Rewire: Advancing CGRA Mapping Through a Consolidated Routing Paradigm

Zhaoying Li[1], Dan Wu[1], Dhananjaya Wijerathne[2§], Dan Chen[1✉], Huize Li[3§], Cheng Tan[4] and Tulika Mitra[1]
[1]*School of Computing, National University of Singapore*, [2]*AMD*, [3]*University of Central Florida*, [4]*Google*
{*zhaoying, danwu20, chend, tulika*}*@comp.nus.edu.sg*, DMD.Wijerathne@amd.com, chengtan@google.com, huize.li@ucf.edu

***Abstract*—Coarse-Grained Reconfigurable Arrays (CGRAs) balance the performance and power efficiency in computing systems. Effective compilers play a crucial role in fully realizing its potential. The compiler maps Data Flow Graphs (DFGs), which represent compute-intensive loop kernels, onto CGRAs. However, existing compilers often tackle DFG nodes individually, neglecting their intricate inter-dependencies. We introduce a novel mapping paradigm called Rewire that can place and route multiple nodes in one shot. Rewire first generates routing information that is shareable among multiple nodes via propagation. Then, Rewire intersects the routing information to generate individual placement candidates for each node. Finally, Rewire innovatively utilizes data dependencies as constraints to quickly find suitable placement for multiple nodes together. Our evaluation demonstrates that Rewire can generate more near-optimal mappings than prior works. Rewire achieves 2.1x and 1.3x performance improvement and 13.5x and 4.7x compilation time reduction, respectively, compared to two popular mappers.**

I. Introduction

Coarse-grained reconfigurable arrays (CGRAs) [1]–[20] have gained significant attention in recent years due to their excellent balance in terms of flexibility, performance, and energy efficiency. The domain-agnostic characteristic allows CGRAs to accelerate a broad spectrum of applications, such as machine learning, video processing, and graph processing. However, achieving superior performance and power efficiency for these applications heavily relies on efficient compilers to explore the enormous mapping space.

Mapping DFG, which represents the compute-intensive loop kernel, onto CGRAs is a well-known challenging problem [10], [14], [21]–[30]. Figure 1 shows a typical 4×4 CGRA architecture. The CGRA is programmable, consisting of processing elements (PE) executing the operations, on-chip memory to store the data, and a Network on Chip (NoC) to route the data among the PEs. The details of the architecture are exposed to the mapper to generate cycle-by-cycle configurations for the programmable units, including the PEs and the routers.

The mapping involves the placement of the DFG nodes onto the PEs of the CGRA, while routing the data dependencies among these nodes. The complex inter-dependencies among the nodes pose significant difficulty to the mapping algorithm. Complex data dependencies refer to rich but irregular connections among multiple nodes. These nodes subtly affect each other's mapping due to resource contention, which are aggravated by intricate dependencies. Conventional mapping approaches iterate over the DFG nodes individually. For each node, they first select the PE placement candidate and then evaluate the routability of the relevant edges. It is theoretically possible to choose the placement of multiple nodes in one shot and then find the routing of the relevant edges together. Unfortunately, such consolidated placement does not yield feasible routing due to complex

§Dhananjaya Wijerathne and Huize Li were with the National University of Singapore when this research was conducted. This research is supported by the National Research Foundation, Singapore under its Competitive Research Program Award NRF-CRP23-2019-0003. The corresponding author is Dan Chen.

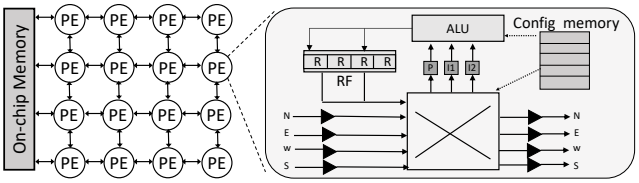


Fig. 1: A typical 4×4 CGRA.

data dependencies. Several works use a hierarchical approach [24]–[26], [31] of partitioning the DFG into several sub-DFGs. However, they still map the nodes individually within each sub-DFG.

There are two disadvantages to existing approaches. First, by mapping each node individually, they lack the ability to handle complex data dependencies collectively. As a result, these approaches need to backtrack whenever they cannot map the current node, causing numerous iterations to explore the multi-node mapping. Nevertheless, they still may fail to reach a feasible mapping within a reasonable compilation time. Second, they are unable to leverage the routing knowledge established for one edge when handling the next edge. Hence, they need to visit the same network repeatedly to evaluate the routability of the PE placement candidates.

