Enhancing CGRA Efficiency Through Aligned Compute and Communication Provisioning

Zhaoying Li*
zhaoying@comp.nus.edu.sg
National University of Singapore
Singapore

Pranav Dangi*
dangi@comp.nus.edu.sg
National University of Singapore
Singapore

Chenyang Yin
ycy@stu.pku.edu.cn
Peking University
China

Thilini Kaushalya Bandara
thilini@comp.nus.edu.sg
National University of Singapore
Singapore

Rohan Juneja
rohan@comp.nus.edu.sg
National University of Singapore
Singapore

Cheng Tan
chengtan@google.com
Google
United States

Zhenyu Bai†
zhenyu.bai@nus.edu.sg
National University of Singapore
Singapore

Tulika Mitra
tulika@comp.nus.edu.sg
National University of Singapore
Singapore

Abstract

Coarse-grained Reconfigurable Arrays (CGRAs) are domain-agnostic accelerators that enhance the energy efficiency of resource-constrained edge devices. The CGRA landscape is diverse, exhibiting trade-offs between performance, efficiency, and architectural specialization. However, CGRAs often overprovision communication resources relative to their modest computing capabilities. This occurs because the theoretically provisioned programmability for CGRAs often proves superfluous in practical implementations.

In this paper, we propose *Plaid*, a novel CGRA architecture and compiler that aligns compute and communication capabilities, thereby significantly improving energy and area efficiency while preserving its generality and performance. We demonstrate that the dataflow graph, representing the target application, can be decomposed into smaller, recurring communication patterns called *motifs*. The primary contribution is the identification of these structural motifs within the dataflow graphs and the development of an efficient collective execution and routing strategy tailored to these motifs. The *Plaid* architecture employs a novel collective processing unit that can execute multiple operations of a motif and route related data dependencies together. The *Plaid* compiler can hierarchically map the dataflow graph and judiciously schedule the motifs. Our design achieves a 43% reduction in power consumption and 46% area savings compared to the baseline high-performance spatio-temporal CGRA, all while preserving its generality and performance levels. In comparison to the baseline energy-efficient spatial CGRA, *Plaid* offers a 1.4× performance improvement and a 48% area savings, with almost the same power.

CCS Concepts: • **Computer systems organization** → *Multicore architectures*; • **Reconfigurable computing**; • **Data flow architectures**;

Keywords: Dataflow computing, Coarse-Grained Reconfigurable Array (CGRA), Motifs

ACM Reference Format:
Zhaoying Li, Pranav Dangi, Chenyang Yin, Thilini Kaushalya Bandara, Rohan Juneja, Cheng Tan, Zhenyu Bai, and Tulika Mitra. 2025. Enhancing CGRA Efficiency Through Aligned Compute and Communication Provisioning. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3669940.3707230

1 Introduction

The surge in new applications like Machine Learning has led to the rapid development of accelerators tailored to specific domains or frequently used computational kernels [11]. Although these specific accelerators excel in performance and power efficiency, edge devices cannot accommodate or utilize many accelerators due to strict power and area constraints [17, 20]. Coarse-grained Reconfigurable Arrays (CGRAs) [12, 35] offer a balanced solution for performance, efficiency, and programmability, making them ideal for edge acceleration.

Several academic [8, 18, 23, 30, 34, 36, 39, 40, 47, 55, 57–59, 61, 63, 66, 69, 71–76] and commercial [19, 21, 31, 52]

*Both authors contributed equally to this research
†Corresponding Author

ASPLOS '25, March 30–April 3, 2025, Rotterdam, Netherlands
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0698-1/25/03.
https://doi.org/10.1145/3669940.3707230

CGRA Architecture	Performance	Energy Efficiency	Generality
Spatio-temporal UE-CGRA [63], HyCUBE [30] ADRES [39], MorphoSys [56]	High	Low	High
Spatial SNAFU [22], Riptide [23]	Medium* or High	High	Medium
Specialized REVAMP [62], REVEL [69] VecPac [58], APEX [40]	High or Ultra-High	High	Low
Ours (Plaid)	High	High	High

* Performance degrades when partitioning complex DFGs to be mapped spatially

Table 1. Reconfigurable architecture landscape

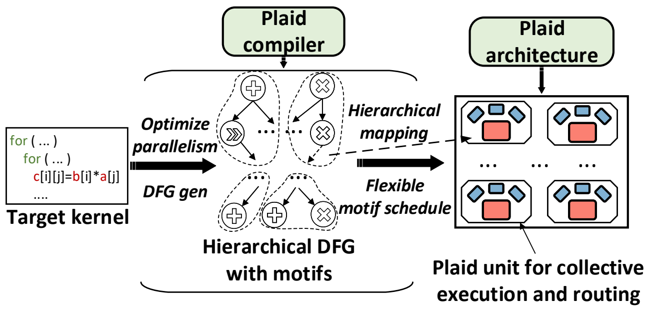


Figure 1. Overview of Plaid co-designed architecture and compiler

CGRA architectures have been proposed over the years. A typical CGRA consists of an array of Processing Elements (PEs) connected by an on-chip network (NoC). Each PE comprises a computing unit (typically an ALU), registers to store the temporal data, a router for interconnections, and a configuration memory to store the instructions statically generated by the compiler for the above hardware modules. The compiler maps the target application, represented by a Dataflow Graph (DFG), onto the CGRA. Computing units execute the DFG nodes, while routers and registers handle the data dependency, i.e., the edges representing communication among these nodes.

The existing landscape of CGRAs navigates trade-offs between performance, efficiency, and generality, where generality refers to the versatility of the scope of target applications or kernels that can be mapped onto the architecture. As shown in Table 1, existing CGRAs fall into three main categories: spatio-temporal CGRAs, spatial CGRAs, and specialized CGRAs.

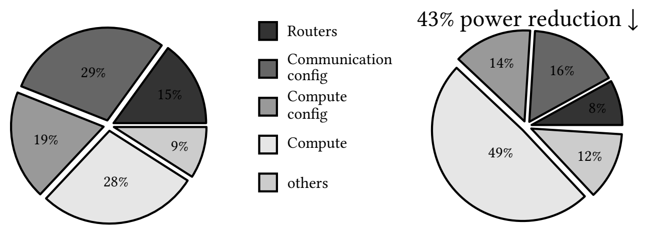
A spatio-temporal architecture allows each PE to reconfigure to a new instruction every cycle, offering higher flexibility. This means that the ALU in each PE can execute a new operation and send the data to a new neighbour every cycle. In contrast, a purely spatial architecture relies on spatial dataflow-based mapping for a code segment, maintaining a fixed configuration of compute and communication during the execution of that segment. Theoretically, a higher degree of reconfigurability enhances flexibility at the cost of higher power consumption. On the other hand, fully spatial mapping ensures extremely low power consumption but with possibly lower performance. Specialized CGRAs, in contrast to the prior ones, are optimized or tuned for a particular application suite or domain. These optimizations are only suitable for a relatively narrow scope of applications and can render the CGRA to be less efficient for others, thereby reducing their versatility. An ideal CGRA would achieve good performance and efficiency while still preserving generality.

Existing works intrinsically facilitate effective execution at the granularity of a single DFG node on a PE. However, they lack insight into the alignment between communication and computation for the collective execution of the entire DFG. First, each PE pairs the ALU with a router designed with adequate degrees of freedom to allow each ALU to individually communicate with any other ALU around it through the NoC. Such highly powerful routers help PEs handle peak routing congestion. An unforeseen consequence is that total communication resources are largely overprovisioned compared to the compute capabilities of the CGRA. This is because such congestion is only faced intermittently throughout total execution. Second, as each PE typically executes one node at a time, the inherent individuality in execution renders the programmability less efficient. For example, even simple inter-PE communication, corresponding to a single edge between two DFG nodes, necessitates configuring at least two routers. This bloats the configuration memory, contributing to 48% of the overall power consumption as shown in Figure 2(a). With the aforementioned resource over-provisioning, the programmability becomes extremely costly.

A straightforward approach to the realignment would be trimming communication resources to a bare minimum. While this strategy is conceptually simple, it can hinder reconfigurability in the architecture and degrade the performance. Therefore, a more nuanced solution should hypothetically involve redesigning the fundamental blocks of the CGRA architecture and the compiler. This redesign should collectively align compute and communication provisioning and improve efficiency while maintaining programmability, considering the inherent characteristics of DFGs and hardware.

Goal and Approach: In this work, we propose a next-gen CGRA architecture and compiler, named *Plaid*, with a hierarchical execution paradigm to address the aforementioned limitation. *Plaid*[1] aims to significantly reduce power and

[1]Despite the fundamental change in CGRA design, *Plaid* is still a spatio-temporal CGRA. When we refer to spatio-temporal CGRA in this paper, we refer to the typical design of spatio-temporal CGRAs.



(a) Spatio-temporal **(b)** Plaid

Figure 2. Power distribution for Plaid and a current state-of-the-art spatio-temporal CGRA. Plaid's execution paradigm reduces CGRA fabric power by 43% while maintaining performance and generality.

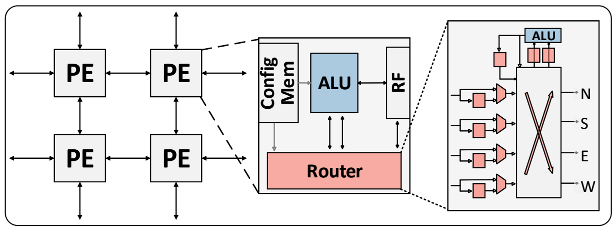


Figure 3. Spatio-temporal CGRA example

```
int main  () {
  # pragma Plaid
  for  (i: 0->N) {
   for  (j: 0->M) {
    c[i][j] = b[i]*k +a[i]*j;
    k[i] = d[i] >> 4;
     ...
    out[i] += c[i][j] +f[j];
}}}
```

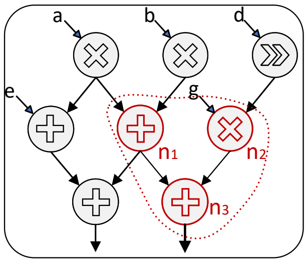


Figure 4. Example of a DFG corresponding to an annotated C code

area while maintaining the performance and generality of current spatio-temporal CGRAs. Figure 1 demonstrates the end-to-end framework of *Plaid* that enables this alignment in computation and communication. *Plaid* takes a target kernel as an annotated C code as an input, converts it into a hierarchical DFG with recurring patterns of communication, i.e., motifs, and maps them onto the proposed CGRA architecture. Below, we delineate the proposed design and highlight the innovative features of our approach:

First, we introduce a novel hierarchical execution paradigm that capitalizes on communication patterns inherent in the DFG structure, termed as "motifs." These motifs, composed of multiple nodes, demonstrate distinctive yet straightforward internal connections, enabling the CGRA to execute these nodes and route related data dependencies collectively with minimal hardware overhead.

