuning algorithm to improve the similarity between contexts in PEs, i.e., in modulo time 0 (M0), we swap "L2" in PE1 with "*nop*" in PE2, swap "≫" in PE3 with "L3" in PE4, and swap "S" in PE4 with "*nop*" in PE1. In this way, all the PEs take only one cycle to fetch the new context and the II is reduced from 4 to 3 cycles. It should be noticed that the number of cycles within M0 is not reduced to 1 cycle because it is bounded not by the context-fetching latency but by the memory access conflict between PE0 and PE3.

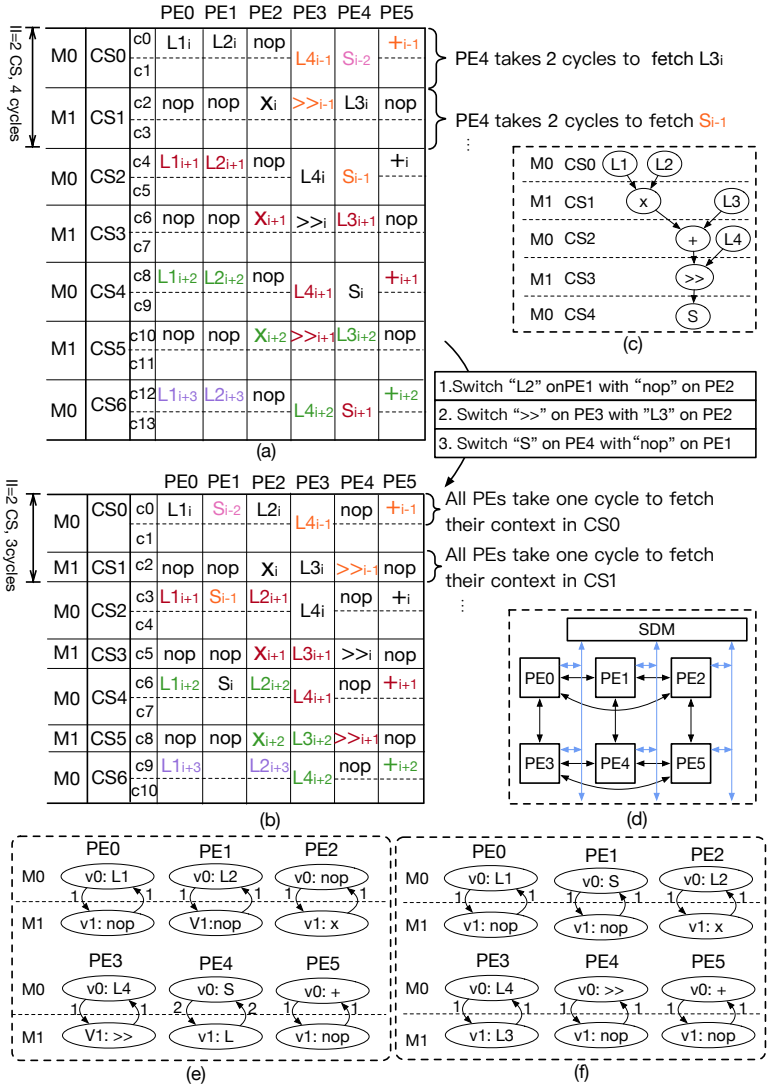


Figure 11. (**a**) The execution scheme of our CGRA under the compilation flow without similarity-aware tuning, (**b**) the execution scheme of our CGRA under the compilation flow with similarity-aware tuning, (**c**) Example DFG, (**d**) the example of our 3 × 2 CGRA, (**e**) context-fetching latency of each PE in every modulo time under the compilation flow without SA tuning, (**f**) context-fetching latency of each PE in every modulo time under the compilation flow with SA tuning.

5.1.2. Similarity-Aware Tuning Algorithm

We use the simulated annealing heuristic to tune the mapping of the operations. The implementation of the similarity-aware tuning algorithm is shown in Algorithm 1. The input of the algorithm is the scheduling $M(D, TEC)$ generated from the modulo scheduling and data-placement algorithms, where every node inside D is mapped to a node in time-extended CGRA (TEC) [24]. The TEC is a resource graph extended in a temporal dimension and models the computational and routing resources of CGRA in temporal and spatial dimensions; each node in TEC represents the PE in a specific modulo time slot. Execution information is the execution latency considering memory conflict within each control step under the original mapping scheme (before similarity-aware tuning). If a PE slot does not have valid operation mapped, we assume there is a *nop* operation mapped to that PE slot.

The algorithm starts by initializing the $Cost_{sum}$ according to the original mapping scheme (line 1). Our algorithm randomly selects a pair of different PEs in the same modulo time and swaps their mapped operations with each other (line 5). The *is_valid_mapping* function checks the success of the swapping and returns true if the routing constraint is still satisfied. For example, in the mapping example of Figure 11d, if we swap the mapping of

operation "L1" and "+", i.e., "L1" is mapped to PE5 and "+" is mapped to PE0. This swap will not succeed because when "+" is mapped to PE0, it cannot get its input "L3" from PE4, because there is no interconnection between PE0 and PE4.

Algorithm 1: Similarity-aware Tuning

Require: MS Mapping: $M(D, TEC)$, Execution information: E, II
Ensure: Tuned Mapping: $M_T(D, TEC)$

```
1:  Cost_sum ← InitializeCost(E, II, M, TEC);
2:  InitializeTemprature();
3:  while True do
4:    for iter = 1 to total_iter do
5:      M_T ← RandomSwitchMappingPairs(M);
6:      if is_valid_mapping(M_T) then
7:        iter ← iter + 1;
8:        enc_context ← ContextEncoding(M_T);
9:        Cost'_sum ← EvaluateCost(enc_context);
10:       accepted ← EvaluateAccept(ΔCost_sum, tmp);
11:       if accepted then
12:         Cost_sum ← Cost'_sum;
13:         M_T ← UpdateMapping();
14:       else
15:         SwitchBack();
16:       end if
17:     end if
18:   end for
19:   if runtime ≤ time_budget or Cost_sum converge then
20:     return M_T;
21:   else
22:     tmp ←UpdateTemprature();
23:   end if
24: end while
```

Typically, there are two factors that will influence the successful rate of the swapping process. First, the swapping between two valid operations is more difficult than swapping one valid operation with *nop*. This is because for two valid operations, both two operations must still satisfy the routing constraints after swapping. However, if there is only one operation is valid, it just needs to consider the routing constraints of this single operation. Second, given more interconnection abilities between PEs, the swap process may be more likely to succeed. For example, in Figure 11b, if there is interconnection between PE0 and PE4, swapping "L1" with "+" will succeed.

If the swap is successful, the simulated annealing strategy is used to decide whether we accept the new swapping by evaluating the cost function according to Equation (6) and the temperature (line 8, 9, 10). The algorithm first uses the *ContextEncoding* function to encode the inactive sections of the original context under the tuned scheduling M_T to improve the similarity between the context of consecutive scheduled operations. The context-encoding example is introduced in Section 5.2. The *EvaluateCost* function tries to evaluate the cost of CFP context-fetching overhead and context-footprint overhead under the tuned scheduling M_T and its corresponding encoded context *enc_context*. Even if the new cost is bigger, there is still a chance to accept the swap according to the *temperature*. This helps to escape from local minima. The temperature is gradually decreased from a high value (line 21). When the swapped mapping pair are accepted, both $Cost_{sum}$ and M_T are updated (line 11 and 12). When the algorithm stops either $Cost_{sum}$ is reduced and cannot be further improved, or the running time of the algorithm reaches the time budget and returns the sub-optimal solution (line 18).

