# Chapter 1
# Coarse Grained Reconfigurable Array (CGRA)

Zhaoying Li, Dhananjaya Wijerathne, and Tulika Mitra

Coarse-Grained Reconfigurable Array (CGRA) is a promising class of spatial accelerator that offers high performance, energy-efficiency, as well as flexibility to support a wide range of application domains. CGRAs can bridge the gap between efficient but inflexible domain-specific accelerators and flexible but inefficient general-purpose processors. A CGRA is essentially an array of word-level processing elements connected via on-chip interconnect. Both the processing elements and the interconnect can be reconfigured per cycle following the on-chip configuration memory content. Thus the compiler needs to map the compute-intensive loop kernels of the application onto the CGRA in a spatio-temporal fashion by setting up the configuration memory. The simplicity and parallelism of the architecture coupled with the efficacy of the compiler enable the CGRA to reach the dual goal of hardware-like efficiency with software-like programmability. We present a comprehensive review of the CGRAs starting with the historical context, sketching the architectural landscape, and providing an extensive overview of the compilation approaches.

## 1.1 Introduction

The history of computing has been dominated by general-purpose processors [1] that can execute any possible application offering unlimited flexibility. Unfortunately, such processors suffer from low performance and energy-efficiency due to the high overhead involved in executing the instructions beyond just the computation (e.g., fetching instructions and data from the memory, decoding instructions, respecting

Zhaoying Li
National University of Singapore, Singapore, e-mail: zhaoying@comp.nus.edu.sg

Dhananjaya Wijerathne
National University of Singapore, Singapore, e-mail: dmd@comp.nus.edu.sg

Tulika Mitra
National University of Singapore, Singapore, e-mail: tulika@comp.nus.edu.sg

the data and control dependencies etc.) [2] and extracting parallelism from sequential instructions stream at runtime. At the other end of the spectrum, we are witnessing the emergence of domain-specific hardware accelerators [3] for many popular tasks such as deep neural networks, image/video processing, cryptography, among others. These ASIC (Application-Specific Integrated Circuit) accelerators provide high performance and energy efficiency but zero flexibility as they are tied to one specific task or application domain. Reconfigurable spatial accelerators present a compromise between the two extremes by supporting ASIC-like efficiency while maintaining flexibility through software programmability [4].

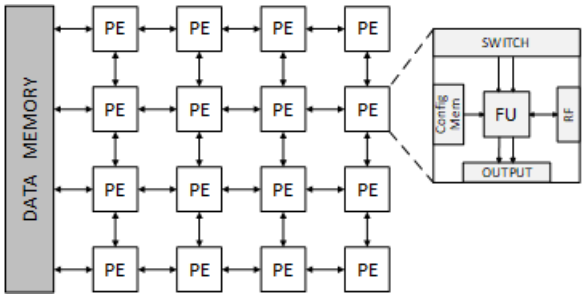



Fig. 1.1: A classic 4x4 CGRA (Coarse-Grained Reconfigurable Array).

A Coarse-Grained Reconfigurable Array (CGRA) is a spatial hardware accelerator with very simple architecture. A generic CGRA comprises of a 2D array of processing elements capable of performing basic arithmetic, logic, and memory operations at word level using the functional unit (FU) and a small register file (RF) as temporary data storage as shown in Figure 1.1. Each processing element is connected to its neighbors through the switch and can transfer the result of the computation to selected neighbors for the next cycle. Both the computation performed by each individual processing element and the routing of the data to the neighbors through the interconnect can be configured on a per cycle basis. This is achieved by storing a predetermined sequence of configurations for limited number of cycles in an on-chip memory (configuration memory). At runtime, the sequence of configurations is repeated in a cyclical fashion. In other words, the CGRA fabric can be configured both in the spatial (restricted by the number of processing elements) and the temporal (restricted by the number of configurations that can be stored on chip) domain. In addition, a new sequence of configurations can be brought into the CGRA from external storage, if necessary, at the cost of runtime delay. We will explore the variations of CGRA architectures in Section 1.3.

The high performance of the CGRA comes from the parallelism offered by the large number of on-chip processing elements. On the other hand, the simplicity of the architecture that just faithfully follows the planned computation and routing (generated by the compiler) without any runtime effort to extract parallelism from the application leads to significantly improved energy efficiency compared to

the general-purpose processors. At the same time, the word-level (coarse-grained) reconfigurations supported by the CGRA compared to bit-level (fine-grained) reconfigurations of the FPGAs (Field-Programmable Gate Arrays) empowers the CGRAs to achieve higher performance and lower power compared to the FPGAs. Finally, the per-cycle temporal configuration dimension of the CGRA is a powerful feature that allows the CGRA to operate with smaller spatial dimensions by time-multiplexing computation and dataflow as opposed to only spatial dimension in the FPGA. The reconfiguration in FPGA, while possible, can only happen over a longer time interval similar to bringing in a new sequence of configurations in CGRA. Thus, CGRA accelerators can have smaller chip area and hence lower power (specially leakage power) compared to the FPGAs. Obviously, the complexity burden is now transferred from the architecture to the compiler.

Let us now focus on how the compiler can exploit the spatio-temporal configuration of the CGRA to accelerate application execution. As the CGRA execution repeatedly cycles through a limited-length sequence of configurations, the application loop kernels are the perfect candidates for acceleration. The compiler is responsible for extracting as much parallelism from the loop kernel as possible (subject to data dependency constraints) and maximize utilization of the array of processing elements. This will lead to reduced temporal length of the configuration sequence and significantly improved runtime of the kernel.

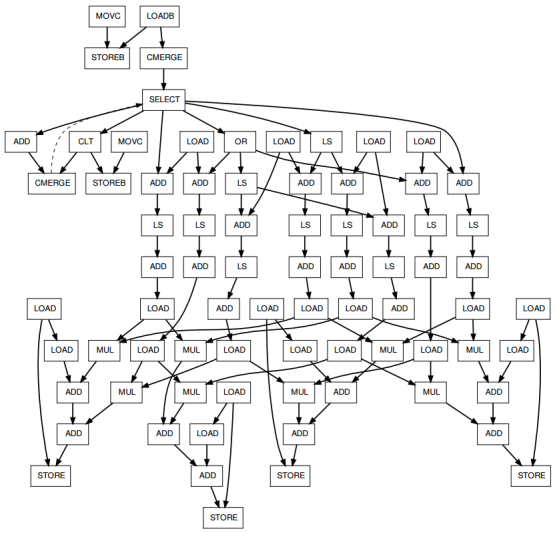



Fig. 1.2: Dataflow graph of the General Matrix Multiply (GEMM) kernel.

The CGRA compiler achieves the mapping by embracing the dataflow computing model. In this model, the compiler exposes all the computations and the flow of data between dependent computations from the high-level sequential code fragment of

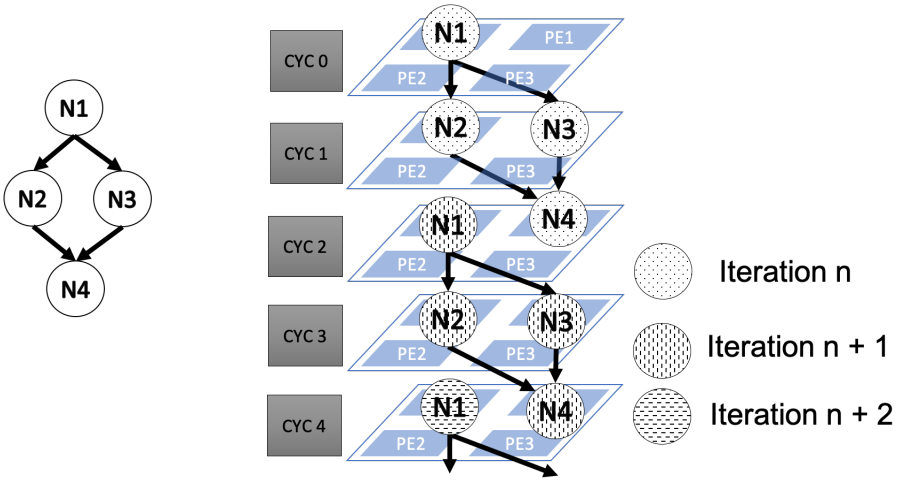



Fig. 1.3: Spatio-tenporal mapping of a simple dataflow graph on a 2x2 CGRA.

the loop kernel. Figure 1.2 shows a dataflow graph of the General Matrix Multiply (GEMM) kernel [5]. This dataflow graph is subsequently mapped onto the CGRA to maximize parallelism while satisfying all the constraints of the architecture as well as data dependencies within the loop kernel. The challenge now is to map the computations within the loop kernel onto the processing elements by finding appropriate spatio-temporal coordinates and route the data dependencies between processing elements. Fig. 1.3 shows the spatio-temporal mapping of a simple dataflow graph on a 2x2 CGRA. We provide an in-depth tour of the diverse mapping approaches with trade-offs between the quality of the mapping and the compilation time in Section 1.4.

Finally, we introduce additional challenges and opportunities in the CGRA accelerator space in terms of data memory management, configuration memory management, and mapping of entire application as opposed to a single isolated loop kernel.

## 1.2 Historical Context

The first general-purpose microprocessor, Intel 4004, was introduced in 1971. The microprocessor industry since then has enjoyed an unprecedented growth in performance due to Moore's Law [6], Dennard scaling [7], and micro-architectural innovations [1] . While Moore's Law was responsible for the sustained increase in clock frequency, the processor performance improved further due to several micro-architectural innovations including processor pipeline, out-of-order execution, speculation and cache memory hierarchy among others. These advancements enabled the

processor to extract instruction-level parallelism (ILP), thereby boosting the critical instructions-per-cycle (IPC) metric [1]. More importantly, as the ILP was extracted transparently by the underlying architecture from single-threaded programs, the software developers enjoyed the performance benefit without any additional effort. Together, the growth in clock frequency and IPC ensued the relentless gain in processor performance spanning over three decades. However, this performance growth has come to an end with power wall due to the breakdown of Dennard scaling, ILP wall, and memory wall [8]. Thus, computing systems made the irreversible transition in early 2000 towards multi- and many-core architectures to gainfully employ the growing number of transistors supported by Moore' Law and exploit thread-level parallelism (TLP) instead of ILP. However, simply increasing the core count in multicores is no longer tenable as the sequential fragment limits the speedup of the entire application according to Amdahl's Law [9].

Against this backdrop, domain-specific accelerators [3, 10, 11, 12] specialized for a particular task such as deep neural networks, image/video processing, encryption etc. have become prevalent from tiny Internet of Things (IoT) devices to the data-centers. Current system-on-chips (SoCs) include a number of special-purpose accelerators. Shao et al. [13] analyzed die photos from three generations of Apple's SoCs: A6 (iPhone 5), A7 (iPhone 5S) and A8 (iPhone 6) to show that consistently more than half of the die area is dedicated to application-specific hardware accelerators and estimated the presence of around 29 accelerators in A8 SoC. The ITRS roadmap predicts hundreds to thousands of customized accelerators by 2022 [14]. These tailor-made ASIC (application-specific integrated circuit) accelerators can provide excellent performance and energy-efficiency but suffer from lack of flexibility as they are restricted to only one particular task. Thus, such accelerators can only be feasible for tasks that are ubiquitous across multiple applications.

Ideally, we want the best of both the worlds, i.e., a general-purpose universal accelerator that can reach close to the performance and efficiency of domain-specific accelerators while maintaining software programmability and flexibility to support multiple tasks. Reconfigurable computing [15] fills this gap between hardware and software with far superior performance potential compared to programmable cores while maintaining higher-level of flexibility than ASICs.

Field-Programmable Gate Arrays (FPGAs) are pre-fabricated semiconductor devices that can be reprogrammed to create almost any digital circuit/system [16]. FPGAs contain an array of computation elements, called configurable logic blocks connected through a set of programmable routing resources. A digital circuit can be fabricated on FPGAs by appropriately setting the configuration bits of the logic blocks for the functionality and connecting these blocks together through reconfigurable routing. This comes at the cost of area, power, and delay: an FPGA requires approximately 20 to 35 times more area than ASIC, has roughly $3-4$ times slower performance than ASIC and consumes about 10 times as much dynamic power [17].