To overcome these limitations, we propose *Rewire**, a novel CGRA mapping paradigm that can quickly generate feasible mapping for multiple nodes in one shot. As existing mappers usually create an initial mapping and improve step by step to achieve a valid mapping, Rewire focuses on amending the initial mapping and can take any initial mapping from other mappers, making Rewire orthogonal to these mappers. Rewire first leverages the inherent DFG structure characteristic to share the routing information among multiple DFG nodes and intersects the routing information to find placement candidates for each of the DFG nodes. Compared to previous path-finding-based routing, Rewire devises a propagation mechanism to generate the routing information for multiple nodes together. Finally, Rewire analyzes routing information and prunes the search space using inter-dependencies as constraints to generate multi-node placement. Therefore, Rewire can simultaneously coordinate the placement and routing of multiple nodes, advancing the current mapping paradigm.

We make the following key contributions:

- We identify a fundamental limitation of traditional CGRA mapping approaches that they can only map the DFG nodes individually, limiting the ability to coordinate multi-node mapping. In contrast, we propose a new mapping paradigm that can generate multi-node mapping in one shot.
- To the best of our knowledge, we are the first to propose sharing the routing information among multiple nodes. Through the sharing, we can reuse the routing information and further coordinate the mapping of multiple nodes.

*We name it Rewire due to the reuse of wire (routing) information and its ability to amend an invalid mapping through minimal alterations.

- Our evaluation demonstrates that Rewire can generate much more near-optimal mappings than prior works. Compared to the two popular works, Rewire can achieve 2.1x and 1.3x performance improvement and 13.5x and 4.7x compilation time reduction, respectively.

II. Related Work

Existing mappers can be classified into three main categories: (1) These approaches choose the placement that minimizes a predefined cost function to map a node [21], [23], [32]–[35]. For example, to map a DFG node, they need to evaluate the multiple placement candidates and select the candidate with the minimal cost. This process necessitates evaluating all potential candidates for the current node and often leads to local minima. If the current node cannot be mapped, a higher cost is assigned to the existing placement, and multiple rounds of partial remappings are performed to achieve a feasible solution. (2) These approaches find any placement that can route the edges connected to the current DFG node but may require many iterations to find a feasible mapping [10], [22], [29], [36]. To map a node, they can select any placement candidate if it can route the current data dependencies, and they do not need to evaluate any other candidates. (3) These approaches partition the DFG or the hardware to reduce the search space using a divide-and-conquer approach [24]–[26], [31], but may still rely on the aforementioned methods for each partitioned DFG. For example, if a partitioned sub-DFG is assigned to a sub-CGRA, they only need to evaluate candidates in the corresponding sub-CGRA. Despite substantial research, state-of-the-art CGRA compilers still only map one node at a time and cannot comprehensively manage multi-node data dependencies together. In contrast, Rewire can simultaneously map multiple DFG nodes together, greatly reducing the number of iterations needed to map multiple DFG nodes.

Furthermore, conventional approaches cannot reuse the routing knowledge, with the exception of EMS [28]. There are three levels in mapping: a mapper needs to map multiple DFG nodes, where mapping a DFG node needs to explore multiple placement candidates, and evaluating a candidate needs to route multiple edges. EMS can reuse the routing knowledge for a single edge when attempting different placement candidates. To map a node, EMS starts routing from its parent node and does not specify the PE candidate for the target node. Upon reaching a potential PE candidate, EMS employs a path search to route the other relevant edges. If unsuccessful, EMS still uses the routing information of the first edge and continues to find new potential PE candidates. EMS can only reuse the routing knowledge for a single edge and cannot extend to other edges. In contrast, Rewire can share the routing information among multiple edges, placement candidates, and DFG nodes, making it possible to coordinate the mapping of multiple DFG nodes.

III. Motivation and Intuition

In this section, we use a motivating example to demonstrate the fundamental limitations of the state-of-the-art CGRA compilation techniques and propose concrete strategies to overcome these limitations. Figure 2(a) shows a DFG example where three nodes A, B, and G have been placed onto PEs in the initial mapping. Figure 2(b) provides an example of initial mapping for the DFG depicted in Figure 2(a) onto a 4×4 CGRA. Figure 2(c) illustrates how typical conventional mappers [22], [26] attempt to map the nodes C, D, E, and F: map each node individually in multiple steps and can require checking multiple placement candidates for each DFG node.