Second, we introduce a novel CGRA architecture *Plaid*, aligning the compute and communication resources for both low-level and high-level execution. This architecture features a novel hierarchical on-chip network, composed of global and local routers, and re-organizes the compute units for collective resource alignment and motif computing.

Third, we develop an end-to-end toolchain consisting of a compiler that takes annotated loops in C code as input and maps them onto the *Plaid* architecture written in RTL. The compiler automatically identifies motifs within DFGs, flexibly schedules motifs, and hierarchically maps DFGs onto the CGRA, ensuring excellent generality and performance. We make the following contributions:

- Identification of a major limitation prevalent in previous CGRAs regarding the alignment of compute and communication resources.
- Insight into DFG's structural regularity, offering a new dimension to enhance efficiency beyond current execution.
- The first proposal of a hierarchical on-chip network within a single CGRA, to the best of our knowledge.
- An effective compiler to map DFG with motifs to achieve hierarchical execution.
- We demonstrate that we can further improve energy and area efficiency with domain-specific optimization.

Results: Our design reduces 43% power and saves 46% area compared to the baseline high-performance spatio-temporal CGRA, while maintaining the generality and performance. At the same time, compared to the baseline energy-efficient spatial CGRA, *Plaid* delivers 1.4× performance and saves 48% area while achieving almost the same power. Furthermore, *Plaid* can achieve 1.22× energy efficiency and 1.25× area efficiency even compared to a domain-optimized CGRA.

2 Background And Motivation

In contemporary CGRAs [1, 6, 9, 10, 13, 15, 28, 32, 43–46, 48–51, 54, 64–66], the PE is typically centered around an ALU as its core component. A PE usually executes one DFG node at a time and simultaneously routes data to and from neighbouring PEs. Figure 3 shows a representative CGRA architecture, detailing the internals of a PE, along with a router and corresponding registers for data buffering. Each PE contains a configuration memory to store instructions that reconfigure the ALU and routers every cycle.

Figure 4 showcases an example DFG corresponding to a loop, annotated with a pragma in the C code. Each DFG node signifies a compute or load-store operation. Typically, each DFG node is mapped to a CGRA PE through a statically compiled instruction configuration. This instruction includes the computation to be executed on the ALU and the communication required with neighbouring PEs. This communication encoding is crucial for selecting the appropriate datapath via a series of multiplexers and crossbars in each cycle.

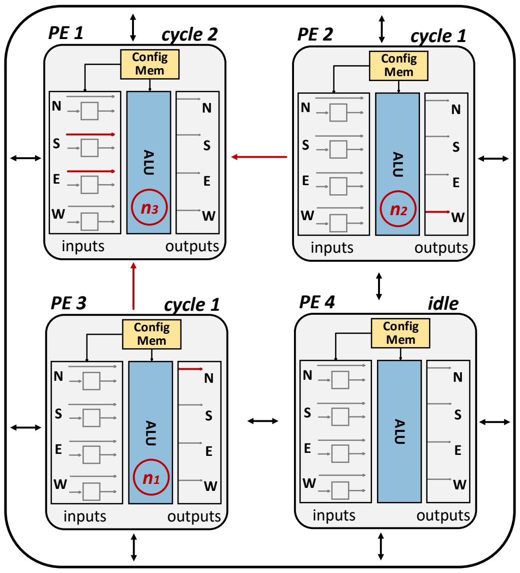


Figure 5. Detailed routing example of the highlighted sub-DFG in Figure 4 on the 2×2 CGRA from Figure 3.

Figure 5 shows the communication, specifically the data routing between three PEs of a CGRA, for the red-highlighted portion of the DFG in Figure 4. In the first cycle, nodes n1 and n2 are executed on PE3 and PE2, respectively, as shown in the figure. The outputs generated are then routed towards PE1: PE3 sends its output to the north, and PE2 sends its output to the west. PE1 receives this data and processes it in the subsequent cycle to produce a new output, which is then sent to the next neighbouring PE.

This example illustrates two main problems of the traditional execution paradigm. First, only a small portion of the network is activated for communication for internal data dependencies of the highlighted sub-DFG. If dependent nodes are placed on neighbouring PEs, such simple and immediate routing does not need powerful routers. These routers benefit from distant routing when we cannot place dependent nodes within such "proximity". Nevertheless, a significant portion of the datapath is only fully engaged when there is extremely high routing congestion at the PE. Overall, in each cycle, only certain parts of the datapath are actively in use, while the other parts serve no functionality. For the whole execution, this overprovisioned communication system does not provide a first-order benefit to the compute efficiency.

Second, as the PE typically executes one node at a time, such individual execution comes with a high cost of programmability. For example, a simple connection in Figure 5 needs to configure two routers. This programmability incurs a significant encoding overhead and leads to unnecessarily complex solutions for simple problems. As shown in Figure 2(a), communication configuration memory and the router consume 29% and 15% of power, respectively. In summary, traditional CGRA designs often fail to align the provisioning of communication with the compute capabilities, and individuality in execution leads to a high cost of programmability, accentuating the inefficiency of resource misalignment.

Through our work, we gain insights into the DFG behavior and identify patterns of communication, or motifs, between collective DFG nodes. We observe that the exhaustive communication possibilities between small groups of DFG nodes can be effectively implemented together in hardware using lightweight primitives. This approach enables hierarchical execution, with the architecture and mapper designed to exploit these motifs and their communication at two levels of granularity, and does not compromise the generality.

3 Hierarchical Execution with Motifs

This section first provides insight into harnessing simple connections in DFG with collective routing. Then, we discuss the design trade-off for collective routing and demonstrate the execution of the structural motifs with collective routing.

3.1 Collective Routing

DFGs are generated from the program by the compiler and represent the corresponding data dependencies; thereby, they do not have entirely arbitrary connections. For example, a single DFG node typically has only two inputs, indicating a limited degree of connection between multiple such nodes. Supposing each node takes a maximum of two inputs, a DFG with n nodes has a maximum of $2n$ edges. The limited number of inputs indicates the prevalence of relatively simple connections among DFG nodes, despite the fact that complex data dependencies also exist in the partial DFG.

These relatively simple connections raise an opportunity to handle the routing among multiple nodes together with a nimble design, instead of using multiple complex routers among multiple compute units. However, simple data dependencies in these sub-DFGs still have distinct connection patterns. To maintain versatility and support different patterns, we cannot resort to fixed connections in the hardware.

Figure 6 shows a prototype of the proposed collective routing scheme: a single router provisions the input/output for a group of ALUs and connects with other such routers. Multiple ALUs execute nodes from a sub-DFG, and a router handles simple data dependencies within the sub-DFG. For example, for the sub-DFG in Figure 4, we can use three ALUs to execute and a router to handle related data dependencies.

Structural motifs refer to a set of sub-DFGs, which have the same number of DFG nodes and simple internal connections, such as the sub-DFGs shown in Figure 7. These motifs are naturally suitable for collective routing: instead of using multiple routers among multiple functional units, we can use a single router to route relevant data dependencies among multiple DFG nodes (functional units). Hence, we can use

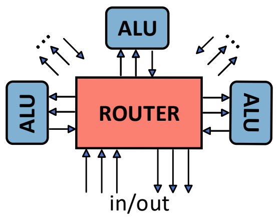


Figure 6. Prototype of collective routing

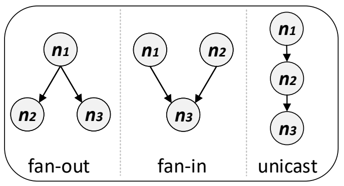
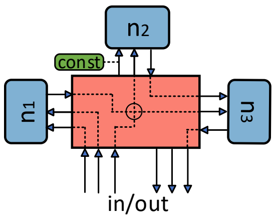
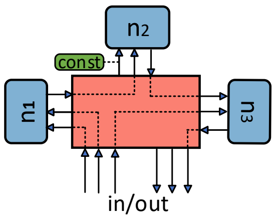


Figure 7. Fundamental motifs for three-node sub-DFGs



(a) *Fan-in* motif



(b) *Unicast* motif

Figure 8. Collective routing for *fan-in* and *unicast* motifs.

minimal hardware to route the data dependencies of a motif and achieve very high utilization of the router.

There are two challenges to leverage motifs for collective routing: The *first challenge* involves determining the optimal number of ALUs (also the number of nodes in the motif) connected to one router for designing the architecture. The hardware cost, including the router and its configuration, escalates significantly as the number of ALUs increases. Moreover, the complex connections among certain nodes in larger sub-DFGs make accommodating all connections with one router impractical.

The *second challenge* is to design a compiler to effectively leverage the collective routing to execute the whole DFG. Although it is possible to disregard the fundamental hardware changes and continue mapping nodes individually, the true potential of these units is realized through more localized routing. In other words, we need an effective compiler to automatically identify motifs and judiciously schedule them to utilize the hardware unit of collective routing. Therefore, addressing this issue requires a comprehensive approach that considers both hardware and software perspectives to develop an efficient collective routing mechanism.

3.2 Three-node Structural Motif

The selection of an optimal motif size, which can benefit from local, collective routing, determines the number of ALUs that should be connected to a single router. A two-node sub-DFG has a fixed, single communication pattern between the nodes. Furthermore, as most DFG nodes have two inputs, the two-node sub-DFG has only one internal connection and needs more communication with the other routers, rendering it similar to individual execution and not feasible to be a motif.

Conversely, larger motifs do not appear frequently in DFGs and can lead to unnecessary fragmentation. Complex connections in certain DFG nodes complicate the identification of large sub-DFGs with straightforward data dependencies. Moreover, these larger motifs tend to fragment the DFG into smaller pieces if not in the motif, diminishing the advantages of collective routing. As we mentioned earlier, accommodating larger communication patterns between nodes escalates hardware costs. Therefore, larger motifs are generally unsuitable for collective routing.