5.1.3. Cost Function Set Up

The cost function considers three factors: context-fetching latency, memory-accessing conflict latency and context footprint. Under the modulo-scheduling scheme, each operation will be iteratively executed during every II control steps, and when there are multiple operations in the PE, these operations are alternately and iteratively executed (for example operation a, b, c, a, b, c and so on). Due to the synchronization scheme, the cycles of fetching the context over all PEs at modulo time m can be calculated by Equation (1):

$$F_m = \max(f_m^0, f_m^1, \ldots, f_m^{N-1}) \tag{1}$$

where $0 \leq m < II$, N is the total number of PEs and f_m^i is the cycles of PE_i to fetch the CFPs for the context of operation that is scheduled at $m+1$. Another factor that the similarity-aware tuning algorithm considers is the execution latency caused by the conflict of memory access. As in the pre-tuning phase, the data-placement optimizations [39] have been used to conquer the memory bank conflicts. Our similarity-aware tuning algorithm only considers the conflicts when PEs are in the same column bus access memory at the same time. The way of calculating the execution cycles within modulo time m is according to Equation (2)

$$E_m = \max(e_m^0, e_m^1, \ldots, e_m^{N-1}) \tag{2}$$

where e_m^i is the cycles of PE_i to execute the operation that is scheduled in modulo time m; the compiler can analyze the execution time of every PE in modulo time m under a specific DFG scheduling.

Thus, the performance cost of modulo time m under the j^{th} swap can be calculated by Equation (3)

$$P_m^j = ReLU(F_m^j - E_m^{j-1}) + ReLU(E_m^j - E_m^{j-1}) \tag{3}$$

where the $ReLU$ function can be defined as: $ReLU(x) = 0$ for $x < 0$ and $ReLU(x) = x$ for $x \geq 0$, and F_m^j and E_m^j are the context-fetching and execution latency after the jth swap at modulo time m, respectively. The intuition behind this cost function is that there will be a cost whenever the swap leads to the increase of either context-fetching or PE execution.

In addition, for the purpose of reducing the context-memory footprint, the similarity-aware tuning algorithm set the context-footprint cost by calculating the total context size under tuned scheduling M_T. The memory footprint cost of centralized (FP_C) and distributed (FP_D) CFP-fetching CGRA can be calculated by Equations (4) and (5) respectively.

$$FP_C = \sum_{m=0}^{II-1} F_m \times (N_{pe} \times 15 + 1) \tag{4}$$

$$FP_D = \sum_{m=0}^{II-1} \sum_{n=0}^{n=N_{pe}-1} f_m^n \times 16; \tag{5}$$

In summary, the cost function which considers both performance and memory footprint during similarity-aware tuning is as per Equation (6)

$$Cost_{sum} = \alpha \times (\sum_{i=0}^{II-1} P_i) + \beta \times (FP_C \; or \; FP_D) \tag{6}$$

where α and β are two parameters to control the performance and context-footprint cost. We set α larger than β. This is because the context-footprint cost is mainly influenced by the sum of the context-fetching latency whereas the performance cost is decided by the context-fetching latency of every modulo time. The context-footprint cost can be reduced when performance cost is reduced, but not vice versa. Therefore, our strategy is to first target finding the tuned schedules with the minimal performance cost and select the one with minimal context-footprint cost.

5.1.4. Inactive Bit Encoding

The inactive bits inside the context can be encoded the same as the corresponding active bits of the previously or subsequently scheduled context to keep the high similarity between contexts.

In our context-fetching scheme, we encode these inactive bits at the granularity of the subsection. If all the bits inside the subsection are do-not-care bits, it is an inactive subsection; otherwise, it is an active one. When encoding each inactive subsection of a context, we first check the corresponding active subsections of the closest previously and subsequently scheduled context. If the two active subsections are the same, the inactive subsections are encoded the same as the active subsection; otherwise, the inactive subsection is encoded to the active subsection the same as the previously or subsequently scheduled context, which is a binary decision. If the compiler is not able to find the active subsection (such as *S*2 in Figure 12a), the inactive subsection will be encoded as a fixed bit stream (for example *Z* in Figure 12b). We then make binary decisions for all such inactive subsections and calculate the performance cost of the encoded context under such a combination. Therefore, the encoding problem is to find a combination of the encoding for all the aforementioned binary decisions so that the sum of the performance cost $\sum_{i=0}^{II-1} P_i$ is minimum.

For example, Figure 12a shows the active subsections of the context. We assume the II is 4 and there are 4 operations scheduled to the same PE. Each capital letter represents a specific encoding for that subsection; each different letter means their subsections are different. In Figure 12b, most of the inactive subsections are encoded to be the same as their corresponding upward or downward active subsections. The subsection *S*2 of all the operations are encoded to Z, which is a fixed bit stream. Therefore, we need to encode the three subsections of *S*1 and *S*7. We calculate the performance cost for all the combinations of encoding the inactive subsections and select the one with minimum performance cost which is shown in Figure 12c.

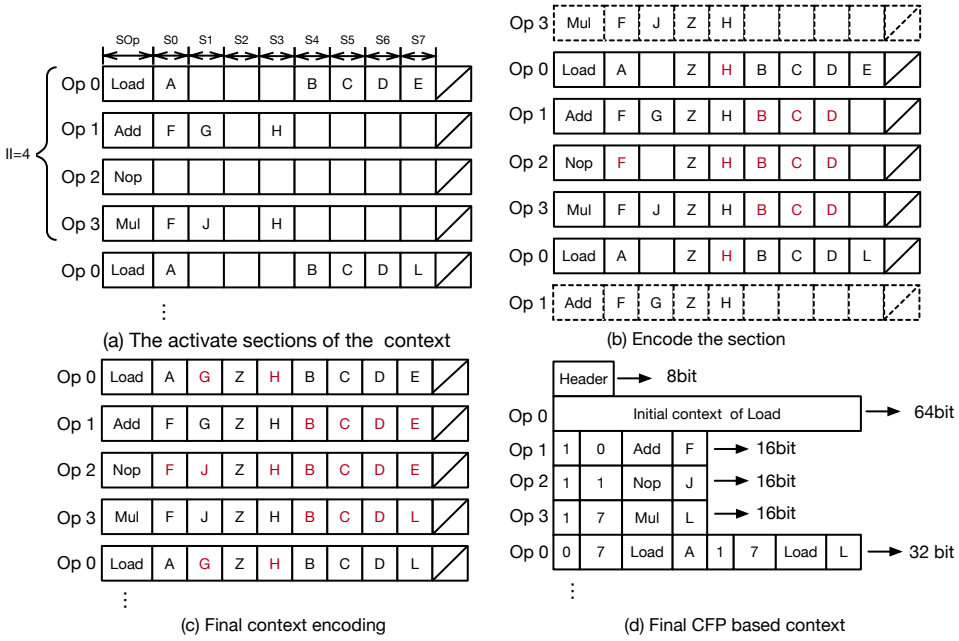


Figure 12. Inactive bit encoding and CFP-based context-generation example.

5.2. CFP Context Generation

After tuning, the compiler will generate the encoded original context for each PE due to the tuning result. The encoded context will finally be compressed to the format of GCFP or LCFP-based context according to specific context compression CGRA framework. Figure 12d shows the example of compressing the context of Figure 12c. In this example, we assume the CGRA has only one PE. Therefore, the format of GCFP and LCFP are the

same. We use an 8-bit header to specify the length of the CFP-based context and then follow the complete context of the first context to be the initial context; the initial context is used to initialize the local context register of each PE before the CGRA execution. Then, the CFP of each operation is consecutively stored in the following address.