Coarse-Grained Reconfigurable Arrays (CGRAs) [18] are promising alternative between ASICs and FPGAs. FPGAs do not have as high an efficiency as ASIC accelerators due to the fine bit-level granularity of reconfiguration that results in lower performance, higher energy consumption, and longer reconfiguration penalty.

In contrast, CGRAs, as the name suggests, comprise of coarse-grained functional units (FUs) connected via typically a mesh-like interconnect as shown in Figure 1.1. The functional units are capable of performing arithmetic/logic operations and can be reconfigured on a per cycle basis by writing to a control (context) register associated with each functional unit. The functional units can exchange data among themselves through the interconnect. As many functional units work in parallel, CGRAs can easily accelerate compute-intensive loop executions by exploiting instruction-level parallelism. The primary challenge lies with the compiler that needs to map and schedule the instructions on the FUs as well as take care of the routing of data among the FUs through the interconnect.

CGRA was introduced around 2000 [19, 20, 21]. Recently, CGRA is witnessing a resurgence in both industry and academia as a promising accelerator architecture that can provide both efficiency and programmability. DARPA has recently launched Software Defined Hardware (SDH) programme [22] to build CGRA-like reconfigurable hardware that would enable near ASIC performance without sacrificing programmability. Various CGRAs are appearing from academia and industry, such as HRL [23], Plasticine [24], HyCUBE [25], Wave DPU [26], Sambanova [27], Samsung Reconfigurable Processor [28], Renesas Dynamically Reconfigurable Processor (DRP) [29] and Intel Configurable Spatial Accelerator [30]. These CGRAs have more processing elements and complex architectures compared to the original designs and thus require more compilation effort to efficiently utilize the hardware resources.

There have been many works on domain-specific spatial accelerators in recent literature [31, 10, 32, 33, 34, 35]. These accelerators target applications in specific domains such as deep neural network, image analysis, and signal processing. The micro-architecture of domain-specific spatial accelerators shares many similarities with CGRAs. Like CGRAs, most of the domain-specific accelerators have an array of processing elements connected in a two-dimensional grid. However, the processing elements have limited and specific computation capability. The interconnection network is designed to support specific data flow and not fully reconfigurable. For example, in the Google Tensor Processing Unit (TPU), the processing elements only support Multiply and Accumulation operations while the interconnection network support systolic data flow for matrix multiplication [10]. These domain-specific accelerators can be viewed as different instantiation of domain-agnostic CGRA accelerator that can be configured in software to support any data flow and computation.

## 1.3 Architecture: A Landscape of Modern CGRA

In this section, we provide a brief overview of the basic CGRA architecture and its variations. For a detailed survey of the CGRA architectures, the readers can refer to [36, 37]

**Basic CGRA architecture** A CGRA consists of a set of Processing Elements (PE) and an on-chip network. A CGRA can reconfigure each PE for different operations and the network for different routing on a per-cycle basis. Figure 1.1 shows an abstract block diagram of a classic 4x4 CGRA. It uses a 2D mesh network and each PE is connected to its neighboring PEs. A PE comprises of a Functional Unit (FU), Register File (RF), crossbar switches, and configuration memory. Each FU can have one or more ALU (Arithmetic-Logic Unit) or other computation units. The on-chip data memory, usually Scratchpad Memory (SPM), feeds data to the whole PE array. The data transfer between the SPM and the off-chip memory takes place through Direct-Memory Access (DMA). In each cycle, a PE reads a configuration from the configuration memory and configures the corresponding modules such as the ALU, the switches and the RF ports. Then PE executes the operation and passed the data to other PEs through the on-chip network.

**Homogeneous and Heterogeneous CGRA** From the perspective of the PEs, the CGRAs can be classified into two categories: homogeneous and heterogeneous. In homogeneous CGRA, all the PE have the same functionality, while in heterogeneous CGRA, the PEs can have different functionality. If a CGRA targets application kernels from some specific domains, special PEs can be useful, such as the ones supporting Multiply-Accumulate (MAC) operations in machine learning. However, if these special PEs are costly in terms of area or power, then the CGRA includes special functionality in only some of the PEs. Most CGRAs provide heterogeneity in terms of memory access functionality. For example, in the CGRA of Fig. 1.1, it is not necessary to let all the PEs access the on-chip data memory. The latency for data memory access is generally much longer than computation, and the SPM also has limited number of ports restricting the number of parallel accesses. Hence, usually only the PEs at the boundary can access the SPM. Another example is RAPID architecture [38] that has 1D array of Special Function Units (SFUs) alongside a 2D array of PEs. The SFU is used to support FP32 operations, and other PEs can only support integer operations.

A recent work REVAMP [39] proposes a generalized automated approach in heterogeneous CGRA exploration that can work across diverse architectures. It is a design space exploration framework that can automatically realize more power-efficient heterogeneous CGRA versions from a given homogeneous CGRA and target application suit. Their micro-architectural optimizations cover a broad scope of heterogeneity, including compute, interconnect, and PE-local storage. It also automatically generates compiler support to map loop kernels onto derived heterogeneous architecture efficiently.

**Spatial CGRA** A CGRA can reconfigure the PEs for different operations and routing per cycle. Each PE is associated with a configuration memory. The configuration memory stores a limited number of configuration words, one per cycle. The PE rotates or loops through these configuration words and accordingly sets the operation of the FU and the routing for the switches and the RF. A special case is a CGRA with only one configuration word and is referred to as a spatial CGRA. A spatial

CGRA can reduce area, power, as well as cycle time (higher clock frequency) as there is no reconfiguration delay involved. The area and power of the configuration memory is considerable for the CGRAs. In [40], the power consumption of a 4KB configuration memory in a 4x4 CGRA is around 40% of the whole chip power. A spatial CGRA is more energy-efficient than traditional CGRAs. However it does not have the advantage of temporal dimension and essentially reduces to FPGA but with coarse-grained reconfigurable units. Note that the limited configuration memory, while area and energy efficient, may not be able to accommodate large kernels or need loop partitioning with runtime configuration reloading to accelerate such kernels.

**On-chip network** The on-chip network connects the PEs to route data. In each PE, there are routing paths from the input ports to the output ports. Also, the data can be stored in the register file while waiting for processing or further routing. The most common network is Neighbor-to-Neighbor (N2N) connection. Each PE is connected to its neighboring PEs and neighbors can be reached in one cycle. Routing to distant PEs requires other intermediate PEs and needs multiple cycles. The simple N2N network, however, provides very limited interconnection on the chip. It needs tremendous compilation effort to achieve good speedup in accelerating kernels with complex data dependencies and even then the speedup can be limited.

A recent CGRA architecture called HyCUBE [25] creates a larger virtual neighborhood for each PE by allowing single-cycle multiple-hop connections. HyCUBE designs a special bypass network to allow intermediate PEs to forward data to other PEs without consuming the data. Thus a PE can send data to distant PEs in one cycle. The HyCUBE chip [41] offers four hops per cycle at maximum clock frequency of 753 MHz. Increasing the number of hops per cycle further will reduce the maximum possible clock frequency. The dataflow graph (DFG) of an application kernel can have complex structure and data dependencies. While N2N networks need multiple cycles when the source and destination nodes corresponding to a data dependency are mapped to distant PEs, HyCUBE only needs one cycle to route most data dependencies leading to better performance both in terms of compilation time (as the mapping becomes easier with larger neighborhood) and actual kernel execution time (due to reduced delay in routing data dependencies). Moreover, in N2N network, a PE that is involved in routing transient data cannot perform computation in the same cycle as the data needs to be stored in the register file of the PE requiring an explicit move operation. HyCUBE allows the intermediate PEs in the bypass path to continue executing operations leading to better utilization of the PEs in performing useful computation.

The above networks cannot scale well with increasing CGRA sizes. A bigger CGRA is usually tiled into blocks and each block is a small sub-CGRA. The network among the blocks often has a higher bandwidth than the one inside a block. An example of such titled architecture is Plasticine [24] that provides scalar and vector communication channels between the blocks.

**Memory hierarchy** Typically, the CGRA memory hierarchy consists of two types of memory: data memory to hold input, output and intermediate data and the configuration memory to hold the configuration directives for the FU, RF, and the switches.

Most CGRA architectures use multi-bank scratchpad memory as the global on-chip data memory [20, 42, 25]. Scratchpad memories are fully software-controlled, meaning that the data movement between the off-chip main memory and on-chip scratchpad memory is explicitly controlled through directives generated by the compiler. Therefore scratchpad memories are more power-efficient than hardware controlled caches. Multi-bank memory SPM is used to increase the data throughput, i.e., the number of parallel accesses between the SPM data memory and the PE array. Usually, each memory bank has a few (one or two) read/write ports, and a subset of PEs have access to each memory bank. The CGRA PEs execute load and store operations to load the input data and store the computed data back into the on-chip memory. Figure 1.4 shows CGRA data memory with four memory banks where only the boundary PEs on the left side have access to the data memory. Some architectures perform the load/store address generation within the PE array, while others have specialized hardware address generation units [43]. Apart from global data memory, some CGRA architectures use shared register files to hold intermediate data. These register files are shared between a subset of PEs. It provides an alternative to the on-chip network for communication between those subsets of PEs.

The CGRA configuration memory, also referred to as context/instruction memory, holds the directives for CGRA execution each cycle including the operation to be executed by the PEs and the routing configurations for the crossbars switches. As CGRAs are specifically used for accelerating loop kernels, the same sequence of configurations are repeated over a fixed number of cycles. The configurations are loaded into the configuration memory before the CGRA execution starts. The configuration memory can be either centralized (global) or decentralized (local), where each PE has a separate configuration memory. Even in a decentralized setting, the configurations for the PEs are fetched and decoded in a lockstep manner. Therefore, program counters of all the PEs have the same value even though they have different configurations.

### Interface between CPU and CGRA

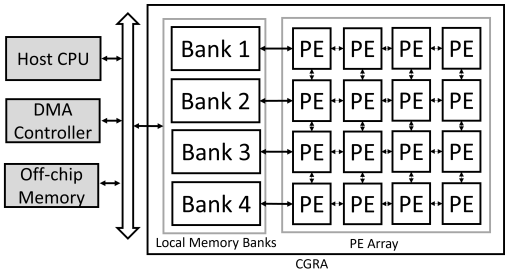



Fig. 1.4: On-chip memory hierarchy of CGRA loosely coupled with host CPU

CGRAs are used to accelerate compute-intensive loop kernels of the applications. Therefore, it needs to be coupled with the host processor for executing a complete application. The host processor is responsible for running the non-loop code, configuring the CGRA, and initiating the DMA data transfers from the main memory to the CGRA local memory.

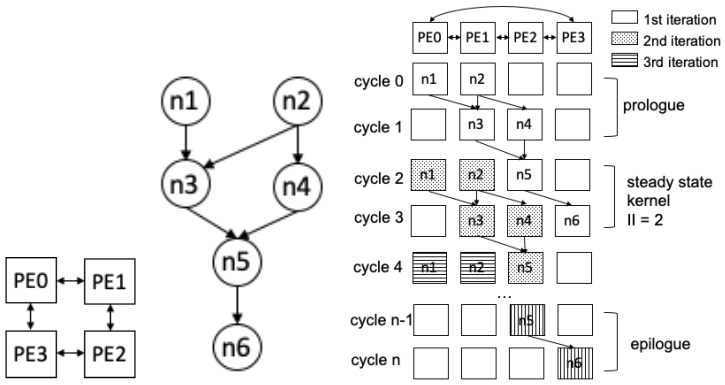
Some CGRAs are closely coupled with the main processor, where CGRA is a part of the main CPU. For example, ADRES [20] CGRA is tightly coupled with the main processor, where the top row of the PE array is a VLIW processor that acts as the main processor. Figure 1.4 shows a loosely coupled CPU where the CGRA is connected to an independent accelerator. MorphoSys CGRA [42] is an example of a loosely coupled CGRA. Loosely coupled CGRAs offer more flexibility in the design phase as they can be designed independently. In a loosely coupled system, both the CPU and the CGRA can execute code in parallel in a non-blocking manner. A tightly coupled system typically cannot execute code in parallel on the CPU and the CGRA as they share the same resources. However, the overheads in data transfer are higher in the loosely coupled system compared to the tightly coupled system.