The three-node motif represents the smallest possible sub-DFG that benefits from reconfigurable communication patterns, i.e., using a router instead of a fixed connection. It aligns perfectly with the two inputs for most DFG nodes. As the smallest reconfigurable motif, it maximizes the number of motifs that can be partitioned from a given DFG and minimizes the number of standalone nodes. This balance makes the three-node motifs ideal for leveraging local, collective routing while maintaining hardware efficiency and reducing complexity. Prior work on patterns in much more complex and random real-world graphs and networks, have similar observations about the recurrence of three-node motifs [41, 70], affirming our approach.

Here, we formalize the solution to better understand the three-node motif. We start by examining Directed Acyclic Graphs (DAGs) with N vertices. We will build our understanding incrementally by considering the cases where the number of vertices, $|N|$, increases.

Base case $|N| = 2$: For $|N| = 2$, the only possible DAG is $G_2 : \{(n_1, n_2)\}$. This represents a DFG where node $n1$ sends data to node $n2$, denoted as $n_1 \longrightarrow n_2$.

$|N| = 3$: consider a new vertex n_3 added to G_2. The exhaustive set of possible DAGs, G_{3i}, that can be created by incorporating the new vertex n_3 can be enumerated as follows (visualized in Figure 7):

- Node n_3 receives the input from n_1 , $E = \{(n_1, n_2), (n_1, n_3)\}$, called ***fan-out** motif.*
- Node n_3 sends its output to n_2, $E = \{(n_1, n_2), (n_3, n_2)\}$, called ***fan-in** motif.*
- Node n_3 sends its output to n_1, or n_2 sends its output to n_3: both are sequential chains, $E = \{(n_1, n_2), (n_2, n_3)\}$ or $E = \{(n_3, n_1), (n_1, n_2)\}$, called ***unicast** motif.*
- Acyclic triangle: $n_3 \longrightarrow n_1 \longrightarrow n_2 \longleftarrow n_3$, $E = \{(n_3, n_1), (n_3, n_2), (n_1, n_2)\}$. The acyclic triangle can be derived from any of these three basic motifs by introducing a single additional edge, and thus is not a basic motif.

The first three basic motifs here serve as exhaustive, fundamental building blocks from which any other DFG with $|N| = 3$ can be constructed.

Hierarchical composition: For a DAG, given all possible subgraphs for $|N| = 3$, any graph with $3 \times n$ nodes

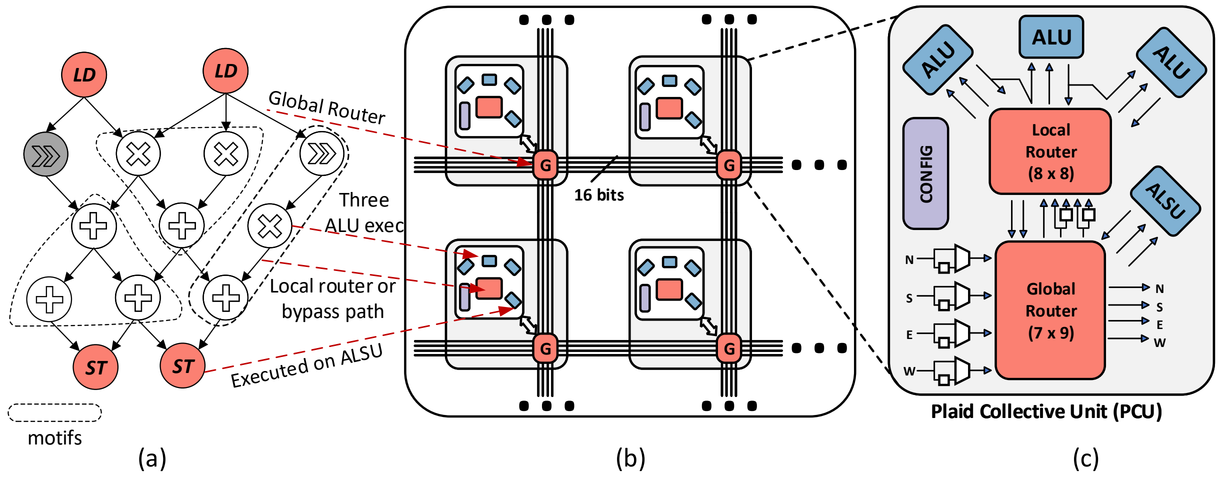


Figure 9. (a) Hierarchical DFG with multiple motifs. (b) Plaid architecture overview (c) Plaid Collective Unit

can be constructed using the above subgraphs as interconnected building blocks. This approach leverages the principle of graph composition, where larger graphs are formed by combining smaller, well-understood subgraphs [7, 33]. We can easily prove that, for DFGs G_{3n+k}, with $3n + k$ nodes, where $0 < k < 3$, the graph can be constructed by using $\left\lfloor \frac{3n+k}{3} \right\rfloor = n$ subgraphs of 3 nodes (three basic motifs), supplemented with k additional standalone helper nodes. Formally, $G_{3n+k} = \left(\bigcup_{i=1}^{n} G_{3i}\right) \cup H_k \cup E''$, where H_k represents the standalone nodes, and E'' represents the additional edges that connect different subgraphs and various standalone nodes with each other. A DFG can thus be constructed by utilizing the three motifs and individual standalone nodes wherever required.

Figure 8 demonstrates the execution of *fan-in* (which forms the highlighted section in Figure 4) and *unicast* motifs using three ALUs and a single router. The internal data dependencies here are managed within the router. For instance, communications such as $n_1 \longrightarrow n_2$ and $n_2 \longrightarrow n_3$ within the unicast motif are routed locally. Other non-constant inputs and outputs, which form the standalone nodes H_k or additional edges E'', engage the local router to communicate with other routers. This arrangement ensures high router utilization and alleviates the demand for global communication by routing internal dependencies locally.

For a better understanding of the hierarchical nature, Figure 9(a) presents a DFG example featuring multiple motifs. We assume the memory access unit is not connected to the collective router and explain the execution of memory operations with a hierarchical on-chip network later. It happens that some standalone nodes are not included in the motif. Such a node can still be executed on any ALU, without the loss of generality. The collective routing mechanism still applies for such nodes, but the corresponding data dependencies might not be routed locally. These motifs enable a ***hierarchical execution*** on the CGRA. To accommodate the high efficiency of the hierarchical execution, we first need to manage the communications among these motifs. As we collectively route these relatively simple dependencies, the communications among motifs are substantial. Second, we need to reorganize the entire CGRA to enhance overall execution efficiency.

4 Plaid Architecture

The *Plaid* CGRA is designed with two levels of granularity in execution as illustrated in Figure 9(b). Each tile executing a motif is called the Plaid Collective Unit (PCU). A PCU incorporates the collective routing circuitry and executes individual motifs, i.e., handles simple DFG connections. Multiple PCUs are interconnected in a mesh network and interact with each other to facilitate overall DFG execution, supporting the communication between various motifs and standalone nodes, including the complex connections in the DFG. Hierarchical execution enables the exploiting of patterns in communication and achieves high efficiency for *Plaid.*

4.1 Plaid PCU

The *Plaid* PCU is specifically designed to execute three-node motifs. The collective routing prototype, as discussed previously, has been adapted for this PCU's micro-architecture, featuring three ALUs (motif compute unit) connected to a local router. Figure 9(c) depicts the internal connections within the PCU. The ALUs, which are 16-bit compute units, support ADD, MUL, SHIFT, and various bit-wise operations, totalling 15 operations. The local router delivers inputs to each of the three ALUs per cycle and collects outputs from them, enabling the execution of all possible three-node motifs.

Although the collective routing prototype can handle all three node motifs with the identically positioned ALUs around the router, we add *virtual* bypass paths between the ALUs. These bypass paths provide shorter routing paths and are prioritized by the mapper, which typically favours an execution sequence from the left-most to the right-most ALU. However, this execution order is not obligatory. We identify that adhering strictly to this order for all motifs can lead to resource under-utilization. To address this, we developed more flexible scheduling options, which we will detail in Section 5. Nevertheless, these bypass paths are still beneficial as they reduce the routing pressure on the local router.

4.2 Hierarchical Network-on-Chip

The local router connects to a global router, which facilitates communication between motifs across different PCUs and links motifs to the data memory. The global router is connected to an Arithmetic-Load-Store Unit (ALSU), which is integral to the PCUs along the edges interfacing with the data memory. The ALSU manages load-store operations within the DFG via a dedicated datapath to the data memory. Additionally, it facilitates the mapping of predication operations and standalone nodes facing routing challenges.

The global router has ports facing the N, S, E, and W directions, enabling inter-motif (inter-PCU) communication. The architecture creates two distinct datapaths: a global datapath, which connects all PCUs across the CGRA through the global router, functioning like a data conveyor belt; a local datapath within each PCU, managed by the local router, which aims to handle all communication internally. Yet, when necessary, it interacts with the global path, either retrieving or depositing data onto the conveyor belt. This hierarchical organization ensures efficient and flexible data management, optimizing both local and global communication within the system.

Considerations for the Global Datapath: The paths between the global and local routers include options for temporal data buffering using registers, as shown in Figure 9(c). Data that continuously cycles from the global router to the local and back again can potentially get stuck in a loop when synthesized. To prevent the formation of such hardware loops, we enforce constraints within the interconnects in the compiler and the hardware. The compiler constraints are designed to block any configuration that could create a closed loop. The hardware constraints ensure no such datapath can be created. These constraints are verified post-synthesis through the EDA tool, which can report any such violations.

4.3 Configuration

With each PCU now managing three ALUs, one ALSU, two routers, and multiple registers along the datapath, the configuration memory encoding bits need a comprehensive redesign. The routers alone consume about half of these encoding bits, highlighting their role in the system. The ALUs connected to the local router require eight-bit constants and four-bit fields for operation selection per cycle. Each instruction, or configuration entry, comprises a total of 120 bits, including the local and global configuration space.