6. Evaluation

6.1. Evaluation Methodology

Architecture configuration and application mapping—Our context-reduction method is evaluated on a general-purpose CGRA processor HReA [18], which contains four 4 × 4 PEAs. Each PEA consists of a 16 KB SDM with 16 banks, a 16 KB GCM with 1024-bit word width and 128 depth, and a 4 × 4 PE array connected with torus network. To map the application onto CGRA, we apply the single-configuration, multiple data (SCMD) computational mode [55] among four PEAs. The compiler generates the configuration or context for one 4 × 4 PEA using our compilation flow and apply the same schedule to four PEAs, and these PEAs accelerate different data sets. We implement three 4 × 4 CGRA frameworks in RTL Verilog and map it to the 65 nm process; the PEA is synthesized with Synopsys Design Compiler, and the SDM and GCM are generated by Memory Compiler. Synopsys PrimeTime PX is used to measure the power consumption under 250 MHz, 1.1 V. Table 2 summarizes the unique features of these three CGRA frameworks. Within each CGRA framework, we implement 2 interconnections between PEs. One is a torus network and another one a torus plus diagonal network where each PE can additionally access its adjacent diagonal for PEs which is adopted by HReA [18]. The CGRA framework A uses the HReA [18] as the original CGRA prototype, and the state-of-the-art modulo-scheduling algorithm [38] and the data-placement algorithm [39]. We use this original context architecture and compilation framework as the **baseline** of our evaluation. The CGRA framework B and C respectively use the centralized and distributed CFP-fetching CGRA as the hardware substrate and the aforementioned modulo scheduling, data-placement algorithm and similarity-aware tuning algorithm in this paper.

Table 2. Different configurations of CGRA architecture and compilation frameworks.

CGRA AF		IN between PEs	Compilation Flow
A: Original CGRA (HReA, Baseline)	A1	torus	MS + DP
	A2	torus + diagonal	
B: Centralized CFP fetching CGRA	B1	torus	MS + DP + SA
	B2	torus + diagonal	
C: Distributed CFP fetching CGRA	C1	torus	MS + DP + SA
	C2	torus + diagonal	

Compilation and simulation—For the compilation flow, we adopt the modulo-scheduling algorithm in [38] and the data-placement algorithm according to [39]. We develop the similarity-aware tuning algorithm and CFP context generation passes into the compilation flow. For loops with single-level or multilevel conditional branches, the loop kernel program is represented using select instructions (in C) to generate the control-independent DFG. Both if/else branches are executed and use *sel* operation to select the right path according to the 1-bit predict signal generated during execution. All the aforementioned algorithms are integrated into the *opt* driver as the separated back end passes in LLVM 8.0.0 [56]. For some imperfect loops, we adopt the loop fission or loop fusion to convert them to perfect nested loops. We use loop flattening in [57] to transform multilevel nested perfect loops to single-level nested loops. Finally, for loops with small loop body, loop unrolling is used to improve the use rate of the array.

We use a C++-based event-driven cycle-level accurate simulator to calculate the runtime of each CGRA framework. To guarantee the correctness of the CGRA simulator, CPU emulations over all the kernels are also performed and the results are compared with the simulation result. The simulator gets the CGRA context generated from the

compiler as the input; the runtime of the simulator considers the context-fetching latency, the on-chip memory-accessing conflict latency, and PE execution latency. We think using a C++-based simulator to evaluate the runtime is more efficient than an RTL-based simulation in terms of the simulation speed, especially when we run more than 30 applications over 6 different CGRA configurations. We measure the running time of our similarity-aware tuning algorithm on a Quad Core Intel Core i5 machine with CPU frequency of 3.3 GHz and 8 GB DDR3 main memory.

Benchmark selection—We select as many computationally intensive loop kernels as possible which cover application domains from Berkeley 13 dwarfs [54]. These loop kernels are selected from existing benchmark suites such as EEMBC, MediaBench [58], MiBench [59], MachSuite [60] and PolyBench [61]. Other loop kernels are extracted from applications such as graph searching (Graph), dynamic programming (DP), digital signal processing (DSP) and computer vision (CV). Table 3 shows the features of the DFG of these kernels, which include the number of nodes, edges, load/store operations, 3-input operations. It should be noticed that each node inside DFG represents one valid PE context instead of the LLVM IR instruction. Our compiler can statically analysis the offset of each direct memory addressing, which avoids using PE to calculate the index of every memory access operation and helps reduce the number of operations inside DFG. The number of Load/Store operations may influence the probability that memory access conflict happens, and the number of 3-input operations may influence the difficulty for the similarity-aware tuning process.

6.2. Performance

Table 4 shows the *II* of deploying loop kernels on different CGRA frameworks in Table 2 in terms of control step and cycle. The second column shows the resource minimal *II* (ResMII) of each loop kernel which applies for all the 6 CGRA configurations in terms of control step. Columns 3 to 6 and 7 to 10 show the performance data of torus and torus plus diagonal interconnect network, respectively. Within each network configuration, we show the *II* in terms of control step (columns 3 and 7); *II* generated by the baseline CGRA framework A (columns 4 and 8) in terms of cycle; and *II* generated by CGRA framework B or C (columns 6 and 10) in terms of cycle. To evaluate how the similarity-aware spatial mapping tuning algorithm influences the performance of the context-reduction CGRA, we also calculate the *II* of our CFP-based fetching CGRAs (CGRAs in framework B or C) using our compilation flow without SA tuning, but necessary inactive bits encoding (columns 5 and 9) in terms of cycle. The centralized and distributed CFP-fetching CGRA generates the same performance. Since the context-fetching latency of the baseline CGRA is 1 cycle for any control step, the number of cycles within each control step only depends on the PE execution latency. One goal of our framework is to prevent the performance from degradation compared with the baseline framework.

When using the torus network, the *II* in terms of cycle of our CFP-based context-fetching CGRA (B or C) is 23.5% larger than the baseline CGRA (A) on average when using the compilation flow without a similarity-aware tuning algorithm, but this number is reduced to 12.4% when using the similarity-aware tuning algorithm in compilation flow. This means that context-fetching can be a performance bottleneck in CFP-based context-fetching CGRA and only inactive bit encoding is not enough. If we compare column 4 with column 5, context-fetching latency exceeds the PE execution latency, which appears in 12 out of 32 selected loop kernels when using CFP-based context-fetching CGRAs. However, using the similarity-aware tuning algorithm, the context-fetching latency is reduced in 8 out of these 12 kernels. The *II* of *fft*, *harris* and *karatsuba2* are reduced from 14 to 10, 15 to 13 and 28 to 20 respectively, but their *II* are still larger than the *II* of the baseline CGRA. Furthermore, the similarity-aware tuning algorithm does not help reduce the *II* of *dfs*, *fil*, *ifft* and *MaxSubString*. One reason is that kernels such as *dfs*, *fil* and *MaxSubString* contain a large percentage of the selection operation, which makes the algorithm harder to search for valid swap.

Table 3. Kernel Features.