## 1.4 Compilation for CGRAs

Given a loop from an application and a CGRA architecture, the goal of compilation is to map the loop onto the CGRA (i.e., generate the configurations for a fixed number of cycles) to maximize the throughput. In general, this compilation is referred as *mapping* in the CGRA world. The loop is represented as a Data Flow Graph (DFG), where the nodes represent the operations and the edges represent the dependency between the nodes.

### 1.4.1 Modulo Scheduling and Modulo Routing Resource Graph (MRRG)

**Modulo Scheduling** Modulo scheduling is a software pipelining technique to exploit the instruction-level parallelism among the loop iterations [44]. There are often inadequate instruction-level parallelism in a single iteration of a loop. Pipelining consecutive loop iterations can provide more parallelism and thus improves the resource utilization. Fig.1.5a shows a 2x2 homogeneous CGRA and we assume each PE can support any operation. Fig.1.5b shows an example DFG where each node represents an operation, such as addition, multiplication, etc. Mapping of the DFG onto the CGRA has two components: placement and routing. The placement decides which PE will execute each operation, and routing makes sure that the data can be routed to the dependent operations in a timely manner.




(a) 2x2 CGRA (b) DFG example (c) DFG mapping example

Fig. 1.5: 2x2 CGRA, a DFG (dataflow graph), and the mapping

Fig.1.5c shows a possible mapping of the DFG in Fig.1.5b onto the CGRA in Fig.1.5a. For the sake of convenience, the 2x2 CGRA in Fig.1.5a has been drawn as a linear array. The mapping has three parts: prologue, steady state kernel, and epilogue. The prologue and epilogue are executed only once at the start and end of the loop execution. The steady state kernel is repeated and include all the operations from one or more iterations. The schedule length of the kernel is called the Initial Interval (II) and indicates the number of cycles between the initiation of consecutive loop iterations. For a loop with a large number of iterations, the execution time is dominated by the II value.

In the mapping of Fig.1.5c, II = 2. Notice that node *n5* of the first loop iteration is executing in the same cycle with *n1* and *n2* from the second loop iteration. Hence the CGRA can start a new loop iteration every two cycles leading to II value of two. The routing is done through the network among the PEs. This figure shows an abstract mapping for convenience. A real mapping will include the detailed routing configuration at each PE.

Given a DFG and a CGRA, the mapper first calculates the Minimum Initial Interval (MII), which is the maximum of the resource-minimal II and the recurrence-minimal II. The resource MII depends on the number of PEs and the number of DFG nodes (assume one PE can process one DFG node). Hence the resource MII cannot be less than the number of DFG nodes divided by the number of PEs. The recurrence MII is determined by the dependency across loop iterations. Let us assume that we have an operation $a[i] = a[i-1] \times b[i]$. The operation of iteration $i$ must wait for the result of the operation of last iteration *i-1*. The recurrence MII can be calculated by traversing the DFG.

Mapping a compute-intensive loop kernel of an application to CGRAs using modulo scheduling was first discussed in the DRESC compiler [45]. The algorithm starts with an II equal to the maximum between the resource-minimal II and recurrence-

minimal II and attempts to schedule the loop. If it fails, it tries with successively larger II values.

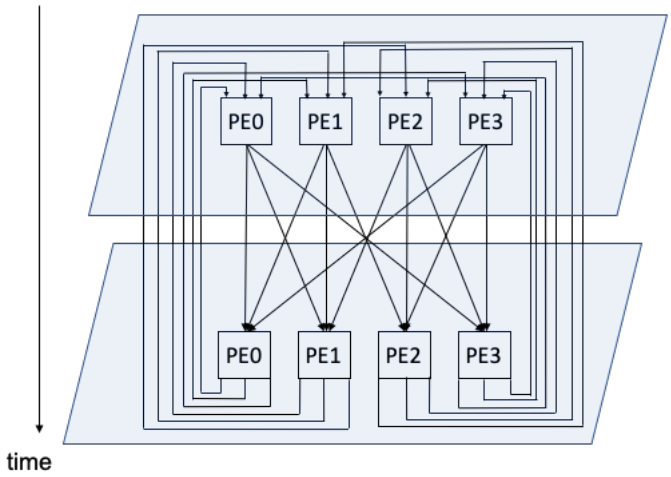



Fig. 1.6: An example of Modulo Routing Resource Graph (MRRG)

**Modulo Routing Resource Graph (MRRG)** Mei et al. [45] proposed the MRRG, which represents the resources and the routing for a time-extended CGRA. The nodes in MRRG represent the ports of the register file, the on-chip network, the ALU inside PE etc. The edges are the connections among the CGRA components represented as nodes. The MRRG is a directed graph $G_{II}$ where $II$ corresponds to the initiation interval. Given a graph $G$, let us denote the vertex set by $V(G)$ and the edge set by $E(G)$. Each node $v \in V(G_{II})$ is a tuple *(n, t)*, where $n$ refers to the resource in CGRA and $t$ is the cycle ($0 \leq t \leq II - 1$). Let $e = (u, v) \in E(G_{II})$ be an edge where $u =$ *(m, t)* and $v =$ *(n, t+1)*. Then the edge $e$ represents a connection from resource $m$ in cycle $t$ to resource $n$ in cycle *t+1*. In general, if resource $m$ is connected to resource $n$ in the CGRA, then node $u =$ *(m, t)* is connected to node $v =$ *(n, t+1)*, $t \geq 0$.

Fig.1.6 shows the MRRG corresponding to the CGRA in Fig. 1.5a when the II is 2. The resources of 2x2 CGRA are replicated every cycle along the time axis. In modulo scheduling, if a node *v=(n, t)* in the MRRG becomes occupied, then all the nodes *v'=(n, t+k×II)* (where $k > 0$) are also occupied. For example, in the Fig. 1.5c, *PE0* is occupied by node *n1* at cycle 0 and the II is 2. Thus the node will occupy *PE0* every $2 \times k$ cycle. Hence after cycle 1, the configuration in cycle 0 will be used to reconfigure the fabric, as the II is 2 and configuration has two items. Thus, there are wrap around edges from the second item back to the first one, as cycle 3 will use the first configuration item. These edges show hardware resource connection along the time axis.

### 1.4.2 CGRA Mapping Approaches

In this section, we present three broad classes of mapping approached based on heuristics, mathematical optimization, and graph theory inspired techniques.

#### 1.4.2.1 Heuristic Approaches

The heuristic approaches propose customized solutions for the CGRA mapping problem.

**Simulated Annealing**

Meta-heuristics are problem-independent approaches that treat the architectural elements as black boxes. Simulated Annealing is one of the most popular meta-heuristic method. Here, we introduce the usage of Simulated Annealing in CGRA mapping as proposed in the DRESC compiler [45]. For a target II value, the algorithm first generates an initial schedule satisfying the dependence constraints but with possibly over-subscribed resources. For example, more than one operations might be scheduled on the same functional unit in the same cycle. The algorithm then iteratively reduces resource overuse and tries to come up with a legal scheduling via simulated annealing that explores different placement and routing options until a valid placement and routing of all operations and data dependencies are found. The cost function used during the simulated annealing is based on the total routing cost, i.e., the combined resource consumption of all the placed operations and the routed data dependencies. In this technique, a huge number of possible routes are evaluated. As a result, the technique has long convergence time, especially for large dataflow graphs.

Routing through the register files and the register allocation problems are further explored in [46], which extends the work in [45]. Register allocation is achieved by constraining the register usage during the simulated annealing place and route process. The imposed constraint is adopted from the meeting graph [47] for solving loop cyclic register allocation in VLIW processors. In post routing phase, the registers are allocated by finding a Hamilton circuit in the meeting graph, which is solved as a traveling salesman problem [46]. This technique is specially designed for CGRAs with rotating register files. [48] and CGRA-ME[49] follow the simulated annealing framework but aim at finding better cost functions for over-used resources.

**Edge-Centric Modulo Scheduling**

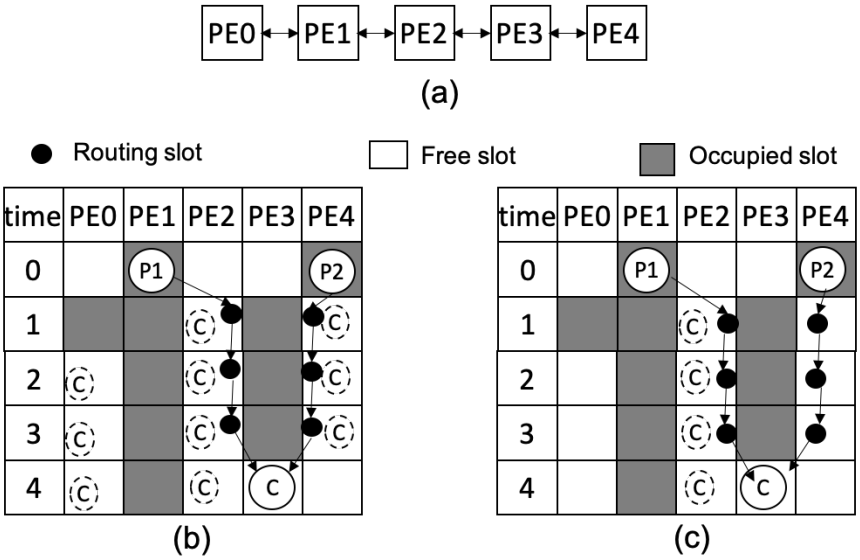



Fig. 1.7: Node-Centric (left) versus Edge-Centric (right) Modulo Scheduling [50]

The DRESC compiler performs node-centric modulo scheduling where the nodes are scheduled and placed first followed by routing of the edges. In contrast, for edge-centric modulo scheduling (EMS) [50], the primary objective is routing efficiency rather than operations assignment. Figure 1.7 taken from [50] shows the difference between the two approaches.

Node-centric approaches place an operation according to the heuristic routing cost. The cost consists of various metrics that reflects the quality of the mapping. The mapper visits the PE candidates and selects the best one, or visits the candidates one by one until it finds a solution. When visiting a candidate, the mapper will try to route the edges from the mapped nodes to the current candidate. Figure 1.7(b) shows how an optimal placement is found with this approach. A DFG including two producers P1 and P2 and a shared consumer C is mapped onto a $1 \times 5$ CGRA in Figure 1.7(a). P1 and P2 are already placed and the mapper places the consumer C by visiting all the empty slots as shown in Figure 1.7(b). The slots with dashed circles are failed attempts as the mapper cannot establish routing from producer P1 and P2. After visiting those slots, the mapper successfully places C on PE3 at time 4 and routes values from P1 and P2.

In an edge-centric approach, the routing function contains the placement of an operation, and the placement decision is made when the routing information is discovered. When scheduling an operation, the mapper picks an edge from the operation's previously-placed producers or consumers and starts routing the edge. The router will search for an empty slot which can execute the target operation. Once a suitable slot is found, the mapper will place the operation and route for other edges.

Figure 1.7(c) shows the same example of Figure 1.7(b), and the consumer is mapped using an edge-centric approach. The scheduler tries to route edge from P1 to C first, instead of placing operation C directly. When an empty slot is found, the scheduler temporarily places the target operation and checks if there are other edges connected to the consumer; if so, it recursively routes those edges. For example, when the router visits slot (PE2,1), it temporarily places C there and recursively calls the router function to route the edge from P2 to C. When it fails to route the edge from P2 to C, routing resumes from slot (PE2,1), and not from P1, and a solution is eventually found at slot (PE3,4).

In general, an edge-centric approach can find a solution faster and achieves better quality mapping compared to a node-centric approach. However, it has a greedy nature in that it optimizes for a single edge at a time, and the solution can easily fall into local minima. There is no search mechanism in the scheduler at the operation level and every decision made in each step is final. This problem can be addressed by employing intelligent routing cost metrics as priorities. The quality of a mapping using specific priorities highly depends on efficient heuristics for assigning these priority values to both the operations and the resources.

### Schedule, Place, and Route (SPR)

SPR [51] is a mature CGRA mapping tool that successfully combines the VLIW-style scheduler and FPGA placement and routing algorithms for CGRA application mapping. It consists of three individual steps namely scheduling (ordering operations in time based on data and control dependencies), placement (assigning operations to functional units), and routing (mapping data signals between operations using wires and registers). SPR uses Iterative Modulo Scheduling (IMS) [44], Simulated Annealing [52] placement with a cooling schedule inspired by VPR [53], and PathFinder [54] and QuickRoute [55] for pipelined routing.