4.4 Efficient Domain-Specialization

Recent advancements in CGRAs have enabled domain-specific optimizations [40, 58, 62] in the architecture. Plaid's execution paradigm, which already exploits communication patterns, can further enhance efficiency through domain-specific specialization. We exploit the existence of recurrent motifs in a domain and specialize a few PEs in architecture that are domain-optimized for those motifs. Essentially, we hardwire a fixed motif inside certain PEs to replace the local router while maintaining the full reconfiguration capability of the global datapath. This approach reduces the configuration space for these specific PEs, compelling the mapper to prioritize them for mapping the corresponding motifs they were specialized for.

5 Plaid Compiler

This section first formulates the mapping problem and then introduces our algorithm for generating motifs, followed by the introduction of the mapping algorithm.

5.1 Mapping Problem Formulation

Given a DFG and a CGRA, application mapping is executed through modulo scheduling. This technique allows for a new loop iteration at every initiation interval (II). Initially, we ascertain the lower bound of II, designated as the Minimum II (MII). This is determined by taking the greater of two metrics: the resource minimum II (ResMII) and the recurrence minimum II (RecMII) caused by inter-iteration data dependency. For each prospective II value, a time-extended resource graph of the CGRA, corresponding to II cycles, is constructed. This graph is referred to as the Modulo Routing Resource Graph (MRRG). Owing to the cyclical nature of the schedule, which repeats every II cycle, there is connectivity between the resources at cycle II-1 and those at cycle 0 within the MRRG—hence the term "modulo." The compiler aims to achieve an optimal mapping of the DFG onto the MRRG, aiming to minimize the II value.

Given a DFG $D = (V_D, E_D)$ and a *Plaid* CGRA instance, the problem is to generate a hierarchical DFG with multiple motifs and standalone nodes, $HD = (M_{HD}, E_{HD})$ (standalone node is a special motif where motif node number is one), and construct a minimally time-extended MRRG of the *Plaid* instance $P_{II} = (V_P, E_P)$, which has a valid mapping $\phi = (\phi M, \phi E)$ from HD.

5.2 Mapping DFG onto CGRA

Algorithm 1 delineates a systematic approach to identify motifs (M_{HD}) to generate a hierarchical DFG (HD). This algorithm is easily extendable to the motif with any number of nodes. Figure 10 shows an example using Algorithm 1. This process starts with the greedy generation of initial motifs, leveraging the fundamental set of motif structure patterns to traverse the DFG and generate motifs (line 1). However, as

Algorithm 1: Motif Generation

```
Input: DFG;
Output: Hierarchical DFG with motifs
1  Generate the initial motifs greedily;
2  while the motif number increases do
3      Randomly break down one motif;
4      Randomly sort standalone nodes;
5      foreach standalone node do
6          if find a motif pattern with this node then
7              Generate the motif and update standalone nodes;
```

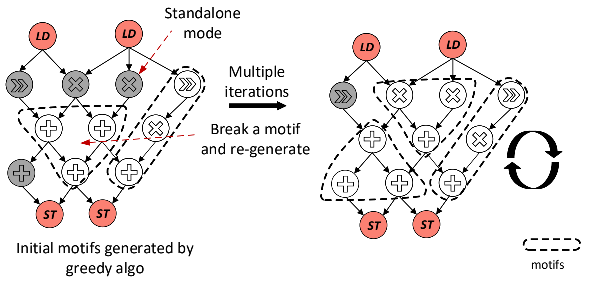

Standalone mode
Multiple iterations
Break a motif and re-generate
Initial motifs generated by greedy algo
motifs
LD
ST

Figure 10. Example with Algorithm 1

shown in Figure 10, the greedy generation is sub-optimal and there are many standalone nodes in the initial generation.

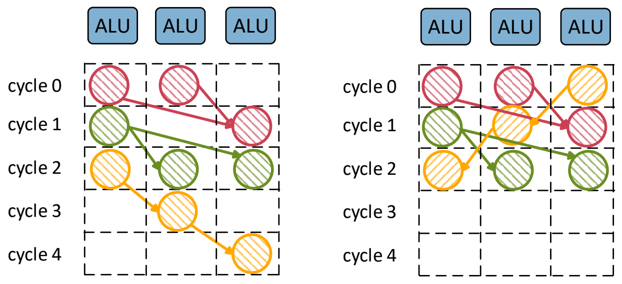
We adopt an iterative algorithm to improve motif identification. The algorithm randomly deconstructs a motif, randomly sorts standalone nodes, and traverses the DFG starting from standalone nodes to find new motifs as long as they match the motif pattern (lines 3-7). For example, in Figure 10, we break the left motif, randomly sort standalone nodes, and try to find sub-DFGs that match motif patterns to generate more motifs. We keep running the deconstruction and re-generation process until the number of motifs does not increase (lines 2-7), or the number of motifs exceeds standalone nodes. The latter is to ensure the utilization of the motif compute unit and ALSU in PCU. The method's iterative nature helps refine motif integration within the DFG, aiming for optimized motif generation.

Algorithm 2 presents the hierarchical mapping to efficiently map DFGs with identified motifs, generated by the Algorithm 1, onto a Plaid CGRA. Algorithm 2 augments simulated annealing [3, 73] to hierarchically schedule motifs. Figure 9 shows the corresponding hardware unit to handle motifs, internal data dependencies, and global communication. We use a cost function to guide the mapping process, which includes metrics such as the number of unmapped nodes (V_D), congestion levels, and the usage of routing resources.

Initially, motifs within the DFG are sorted according to the data dependency (line 1), which prioritizes critical paths for early mapping. The algorithm starts with the MII. If no valid mapping is found at the current II, it increments

Algorithm 2: Hierarchical mapping

```
Input: Hierarchical DFG, CGRA architecture description;
Output: Mapping
1  Sort motifs by data dependency;
2  while Mapping is not valid do
3      foreach motif do
4          Map the motif to a PE with the least routing resource;
5      while not find a valid mapping or exceed time limitation do
6          Unmap one motif ;
7          Randomly select one placement candidate;
8          Generate different motif schedules;
9          foreach motif schedule do
10             Route this motif's operands and dependencies to
                 the network using Djkstra's algorithm;
11             Select the combination yielding the highest
                 objective;
12     II = II + 1 ;
```



(a) Stringent sequential schedule **(b)** Flexible Schedule

Figure 11. Schedules for three motifs. The right one has a higher resource utilization.

the II by one, continuing this process until it either finds a valid mapping or exceeds the maximum II determined by the configuration memory size (lines 2-12). In each II, initially, sorted motifs are mapped onto the hardware unit V_P with the least routing resources (lines 3 and 4). If a valid mapping is not achieved within the initial step, the algorithm uses an iterative methodology to generate a valid mapping (lines 4-10). In each iteration, one motif (M_{HD}) is unmapped (line 6), and we randomly select one motif unit (line 7) as the placement candidate. We can place a single node (special M_{HD}) onto any functional unit V_P if it supports the operation.

The motif compute unit's bypass connection does not restrict the node placement within a motif from left to right. For example, in the unicast motif, the first node and the second node are not required to be placed onto the leftmost ALU and the middle ALU, respectively. Instead, the first node can be placed onto the rightmost ALU. Of course, the bypass connection is not utilized in such case. A stringent placement can cause motif compute units to be under-utilized if any ALU is not available.

Figure 11 illustrates two distinct schedules for three motifs on the motif compute unit. In Figure 11(a), the scheduling order is strict to the above-mentioned sequential order: the first node is allocated to the leftmost ALU, the second node to the middle ALU, and the third node to the rightmost ALU. However, this stringent scheduling sequence leads to an under-utilization of resources. In contrast, Figure 11(b) depicts a more flexible scheduling approach, where the *unicast* motif adopts a 'reversed' sequence compared to that in Figure 11(a), thereby enhancing the utilization of the collective unit.

To support flexible scheduling, we generate several schedule templates for each motif type. For example, for the fan-out motif, we have the following schedule templates: $\{(n1, c), (n2, c+1), (n3, c+1)\}$, $\{(n1, c), (n2, c+1), (n3, c+2)\}$, $\{(n1, c), (n2, c+2), (n3, c+1)\}$, $\{(n3, c+1), (n2, c+1), (n1, c)\}$, $\{(n3, c+2), (n2, c+1), (n1, c)\}$, & $\{(n3, c+1), (n2, c+2), (n1, c)\}$. Each template comprises three tuples representing an ALU in left-to-right order, with each tuple containing the node in motif and its scheduled cycle. These templates ensure sufficient flexibility and obviate exhaustive searches on the motif compute unit. For the placement candidate, we use Dijkstra's algorithm to assess each schedule (line 10). The schedule that yields the highest objective value (the least cost) is selected (line 11). Like typical simulated annealing, we can occasionally accept a "worse" movement to overcome the local minimum. We repeat this process until we find a valid mapping or exceed the mapping time limitation.

6 Experimental Methodology

We describe the setup and methodology to evaluate the end-to-end framework for *Plaid* in this section.

6.1 Architecture Synthesis

We implement the *Plaid* architecture using Verilog RTL and synthesize it with Cadence Genus targeting a 22nm FDSOI technology node at 100MHz. The power consumption metrics are obtained from post-synthesis estimates provided by the synthesis tool. *Plaid* features a 2×2 PCU array and four 4KB data memory banks, with each PCU having a 16-entry configuration memory. As each PCU has four functional units, the 2×2 *Plaid* has the same number of functional units as typical 4×4 CGRA. Moreover, we also evaluate the scaled 3×3 *Plaid* with the same number of functional units as a 6×6 CGRA.

6.2 Compiler and Execution

The *Plaid* framework processes annotated C code, as shown in Figure 4, and generates static configurations for the *Plaid* CGRA using the Morpher toolchain[73], which includes a mapper and a cycle-accurate C++ simulator. The compiler typically maps the kernel in a few minutes. As typical CGRAs are statically scheduled, the performance is deterministic and known at compilation time. With the II and the total number of loop iterations, we can precisely calculate the overall performance (cycles). The primary purpose of the simulation is to verify the mapping and hardware design.

domain	kernel	unroll	char[1]	kernel	unroll	char
Linear Algebra	atax	2	15,6,6	atax	4	27,14,11
	bicg	2	23,11,10	bicg	4	42,23,19
	doitgen	2	18,9,9	doitgen	4	34,21,10
	gemm	2	21,12,12	gemm	4	37,24,23
	gemver	2	21,11,10	gemver	4	41,23,19
	gesumm	2	22,9,8	gesumm	4	38,19,16
Machine Learning	conv2x2	1	20,12,10	conv3x3	1	37,26,17
	dwconv	1	7,3,2	dwconv	5	31,19,13
	fc	1	17,8,7			
Image	cholesky	2	14,5,4	cholesky	4	28,11,8
	durbin	2	14,7,4	durbin	4	28,15,8
	fdtd	2	16,7,6	fdtd	4	32,15,12
	gramsc	2	15,5,4	gramsc	4	25,11,8
	jacobi	1	16,7,5	jacobi	2	30,15,12
	jacobi	4	54,30,27	seidel	1	22,11,9
	seidel	2	44,23,21			

1. Characteristics contain the number of DFG nodes, the number of compute nodes, and the number of compute nodes covered by motifs.