Loop Kernel	Benchmark	Nodes	Edges	Load/Store	Sel
aes3	EEMBC	24	12	12	0
aes5	EEMBC	22	20	10	0
auctor1	EEMBC	32	37	16	2
bezier1	EEMBC	27	28	10	0
conven1	EEMBC	21	22	9	3
correlation	PolyBench	7	5	5	0
dfs1	Graph	43	35	18	4
dijstra	Mibench	22	28	8	4
dp4	DP	49	54	21	3
fft	EEMBC	58	58	24	0
fil2	EEMBC	40	56	16	8
fir3	EEMBC	34	33	18	0
gauss	DSP	7	5	4	0
gemm1	PolyBench	48	40	32	0
gesummv	PolyBench	10	9	5	0
harris	CV	68	77	26	12
ifft5	EEMBC	58	58	24	0
karatsuba2	DSP	103	117	23	8
laplace	DSP	9	8	5	0
lcs5	DP	30	15	30	0
MaxSubString1	DP	14	16	6	2
md5	MediaBench	57	62	21	0
montecarlo1	EEMBC	22	22	9	1
montecarlo2	EEMBC	17	14	5	0
mvt	PolyBench	64	63	33	0
nbody3	MachSuite	22	21	8	2
spmv1	MachSuite	28	26	16	2
strassen1	DSP	56	55	29	0
symm	PolyBench	10	9	4	0
unstructured1	EEMBC	35	27	18	0
unstructed2	EEMBC	17	17	4	0
wavelet	DSP	32	30	18	0

With the torus plus diagonal network, the *II* of baseline CGRA in terms of cycle is reduced by 4.3% on average compared with the one using the torus network. This is because the PE execution latency is improved when the interconnection density increases. Under this interconnection, the context-fetching latency of 8 out of 32 kernels exceeds their execution latency when compiled without similarity-aware tuning algorithm. The *II* of all these kernels is reduced after using the similarity-aware tuning algorithm. It turns out that the *II* cycles of CFP-based context-fetching CGRA using torus plus diagonal interconnection and similarity-aware tuning only have 2.3% performance degradation compared with the baseline CGRA.

Table 4. Runtime comparison between different 4 × 4 CGRA and compilation frameworks.

NT	**Both**	**Torus**				**Torus + Diagonal**			
Config	**All**	**All**	**A**	**B/C**	**B/C**	**All**	**A**	**B/C**	**B/C**
Kernels	**ResMII**	II_{cs}	II	II	II_{sa}	II_{cs}	II	II	II_{sa}
aes3	2	2	6	6	6	2	6	6	6
aes5	2	2	4	6	4	2	4	6	4
autocr	2	3	6	9	6	3	6	9	6
bezier	2	2	6	7	6	2	6	6	6
conven	2	2	5	5	5	2	5	5	5
correlation	1	1	3	3	3	1	3	3	3
dfs	3	3	6	8	8	3	6	8	6
dijstra	2	2	3	3	3	2	3	3	3
dp	4	4	9	15	9	4	9	12	9
fft	4	5	9	14	10	5	9	9	9
fil	3	3	6	8	8	3	6	6	6
fir	3	3	8	8	8	3	8	8	8
gauss	1	1	2	2	2	1	2	2	2
gemm	3	4	15	15	15	4	15	15	15
gesummv	1	1	2	2	2	1	2	2	2
harris	5	5	9	15	13	5	9	15	9
ifft	4	5	8	10	10	5	8	10	9
karatsuba	7	10	11	28	20	10	10	28	16
laplace	1	1	2	2	2	1	2	2	2
lcs	2	2	8	8	8	2	8	8	8
MaxSubString	1	3	3	8	8	1	2	2	2
md5	4	4	7	7	7	4	7	7	7
montecarlo	2	2	7	7	7	2	7	7	7
montecarlo1	2	2	7	7	7	2	5	7	5
mvt	4	5	11	11	11	4	11	11	11
nbody	2	2	10	10	10	2	10	10	10
spmv	2	2	6	6	6	2	6	6	6
strassen	4	4	10	10	10	4	10	10	10
symm	1	1	2	2	2	1	2	2	2
unstructured	2	3	7	7	7	3	7	7	7
unstructured2	3	3	6	8	6	1	2	2	2
wavelet	2	2	7	7	7	2	7	7	7

6.3. Context-Footprint Compression Ratio

We calculate the compression ratio of two CFP-based context-fetching frameworks by evaluating their corresponding context footprint relative to the baseline. We also calculate the context-footprint compression ratio when applying the NOP optimization [62] over the CGRA framework A, where the NOP operations are removed to save the memory

space. Figure 13 shows the context compression ratio of different kernels and total kernels using the centralized and distributed CFP-fetching CGRAs with different interconnections, and the ones applying the NOP-removal technique on the baseline CGRA.

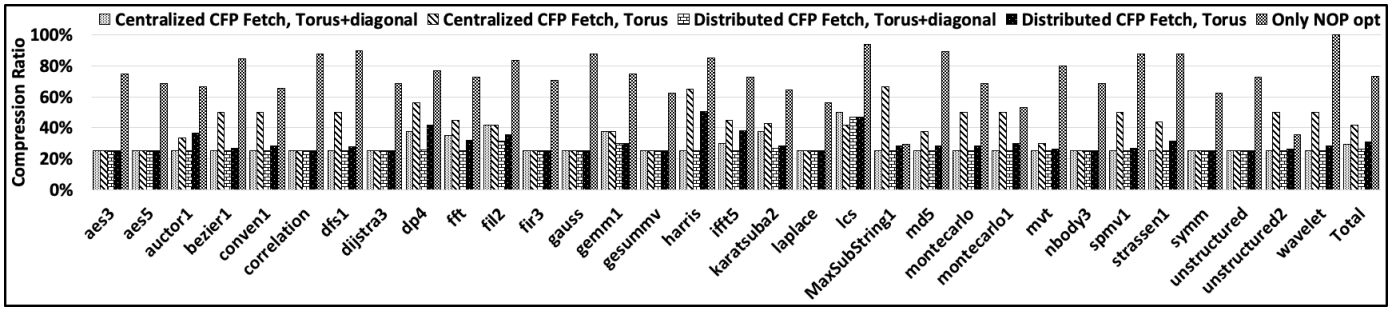


Figure 13. The context compression ratio of the centralized and decentralized CFP-fetching CGRA, and the baseline CGRA using NOP-removal technique [62].

According to the data, the compression ratio of the total kernel size when using NOP-removal technique is 73%. For centralized CFP-fetching CGRA, the average compression ratio using torus and torus plus diagonal interconnections are 42% and 29% respectively. For the distributed CFP-fetching CGRA, the average compression ratio of using torus and torus plus diagonal interconnections are 31% and 26% respectively. This means our approaches under any architecture configuration can generate lower memory footprint than the NOP-removal technique. The compression ratio is smaller for both CGRA frameworks when the interconnection density increases. This is because the similarity-aware tuning algorithm increases a successful rate of the valid swap when the interconnection density increases, which enlarges the tuning space and helps further improve the similarity.

Furthermore, the compression ratio of the centralized CFP-fetching CGRA is more sensitive to the interconnect density. This is mainly because of the redundant CFPs fetched by centralized CFP-fetching CGRA due to the imbalance CFP context-fetching latency between each PE.

6.4. Compilation Time

The compilation time of our similarity-aware tuning ranges from 3.1 ms to 20 min. We set 1200 s as the time budget of our algorithm. The reason for this is that according to the kernels we have tested, the algorithm is able to converge within 20 min in general; otherwise, it is not able to converge to the optimal point when the algorithm run time is larger than 20 min.