IMS is a VLIW inspired loop instruction scheduling algorithm. IMS heuristically assigns operations to a schedule specifying the start time for each instruction, taking into account resource constraints and data and control dependencies. SPR uses IMS for initial operation scheduling and extends IMS to support rescheduling with feedback from the placement algorithm, letting it handle the configurable interconnects of CGRAs.

FPGA mapping tools historically use Simulated Annealing for placement and PathFinder for routing. VPR [53], which has become the de facto standard for FPGA architecture exploration, is similar to SPR in that it seeks to be a flexible and open mapping tool that can provide high quality mappings and support a wide spectrum of architectural features. Unfortunately, it only applies to FPGAs. SPR adopts similar algorithms but extended for CGRAs to support multiplexing of resources across cycles and solving the placement and routing issues that arise when using a fixed frequency device. SPR uses QuickRoute to solve the pipelined routing problem.

### List Scheduling

A list scheduling algorithm is adapted in the mapping algorithms of [56]. Priority-based list scheduling heuristic is used in [56] to map data-dependent operations in the kernel onto spatially close PEs in the CGRA. Each operation in the kernel is mapped onto a PE considering the operation priority and ability to route data from already mapped parent operations. They maintain a PE list based on topology traversal order and an operation list based on scheduling priority. Topology traversal order is the order in which PEs are traversed in the CGRA while mapping operations to PEs. The experiment results show that spiral traversal order performs best. According to a scheduling priority, the operation list is maintained, which gives preference to operations on the longest data dependency paths. Operation with the highest priority is mapped on the next available PE in the PE list if there is a valid route from already mapped parent operations. Scheduling is done on a cycle by cycle basis. Each cycle, the algorithm schedules operations on each PE and then increments the cycle when the PE list is exhausted. This process is continued until all the operations in the kernel have been scheduled. Unfortunately, the list scheduling algorithms do not produce a software pipelined schedule and are thus unable to exploit inter-iteration parallelism.

### Evolutionary Algorithm

The mapping approach in [57] presents a fast heuristic using a quantum-inspired evolutionary algorithm. This evolutionary algorithm uses an initial solution obtained from list scheduling as a starting point. The algorithm uses Q-Bit encoding to represent the hundreds of possible mapping results and evaluates each case to choose the best solution that satisfies the data dependency constraints. Q-Bit encoding allows compact maintenance of potential mappings, enabling fast design space exploration compared to other evolutionary algorithms. Fitness function is the performance, which is the inverse of the total latency. The algorithm iteratively improves the solution until it finds a solution with the lower bound of optimal latency or there is no improvement during a given time interval. However, the experimental evaluation is limited to small loop kernels with few DFG operations and CGRAs with limited reconfigurability.

### Machine Learning

A reinforcement learning-based mapping approach for CGRAs has been proposed in RLMap [58]. The CGRA mapping problem is formulated as an agent in reinforce-

ment learning. Each mapping state is represented as a distinct image that captures operation placement and inter-PE routing. Agent action is defined as the interchange of operations on neighbour PEs to keep the action space small. The reward function is defined based on a cost function that captures interconnect power requirements, utilized compute PEs, routing PEs, and empty PEs. Reward function helps the agent obtain valid and high-quality mapping in terms of power, area and performance.

Inspired by the progress in deep learning, [59] proposed DFGNet, a convolutional neural network-based mapping approach. They present dual input neural network to capture kernel DFG and CGRA architecture. CGRA mapping problem is translated into an image-based classification problem in a convolutional neural network. Input DFG is represented as an adjacency matrix, and a matrix represents the CGRA architecture state. The neural network consists of convolutional, max pooling, concatenate and fully connected layers. The issue with any deep learning method for application mapping on CGRAs is the difficulty in obtaining the abundant training data required for such approaches.

CGRAs differ in the network and the PE function. Existing compilers [41, 60, 61] usually leverage special characteristics of the architecture to generate quality mapping. These compilers, however, are usually hand-crafted, making it challenging from the time-to-market perspective. [62] proposed a portable framework, LISA, to map DFGs onto diverse CGRAs. LISA uses Graph Neural Network (GNN) to analyze Data Flow Graph (DFG) to derive *labels* that describe how the DFG should be mapped, e.g, the estimated routing resource required by an edge and the predicted mapping distance between the DFG nodes. With trained GNNs, these labels can reflect characteristics of the accelerator. Moreover, LISA provides a global view in mapping by describing the whole DFG and accelerator characteristics. For a new accelerator, the portable compiler re-trains the GNN model to adapt the labels according to the accelerator characteristics.

#### 1.4.2.2 Mathematical Optimization Techniques

We now present two mathematical optimization techniques for CGRA mapping.

**Integer Linear Programming (ILP)**

ILP-based formalization of the CGRA mapping problem has been proposed in the literature [63, 64]. The ILP formulation consists of all the requirements and the constraints that must be satisfied by a valid schedule. The formulation is built from the DFG and the MRRG and hence highly portable as shown recently in the CGRA-ME project [64]. However, it is not clear whether the ILP modeling can be effective for all possible architectural features and more importantly, scalability is a huge issue with ILP techniques that can only be applied to very simple loop kernels.

**Boolean Satisfiability (SAT) solvers**

A SAT solver based application mapping approach for CGRAs has been proposed by Wave Computing for their CGRA architecture [65]. This technique has been demonstrated to automatically compile dataflow programs onto a commercial massively parallel CGRA-based dataflow processor (DPU) containing 16,000 processing elements. This is an innovative application of Boolean Satisfiability to formally solve this complex and irregular optimization problem, and produce high-quality results comparable to hand-written assembly code produced by human experts. The approach is reported to be efficient in utilizing processing elements with rich micro-architectural features such as complex instructions, multi-precision data paths, local memories, register files, switches etc. However, the approach requires custom algorithms to handle the complexity of the SAT-based solutions in offering scalable, robust technique. A constraint-based approach is also used for the Silicon Hive CGRA architecture [66] though the details are not publicly available.

#### 1.4.2.3 Graph Theory Inspired Techniques

Many CGRA mapping approaches use graph theory concepts to formulate the CGRA mapping problem. Those approaches transform the CGRA mapping problem into well known graph theoretic formulations and leverage the existing techniques to solve the problem. This section categorises the graph theory inspired mapping techniques based on the graph theory formalism they use. We also discuss, in more detail, prominent CGRA mapping techniques that correspond to each formalisation. Table 1.1 summarises different aspects of five such notable works.

Table 1.1: Notable works using graph theory concepts for CGRA mapping problem

| Work | Graph Theory Concept | Solution | What is new? |
|---|---|---|---|
| [67] | Homeomorphism | Greedy algorithm for transformation | Mapping DFG substructures |
| EpiMap [68] | Epimorphism | Heuristic based search | Re-computation to solve out-degree problem |
| Graph Minor [69] | Graph Minor | Tree search method | Allow route sharing |
| RegiMap [60] | Compatibility graph | Finding a max clique | Allow both route sharing, recomputation |
| SPKM [70] | Graph Drawing | Split and Push approach | Support heterogeneous architectures |

Following graph theory concepts are widely used to formalize and solve CGRA application mapping problem:

1. Subgraph Homeomorphism
2. Graph Epimorphism
3. Graph Minor
4. Compatibility Graph
5. Graph drawing

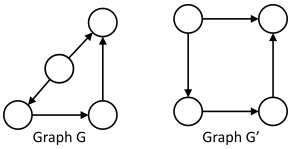



Fig. 1.8: Graph Isomorphism

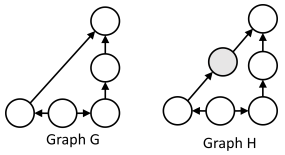



Fig. 1.9: Graph Subdvision

To understand the above graph theory concepts, we need to first present few related definitions. Therefore, let us first look at the definitions of graph isomorphism, graph subdivision, graph homeomorphism, and induced subgraph.

**Definition 1.1** A directed graph $G = (V, E)$ is a pair where $V$ is a set of vertices and $E \subseteq V \times V$ is a set of edges. Let $G$ and $G'$ be two graphs where $G = (V, E)$ and $G' = (V', E')$.

**Definition 1.2** Graph Isomorphism: An isomorphism from $G$ to $G'$ is a bijective function $f : V \to V'$ such that $(u, v) \in E \iff (f(u), f(v)) \in E'$.

Two graphs are isomorphic when both graphs contain the same number of vertices connected in the same way. Figure 1.8 shows two isormorphic graphs. Graph isomorphism is an equivalence relation on directed graphs.

**Definition 1.3** Graph Subdivision: The subdivision of some edge $e = (u, v) \in E$ yields a graph containing one new vertex $w$ and with an edge set replacing $e$ by two new edges, $(u, w)$ and $(w, v)$.

The definition of graph subdivision is self explanatory. In Figure 1.9, graph H is formed by subdivision of graph G.

**Definition 1.4** Graph Homeomorphism: Two graphs $G$ and $G'$ are homeomorphic if there is a graph isomorphism from some subdivision of $G$ to some subdivision of $G'$. In general, a subdivision of a graph $G$ is a graph resulting from the subdivision of edges in $G$.

In Figure 1.10, graph $H$ can be created by subdivision of edges of $G$ and also by subdivision of edges of $G'$. Therefore, graph $G$ and graph $G'$ are homeomorphic.

**Definition 1.5** Induced Subgraph: Let $U \subseteq V$ be a subset of vertices of $G$. The subgraph of $G$ induced by $U$ is $G_{\downarrow U} = (U, E \cap (U \times U))$.

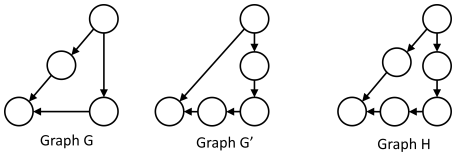



Fig. 1.10: Graph Homeomorphism

Induced subgraph is a graph formed by a subset of vertices of another graph with all of the edges that connect the vertices in that subset.

**Subgraph Homeomorphism Based Techniques**

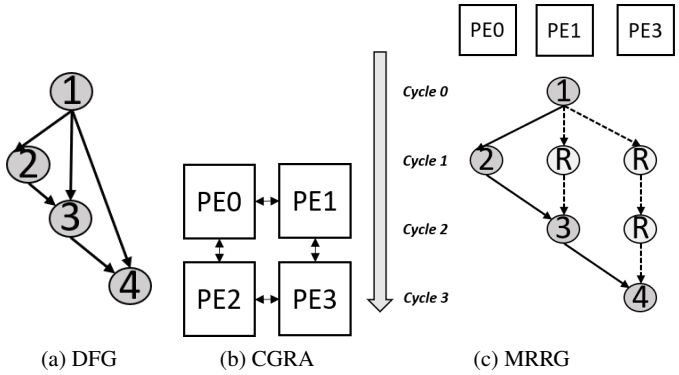
Formal definition of subgraph homeomorphism is as follows:

**Definition 1.6** Subgraph Homeomorphism: A subgraph homeomorphism from $G$ to $G'$ is a homeomorphism $f$ from an induced subgraph $G_{\downarrow U}$ of $G$ to $G'$.

Let $G$ be a directed graph representing the DFG and $H_{II}$ be a directed graph representing the MRRG with initiation interval II. In the ideal scenario of full connectivity among the PEs, we can map all the data dependencies in the DFG to direct edges in the MRRG. That is, for any edge $e = (u, v) \in E(G)$, there is an edge $e = (f(u), f(v)) \in E(H)$ where $f$ represents the vertex mapping function from the DFG to the MRRG. This matches the definition of subgraph isomorphism. However, data may need to be routed through a series of nodes rather than direct links in reality. If an edge $e = (u, v) \in E(G)$ in the DFG can be mapped to a path from $f(u)$ to $f(v)$ in the MRRG $H$, it matches the subgraph homeomorphism definition. The idea is to test if the DFG $G$ representing the loop kernel is subgraph homeomorphic to the MRRG $H_{II}$ representing the CGRA resources and their interconnects.