Table 2. Evaluated workloads

The host processer loads the CGRA configuration bits generated by the compiler to the CGRA fabric and sends input data to the SPM. Then, it triggers the CGRA to fetch the configuration, load the data, perform computations, and store the results back in the SPM. Spatio-temporal CGRA can read the configuration memory per cycle in a modulo way. Ultimately, data from the SPM is moved back to the main memory for further processing by the host processor.

6.3 Baseline CGRAs

Given the significant redesign of the CGRA architecture, we benchmark our results against several baselines:
Spatio-temporal CGRA, like typical CGRAs [14, 39, 58, 67] as shown in Figure 3, has a 4×4 PE array with a mesh network and the same SPM configuration with 2×2 *Plaid*. Each PE has 16-entry configuration memory.
Spatial CGRA follows the state-of-the-art energy-efficient spatial CGRAs [22, 23, 55] but with a mesh network. The spatial CGRA has a 4×4 PE array. Mapping complex kernels ($II > 1$) onto spatial CGRAs requires partitioning the DFG into several DFGs. We develop a Python script to partition DFGs. Additional loads and stores are introduced during partition to put intermediate data in SPM.
Domain-optimized spatio-temporal CGRA is based on the spatio-temporal CGRA and optimized for the machine learning domain [62]. This lowers the performance of the spatio-temporal CGRA for other domains but improves the energy and area efficiency of the target domain.
Baselines are implemented using the same technology node for consistent area and power comparisons. We use two mappers for these baselines and select the one with higher performance. The first is PathFinder, adapted from [38, 60,

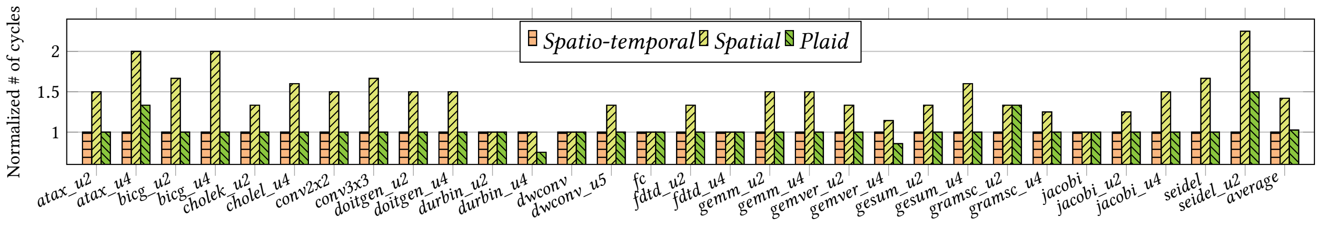


Figure 12. Performance of *Plaid* and a spatial CGRA normalized to a spatio-temporal CGRA. *u2* :unrolling factor of 2.

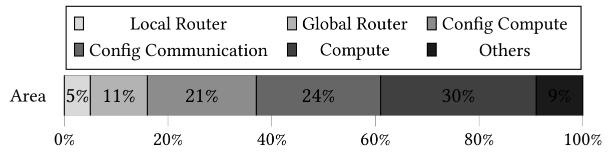


Figure 13. Area breakdown for Plaid's CGRA fabric

73], consisting of around 3K lines of C++ code. The second mapper utilizes simulated annealing, as detailed in [3, 68, 73], implemented with around 2K lines of C++ code.

6.4 Workloads

To demonstrate that *Plaid* maintains the versatility of CGRAs, we benchmark our work using a diverse set of kernels from various applications and benchmark suites, as detailed in Table 2. We unroll DFGs to evaluate the capability of *Plaid* to handle complex data dependencies, leading to 30 DFGs in total. We use five kernels from TinyML[2] to represent typical machine learning workloads. For *dwconv*, we unroll with 5 as the trip count is not divisible by other less unrolling factors. We evaluate the linear algebra benchmark suite and image process kernels from PolyBench [29], assessing the performance of kernels from this suite. To ensure a similar number of kernels from each domain, we use the first six kernels from the PolyBench linear algebra suite. For each DFG, Table 2 shows the number of nodes, compute nodes, and nodes covered by motifs. As motif compute unit does not provide memory access, all the nodes in motifs are compute nodes. Moreover, we also execute the two-node motif with the motif compute unit. As we provide flexible schedules, this does not affect the schedule of three-node motifs.

Application-level mapping: to evaluate system-level performance, we evaluate three DNN applications adapted from TinyML [2]. These three DNN applications comprise 10, 13, and 16 layers, respectively. Most layers are Convolution layers and DepthWiseConv layers.

7 Evaluation

To highlight the overprovisioned communication in CGRAs, we compare CGRAs with equivalent theoretical throughput (the same number of functional units) from three perspectives: performance, energy consumption, and performance per area. Second, we showcase *Plaid*'s scalability by evaluating larger architecture versions using a 3×3 PCU array, to highlight *Plaid* can maintain efficiency as it scales. Third, to evaluate the effectiveness of our mapper, we compare it with popular CGRA mappers on *Plaid.* Finally, we implement a domain-optimized *Plaid* and compare it with a domain-optimized (specifically machine learning) spatio-temporal CGRA.

A *2×2* prototype of *Plaid* reveals that the CGRA fabric occupies 33,366 μm^2 of space, while the scratchpad memories take up an additional 30,000 μm^2. As illustrated in Figure 13, the breakdown of the CGRA fabric area shows some interesting insights. The communication hardware, including routers and their configuration elements, makes up about 40% of the area. Meanwhile, the compute hardware and its configuration take up 50% of the on-chip area. *Plaid* trims down a significant portion of the redundant communication resources and achieves a balance by aligning compute and communication resources, leading to high utilization of both.

Our evaluation shows that *Plaid* delivers 1.40× performance and saves 48% of the area compared to the energy-minimal CGRA while achieving almost the same power. Compared to high-performance spatio-temporal CGRA, *Plaid* reduces 43% power and saves 46% area without sacrificing performance and generality. Furthermore, compared to the domain-optimized spatio-temporal CGRA, *Plaid* can still significantly improve energy and area efficiency.

7.1 Plaid Achieves High Performance and High Efficiency

Performance: We present the performance of each kernel in terms of the number of cycles it takes for a complete execution on each architecture. Figure 12 presents the performance comparison of *Plaid* and the baselines, all normalized to the performance of a spatio-temporal CGRA. All three CGRAs have 16 functional units (ALU and ALSU).

Overall, *Plaid* achieves almost the same performance compared to a spatio-temporal CGRA. While maintaining the same performance, it also achieves a 43% reduction in power and 46% reduction area compared to the spatio-temporal

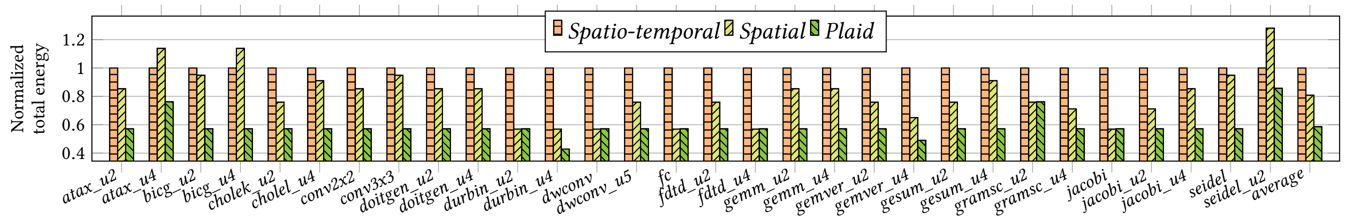


Figure 14. Energy consumption of *Plaid* and a spatial CGRA normalized to a spatio-temporal CGRA.

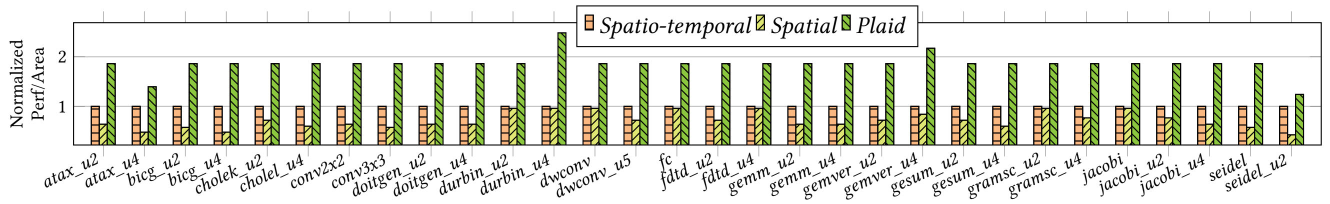


Figure 15. Performance per area of *Plaid* and a spatial CGRA normalized to a spatio-temporal CGRA

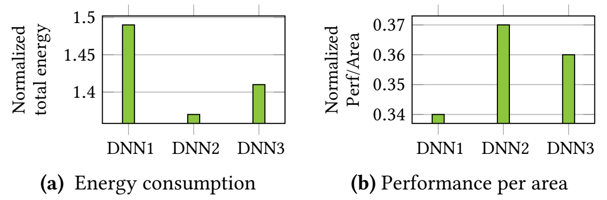


Figure 16. Comparison of a spatial CGRA and *Plaid* (normalized to *Plaid*) on three DNN applications adapted from TinyML [2].