Figure 14 plots the compilation time of the tuning algorithm for our CGRA with torus and torus plus diagonal interconnections When using torus network, the similarity-aware tuning is not able converge to the optimal point within the time budget for loop kernels like *conven*1, *MaxSubString*1, *montecarlo*, *spmv*1, $fil2$ and *strassen*. This is mainly because the updating speed of the $Cost_{sum}$ is very slow. Either the tuning algorithm is unable to find a valid swap or the $Cost_{sum}$ cannot be reduced, as the DFG of these kernels have multiple high-degree nodes which contain more than two inputs or high fan-out operations with multiple outputs. This brings the challenge of finding a valid swap when the interconnection ability between PEs is lower, because these special operations bring more routing constraints. However, increasing the interconnection ability between PEs using torus plus diagonal can make the tuning algorithm converge on these kernels. According to Figure 14, the similarity-aware tuning algorithm can converge on all these kernels within the time budget.

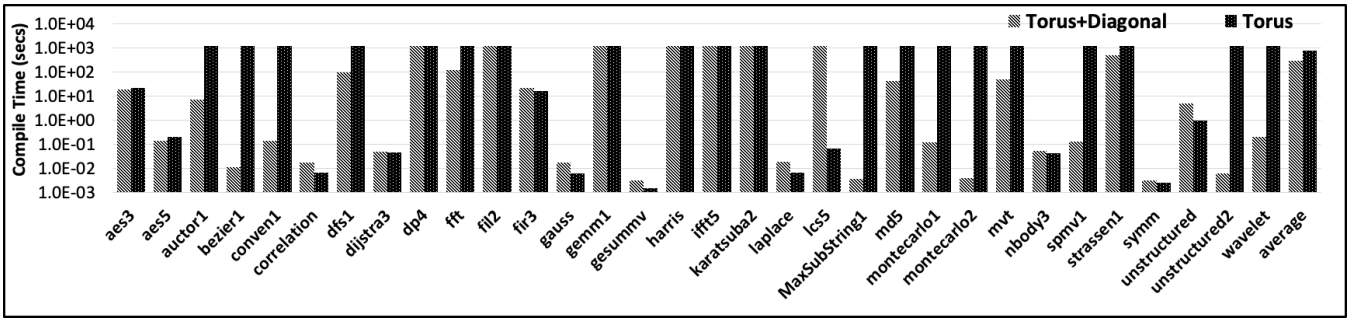


Figure 14. The compilation time of CFP-based context-fetching CGRA under torus and torus + diagonal interconnections.

6.5. Power and Area

Figure 15a,b plot the power and area distribution of the three CGRA architectures with different interconnection configurations. The modules we evaluate include PEA, GCM and SDM. We calculate the GCM area and power of the distributed CFP-fetching CGRA by evaluating the total area and power of the CLCM that has been placed closer to each PE. Since the compression ratio of the CLCMs of each PE are various due to the distributed CFP-fetching mechanism, and the compression ratio of each PE is generated in the post-silicon compile time, it is unrealistic to physically assign the different-sized CLCM to each PE. We align the depth of all the CLCM to the one generating the largest compression ratio. The meaning of A1, A2, B1, B2, C1, C2 can be referred from Table 2.

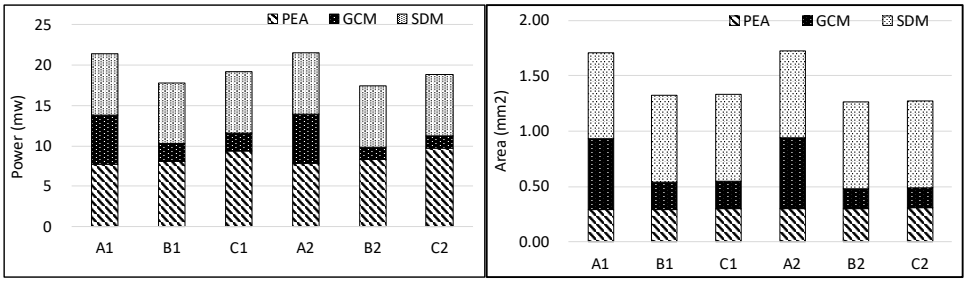


Figure 15. The area and power of 4x4 CGRAs under different frameworks.

6.5.1. Power Comparison

Figure 15a shows the power distribution of different CGRA frameworks. It turns out the GCM power consumption of the centralized and distributed CGRA (B1 and C1) with the torus network are the same, and they both generate 64% power reduction compared with the GCM power consumption of the baseline CGRA (A1). The reason for this is that even though the total context-memory footprint generated by the distributed CFP-fetching CGRA is smaller than the centralized one, the physical implementation of the CLCM of the distributed CFP must align to the one with the maximum compression ratio. The total power of the centralized and distributed CGRA with torus network is reduced by 17% and 11%, respectively, compared with the total power of the baseline CGRA. The distributed CFP-fetching CGRA generates larger total power due to the overhead of the local synchronizer inside its PE, which brings 14% of the PEA power of the centralized CFP-fetching CGRA. When using torus plus diagonal interconnection, the GCM power consumption of both centralized and distributed context-fetching CGRA (B2 and C2) are reduced by 74% compared with the corresponding baseline CGRA (A2). The total power reduction is 19% and 12%, respectively, compared with their corresponding baseline. According to the power distribution data, the benefit of context-memory reduction outweighs the overhead of enhancing the density of the interconnect network between PEs. For example, if we compare the PEA and GCM power consumption of B1 with B2, using torus plus diagonal interconnection generates 0.26 mw more power but saves 0.65 mw on context

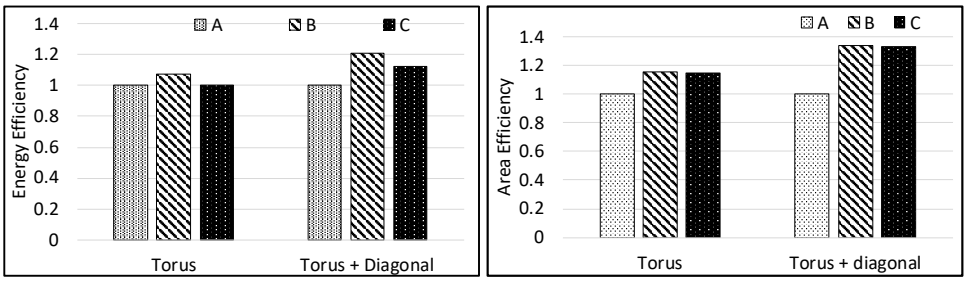
memory. Finally, if we compare the total power consumption of PEA between A1 and B1, the power overhead of the CFP context-fetching unit only takes 4% of the total PEA of the baseline CGRA.

6.5.2. Area Comparison

Under the torus network, the GCM area of both centralized and decentralized CFP-fetching CGRAs (B1 and C1) are the same, and they both generate 61% area reduction compared with the GCM area of the original CGRA (A1). The total area of the centralized and distributed CFP-fetching CGRA with torus network is reduced by 23% and 22% respectively compared with the one of the baseline CGRA. The local synchronizer only brings 1% area overhead but leads to 14% power overhead. When using torus plus diagonal interconnection, the GCM area of both centralized and distributed CFP-fetching CGRA (B2 and C2) are reduced by 72% compared with the corresponding baseline CGRA (A2). The total area reduction is 27% and 26%, respectively, compared with their corresponding baseline. According to this area distribution data, the benefit of the context-memory reduction also outweighs the overhead of enhancing the density of the interconnect network. For example, if we compare the PEA and GCM area consumption of B1 with B2, using torus plus diagonal interconnection generates 0.01 mm^2 more area but saves 0.07 mm^2 on context memory. Finally, if we compare the area of PEA between A1 and B1, the area overhead of the CFP context-fetching unit takes 3% of the total PEA area of baseline CGRA.