Figure 1.11 illustrates the subgraph homeomorphism formulation. Figure 1.11a shows a simple DFG being mapped onto a 2x2 homogeneous mesh CGRA shown in Figure 1.11b. The DFG is homeomorphic to the subgraph of the MRRG shown in Figure 1.11c and thus the subgraph represents a valid mapping (for simplicity we have removed additional nodes of the MRRG). In this homeomorphic mapping, edges (1,3) and (1,4) have been routed through three additional routing nodes marked by R. Notice that each routing node has degree 2 and has been added through edge subdivision (marked by dashed edges). Alternatively, we can smooth out the routing nodes in the MRRG subgraph to obtain the original DFG.

The subgraph homeomorphism techniques for CGRA mapping problem has been adopted in [71, 67, 72]. Authors in [71] formulate the mapping problem as finding a node disjoint subgraph homeomorphism between the DFG and the MRRG. The mapping algorithm is adapted from Modulo Scheduling with Integrated Register Spilling (MIRS) [73], a modulo scheduler capable of instruction scheduling with register constraints. [67] partitions the DFG into subgraphs called HyperOps, and




(a) DFG (b) CGRA (c) MRRG

Fig. 1.11: Subgraph homeomorphism formulation of CGRA mapping problem.

these HyperOps are synthesized into hardware configurations. The synthesis is carried out through a homeomorphic transformation of the dependency graph of each HyperOp onto the hardware resource graph. They employ a greedy algorithm for the transformation. [72] also formalizes the CGRA mapping as a subgraph homeomorphism problem. However, they consider general application kernels rather than loops.

However, subgraph homeomorphism requires the edge mappings to be node disjoint (except at endpoints) or edge-disjoint [74]. As a result, subgraph homeomorphism based techniques exclude the possibility of sharing the routing nodes among single source multiple target edges [75] (also called multi-net), leading to possible wastage of precious routing resources.

### Graph Epimorphism Based Technique

Graph epimorphism is defined based on graph homomorphism. Therefore, let us first look at the definition of graph homomorphism. A graph homomorphism defines a mapping between two graphs in which adjacent vertices in the first graph are mapped to adjacent vertices in the second graph. Unlike isomorphism, homomorphism can be from a bigger to a smaller graph. The formal definition of a homomorphism is as follows:

**Definition 1.7** Graph Homomorphism: A homomorphism from $G$ to $G'$ is a function $f : V \rightarrow V'$ such that $(u, v) \in E \implies (f(u), f(v)) \in E'$.

Graph epimorphism relaxes the bijection constraint of graph isomorphisms to a surjection constraint on both vertices and edges (hence the terminology of epimorphism). Several vertices of $G$ may be mapped on the same vertex of $G'$.

**Definition 1.8** Graph Epimorphism: An epimorphism from $G$ to $G'$ is a surjective function $f : V \rightarrow V'$ such that

- if $(u, v) \in E \implies (f(u), f(v)) \in E'$ (graph homomorphism)
- if $(u', v') \in E'$ then there exists $(u, v) \in E$ such that $f(u) = u'$ and $f(v) = v'$ (surjectivity on edges)

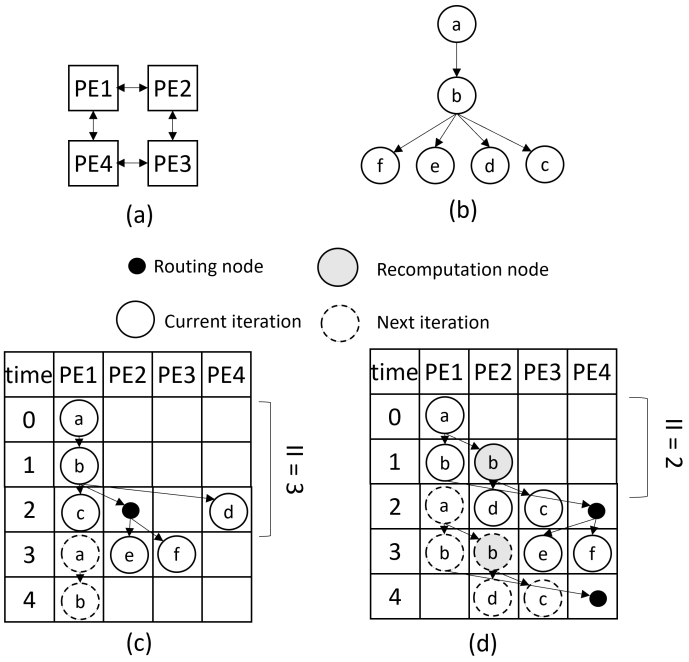



Fig. 1.12: A valid mapping without using re-computation (left) versus with re-computation (right) in EpiMap [68]

EPIMap [68] formalizes the CGRA mapping problem as a graph epimorphism problem with the additional feature of recomputations. Re-computation allows for the same operation to be performed on multiple PEs if it leads to better routing. In the EPIMap approach, the DFG $G$ is morphed into another graph $G'$ (through the introduction of routing/recomputation nodes and other transformations) such that there exists subgraph epimorphism from $G'$ to $G$ (many to one mapping of vertices from $G'$ to $G$ and adjacent vertices in $G'$ map to adjacent vertices in $G$). Then EPImap attempts to find the maximal common subgraph (MCS) between $G'$ and the MRRG graph $H$ using standard MCS identification procedure. If the resulting MCS is isomorphic to $G'$, then a valid mapping has been obtained; otherwise, $G$ is morphed differently in the next iteration and the process repeats. EPIMap can generate better scheduling results compared to EMS with a similar compilation time. Figure 1.12

taken from [68] shows the benefit of recomputation. The mapping in Figure 1.12 (d) computes the node $b$ in both PE1 and PE2 at cycle 1. This recomputation results in a better mapping (II=2) compared to mapping without recomputation (II=3) in Figure 1.12 (c).

### Graph Minor Based Technique

**Definition 1.9** Graph Minor: An undirected graph $G$ is called a minor of the graph $G'$ if $G$ is isomorphic to a graph that can be obtained by zero or more edge contractions on a subgraph of $G'$. An edge contraction is an operation that removes an edge from a graph while simultaneously merging together the two vertices it used to connect. A model of $G$ in $G'$ is a mapping $\phi$ that assigns to every edge $e \in E$ an edge $\phi(e) \in E'$, and to every vertex $v \in V$ a non-empty connected tree subgraph $\phi(v) \subseteq G'$ such that

- the graphs $\phi(v)|v \in V$ are mutually vertex-disjoint and the edges $\phi(e)|e \in E$ are pairwise distinct; and
- if $e = (u, v) \in E$, the edge $\phi(e)$ connects $\phi(u)$ with $\phi(v)$.

$G$ is isomorphic to a minor of $G'$ if and only if there exists a model of $G$ in $G'$ .

Graph minor [69] models the CGRA mapping problem as a graph minor containment problem that can explicitly model route sharing. As explained in the definition, a graph H is a minor of graph G if H can be obtained from a subgraph of G by a (possibly empty) sequence of edge contractions [76]. The graph minor is initially defined for undirected graphs, but the authors in [69] adapt the definition to directed graphs for CGRA mapping. In this context, we need to test if the DFG is a minor of the MRRG, where the edges to be contracted represent the routing paths in the MRRG. The mapping algorithm is inspired by the tree search method, which is widely used to solve graph matching problems. Unlike edge subdivision (or its reverse operation smoothing), edge contractions are not restricted to simple paths. Thus graph minor formalism naturally allows for route sharing. Figure 1.13 shows the difference between mappings under graph minor approach (Figure 1.13 (d)) and subgraph homeomorphism approach (Figure 1.13 (c)). The number of routing nodes are reduced from 3 (in subgraph homeomorphism mapping) to 2 (in graph minor mapping) through route sharing.

### Compatibilty Graph Based Technique

REGIMap [60] presents a general formulation of the problem of mapping a kernel on the CGRA while using its registers to minimize II. The formulation partitions the problem into a scheduling problem and an integrated placement and register

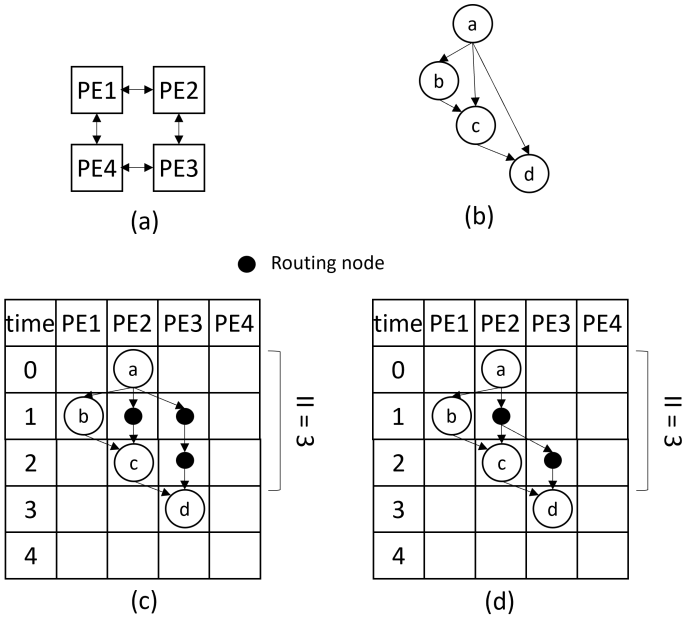



Fig. 1.13: Subgraph homeomorphism (left) versus Graph Minor formulation [69] (right) of CGRA mapping problem

allocation problem. They first create a compatibility graph, a subgraph of the product of the DFG $G$ and MRRG $H$. The vertices of the compatibility graph denote the operation-resource pair, which represent possible mapping pairs. The edges of the graph denote the compatibility of two corresponding mapping pairs. The mapping problem is reduced to one of finding the largest clique in the compatibility graph under the constraints of register resources. Then an efficient and constructive heuristic is used to solve the mapping problem.

### Graph Drawing Based Technique

SPKM [70] adopts the split and push technique [77] for planar graph drawing and focuses on spatial mappings for CGRAs. The mapping in SPKM starts from an initial drawing where all DFG nodes reside in the same group. One group represents a single processing element. The group is then split into two, and a set of nodes are pushed to the newly generated group. The split process continues until each group contains only one node, representing a one-to-one map ping from DFG to the planar resource graph of CGRA.

### 1.4.3 Other Compilation-related Issues

#### 1.4.3.1 Challenges Related to Data Access

The works presented for computation mapping have largely ignored the impact of data placement in the on-chip memory and the communication between the CPU and CGRA. Those works mostly assume data is already present in the local data memory. They also assume all PEs have access to all data memories, i.e., infinite memory bandwidth between local data memory and PE array. However, in reality, CGRA local memory bank has non-uniform memory access architecture where only a subset of the PEs have access to a memory bank with limited number of read/write ports [78]. Even when CGRA mapping achieves higher compute utilization under the assumption of ideal memory, the memory limitations could cause overall performance degradation in the actual setting. Thus, the compiler should be aware of the data memory limitations to minimize the effects of the memory bottleneck.

Figure 1.14a shows the simplified DFG of array-addition loop kernel. The kernel adds elements in two arrays $A[], B[]$ and stores the results in array $C[]$. Shaded nodes represent memory access operations; two load operations ($L$) and one store operation ($S$). Memory address of each array ($\&A[i], \&B[i], \&C[i]$) are computed in the nodes above the $L/S$ nodes based on the iteration variable $i$. Figure 1.14b shows the mapping of the DFG on CGRA coupled with on-chip local memory with four banks. Only boundary PEs have access to a directly connected memory bank. Array $A[], B[]$, and $C[]$ are placed in memory banks 1, 2, and 4 respectively. The CGRA mapper should be aware of the data placement to correctly place the load/store operations on the PEs. Therefore, data placement and CGRA mapping are interdependent tasks. Host CPU manages the data movement using a DMA controller based on the data placement decided by the compiler.