CGRA, demonstrating the need to the address the over-provisioning. Moreover, *Plaid* even outperforms the spatio-temporal CGRA on *durbin_u4* and *gemver_u4*. The reason is that the unrolling for these kernels enlarges the DFG but does not significantly complicate the data dependency. For an enlarged mapping space of the problem, the spatio-temporal architecture tries to provide a complex solution to a simple problem and suffers from it. In contrast, *Plaid* can still utilize motifs to generate superior performance. Moreover, the spatio-temporal CGRA achieves better performance on *atax_u4*, *gramsc_u2*, and *sedidel_u2* than *Plaid*. The reason is that more complex and long data dependencies are introduced across multiple parts of the DFG during the unrolling of these kernels, and motifs need more frequent long latency communication with other nodes. Nevertheless, the average performance is almost the same between *Plaid* and the spatio-temporal CGRA.

Compared to the baseline spatial CGRA, *Plaid* delivers a 1.40× improvement in performance. The spatial CGRA needs to partition the DFG to handle complex kernels, introducing more load and store operations to put intermediate data in SPM. Spatial CGRA achieves the same performance with *Plaid* and spatio-temporal CGRA for a few kernels, such as *fc*, *durbin_u2*, *dwconv*, *gramsc*, and *jacobi*, The reason is that these kernels have relatively simple data dependency and do not need a lot of additional loads and stores for partitioning.

Energy consumption Figure 14 shows fabric energy consumption comparison normalized to the spatio-temporal CGRA. *Plaid* achieves 42.0% and 27.7% energy reduction compared to spatio-temporal CGRA and spatial CGRA, respectively. The similar performance and *Plaid's* significant reduction in power directly translates to a reduction in energy consumption over the spatio-temporal CGRA. Furthermore, *Plaid* improves the energy efficiency compared to spatial CGRA, because it can deliver much better performance with roughly the same power. A typical spatio-temporal or spatial CGRA usually only achieves high performance or high energy efficiency. ***Plaid* demonstrates that through aligning resource provisioning, a CGRA can be high-performance, general, and efficient at the same time.**

Performance per area Figure 15 presents the performance per area comparison normalized to the baseline spatio-temporal CGRA. *Plaid* achieves significant improvement in area efficiency compared to the spatial and spatio-temporal CGRAs. Spatial CGRA can achieve higher energy efficiency but lower area efficiency compared to spatio-temporal CGRA. This is because the spatial CGRAs clock-gate the configuration memory and enable spatial dataflow-based mapping, directly reducing the power, while still requiring similar area. Due to the relatively complex data dependency in *linear algebra*, spatial CGRA achieves less performance per area than the other two domains. *Plaid* achieves a stable improvement compared

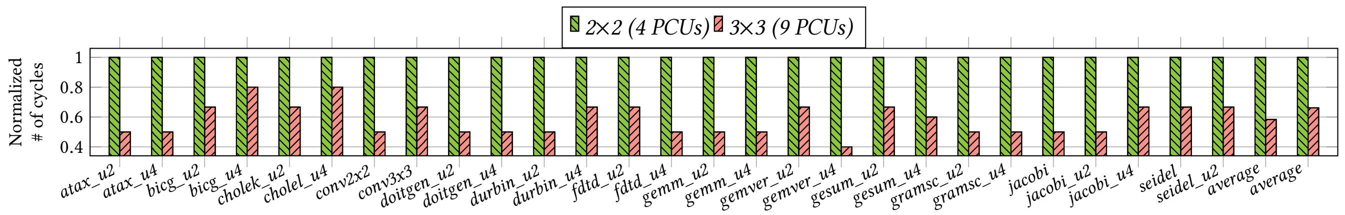


Figure 17. Scalability analysis: Performance of 3×3 Plaid compared to 2×2 *Plaid*

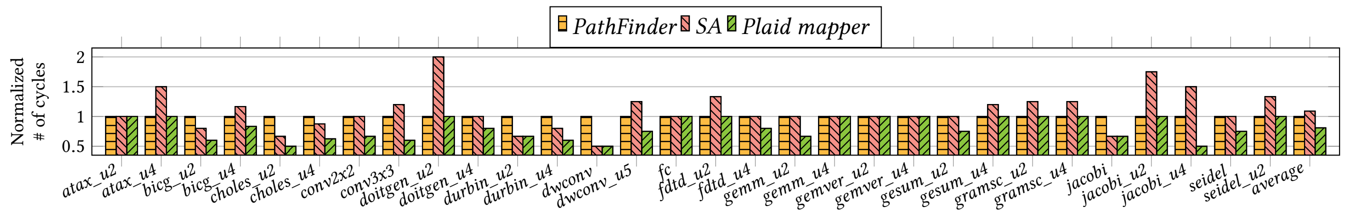


Figure 18. Performance of *Plaid* on using *Plaid's* mapper compared to using a generic mapper

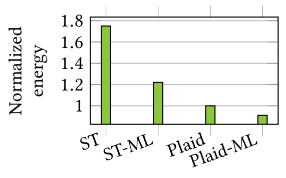
to spatio-temporal CGRA across the three domains, demonstrating the capability to handle various applications.

Application-level comparison Figure 16 presents the energy consumption and performance per area comparison between *Plaid* and spatial CGRA. We do not include spatio-temporal CGRA as it achieves the same performance with *Plaid* on every machine learning kernel. Spatial CGRA consumes 1.42× energy and achieves 36% performance per area compared to *Plaid* on average. With application-level comparison, we can find *Plaid* still achieves significant improvement in energy consumption and performance per area.

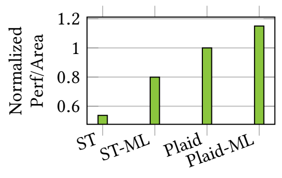
In summary, *Plaid* can achieve high performance, high energy efficiency, and high performance per area, while spatio-temporal CGRA can only achieve high performance, and spatial CGRA can achieve high energy efficiency. Nevertheless, compared to an energy-efficient spatial CGRA, *Plaid* achieves higher energy efficiency. This demonstrates that we optimally align the computing and communication resource provisioning with the proposed design.

7.2 Plaid's Hierarchical Execution is Scalable

Figure 17 shows the performance comparison between 2×2 PCU array and 3×3 PCU array. We exclude DFGs that the 3x3 *Plaid* cannot enhance the performance due to inter-iteration data dependencies. 3×3 *Plaid* can achieve 1.71× performance compared to 2×2 *Plaid*, demonstrating the scalability of the *Plaid* architecture. The performance increase with the 3x3 *Plaid* does not reach the theoretical maximum for two reasons. First, the performance might saturate at 2×2 *Plaid*, limiting the additional benefits from the larger PCU array. Secondly, the relationship between the number of nodes in a DFG and the number of functional units affects performance scaling. For instance, if a DFG comprises 40 nodes, the



(a) Energy consumption



(b) Performance per area

Figure 19. Comparison of domain-specialization in Spatio-Temporal (ST) and *Plaid*. Numbers are normalized to *Plaid*.

theoretical minimum resource II for the 2x2 *Plaid* (with 16 functional units) and the 3x3 *Plaid* (with 36 functional units) would be 3 and 2, respectively. If we can achieve minimal II on both, 3×3 *Plaid* can only achieve 1.5× performance.

Figure 18 shows the performance comparison on *Plaid* CGRA among the aforementioned CGRA mapper PathFinder, Simulated Annealing (SA), and the *Plaid* mapper. The *Plaid* mapper augments SA to support motif-based hierarchical mapping. Thus, the main difference between the two mappers is motif scheduling. The *Plaid* mapper can achieve 1.25× and 1.28× performance improvements compared to the PathFinder and SA mapper, respectively. Despite this, both generic mappers can still utilize the hardware designed for collective routing because the routing path is much shorter via local routers. *Plaid*, in conjunction with PathFinder and SA, can achieve comparable performance on several DFGs with spatio-temporal CGRA, demonstrating the effectiveness of the hardware design. However, both generic mappers cannot fully utilize the *Plaid* architecture for more complex DFGs, as they lack the capability to recognize and exploit motifs within DFGs, and are unable to handle the high routing congestion because of the trimmed-down communication circuitry. In contrast, our compiler can automatically identify these motifs and exploit the hierarchical execution paradigm, thereby maximizing the architectural benefits of *Plaid*.

7.3 Plaid Enhances Domain-Specialization

Figure 19 presents the comparison for various CGRAs: a general spatio-temporal CGRA (ST), a machine learning-optimized spatio-temporal CGRA (ST-ML), general-purpose *Plaid*, and *Plaid* optimized for machine learning (Plaid-ML). ST-ML is generated by pruning the function and bit width to achieve higher energy efficiency and higher performance per area [62]. As mentioned in Section 4.4, for the Plaid-ML, connections for the motif compute units are hardwired instead of being routed through local routers. We manually check the machine learning DFGs and design Plaid-ML. Specifically, Plaid-ML uses 2 hardwired *fan-in* motifs, 1 *unicast* motif, and 1 *fan-out* motif for four PCUs, effectively accommodating the motif requirements of our machine learning DFGs. In terms of performance and energy efficiency, general-purpose *Plaid* outperforms the domain-optimized ST-ML by reducing 18% energy consumption and achieving a 1.26 × performance per area. When comparing Plaid-ML to domain-optimized ST-ML, Plaid-ML shows a significant improvement, reducing energy consumption by 25.5% and achieving a 1.46 × performance per area.

8 Related Work

Compute and communication resource provisioning: HyCUBE [30] designs a single-cycle multi-hop NoC to improve the capability to handle complex data dependencies. Zhang et al. [77] propose a scalable hybrid network to improve energy efficiency. Marionette [16] enhances network efficiency by separating the control and data networks. Softbrain [47], Tartan [42], and Piperench [25] feature dedicated PEs for each instruction. Snafu [22], Manic [24], Riptide [23], Dyser [26], and Plasticine [52] enhance PEs with limited dataflow semantics. While these designs emphasize resource provisioning, they fail to check the collective misalignment between compute and communication provisioning.

Enhancing efficiency: REVAMP [62] offers a framework that derives low-power heterogeneous CGRAs from homogeneous ones based on user workloads. Vecpac [58] presents a precision-aware CGRA that utilizes multiple 16-bit vector units to support higher data precision. OpenCGRA [60] provides a framework for automatically generating hardwired processing elements for typically linear operation "chains" in an application. APEX [40] and BERET [27] automatically design, i.e., hard-wire parts of their circuit for frequent sub-DFGs within a given domain, selecting one block to process each code section. They potentially suffer from under-utilization and over-provisioning of resources [53]. ML-CGRA [37] offers high-level compiler optimizations for machine learning applications on CGRAs. CCA [4, 5] is one of the early works to identify and accelerate subgraphs. It features composable rows of functional units to accelerate commonly observed dataflow semantics. However, its network structure offers less flexibility with varying depths, corresponding timing constraints, and configurations to explore depending on the targeted applications.