The key problem is that such approaches cannot simultaneously coordinate the mapping of multiple nodes. For example, when it

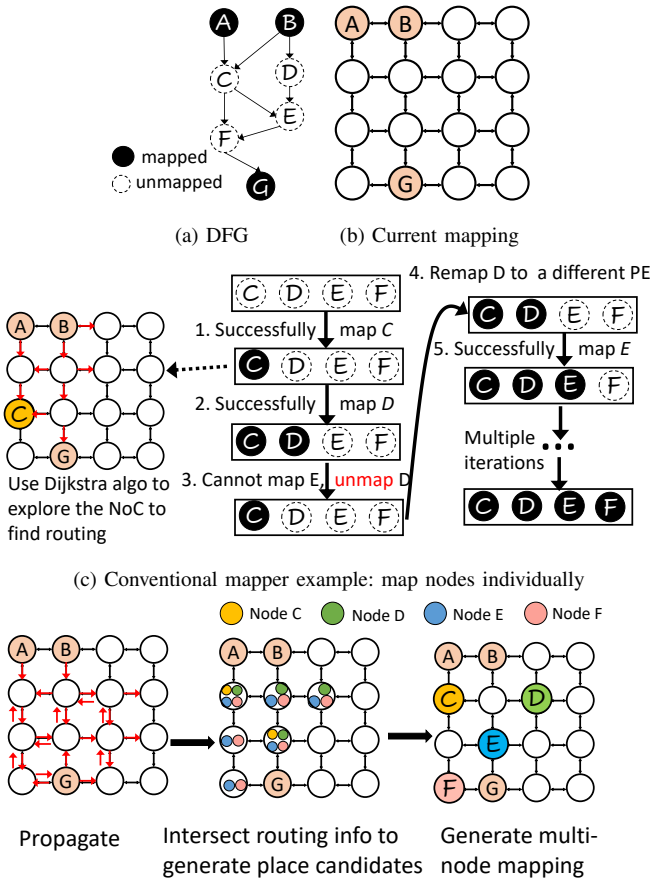


Fig. 2: Comparison between conventional mappers and proposed mapper

attempts to place node C onto a PE, it cannot guarantee that this placement leads to a successful mapping of the remaining nodes D, E, and F. Thus, when node E cannot be mapped (step 3), it usually performs partial remapping by removing a part of the current mapping and changing the placement of the affected nodes to different PEs. In this example, we unmap node D and move it to a different PE. Hence these conventional mappers need multiple backtracking iterations to find a feasible mapping. Another issue is that these conventional mappers cannot reuse routing knowledge either for multiple edges, placement candidates, or DFG nodes. For each edge or placement candidate, the mapper needs to explore the NoC to establish the routing and need to visit the same network multiple times. Yet, they did not reuse the routing information. Furthermore, simple reuse or routing information memorization cannot be applied as the resource status can change from routing one edge to another edge.

In this paper, we demonstrate that it is possible to simultaneously map multiple nodes and share the routing information among relevant edges. Our approach is shown in Figure 2(d). We first propagate the output value of A along the network without specifying any destination node. Any reachable PEs from the PE containing DFG node A can be a potential placement candidate for node C, which is dependent on node A. Moreover, as the nodes E and F are dependent on node C, by transitivity, they are also dependent on node A. Therefore, if C is placed on a reachable PE from A, we can place E and F on the PEs reachable from C. All these reachable PEs that are potential placement candidates for DFG nodes E and F have been identified during the propagation from A. We can use

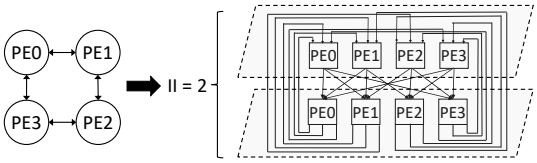


Fig. 3: High-level MRRG example for 2×2 CGRA.

the same propagation information to route $A \rightarrow C$, $C \rightarrow E$, and $C \rightarrow F$. Similarly, we can propagate the output of B and use this information to find potential placement candidates for the nodes D, E, and F, and routing the edges $B \rightarrow D$, $D \rightarrow E$, and $E \rightarrow F$. Finally, we need to conduct backward propagation from node G that has already been mapped, just as we do forward propagation from nodes A and B. The above process identifies the potential placement candidates via forward and backward reachability searches from the already placed ancestor nodes (A and B in the example) and the descendant nodes (G in the example).