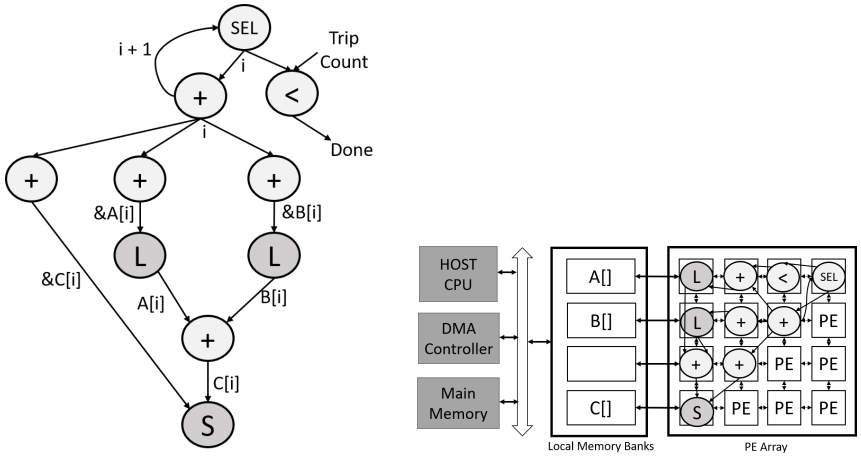
This section discusses compiler based solutions for challenges related to data access in CGRA.

**Memory Aware Compilation**

Effective memory-aware compiling strategy should

- Place the data without under-utilizing the memory banks
- Consider the limited connections between the PE array and the memory banks
- Prevent memory access conflicts
- Maximize data reuse by avoiding data duplication

[78] proposes a memory aware mapping solution that considers the effects of various memory architecture parameters, including the number of banks, local memory size and the communication bandwidth between the local memory and the external main memory. Their heuristic-based mapping approach considers minimizing duplicate arrays, balancing bank utilization, and balancing computation and data transfer time.




(a) DFG of the array-addition loop kernel (b) Mapping of array-addition loop kernel

Fig. 1.14: Memory-aware Loop Mapping on CGRA

Memory access conflict arises when the number of data accesses per bank per cycle is higher than the number of memory ports in one memory bank. Such access conflicts can be resolved either by data duplication or hardware queue with arbiters. Both approaches result in higher cost in terms of performance and power. A better solution would be to let the compiler partition the data into memory banks to avoid access conflicts. The application mapping technique in [79] consider the memory banking architecture and map operations and data to avoid memory bank conflicts. The initial schedule is generated without considering the data mapping. Subsequently, it uses array clustering and conflict-free scheduling until a conflict-free mapping is found. They also consider a hardware solution called DAMQ (Dynamically Allocated, Multi-Queue buffer), which uses a request queue and arbiter to resolve access conflicts. This hardware approach increases the access latency of the memory operation. They show the software solution is 8.5% better than a hardware solution. [80, 81] also proposes memory access conflict-free loop mapping strategy and a joint mapping flow by integrating modulo scheduling and memory partitioning. Dual-force directed scheduling algorithm is designed to solve the CGRA mapping problem and memory partitioning problem jointly.

When supporting kernels with multiple accesses for the same data array, a naive data placement in multi-bank memory could result in many access conflicts. [82] propose a memory partitioning scheme for multi-dimensional arrays based on a linear transformation. It partitions the multi-dimensional array among different banks to place each parallelly accessed data element in a separate memory bank.

[83] proposes memory aware mapping technique which uses shared data memory as the routing resource. They argue that routing some data dependencies through memory could improve the performance. Therefore, data dependencies that consume multiple PE routing resources are replaced by memory access operations. They

divide the mapping problem into two subproblems: 1) replacing the dependence with memory access operations, and 2) integrating placement and routing with the PE allocation for memory operations. Then, those two subproblems are solved to find a valid mapping. They establish a precise formulation for the CGRA mapping problem while using shared local data memory as a routing resource and present a practical approach for mapping loops to CGRAs.

**Memory Address Generation**

One other main challenge related to data access is the way data memory addresses are generated. [43] shows a substantial amount of instructions in loop kernels correspond to address generation (ranging from 20% to 80%). One solution is to offload the address generation to specialized address generation units since address generation involves a common operation pattern.

[84, 43] advocate the separation of execution and memory address generation due to the overhead of address generation in CGRA. [84] proposes a decoupled access-execute CGRA with complex on-chip memory hierarchy and a stream-based programming interface. In decoupled access-execute CGRA model, memory address generation is decoupled from the execution of main computation. [43] propose a decoupled access-execute CGRA with software and hardware support for conflict-free memory access.

#### 1.4.3.2 Nested Loop Mapping.

The application mapping approaches presented in the previous subsections consider a single innermost loop. In recent literature, several works explore the mapping of loop-nests beyond the innermost level.

There are two main motivations for going beyond the innermost loop level. First, nested loops offer more parallelism than what is available in a single innermost loop level. Therefore going beyond the innermost loop level could improve the available instruction-level parallelism. Secondly, the host processor needs to invoke the CGRA multiple times to support imperfect nested loops when only the innermost loop body is mapped to the CGRA. Multiple invocations lead to overheads, including pipeline filling/draining and the initialization of loop variables and pipeline parameters on the CGRA.

**Mapping Approaches**

We categorise existing works for nested loop mapping based on their approach.

**Polyhedral Model Based:** The polyhedral model is a robust framework that is widely used as a loop transformation technique. Polyhedral based transformations can be applied to nested loops where the loop bounds and array references are affine functions of loop iterators and program parameters. [85] use the polyhedral model to map the innermost two loops of multi-level loop nests. They use the polyhedral model to transform the two-dimensional nested loops to a new iteration domain with a parallelogram shape. Then they tile the parallelogram into multiple tiles where each tile consists of numerous iterations in the original program. Operators in each tile are mapped to CGRA using a place and route algorithm. The objective of the problem formulation is to reduce the total execution time by determining tile parameters. They adopt genetic algorithm to solve the problem.

**Loop Flattening Based** Loop flattening can convert imperfect loop nests into a single nested loop and can be executed in a single invocation. However, loop flattening comes with the overhead of increased code size. [86] argue that overhead from increased code size is lower than the overheads from multiple invocations. To limit the negative impact of loop flattening, they combine loop fission with flattening and introduce few specialized operations to CGRA PEs called nested iterators, extended accumulator, and periodic store.

**Systolic Mapping Based** HiMap [87, 88] proposes a hierarchical mapping approach to map regular multi-dimensional kernels on CGRAs. They use systolic mapping [89] as an intermediate abstraction layer to guide the hierarchical mapping. Each iteration in the multi-dimensional iteration space is mapped to a virtual PE cluster on CGRA based on a space-time mapping matrix derived from systolic mapping algorithm. Then operations in each iteration are mapped to physical CGRA PEs. They only generate detailed mapping for the few unique iterations with unique computation and routing patterns. The mappings of the unique iterations are replicated to obtain the final valid mapping. Therefore, HiMap is fast and scalable for mapping regular multi-dimensional kernels.

### Nested Loop Mapping Under Limited Configuration Memory

[90] proposes a method to map two of the innermost levels of loop-nests using less configuration memory compared to [86]. In this context, configuration memory size constraint presents a significant limitation. To solve this configuration memory capacity constraint, recent works [91, 92] propose architectural improvements to use the configuration memory as a cache that stores the most recently accessed loop-nests at runtime. The dynamic caching leads to performance improvement because more application segments can be accelerated, and the data transfer between the host and the CGRA is minimized. It is possible to naively employ caching within a loop-nest to expand the mappable loop-nests beyond the innermost loops. Still, the frequent context switching between outer and inner loops may incur significant overhead. DNestMap [40], a partitioning and mapping tool for CGRAs, can judiciously extract

the most beneficial code segments of multiple deeply-nested loops and effectively cache them together statically in the configuration memory through spatio-temporal partitioning.

#### 1.4.3.3 Application-level Mapping

An application usually has several kernels, which can be a sequential code, a single loop, or multiple loops, or any combination of them. CGRA can reconfigure the functionality of PE and the routing of on-chip network to accelerate any kernel. Application-Specific Integrated Circuits (ASICs) always target specific kernels and lose the flexibility to process other kernels. Field-Programmable Gate Array (FPGA) can be reconfigured to accelerate any kernel. Due to the time cost of reconfiguration, however, FPGA cannot change the configuration frequently to execute different kernels. When mapping an application, FPGA usually needs to map all the target kernels spatially, which is limited by the area and cannot do a spatio-temporal mapping. On the contrary, CGRA can re-configure the PE and on-chip network per cycle, thus leading to a spatio-temporal mapping.

**Partitioning between CPU and CGRA**

An application can have multiple kernels. Some kernels can significantly benefit from CGRA because of adequate instruction-level parallelism, while other kernels may not. With limited on-chip memory, offloading all the kernels to a tiny CGRA might be not a good choice as it might need to use main memory to store intermediate data. [93] explores how to execute the whole application onto CGRA and host processor. This work first profiles the execution time and memory requirement of each Kernel. Then it uses Integer Linear Programming (ILP) to select which kernel to execute on CGRA and which kernel to execute on host processor. Through this method, it can maximize the utilization of CGRA and reduce the data transfer between host processor and CGRA. CGRA can be reconfigured to execute the selected kernels. However, this work only focuses on a small $4 \times 4$ CGRA and did not explore how to run multiple kernels concurrently on CGRA.

**Synchronous DataFlow (SDF)**

Synchronous DataFlow (SDF) is a suitable representation for application-level mapping on CGRAs. The SDF has several actors, and data is encapsulated in an object called a token. Each actor either consumes data tokens or produces tokens or both in each invocation. An actor in an SDF can be a sequential code, a single loop, or multiple loops, or any combination of them. Fig. 1.15 shows a SDF example which has three actors: A, B, and C. Each invocation of A produces 20 tokens.

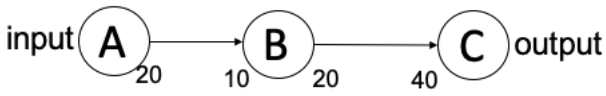
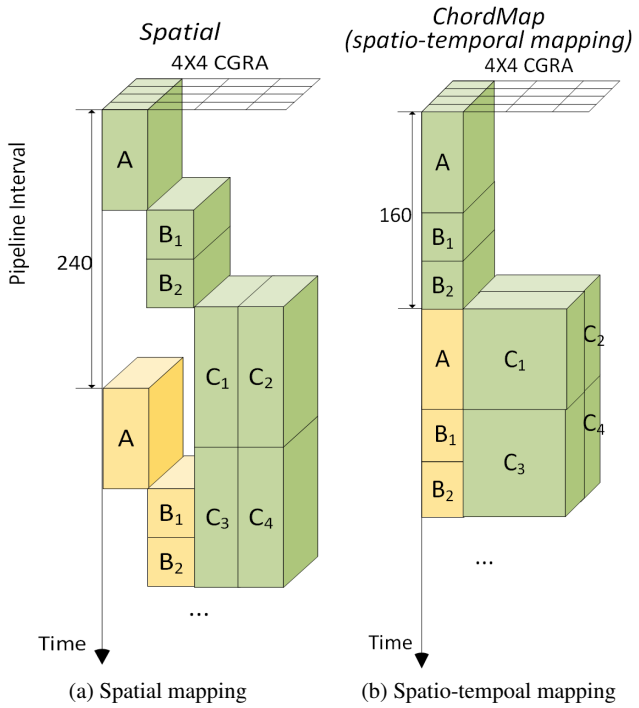



Fig. 1.15: SDF example

Each invocation of B consumer 10 tokens from A and produces 20 tokens. Thus the SDF needs a schedule which can balance the execution of actors. $A(BC^2)^2$ and $AB^2C^4$ are some of the many possible bounded-buffer schedules for our example, where $A(BC^2)^2$ indicates the execution order $ABCCBCC$ and $AB^2C^4$ represents $ABBCCCC$. These schedules trades-off between buffer requirement and throughput.




(a) Spatial mapping (b) Spatio-tempoal mapping

Fig. 1.16: Comparison between a Spatial and Spatio-Temporal mapping.

Fig. 1.16 shows the difference between spatial and spatio- temporal mapping. In FPGA, these actors are placed spatially and scheduled to respect to the data dependency among the actors. Each actor occupy a region throughout. However, this method cannot utilize the advantage the spatio-temporal mapping for CGRA. Specially, the SDF needs a schedule to orchestrate the actors to satisfy the data

dependencies. It is hard to achieve balanced execution of the actors under the SDF schedule constraints with spatial-only mapping. On the other side, the spatio-tempoal mapping of the actors provides a 3D space for the schedule that has more flexibility to map these actors.