In contrast to the above architectures, Plaid's reconfigurable network is designed with a more generalized understanding of (dataflow) graph structures independent of any particular application. This allows for flexible and resource-efficient compositions of functional units, capable of supporting any arbitrary range of patterns. Based on compositions of 3-node motifs within DFGs, Plaid's approach is inherently more general, adaptable, and future-proof, as this structural foundation will remain consistent even as application kernels change.

9 Conclusions

We introduce *Plaid*, a novel CGRA architecture and compiler that addresses the misalignment of compute and communication resources. *Plaid* reduces power by 43% and area by 46% compared to baseline spatio-temporal CGRAs while maintaining high performance and generality. *Plaid* offers 1.40× performance and 48% area savings compared to energy-efficient spatial CGRAs. Overall, *Plaid* represents a significant advancement in CGRA design, providing a balanced solution that meets the demands of edge devices while achieving high performance, high energy efficiency, and high generality.

Beyond its immediate benefit on CGRA design, our motif-based execution paradigm offers an alternative to current ad-hoc specialization approaches, where disparate accelerator designs emerge due to the high cost of composition. The fundamental insight of leveraging structural motifs for collective execution could inform the design of other non-CGRA architectures, providing a systematic framework to improve efficiency while maintaining generality.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments. This research is supported by the National Research Foundation, Singapore under its Competitive Research Program Award NRF-CRP23-2019-0003.

References

[1] Mahesh Balasubramanian and Aviral Shrivastava. 2022. PathSeeker: A Fast Mapping Algorithm for CGRAs. In 2022 Design, Automation Test in Europe Conference Exhibition (DATE). 268–273.

[2] Colby Banbury, Vijay Janapa Reddi, Peter Torelli, Jeremy Holleman, Nat Jeffries, Csaba Kiraly, Pietro Montino, David Kanter, Sebastian Ahmed, Danilo Pau, et al. 2021. Mlperf tiny benchmark. *arXiv preprint arXiv:2106.07597* (2021).

[3] S Alexander Chin, Noriaki Sakamoto, Allan Rui, Jim Zhao, Jin Hee Kim, Yuko Hara-Azumi, and Jason Anderson. 2017. CGRA-ME: A unified framework for CGRA modelling and exploration. In *2017 IEEE 28th*

International Conference on Application-specific Systems, Architectures and Processors (ASAP). IEEE, 184–189.

[4] Nathan Clark, Jason Blome, Michael Chu, Scott Mahlke, Stuart Biles, and Krisztian Flautner. 2005. An architecture framework for transparent instruction set customization in embedded processors. In *32nd International Symposium on Computer Architecture (ISCA'05)*. IEEE, 272–283.

[5] Nathan Clark, Manjunath Kudlur, Hyunchul Park, Scott Mahlke, and Krisztian Flautner. 2004. Application-specific processing on a general-purpose core via transparent instruction set customization. In *37th international symposium on microarchitecture (MICRO-37'04)*. IEEE, 30–40.

[6] Jason Cong, Hui Huang, Chiyuan Ma, Bingjun Xiao, and Peipei Zhou. 2014. A fully pipelined and dynamically composable architecture of CGRA. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 9–16.

[7] Gérard Cornuéjols and William H. Cunningham. 1985. Compositions for perfect graphs. *Discret. Math.* 55, 3 (1985), 245–254. https://doi.org/10.1016/S0012-365X(85)80001-7

[8] Vidushi Dadu, Sihao Liu, and Tony Nowatzki. 2021. Polygraph: Exposing the value of flexibility for graph processing accelerators. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 595–608.

[9] Vidushi Dadu and Tony Nowatzki. 2022. Taskstream: Accelerating task-parallel workloads by recovering program structure. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 1–13.

[10] Vidushi Dadu, Jian Weng, Sihao Liu, and Tony Nowatzki. 2019. Towards general purpose acceleration by exploiting common data-dependence forms. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 924–939.

[11] William J Dally, Yatish Turakhia, and Song Han. 2020. Domain-specific hardware accelerators. *Commun. ACM* 63, 7 (2020), 48–57.

[12] Pranav Dangi, Thilini Kaushalya Bandara, Saeideh Sheikhpour, Tulika Mitra, and Lieven Eeckhout. 2024. Sustainable Hardware Specialization. *arXiv preprint arXiv:2411.09315* (2024).

[13] Satyajit Das, Davide Rossi, Kevin JM Martin, Philippe Coussy, and Luca Benini. 2017. A 142mops/mw integrated programmable array accelerator for smart visual processing. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 1–4.

[14] Shail Dave, Mahesh Balasubramanian, and Aviral Shrivastava. 2018. RAMP: Resource-aware mapping for CGRAs. In *Proceedings of the 55th Annual Design Automation Conference*. 1–6.

[15] Shail Dave, Mahesh Balasubramanian, and Aviral Shrivastava. 2018. Ureca: Unified register file for cgras. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1081–1086.

[16] Jinyi Deng, Xinru Tang, Jiahao Zhang, Yuxuan Li, Linyun Zhang, Boxiao Han, Hongjun He, Fengbin Tu, Leibo Liu, Shaojun Wei, et al. 2023. Towards Efficient Control Flow Handling in Spatial Architecture via Architecting the Control Flow Plane. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 1395–1408.

[17] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark silicon and the end of multicore scaling. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. 365–376.

[18] Kathleen Feng, Taeyoung Kong, Kalhan Koul, Jackson Melchert, Alex Carsello, Qiaoyi Liu, Gedeon Nyengele, Maxwell Strange, Keyi Zhang, Ankita Nayak, et al. 2023. Amber: A 16-nm System-on-Chip With a Coarse-Grained Reconfigurable Array for Flexible Acceleration of Dense Linear Algebra. *IEEE Journal of Solid-State Circuits* (2023).

[19] Kermin E Fleming, Kent D Glossop, Simon C Steely Jr, Jinjie Tang, Alan G Gara, et al. 2020. Processors, methods, and systems with a configurable spatial accelerator. US Patent 10,558,575.

[20] A. Fuchs and D. Wentzlaff. 2019. The Accelerator Wall: Limits of Chip Specialization. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–14. https://doi.org/10.1109/HPCA.2019.00023

[21] Taro Fujii, Takao Toi, Teruhito Tanaka, Katsumi Togawa, Toshiro Kitaoka, Kengo Nishino, Noritsugu Nakamura, Hiroki Nakahara, and Masato Motomura. 2018. New generation dynamically reconfigurable processor technology for accelerating embedded AI applications. In *2018 IEEE symposium on VLSI circuits*. IEEE, 41–42.

[22] Graham Gobieski, Ahmet Oguz Atli, Kenneth Mai, Brandon Lucia, and Nathan Beckmann. 2021. Snafu: an ultra-low-power, energy-minimal cgra-generation framework and architecture. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 1027–1040.

[23] Graham Gobieski, Souradip Ghosh, Marijn Heule, Todd Mowry, Tony Nowatzki, Nathan Beckmann, and Brandon Lucia. 2022. Riptide: A programmable, energy-minimal dataflow compiler and architecture. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[24] Graham Gobieski, Amolak Nagi, Nathan Serafin, Mehmet Meric Isgenc, Nathan Beckmann, and Brandon Lucia. 2019. Manic: A vector-dataflow architecture for ultra-low-power embedded systems. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 670–684.

[25] Seth Copen Goldstein, Herman Schmit, Mihai Budiu, Srihari Cadambi, Matthew Moe, and R Reed Taylor. 2000. PipeRench: A reconfigurable architecture and compiler. *Computer* 33, 4 (2000), 70–77.

[26] Venkatraman Govindaraju, Chen-Han Ho, Tony Nowatzki, Jatin Chhugani, Nadathur Satish, Karthikeyan Sankaralingam, and Changkyu Kim. 2012. Dyser: Unifying functionality and parallelism specialization for energy-efficient computing. *IEEE Micro* 32, 5 (2012), 38–51.

[27] Shantanu Gupta, Shuguang Feng, Amin Ansari, Scott Mahlke, and David August. 2011. Bundled execution of recurring traces for energy-efficient general purpose processing. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. 12–23.

[28] Mahdi Hamzeh, Aviral Shrivastava, and Sarma Vrudhula. 2014. Branch-aware loop mapping on cgras. In *Proceedings of the 51st Annual Design Automation Conference*. 1–6.

[29] Jeyhun Karimov, Tilmann Rabl, and Volker Markl. 2018. Polybench: The first benchmark for polystores. In *Technology Conference on Performance Evaluation and Benchmarking*. Springer, 24–41.

[30] Manupa Karunaratne, Aditi Kulkarni Mohite, Tulika Mitra, and Li-Shiuan Peh. 2017. Hycube: A cgra with reconfigurable single-cycle multi-hop interconnect. In *Proceedings of the 54th Annual Design Automation Conference 2017*. 1–6.

[31] Changmoo Kim, Mookyoung Chung, Yeongon Cho, Mario Konijnenburg, Soojung Ryu, and Jeongwook Kim. 2014. Ulp-srp: Ultra low-power samsung reconfigurable processor for biomedical applications. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 7, 3 (2014), 1–15.

[32] Yoonjin Kim and Rabi N Mahapatra. 2009. Hierarchical reconfigurable computing arrays for efficient CGRA-based embedded systems. In *Proceedings of the 46th Annual Design Automation Conference*. 826–831.

[33] Arnold Knopfmacher and M. Mays. 2003. Graph Compositions I: Basic Enumeration. (03 2003).

[34] Zhaoying Li, Dhananjaya Wijerathne, Xianzhang Chen, Anuj Pathania, and Tulika Mitra. 2021. ChordMap: Automated Mapping of Streaming Applications onto CGRA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2021).

[35] Zhaoying Li, Dhananjaya Wijerathne, and Tulika Mitra. 2022. Coarse-Grained Reconfigurable Array (CGRA). *Handbook of Computer Architecture* (2022), 1–41.

[36] Zhaoying Li, Dan Wu, Dhananjaya Wijerathne, and Tulika Mitra. 2022. Lisa: Graph neural network based portable mapping on spatial accelerators. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 444–459.