6.6. Energy Efficiency

Figure 16a,b shows the energy efficiency in terms of *GOPS*/*W* and area efficiency in terms of mm^2/W of the centralized and distributed CFP-fetching CGRA relative to the baseline CGRA. We generate the throughput data by calculating the throughput of running all the loop kernels. When using the torus interconnect network, the centralized CGRA can generate 7% higher energy efficiency, but the distributed CFP-fetching CGRA does not show any improvement. However, the centralized and distributed CFP-fetching CGRA with the same interconnect work acquires 16% and 15% higher area efficiency, respectively. When using the torus plus diagonal interconnect network, the centralized and decentralized CFP-fetching CGRA reaches 21% and 12% higher energy efficiency, and 34% and 33% higher area efficiency, respectively. Therefore, the best CGRA configuration is to use centralized CFP CGRA with torus plus diagonal interconnect network between PEs.



(a) Energy efficiency comparison **(b) Area efficiency comparison**

Figure 16. The normalized energy efficiency and area-efficiency comparison.

6.7. Comparison with the Other Context-Reduction Frameworks

It is non-trivial to make an apple-to-apple comparison between different context-reduction techniques. There are several reasons:

1. First, the efficiency of compression technique mainly depends on the original context architecture, which means context-reduction technique is an architecture-sensitive technique. For example, our approach mainly targets homogeneous CGRAs, whereas the state-of-the-art compression technique [36] mainly targets heterogeneous CGRAs [10]. Homogeneous and heterogeneous CGRAs reveal different architecture design philosophies

(see Section 1) which directly influence the context architecture and subsequently influence the following compilation flow.

2. Secondly, different context-reduction techniques use different compilation flows. Since existing context-reduction works mainly target the context encoding and decoding logic design, and apply a post-context-generation method, there is not much information on how the code is deployed onto the CGRA. For example, how the loop is converted to DFG may significantly influence the node number inside DFG; different compiler frameworks may lead to different DFGs, and the mapping techniques are different.

3. Finally, the target test benches are different to each other. This is mainly decided by the application domains that different CGRAs target.

Therefore, most of the existing works are not able to make a fair comparison with other context-reduction techniques. However, we still try to compare several metrics that can help us understand the insight of our context-reduction technique

Compression ratio—The state-of-the-art compression technique uses a memory-partitioning approach [36], which can reach 39% for total kernels. Our approach generates 29% for all the kernels we tested.

Area and energy overhead of the decoder logic—The state-of-the-art compression technique [36] acquires 7% and 8% energy and area overhead, respectively, on the decoder logic. In our framework, the CFP-fetching unit acts like a decoder, and its area and energy overhead relative to the original PEA is 3% and 4%, respectively.

Area and energy reduction—The state-of-the-art compression technique [36] obtains 49.5% energy reduction and nearly 52% area reduction; however, the reduction number inside their paper is relative to the total energy of the configuration memory, control logic and buffer register. It is not clear whether the energy of PE array and data memory is counted in. Our work obtains a 19% energy reduction and 27% area saving relative to the whole system, which includes configuration memory, data memory and PE array.

Portability—Even though our approach is applied to homogeneous CGRA, it can still be applied to heterogeneous CGRA with the change of separately designing the CFP and context partition for PEs with different functions. However, since there are more scheduling restrictions for heterogeneous CGRA, it will be more challenging to explore the tuning space for the similarity-aware tuning algorithm. As our method mainly relies on scheduling, the CGRAs with fewer scheduling restrictions (such as using homogeneous CGRA) will be more suitable to our approach.

Discussion—Finally, we want to provide some high-level insight of our method from the other works. To the best of our knowledge, this is the first work that leverages spatial mapping within the compilation flow to help compressing the context for modulo-scheduled CGRA. Unlike previous works, which focus more on the hardware decoder or physical memory partitioning, our work mainly focuses on stress.

Therefore, compared to other code compression works, we are trying to explore another trade-off point when designing the compression framework for modulo-scheduled CGRA. We believe this is a good start for exploring how software technique can help reduce the control overhead of module scheduled CGRA.

7. Conclusions

In this paper, we use an architecture and compiler co-designed approach to reduce the context-memory and context-fetching overhead. We leverage the redundant inactive subsection inside the context and propose a novel context-fetching scheme based on context-fetching primitive (CFP). We integrate the similarity-tuning algorithm to the CGRA compilation flow with state-of-the-art modulo scheduling and data-placement algorithm to schedule the data flow graph and improve the similarities between the context of the consecutive operations in each PE. We make a design space exploration by exploring the energy efficiency of the centralized and distributed CFP-fetching CGRA with different interconnection networks between PEs. The evaluation result shows that the CFP-based

context-fetching CGRA can generate 21% higher energy efficiency and 34% area efficiency compared with the state-of-the-art CGRA baseline.

Author Contributions: Conceptualization, Z.Z., W.S. and Q.W.; data curation, Z.Z., J.L. and P.Y.; methodology, Z.Z.; project administration, Z.M.; Software, Z.Z. and P.Y.; writing—original draft, Z.Z.; writing—review and editing, W.S; supervision, Z.M. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: Data presented in this study are available on request from the corresponding author. The data are not publicly available due to internal policies of the industry research partner.

Acknowledgments: The authors want to thank the anonymous reviewers for their constructive comments on this paper.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Prabhakar, R.; Zhang, Y.; Koeplinger, D.; Feldman, M.; Zhao, T.; Hadjis, S.; Pedram, A.; Kozyrakis, C.; Olukotun, K. Plasticine: A Reconfigurable Architecture For Parallel Paterns. In Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA), Toronto, ON, Canada, 24–28 June 2017; pp. 389–402.
2. Chen, D.C. Programmable Arithmetic Devices for High Speed Digital Signal Processing. Ph.D. Thesis, University of California at Berkeley, Berkeley, CA, USA, 1992.
3. Becker, J.; Vorbach, M. Architecture, memory and interface technology integration of an industrial/ academic configurable system-on-chip (CSoC). In Proceedings of the IEEE Computer Society Annual Symposium on VLSI, Tampa, FL, USA, 20–21 February 2003; pp. 107–112.
4. Miyamori, T.; Olukotun, U. A quantitative analysis of reconfigurable coprocessors for multimedia applications. In Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM), Napa Valley, CA, USA, 17 April 1998; pp. 2–11. [CrossRef]
5. Liu, L.; Wang, D.; Zhu, M.; Wang, Y.; Yin, S.; Cao, P.; Yang, J.; Wei, S. An Energy-Efficient Coarse-Grained Reconfigurable Processing Unit for Multiple-Standard Video Decoding. *IEEE Trans. Multimed.* **2015**, *17*, 1706–1720. [CrossRef]
6. Toi, T.; Nakamura, N.; Fujii, T.; Kitaoka, T.; Togawa, K.; Furuta, K.; Awashima, T. Optimizing time and space multiplexed computation in a dynamically reconfigurable processor. In Proceedings of the 2013 International Conference on Field-Programmable Technology (FPT), Kyoto, Japan, 9–11 December 2013; pp. 106–111. [CrossRef]
7. Goldstein, S.C.; Schmit, H.; Moe, M.; Budiu, M.; Cadambi, S.; Taylor, R.R.; Laufer, R. PipeRench: A Co/Processor for Streaming Multimedia Acceleration. In *International Symposium on Computer Architecture (ISCA)*; IEEE Computer Society: Atlanta, GA, USA, 1999; pp. 28–39. [CrossRef]
8. Mei, B.; Vernalde, S.; Verkest, D.; De Man, H.; Lauwereins, R. ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix. In *Field Programmable Logic and Application*; Springer: Berlin/Heidelberg, Germany, 2003; pp. 61–70.
9. Singh, H.; Lee, M.-H.; Lu, G.; Kurdahi, F.J.; Bagherzadeh, N.; Chaves Filho, E.M. MorphoSys: An integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Trans. Comput.* **2000**, *49*, 465–481. [CrossRef]
10. Suh, D.; Kwon, K.; Kim, S.; Ryu, S.; Kim, J. Design space exploration and implementation of a high performance and low area Coarse Grained Reconfigurable Processor. In Proceedings of the International Conference on Field-Programmable Technology (FPT), Seoul, Korea, 10–12 December 2012; pp. 67–70.
11. Kim, C.; Chung, M.; Cho, Y.; Konijnenburg, M.; Ryu, S.; Kim, J. ULP-SRP: Ultra low power Samsung Reconfigurable Processor for biomedical applications. In Proceedings of the International Conference on Field-Programmable Technology (FPT), Seoul, Korea, 10–12 December 2012, pp. 329–334. [CrossRef]
12. Lee, J.; Shin, Y.; Lee, W.; Ryu, S.; Kim, J. Real-time ray tracing on coarse-grained reconfigurable processor. In Proceedings of the International Conference on Field-Programmable Technology (FPT), Kyoto, Japan, 9–11 December 2013; pp. 192–197.
13. Karunaratne, M.; Mohite, A.K.; Mitra, T.; Peh, L. HyCUBE: A CGRA with reconfigurable single-cycle multi-hop interconnect. In Proceedings of the 54th ACM/EDAC/IEEE Design Automation Conference (DAC), Austin, TX, USA, 18–22 June 2017; pp. 1–6. [CrossRef]
14. Chen, Y.; Krishna, T.; Emer, J.S.; Sze, V. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE J. Solid-State Circuits (JSSC)* **2017**, *52*, 127–138. [CrossRef]
15. Jouppi, N.P.; Young, C.; Patil, N.; Patterson, D.; Agrawal, G.; Bajwa, R.; Bates, S.; Bhatia, S.; Boden, N.; Borchers, A.; et al.. In-Datacenter Performance Analysis of a Tensor Processing Unit. In Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA), Toronto, ON, Canada, 24–28 June 2017; pp. 1–12.