ChordMap [94] explores the mapping of the SDF onto CGRA for high throughput, spatio-temporal mapping. Given a limited scratchpad memory for the buffers, ChordMap uses a divide-and-conquer approach to partition the SDF and CGRA to reduce the complextity. ChordMap maps each sub-SDF onto corresponding sub-CGRA and uses an iterative approach to improve the overall mapping. ChordMap can exploit the instruction level parallelism inside actor, the parallelism among actors and their instances, and the pipeline parallelism among sub-SDFs.

#### 1.4.3.4 Handling Loops with Control Flow

The statically scheduled CGRAs rely on predication to handle the loops with complex control flow [95]. Predication effectively translates the control flow instructions with dataflow instructions. The compiler maps both paths of each conditional branch onto the CGRA, but the instructions from the taken path are permitted to execute at run time. This leads to resource underutilization due to static allocation of duplicate resources which are unused at runtime. A recent work 4D-CGRA [96] proposes a new execution paradigm to handle control divergence at low overhead. 4D-CGRA architecture follows a semi-triggered execution model, a hybrid between sequential execution and triggered execution to accelerate loops with complex control flows. 4D-CGRA compiler places multiple shards of instructions (a portion of a basic block) from mutually exclusive execution paths on a PE and triggers a specific shard at runtime.

#### 1.4.3.5 Scalable CGRA Mapping.

Most CGRA mapping approaches are not scalable, i.e., they fail to generate high-quality mappings within an acceptable compilation time for larger CGRAs and complex application kernels. Operation placement and routing become increasingly difficult in larger CGRAs due to limited routing resources and complicated data dependencies in bigger kernels. Therefore, most CGRA mappers are only evaluated on small benchmark kernels and small CGRA sizes. Table 1.2 shows the DFG size, CGRA size, and compilation time of prominent CGRA mappers. SPR [51] is the most scalable compiler evaluated on benchmark kernels with an average of 263 nodes and 16x16 CGRA.

Panorama is a fast and scalable mapper that generates quality mapping for complex dataflow graphs onto larger CGRA using a divide-and-conquer approach [97]. It is a portable solution that can be combined with existing low-level CGRA mappers to achieve enhanced performance in a shorter compilation time. Panorama implements

Table 1.2: Summary of CGRA mappers.

| | DFG Nodes | CGRA Size | Compilation Time |
|---|---|---|---|
| CGRA-ME [49] | 12 | 4x4 | NA |
| SPKM [70] | 16 | 4x4 | ~1s |
| G-Minor [69] | 35 | 4x4, 16x16 | 0.2s, 7s |
| EPIMAP [68] | 35 | 4x4, 16x16 | 54s, 23min |
| DRESC [101] | 56 | 4x4 | ~15min |
| EMS [75] | 4~142 | 4x4 | ~37min |
| SPR [51] | 263 | 16x16 | NA |

a high-level mapping step that finds clusters of nodes in the dataflow graph and performs cluster level mapping to place closely related clusters on nearby CGRA PE clusters. The higher-level mapping guides the lower-level mapping, reducing overall complexity. HiMap is another fast and scalable mapping technique, although it is only specialized for mapping regular highly parallel kernels, as mentioned in the nested loop mapping section [87, 88]. Similar approaches of exploring multi-level parallelism has been studied in the context of FPGAs [98, 99, 100].

## 1.5 Conclusions

Coarse-Grained Reconfigurable Array (CGRA) has emerged as a popular, general-purpose, spatial accelerator that can support high-performance, energy-efficiency, and flexibility across multiple application domains. In this article, we presented an overview of the architectural and compilation innovations over the last two decades to better realize the potential of the CGRAs. There remain multiple challenges and opportunities in this space, including but not limited to, scalability for more complex applications, memory management, runtime power management, and specializations for important and emerging application domains.

## 1.6 Acknowledgement

This work is partially supported by the National Research Foundation, Singapore under its Competitive Research Programme Award NRF-CRP23-2019-0003.

## References

1. J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.
2. R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose

chips," in *Proceedings of the 37th annual international symposium on Computer architecture*, 2010, pp. 37–47.
3. W. J. Dally, Y. Turakhia, and S. Han, "Domain-specific hardware accelerators," *Communications of the ACM*, vol. 63, no. 7, pp. 48–57, 2020.
4. T. Mitra, "Heterogeneous multi-core architectures," *Information and Media Technologies*, vol. 10, no. 3, pp. 383–394, 2015.
5. B. Kågström, P. Ling, and C. Van Loan, "Gemm-based level 3 blas: high-performance model implementations and performance evaluation benchmark," *ACM Transactions on Mathematical Software (TOMS)*, vol. 24, no. 3, pp. 268–302, 1998.
6. G. E. Moore *et al.*, "Cramming more components onto integrated circuits," 1965.
7. R. H. Dennard, F. H. Gaensslen, H.-N. Yu, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted mosfet's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.
8. D. A. Patterson, "Future of computer architecture," in *Berkeley EECS Annual Research Symposium (BEARS), College of Engineering, UC Berkeley, US*, 2006.
9. G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*, 1967, pp. 483–485.
10. N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th annual international symposium on computer architecture*, 2017, pp. 1–12.
11. J. Ghorpade, J. Parande, M. Kulkarni, and A. Bawaskar, "Gpgpu processing in cuda architecture," *arXiv preprint arXiv:1202.4347*, 2012.
12. M. Rashid, M. Imran, A. R. Jafri, and T. F. Al-Somani, "Flexible architectures for cryptographic algorithms—a systematic literature review," *Journal of Circuits, Systems and Computers*, vol. 28, no. 03, p. 1930003, 2019.
13. Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks, "The aladdin approach to accelerator design and modeling," *IEEE Micro*, vol. 35, no. 3, pp. 58–70, 2015.
14. J.-A. Carballo, W.-T. J. Chan, P. A. Gargini, A. B. Kahng, and S. Nath, "Itrs 2.0: Toward a re-framing of the semiconductor technology roadmap," in *2014 IEEE 32nd International Conference on Computer Design (ICCD)*. IEEE, 2014, pp. 139–146.
15. K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software," *ACM Computing Surveys (csuR)*, vol. 34, no. 2, pp. 171–210, 2002.
16. I. Kuon, R. Tessier, and J. Rose, *FPGA architecture: Survey and challenges*. Now Publishers Inc, 2008.
17. I. Kuon and J. Rose, "Measuring the gap between fpgas and asics," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 203–215, 2007.
18. K. Choi, "Coarse-grained reconfigurable array: Architecture and application mapping," *IPSJ Transactions on System LSI Design Methodology*, vol. 4, pp. 31–46, 2011.
19. H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. Chaves Filho, "Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications," *IEEE transactions on computers*, vol. 49, no. 5, pp. 465–481, 2000.
20. B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix," in *Proceedings of the 13th International Conference on Field Programmable Logic and Application*, ser. FPL'03. Springer, 2003, pp. 61–70.
21. V. Baumgarte, G. Ehlers, F. May, A. Nückel, M. Vorbach, and M. Weinhardt, "Pact xpp—a self-reconfigurable data processing architecture," *the Journal of Supercomputing*, vol. 26, no. 2, pp. 167–184, 2003.
22. "Darpa software defined hardware." [Online]. Available: https://www.darpa.mil/program/software-defined-hardware

23. M. Gao and C. Kozyrakis, "Hrl: Efficient and flexible reconfigurable logic for near-data processing," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Ieee, 2016, pp. 126–137.
24. R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, "Plasticine: A reconfigurable architecture for parallel patterns," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 389–402.
25. M. Karunaratne, A. K. Mohite, T. Mitra, and L.-S. Peh, "Hycube: A cgra with reconfigurable single-cycle multi-hop interconnect," in *Design Automation Conference (DAC), 2017 54th ACM/EDAC/IEEE*. IEEE, 2017, pp. 1–6.
26. C. Nicol, "A coarse grain reconfigurable array (cgra) for statically scheduled data flow computing," *Wave Computing White Paper*, 2017.
27. M. Emani, V. Vishwanath, C. Adams, M. E. Papka, R. Stevens, L. Florescu, S. Jairath, W. Liu, T. Nama, and A. Sujeeth, "Accelerating scientific applications with sambanova reconfigurable dataflow architecture," *Computing in Science & Engineering*, vol. 23, no. 2, pp. 114–119, 2021.
28. D. Suh, K. Kwon, S. Kim, S. Ryu, and J. Kim, "Design space exploration and implementation of a high performance and low area coarse grained reconfigurable processor," in *2012 international conference on field-programmable technology*. IEEE, 2012, pp. 67–70.
29. T. Fujii, T. Toi, T. Tanaka, K. Togawa, T. Kitaoka, K. Nishino, N. Nakamura, H. Nakahara, and M. Motomura, "New generation dynamically reconfigurable processor technology for accelerating embedded ai applications," in *2018 IEEE Symposium on VLSI Circuits*. IEEE, 2018, pp. 41–42.
30. K. E. Fleming, K. D. Glossop, S. C. Steely Jr, J. Tang, A. G. Gara *et al.*, "Processors, methods, and systems with a configurable spatial accelerator," Feb. 11 2020, uS Patent 10,558,575.
31. Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 292–308, 2019.
32. W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, "Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 553–564.
33. F. Tu, S. Yin, P. Ouyang, S. Tang, L. Liu, and S. Wei, "Deep convolutional neural network architecture with reconfigurable computation patterns," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 8, pp. 2220–2233, 2017.
34. J. Kwong and A. P. Chandrakasan, "An energy-efficient biomedical signal processing platform," *IEEE Journal of Solid-State Circuits*, vol. 46, no. 7, pp. 1742–1753, 2011.
35. J. Yoo, L. Yan, D. El-Damak, M. A. B. Altaf, A. H. Shoeb, and A. P. Chandrakasan, "An 8-channel scalable eeg acquisition soc with patient-specific seizure classification and recording processor," *IEEE journal of solid-state circuits*, vol. 48, no. 1, pp. 214–228, 2012.
36. L. Liu, J. Zhu, Z. Li, Y. Lu, Y. Deng, J. Han, S. Yin, and S. Wei, "A survey of coarse-grained reconfigurable architecture and design: Taxonomy, challenges, and applications," *ACM Computing Surveys (CSUR)*, vol. 52, no. 6, pp. 1–39, 2019.
37. A. Podobas, K. Sano, and S. Matsuoka, "A survey on coarse-grained reconfigurable architectures from a performance perspective," *IEEE Access*, vol. 8, pp. 146 719–146 743, 2020.
38. S. Venkataramani, J. Choi, V. Srinivasan, W. Wang, J. Zhang, M. Schaal, M. J. Serrano, K. Ishizaki, H. Inoue, E. Ogawa *et al.*, "Deeptools: Compiler and execution runtime extensions for rapid ai accelerator," *IEEE Micro*, vol. 39, no. 5, pp. 102–111, 2019.
39. T. M. L.-S. P. Thilini Kaushalya Bandara, Dhananjaya Wijerathne, "Revamp: A systematic framework for heterogeneous cgra realization," in *27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2022.
40. M. Karunaratne, C. Tan, A. Kulkarni, T. Mitra, and L.-S. Peh, "Dnestmap: mapping deeply-nested loops on ultra-low power cgras," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE, 2018, pp. 1–6.