[37] Yixuan Luo, Cheng Tan, Nicolas Bohm Agostini, Ang Li, Antonino Tumeo, Nirav Dave, and Tong Geng. 2023. ML-CGRA: an integrated compilation framework to enable efficient machine learning acceleration on CGRAs. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[38] Larry McMurchie and Carl Ebeling. 1995. PathFinder: A negotiation-based performance-driven router for FPGAs. In *Proceedings of the 1995 ACM third international symposium on Field-programmable gate arrays*. 111–117.

[39] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. 2003. ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix. In *Field Programmable Logic and Application: 13th International Conference, FPL 2003, Lisbon, Portugal, September 1-3, 2003 Proceedings 13*. Springer, 61–70.

[40] Jackson Melchert, Kathleen Feng, Caleb Donovick, Ross Daly, Ritvik Sharma, Clark Barrett, Mark A Horowitz, Pat Hanrahan, and Priyanka Raina. 2023. Apex: A framework for automated processing element design space exploration using frequent subgraph analysis. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 33–45.

[41] Ron Milo, Shai Shen-Orr, Shalev Itzkovitz, Nadav Kashtan, Dmitri Chklovskii, and Uri Alon. 2002. Network motifs: simple building blocks of complex networks. *Science* 298, 5594 (2002), 824–827.

[42] Mahim Mishra, Timothy J Callahan, Tiberiu Chelcea, Girish Venkataramani, Seth C Goldstein, and Mihai Budiu. 2006. Tartan: evaluating spatial computation for whole program execution. *ACM SIGARCH Computer Architecture News* 34, 5 (2006), 163–174.

[43] Takashi Miyamori and Kunle Olukotun. 1999. REMARC: Reconfigurable multimedia array coprocessor. *IEICE Transactions on information and systems* 82, 2 (1999), 389–397.

[44] Quan M Nguyen and Daniel Sanchez. 2021. Fifer: Practical acceleration of irregular applications on reconfigurable architectures. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 1064–1077.

[45] Chris Nicol. 2017. A coarse grain reconfigurable array (CGRA) for statically scheduled data flow computing. *Wave computing white paper* (2017), 1–9.

[46] Tony Nowatzki, Newsha Ardalani, Karthikeyan Sankaralingam, and Jian Weng. 2018. Hybrid optimization/heuristic instruction scheduling for programmable accelerator codesign. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*. 1–15.

[47] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. 2017. Stream-dataflow acceleration. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 416–429.

[48] Tony Nowatzki, Vinay Gangadhar, and Karthikeyan Sankaralingam. 2015. Exploring the potential of heterogeneous von neumann/dataflow execution models. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 298–310.

[49] Nobuaki Ozaki, Yoshihiro Yasuda, Mai Izawa, Yoshiki Saito, Daisuke Ikebuchi, Hideharu Amano, Hiroshi Nakamura, Kimiyoshi Usami, Mitaro Namiki, and Masaaki Kondo. 2011. Cool mega-arrays: Ultralow-power reconfigurable accelerator chips. *IEEE Micro* 31, 6 (2011), 6–18.

[50] Angshuman Parashar, Michael Pellauer, Michael Adler, Bushra Ahsan, Neal Crago, Daniel Lustig, Vladimir Pavlov, Antonia Zhai, Mohit Gambhir, Aamer Jaleel, et al. 2013. Triggered instructions: A control paradigm for spatially-programmed architectures. *ACM SIGARCH Computer Architecture News* 41, 3 (2013), 142–153.

[51] Hyunchul Park, Yongjun Park, and Scott Mahlke. 2009. Polymorphic pipeline array: a flexible multicore accelerator with virtualized execution for mobile multimedia applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. 370–380.

[52] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2017. Plasticine: A reconfigurable architecture for parallel patterns. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 389–402.

[53] Megan Wachs Omid Azizi Alex Solomatnikov Benjamin C. Lee Stephen Richardson Christos Kozyrakis Rehan Hameed, Wajahat Qadeer and Mark Horowitz. 2023. RETROSPECTIVE: Understanding Sources of Inefficiency in General-purpose Chips. In *ISCA@50 25-Year Retrospective: 1996-2020*, José F. Martínez and Lizy K. John (Eds.). ACM SIGARCH and IEEE TCCA. https://bit.ly/isca50_retrospective

[54] Karu Sankaralingam, Tony Nowatzki, Greg Wright, Poly Palamuttam, Jitu Khare, Vinay Gangadhar, and Preyas Shah. 2021. Mozart: Designing for software maturity and the next paradigm for chip architectures. In *2021 IEEE Hot Chips 33 Symposium (HCS)*. IEEE, 1–20.

[55] Nathan Serafin, Souradip Ghosh, Harsh Desai, Nathan Beckmann, and Brandon Lucia. 2023. Pipestitch: An energy-minimal dataflow architecture with lightweight threads. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 1409–1422.

[56] Hartej Singh, Ming-Hau Lee, Guangming Lu, Fadi J Kurdahi, Nader Bagherzadeh, and Eliseu M Chaves Filho. 2000. MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE transactions on computers* 49, 5 (2000), 465–481.

[57] Cheng Tan, Miaomiao Jiang, Deepak Patil, Yanghui Ou, Zhaoying Li, Lei Ju, Tulika Mitra, Hyunchul Park, Antonino Tumeo, and Jeff Zhang. 2014. ICED: An Integrated CGRA Framework Enabling DVFS-Aware Acceleration. In *Proceedings of the 57th Annual IEEE/ACM International Symposium on Microarchitecture*.

[58] Cheng Tan, Deepak Patil, Antonino Tumeo, Gabriel Weisz, Steve Reinhardt, and Jeff Zhang. 2023. VecPAC: A Vectorizable and Precision-Aware CGRA. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 1–9.

[59] Cheng Tan, Chenhao Xie, Tong Geng, Andres Marquez, Antonino Tumeo, Kevin Barker, and Ang Li. 2021. ARENA: Asynchronous Reconfigurable Accelerator Ring to Enable Data-Centric Parallel Computing. *IEEE Transactions on Parallel and Distributed Systems* 32, 12 (2021), 2880–2892.

[60] Cheng Tan, Chenhao Xie, Ang Li, Kevin J Barker, and Antonino Tumeo. 2020. OpenCGRA: An open-source unified framework for modeling, testing, and evaluating CGRAs. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*. IEEE, 381–388.

[61] Cheng Tan, Chenhao Xie, Ang Li, Kevin J Barker, and Antonino Tumeo. 2021. Aurora: Automated refinement of coarse-grained reconfigurable accelerators. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1388–1393.

[62] Kaushalya Bandara Thilini, Dhananjaya Wijerathne, Tulika Mitra, and Li-Shiuan Peh. 2022. REVAMP: A Systematic Framework for Heterogeneous CGRA Realization. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.

[63] Christopher Torng, Peitian Pan, Yanghui Ou, Cheng Tan, and Christopher Batten. 2021. Ultra-Elastic CGRAs for Irregular Loop Specialization. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 412–425.

[64] Matthew Vilim, Alexander Rucker, Yaqi Zhang, Sophia Liu, and Kunle Olukotun. 2020. Gorgon: Accelerating machine learning from relational data. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 309–321.

[65] Dani Voitsechov and Yoav Etsion. 2014. Single-graph multiple flows: Energy efficient design alternative for gpgpus. *ACM SIGARCH computer architecture news* 42, 3 (2014), 205–216.

[66] Dani Voitsechov, Oron Port, and Yoav Etsion. 2018. Inter-thread communication in multithreaded, reconfigurable coarse-grain arrays. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 42–54.

[67] Bo Wang, Manupa Karunarathne, Aditi Kulkarni, Tulika Mitra, and Li-Shiuan Peh. 2019. 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, 133–136.

[68] Jian Weng, Sihao Liu, Vidushi Dadu, Zhengrong Wang, Preyas Shah, and Tony Nowatzki. 2020. Dsagen: Synthesizing programmable spatial accelerators. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 268–281.

[69] Jian Weng, Sihao Liu, Zhengrong Wang, Vidushi Dadu, and Tony Nowatzki. 2020. A hybrid systolic-dataflow architecture for inductive matrix algorithms. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 703–716.

[70] Sebastian Wernicke. 2006. Efficient detection of network motifs. *IEEE/ACM transactions on computational biology and bioinformatics* 3, 4 (2006), 347–359.

[71] Dhananjaya Wijerathne, Zhaoying Li, Thilini Kaushalya Bandara, and Tulika Mitra. 2022. PANORAMA: divide-and-conquer approach for mapping complex loop kernels on CGRA. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*. 127–132.

[72] Dhananjaya Wijerathne, Zhaoying Li, Manupa Karunarathne, Anuj Pathania, and Tulika Mitra. 2019. Cascade: High throughput data streaming via decoupled access-execute cgra. *ACM Transactions on Embedded Computing Systems (TECS)* 18, 5s (2019), 1–26.

[73] Dhananjaya Wijerathne, Zhaoying Li, Manupa Karunaratne, Li-Shiuan Peh, and Tulika Mitra. 2022. Morpher: An open-source integrated compilation and simulation framework for cgra. In *Fifth Workshop on Open-Source EDA Technology (WOSET)*.

[74] Dhananjaya Wijerathne, Zhaoying Li, and Tulika Mitra. 2023. Accelerating Edge AI with Morpher: An Integrated Design, Compilation and Simulation Framework for CGRAs. *arXiv preprint arXiv:2309.06127* (2023).

[75] Dhananiaya Wijerathne, Zhaoying Li, Anuj Pathania, Tulika Mitra, and Lothar Thiele. 2021. Himap: Fast and scalable high-quality mapping on CGRA via hierarchical abstraction. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1192–1197.

[76] Dan Wu, Peng Chen*, Thilini Kaushalya Bandara, Zhaoying Li, and Tulika Mitra. 2023. Flip: Data-Centric Edge CGRA Accelerator. *ACM Transactions on Design Automation of Electronic Systems* 29, 1 (2023), 1–25.

[77] Yaqi Zhang, Alexander Rucker, Matthew Vilim, Raghu Prabhakar, William Hwang, and Kunle Olukotun. 2019. Scalable interconnects for reconfigurable spatial architectures. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 615–628.