16. Kwon, H.; Samajdar, A.; Krishna, T. MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Reconfigurable Interconnects. In Proceedings of the Architectural Support for Programming Languages and Operating Systems (ASPLOS), Williamsburg, VA, USA, 24–28 March 2018; pp. 461–475.
17. Nowatzki, T.; Gangadhar, V.; Ardalani, N.; Sankaralingam, K. Stream-Dataflow Acceleration. In Proceedings of the International Symposium on Computer Architecture (ISCA), Toronto, ON, Canada, 24–28 June 2017; pp. 416–429.
18. Liu, L.; Li, Z.; Yang, C.; Deng, C.; Yin, S.; Wei, S. HReA: An Energy-Efficient Embedded Dynamically Reconfigurable Fabric for 13-Dwarfs Processing. *IEEE Trans. Circuits Syst. II Express Briefs* **2018**, *65*, 381–385. [CrossRef]
19. Wijtvliet, M.; Waeijen, L.; Corporaal, H. Coarse grained reconfigurable architectures in the past 25 years: Overview and classification. In Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS), Samos Island, Greece, 18–21 June 2016, pp. 235–244.
20. Dave, S.; Shrivastava, A. CCF: A CGRA Compilation Framework. Available online: https://github.com/MPSLab-ASU/ccf (accessed on 1 September 2021).
21. Mei, B.; Vernalde, S.; Verkest, D.; Man, H.D.; Lauwereins, R. DRESC: A retargetable compiler for coarse-grained reconfigurable architectures. In Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT), Hong Kong, China, 16–18 December 2002; pp. 166–173.
22. Friedman, S.; Carroll, A.; Van Essen, B.; Ylvisaker, B.; Ebeling, C.; Hauck, S. SPR: An Architecture-adaptive CGRA Mapping Tool. In Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA), Monterey, CA, USA, 22–24 February 2009; pp. 191–200.
23. Chen, L.; Mitra, T. Graph minor approach for application mapping on cgras. *Trans. Reconfigurable Technol. Syst. (TRETS)* **2014**, *7*, 21. [CrossRef]
24. Mahdi, H.; Shrivastava, A.; Vrudhula, S. EPIMap: Using epimorphism to map applications on CGRAs. In Proceedings of the Design Automation Conference (DAC), San Francisco, CA, USA, 3–7 June 2012; pp. 1280–1287.
25. Hamzeh, M.; Shrivastava, A.; Vrudhula, S. REGIMap: Register-aware application mapping on Coarse-Grained Reconfigurable Architectures (CGRAs). In Proceedings of the ACM/EDAC/IEEE Design Automation Conference (DAC), Austin, TX, USA, 29 May–7 June 2013; pp. 1–10.
26. Dave, S.; Balasubramanian, M.; Shrivastava, A. RAMP: Resource-Aware Mapping for CGRAs. In Proceedings of the 55th ACM/ESDA/IEEE Design Automation Conference (DAC), San Francisco, CA, USA, 24–28 June 2018; pp. 1–6.
27. Zhao, Z.; Sheng, W. Resource-saving compile flow for coarse-grained reconfigurable architectures. In Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig), Cancun, Mexico, 10 August 2015; pp. 1–8.
28. Fell, A.; Rakossy, Z.E.; Chattopadhyay, A. Force-directed scheduling for Data Flow Graph mapping on Coarse-Grained Reconfigurable Architectures. In Proceedings of the 2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14), Cancun, Mexico, 8–10 December 2014; pp. 1–8.
29. Yoon, J.W.; Shrivastava, A.; Park, S.; Ahn, M.; Paek, Y. A Graph Drawing Based Spatial Mapping Algorithm for Coarse-Grained Reconfigurable Architectures. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2009**, *17*, 1565–1578. [CrossRef]
30. Dimitroulakos, G.; Georgiopoulos, S.; Galanis, M.D.; Goutis, C.E. Resource aware mapping on coarse grained reconfigurable arrays. *Microprocess. Microsyst.* **2009**, *33*, 91–105. [CrossRef]
31. Aslam, N.; Milward, M.J.; Erdogan, A.T.; Arslan, T. Code Compression and Decompression for Coarse-Grain Reconfigurable Architectures. *TVLSI* **2008**, *16*, 1596–1608. [CrossRef]
32. Chung, M.; Cho, Y.; Ryu, S. Efficient code compression for coarse grained reconfigurable architectures. In Proceedings of the IEEE 30th International Conference on Computer Design (ICCD), Montreal, QC, Canada, 30 September–3 October 2012; pp. 488–489.
33. Chung, M.; Kim, J.; Cho, Y.; Ryu, S. Adaptive compression for instruction code of Coarse Grained Reconfigurable Architectures. In Proceedings of the 2013 International Conference on Field-Programmable Technology (FPT), Kyoto, Japan, 9–11 December 2013; pp. 394–397.
34. Kim, Y.; Mahapatra, R.N. Dynamic Context Compression for Low-Power Coarse-Grained Reconfigurable Architecture. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2010**, *18*, 15–28. [CrossRef]
35. Egger, B.; Lee, H. A Space- and Energy-Efficient Code Compression/Decompression Technique for Coarse-Grained Reconfigurable Architectures. In *International Symposium on Code Generation and Optimization (CGO)*; IEEE Press: Austin, TX, USA, 2017; pp. 197–209.
36. Lee, H.; Moghaddam, M.S. Improving Energy Efficiency of Coarse-Grain Reconfigurable Arrays Through Modulo Schedule Compression/Decompression. *ACM Trans. Archit. Code Optim.* **2018**, *15*, 1–26. [CrossRef]
37. Yin, S.; Yin, C. Configuration context reduction for coarse-grained reconfigurable architecture. *IEICE Trans. Inf. Syst.* **2012**, *95*, 335–344. [CrossRef]
38. Zhao, Z.; Sheng, W.; Wang, Q.; Yin, W.; Ye, P.; Li, J.; Mao, Z. Towards Higher Performance and Robust Compilation for CGRA Modulo Scheduling. *IEEE Trans. Parallel Distrib. Syst.* **2020**, *31*, 2201–2219. [CrossRef]
39. Zhao, Z.; Liu, Y. Optimizing the data placement and transformation for multi-bank CGRA computing system. In Proceedings of the 2018 Design, Automation Test in Europe Conference Exhibition (DATE), Dresden, Germany, 19–23 March 2018; pp. 1087–1092.
40. Liu, L.; Yang, C.; Yin, S.; Wei, S. CDPM: Context-Directed Pattern Matching Prefetching to Improve Coarse-Grained Reconfigurable Array Performance. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **2018**, *37*, 1171–1184. [CrossRef]