41. B. Wang, M. Karunarathne, A. Kulkarni, T. Mitra, and L.-S. Peh, "Hycube: A 0.9 v 26.4 mops/mw, 290 pj/op, power efficient accelerator for iot applications," in *2019 IEEE Asian Solid-State Circuits Conference (A-SSCC)*. IEEE, 2019, pp. 133–136.
42. H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. Chaves Filho, "MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications," *IEEE Transactions on Computers*, vol. 49, no. 5, pp. 465–481, 2000.
43. D. Wijerathne, Z. Li, M. Karunarathne, A. Pathania, and T. Mitra, "Cascade: High throughput data streaming via decoupled access-execute cgra," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 5s, pp. 1–26, 2019.
44. B. R. Rau, "Iterative modulo scheduling: An algorithm for software pipelining loops," in *Proceedings of the 27th annual international symposium on Microarchitecture*. ACM, 1994, pp. 63–74.
45. B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling," in *Proceedings of the 2003 Conference on Design, Automation and Test in Europe*, ser. DATE'03. IEEE, 2003, pp. 296–301.
46. B. De Sutter, P. Coene, T. Vander Aa, and B. Mei, "Placement-and-routing-based register allocation for coarse-grained reconfigurable arrays," in *Proceedings of the 2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers and Tools for Embedded System*, ser. LCTES'08. ACM, 2008, pp. 151–160.
47. C. Eisenbeis, S. Lelait, and B. Marmol, "The meeting graph: a new model for loop cyclic register allocation," in *Proceedings of the 1995 International Federation for Information Processing Working Group*, 1995, pp. 264–267.
48. A. Hatanaka and N. Bagherzadeh, "A modulo scheduling algorithm for a coarse-grain reconfigurable array template," in *Proceedings of the 21th International Parallel and Distributed Processing Symposium*, ser. IPDPS'07. IEEE, 2007, pp. 1–8.
49. S. A. Chin, N. Sakamoto, A. Rui, J. Zhao, J. H. Kim, Y. Hara-Azumi, and J. Anderson, "Cgrame: A unified framework for cgra modelling and exploration," in *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2017, pp. 184–189.
50. H. Park, K. Fan, S. A. Mahlke, T. Oh, H. Kim, and H.-s. Kim, "Edge-centric modulo scheduling for coarse-grained reconfigurable architectures," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT'08. ACM, 2008, pp. 166–176.
51. S. Friedman, A. Carroll, B. Van Essen, B. Ylvisaker, C. Ebeling, and S. Hauck, "SPR: an architecture-adaptive CGRA mapping tool," in *Proceedings of the 17th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA'09. ACM, 2009, pp. 191–200.
52. S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *science*, vol. 220, no. 4598, pp. 671–680, 1983.
53. V. Betz and J. Rose, "Vpr: A new packing, placement and routing tool for fpga research," in *International Workshop on Field Programmable Logic and Applications*. Springer, 1997, pp. 213–222.
54. L. McMurchie and C. Ebeling, "Pathfinder: a negotiation-based performance-driven router for fpgas," in *Reconfigurable Computing*. Elsevier, 2008, pp. 365–381.
55. S. Li and C. Ebeling, "Quickroute: a fast routing algorithm for pipelined architectures," in *Field-Programmable Technology, 2004. Proceedings. 2004 IEEE International Conference on*. IEEE, 2004, pp. 73–80.
56. N. B. S. G. N. Dutt and A. Nicolau, "Analysis of the performance of coarse-grain reconfigurable architectures with different processing element configurations."
57. G. Lee, K. Choi, and N. D. Dutt, "Mapping multi-domain applications onto coarse-grained reconfigurable architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 5, pp. 637–650, 2011.

58. D. Liu, S. Yin, G. Luo, J. Shang, L. Liu, S. Wei, Y. Feng, and S. Zhou, "Data-flow graph mapping optimization for cgra with deep reinforcement learning," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 12, pp. 2271–2283, 2018.
59. S. Yin, D. Liu, L. Sun, L. Liu, and S. Wei, "Dfgnet: Mapping dataflow graph onto cgra by a deep learning approach," in *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2017, pp. 1–4.
60. M. Hamzeh, A. Shrivastava, and S. Vrudhula, "Regimap: Register-aware application mapping on coarse-grained reconfigurable architectures (cgras)," in *Proceedings of the 50th Annual Design Automation Conference*, 2013, pp. 1–10.
61. S. Dave, M. Balasubramanian, and A. Shrivastava, "Ramp: Resource-aware mapping for cgras," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE, 2018, pp. 1–6.
62. Z. Li, D. Wu, D. Wijerathne, and T. Mitra, "Lisa: Graph neural network based portable mapping on spatial accelerators," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2022.
63. M. Ahn, J. W. Yoon, Y. Paek, Y. Kim, M. Kiemb, and K. Choi, "A spatial mapping algorithm for heterogeneous coarse-grained reconfigurable architectures," in *Proceedings of the conference on Design, automation and test in Europe: Proceedings*. European Design and Automation Association, 2006, pp. 363–368.
64. S. A. Chin and J. H. Anderson, "An architecture-agnostic integer linear programming approach to cgra mapping," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE, 2018, pp. 1–6.
65. S. Chaudhuri and A. Hetzel, "Sat-based compilation to a non-von neumann processor," in *Proceedings of the 36th International Conference on Computer-Aided Design*. IEEE Press, 2017, pp. 675–682.
66. G. F. Burns, M. Jacobs, M. Lindwer, and B. Vandewiele, "Exploiting parallelism, while managing complexity using silicon hive programming tools," *White paper*, vol. 42, p. 43, 2004.
67. M. Alle, K. Varadarajan, R. C. Ramesh, J. Nimmy, A. Fell, A. Rao, S. Nandy, and R. Narayan, "Synthesis of application accelerators on runtime reconfigurable hardware," in *2008 International Conference on Application-Specific Systems, Architectures and Processors*. IEEE, 2008, pp. 13–18.
68. M. Hamzeh, A. Shrivastava, and S. Vrudhula, "Epimap: Using epimorphism to map applications on cgras," in *Proceedings of the 49th Annual Design Automation Conference*, 2012, pp. 1284–1291.
69. L. Chen and T. Mitra, "Graph minor approach for application mapping on cgras," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 7, no. 3, pp. 1–25, 2014.
70. J. W. Yoon, A. Shrivastava, S. Park, M. Ahn, and Y. Paek, "A graph drawing based spatial mapping algorithm for coarse-grained reconfigurable architectures," *IEEE transactions on very large scale integration (VLSI) systems*, vol. 17, no. 11, pp. 1565–1578, 2009.
71. M. A. A. Tuhin and T. S. Norvell, "Compiling parallel applications to coarse-grained reconfigurable architectures," in *2008 Canadian Conference on Electrical and Computer Engineering*. IEEE, 2008, pp. 001 723–001 728.
72. J. A. Brenner, S. P. Fekete, and J. C. Van Der Veen, "A minimization version of a directed subgraph homeomorphism problem," *Mathematical Methods of Operations Research*, vol. 69, no. 2, pp. 281–296, 2009.
73. J. Zalamea, J. Llosa, E. Ayguadé, and M. Valero, "Mirs: Modulo scheduling with integrated register spilling," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2001, pp. 239–253.
74. S. Fortune, J. Hopcroft, and J. Wyllie, "The directed subgraph homeomorphism problem," *Theoretical Computer Science*, vol. 10, no. 2, pp. 111–121, 1980.

75. H. Park, K. Fan, S. A. Mahlke, T. Oh, H. Kim, and H.-s. Kim, "Edge-centric modulo scheduling for coarse-grained reconfigurable architectures," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008, pp. 166–176.
76. N. Robertson and P. D. Seymour, "Graph minors. ix. disjoint crossed paths," *Journal of Combinatorial Theory, Series B*, vol. 49, no. 1, pp. 40–77, 1990.
77. G. Di Battista, M. Patrignani, and F. Vargiu, "A split&push approach to 3d orthogonal drawing," in *International Symposium on Graph Drawing*. Springer, 1998, pp. 87–101.
78. Y. Kim, J. Lee, A. Shrivastava, J. Yoon, and Y. Paek, "Memory-aware application mapping on coarse-grained reconfigurable arrays," in *International conference on High-Performance Embedded Architectures and Compilers*. Springer, 2010, pp. 171–185.
79. Y. Kim, J. Lee, A. Shrivastava, and Y. Paek, "Operation and data mapping for cgras with multi-bank memory," *ACM Sigplan Notices*, vol. 45, no. 4, pp. 17–26, 2010.
80. S. Yin, X. Yao, T. Lu, D. Liu, J. Gu, L. Liu, and S. Wei, "Conflict-free loop mapping for coarse-grained reconfigurable architecture with multi-bank memory," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 9, pp. 2471–2485, 2017.
81. S. Yin, X. Yao, T. Lu, L. Liu, and S. Wei, "Joint loop mapping and data placement for coarse-grained reconfigurable architecture with multi-bank memory," in *Proceedings of the 35th International Conference on Computer-Aided Design*, 2016, pp. 1–8.
82. Y. Wang, P. Li, P. Zhang, C. Zhang, and J. Cong, "Memory partitioning for multidimensional arrays in high-level synthesis," in *Proceedings of the 50th Annual Design Automation Conference*, 2013, pp. 1–8.
83. S. Yin, X. Yao, D. Liu, L. Liu, and S. Wei, "Memory-aware loop mapping on coarse-grained reconfigurable architectures," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 5, pp. 1895–1908, 2015.
84. T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, "Stream-dataflow acceleration," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 416–429.
85. D. Liu, S. Yin, L. Liu, and S. Wei, "Polyhedral model based mapping optimization of loop nests for cgras," in *Proceedings of the 50th Annual Design Automation Conference*. ACM, 2013, p. 19.
86. J. Lee, S. Seo, H. Lee, and H. U. Sim, "Flattening-based mapping of imperfect loop nests for cgras," in *Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis*. ACM, 2014, p. 9.
87. D. Wijerathne, Z. Li, A. Pathania, T. Mitra, and L. Thiele, "Himap: Fast and scalable high-quality mapping on cgra via hierarchical abstraction," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.
88. ——, "Himap: Fast and scalable high-quality mapping on cgra via hierarchical abstraction," pp. 1192–1197, 2021.
89. P. Lee and Z. M. Kedem, "Mapping Nested Loop Algorithms into Multidimensional Systolic Arrays," *IEEE transactions on Parallel and Distributed Systems*, vol. 1, no. 1, pp. 64–76, 1990.
90. S. Yin, X. Lin, L. Liu, and S. Wei, "Exploiting parallelism of imperfect nested loops on coarse-grained reconfigurable architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 11, pp. 3199–3213, 2016.
91. P. Cao, B. Liu, J. Yang, J. Yang, M. Zhang, and L. Shi, "Context management scheme optimization of coarse-grained reconfigurable architecture for multimedia applications," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 17, 2017.
92. S. M. A. H. Jafri, M. A. Tajammul, A. Hemani, K. Paul, J. Plosila, P. Ellervee, and H. Tenuhnen, "Polymorphic configuration architecture for cgras," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 1, pp. 403–407, 2015.
93. H. Lee, D. Nguyen, and J. Lee, "Optimizing stream program performance on cgra-based systems," in *Proceedings of the 52nd Annual Design Automation Conference*, 2015, pp. 1–6.

94. Z. Li, D. Wijerathne, X. Chen, A. Pathania, and T. Mitra, "Chordmap: Automated mapping of streaming applications onto cgra," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.
95. K. Han, J. Ahn, and K. Choi, "Power-efficient predication techniques for acceleration of control flow execution on cgra," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 10, no. 2, pp. 1–25, 2013.
96. M. Karunaratne, D. Wijerathne, T. Mitra, and L.-S. Peh, "4d-cgra: Introducing branch dimension to spatio-temporal application mapping on cgras," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2019, pp. 1–8.
97. D. Wijerathne, Z. Li, T. K. Bandara, and T. Mitra, "Panorama: Divide-and-conquer approach for mapping complex loop kernels on cgra," in *2022 59th ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2022, pp. 1–6.
98. G. Zhong, V. Venkataramani, Y. Liang, T. Mitra, and S. Niar, "Design space exploration of multiple loops on fpgas using high level synthesis," in *2014 IEEE 32nd international conference on computer design (ICCD)*. IEEE, 2014, pp. 456–463.
99. G. Zhong, A. Prakash, Y. Liang, T. Mitra, and S. Niar, "Lin-analyzer: A high-level performance analysis tool for fpga-based accelerators," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2016, pp. 1–6.
100. G. Zhong, A. Prakash, S. Wang, Y. Liang, T. Mitra, and S. Niar, "Design space exploration of fpga-based accelerators with multi-level parallelism," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017, pp. 1141–1146.
101. B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "DRESC: A retargetable compiler for coarse-grained reconfigurable architectures," in *2002 IEEE International Conference on Field-Programmable Technology, 2002.(FPT). Proceedings.* IEEE, 2002, pp. 166–173.