41. Park, H.; Fan, K. Edge-centric Modulo Scheduling for Coarse-grained Reconfigurable Architectures. In Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT), New York, NY, USA, 25–29 October 2008; pp. 166–176.
42. Mei, B.; Vernalde, S.; Verkest, D.; Man, H.D.; Lauwereins, R. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. In Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE), Munich, Germany, 3–7 March 2003; pp. 296–301.
43. Oh, T.; Egger, B.; Park, H.; Mahlke, S. Recurrence Cycle Aware modulo Scheduling for Coarse-Grained Reconfigurable Architectures. In Proceedings of the 2009 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, Dublin, Ireland, 19–20 June 2009; Association for Computing Machinery: New York, NY, USA, 2009; pp. 21–30. [CrossRef]
44. Theocharis, P.; Sutter, B.D. A Bimodal Scheduler for Coarse-Grained Reconfigurable Arrays. *ACM Trans. Archit. Code Optim.* **2016**, *13*, 15:1–15:26. [CrossRef]
45. Yin, S.; Yao, X.; Lu, T.; Liu, D.; Gu, J.; Liu, L.; Wei, S. Conflict-Free Loop Mapping for Coarse-Grained Reconfigurable Architecture with Multi-Bank Memory. *IEEE Trans. Parallel Distrib. Syst. (TPDS)* **2017**, *28*, 2471–2485. [CrossRef]
46. Bougard, B.; De Sutter, B.; Verkest, D.; Van der Perre, L.; Lauwereins, R. A Coarse-Grained Array Accelerator for Software-Defined Radio Baseband Processing. *IEEE Micro* **2008**, *28*, 41–50. [CrossRef]
47. Kim, Y.; Lee, J.; Shrivastava, A.; Paek, Y. Operation and Data Mapping for CGRAs with Multi-bank Memory. In Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES), Stockholm, Sweden, 13–15 April 2010; ACM: New York, NY, USA 2010; pp. 17–26.
48. Lee, J.; Jeong, Y.; Seo, S. Fast shared on-chip memory architecture for efficient hybrid computing with CGRAs. In Proceedings of the 2013 Design, Automation Test in Europe Conference Exhibition (DATE), Grenoble, France, 18–22 March 2013; pp. 1575–1578.
49. Wang, Y.; Li, P.; Zhang, P.; Zhang, C.; Cong, J. Memory partitioning for multidimensional arrays in high-level synthesis. In Proceedings of the 2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC), Austin, TX, USA, 29 May 2013; pp. 1–8.
50. Wijerathne, D.; Li, Z.; Karunarathne, M.; Pathania, A.; Mitra, T. CASCADE: High Throughput Data Streaming via Decoupled Access-Execute CGRA. *ACM Trans. Embed. Comput. Syst.***2019** *18*, 1–26. [CrossRef]
51. Kim, Y.; Kiemb, M.; Park, C.; Jung, J.; Choi, K. Resource sharing and pipelining in coarse-grained reconfigurable architecture for domain-specific optimization. In *Design, Automation and Test in Europe (DATE)*; IEEE Computer Society: Washington, DC, USA, 2005; pp. 12–17. Volume 1.
52. Kim, Y.; Mahapatra, R.N.; Park, I.; Choi, K. Low Power Reconfiguration Technique for Coarse-Grained Reconfigurable Architecture. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2009**, *17*, 593–603. [CrossRef]
53. Park, S.; Choi, K. An approach to code compression for CGRA. In Proceedings of the 2011 3rd Asia Symposium on Quality Electronic Design (ASQED), Kuala Lumpur, Malaysia, 19–20 July 2011; IEEE: Piscataway, NJ, USA, 2011; pp. 240–245.
54. Asanovic, K.; Bodik, R.; Catanzaro, B.C.; Gebis, J.J.; Husbands, P.; Keutzer, K.; Patterson, D.A.; Plishker, W.L.; Shalf, J.; Williams, S.W.; et al. *The Landscape of Parallel Computing Research: A View from Berkeley*; Technical Report; UC: Berkeley, CA, USA, 2006.
55. Liu, L.; Zhu, J.; Li, Z.; Lu, Y.; Deng, Y.; Han, J.; Yin, S.; Wei, S. A Survey of Coarse-Grained Reconfigurable Architecture and Design: Taxonomy, Challenges, and Applications. *ACM Comput. Surv.* **2019**, *52*, 1–39. [CrossRef]
56. Lattner, C.; Adve, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In Proceedings of the International Symposium on Code Generation and Optimization (CGO) , Palo Alto, CA, USA, 20–24 March 2004; p. 75.
57. Lee, J.; Seo, S.; Lee, H.; Sim, H.U. Flattening-based mapping of imperfect loop nests for CGRAs. In Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), Uttar Pradesh, India, 12–17 October 2014; pp. 1–10.
58. Lee, C.; Potkonjak, M.; Mangione-Smith, W.H. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communicatons Systems. In Proceedings of the International Symposium on Microarchitecture (MICRO), Research Triangle Park, NC, USA, 1–3 December 1997; pp. 330–335.
59. Guthaus, M.R.; Ringenberg, J.S.; Ernst, D.; Austin, T.M.; Mudge, T.; Brown, R.B. MiBench: A free, commercially representative embedded benchmark suite. In Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization, WWC-4 (Cat. No.01EX538), Austin, TX, USA, 2 December 2001; pp. 3–14.
60. Reagen, B.; Adolf, R. MachSuite: Benchmarks for Accelerator Design and Customized Architectures. In Proceedings of the IEEE International Symposium on Workload Characterization, Raleigh, NC, USA, 26–28 October 2014.
61. Pouchet, L.N. Polybench: The Polyhedral Benchmark Suite. Available online: http://www.cs.ucla.edu/pouchet/software/polybench (accessed on 1 March 2012).
62. Jin, T.; Ahn, M.; Yoo, D.; Suh, D.; Choi, Y.; Kim, D.H.; Lee, S. Nop compression scheme for high speed DSPs based on VLIW architecture. In Proceedings of the 2014 IEEE International Conference on Consumer Electronics (ICCE), Las Vegas, NV, USA, 10–13 January 2014; pp. 304–305.