However, the placement of a DFG node X needs to satisfy multiple constraints as it needs to be reachable from all the source nodes with which it has dependencies and all the destination nodes that are dependent on this node X. For example, node C should be on the forward propagation routes of nodes A, B, and the backward propagation routes of node G. A simple intersection operation among the propagation routes from the nodes A, B, and G can identify the set of potential feasible placement candidates for node C. Similarly, we can identify the set of potential candidates for nodes D, E, and F. Given these potential candidates for all the nodes we intend to map, we have to select the actual placement candidate for multi-node such that (a) only one DFG node is mapped to a PE candidate, (b) we can route all the data dependencies among these nodes together.

IV. Rewire Algorithm

In this section, we first formulate the mapping problem and then present the Rewire mapper that maps the DFG onto the CGRA. The CGRA mapper first generates a DFG representing the application program and then maps the DFG onto the CGRA with respect to the data dependency. CGRA uses Initiation Interval (II) to evaluate the mapping quality, where II is the number of cycles between consecutive iterations of a loop kernel. The lower the II, the better the performance.

A. Problem Formulation

DFG: We define DFG $D = (V_D, E_D)$ as a directed acyclic graph with V_D representing operations and E_D representing dependencies between operations.

CGRA Graph: CGRA is defined as a graph $G = (V_G, E_G)$. Modulo Routing Resource Graph (MRRG) $H_{II} = (V_H, E_H)$ is a resource graph of the CGRA that is time extended to II cycles [37]. V_H consists of two types of nodes: ALUs (V_H^F) in each PE and ports (V_H^P) in interconnects and Register Files (RFs) [37]. As the CGRA schedule repeats after II cycles, the resources at cycle $II-1$ have connectivity with the resources at cycle 0 in the MRRG. $H^{'} = (V_{H'}, E_{H'})$ is defined as the time extended resource graph of the $G^{'}$ ($V_{H'} \subseteq \mathcal{P}(V_H)$). Figure 3 shows a high-level MRRG example for 2×2 CGRA where we only show the inter-PE links and do not include internal links in the PE.

Problem Definition: Given a kernel and a CGRA, the problem is to construct a time extended MRRG $H_{II} = (V_H, E_H)$ of the CGRA for which there exists a mapping $\phi = (\phi_V, \phi_E)$ from $D = (V_D, E_D)$ to H_{II} which minimizes II. We decompose the problem into three sub-problems: selecting multiple DFG nodes to form a cluster U, propagation for U, and generating the multi-node mapping of U.

Algorithm 1: Rewire Algorithm

```
Input: DFG, CGRA
Output: Valid mapping
1  Sort DFG nodes by topological order;
2  minII = Minimum(recurrenceMII, resourceMII);
3  while mapping is not valid do
4      Generate the initial mapping for current II ;
5      while mapping is not valid do
6          Generate target cluster U;
7          while U is not mapped and is less than α do
8              Propagate for U;
9              Intersect propagation tuples to find placement candidates;
10             if Generate multi-node mapping then
11                 Break;
12             else
13                 Append a node to U;
14         if U are not mapped then
15             Break;
16     II = II + 1;
```

B. Overall Algorithm

Algorithm 1 shows the Rewire algorithm. We first sort DFG nodes by topological order (line 1). The mapper calculates the theoretical Minimum Initiation Interval (MII) based on the hardware resource and inter-iteration data dependency [38] (line 2). Given a DFG and a CGRA, we initiate the mapping process starting from MII. If the mapping is not successful, we increase the II by one and attempt to map again. This process continues until we reach the maximum II (hardware-decided) or exceed the mapping time limit (lines 3-16).

As we mentioned earlier, Rewire takes the initial mapping from conventional approaches [22] (line 2) and focuses on amending the initial invalid mapping to achieve a valid mapping (lines 5-15). Given the invalid initial mapping, Rewire identifies nodes that are non-mapped or have congested routing as ill-mapped nodes. Rewire randomly selects several unmapped connected nodes to let them share the routing information, denoted as U (line 6). We need to limit the size of U. The reason is: (1) too many DFG nodes in U will significantly increase the search space and make it very time-consuming to find suitable placement candidates for U; (2) increasing the number of DFG nodes might not always work as the DFG is not mappable at current II (MII is not always achievable). In practice, we limit the size of U to 15. Given U, Rewire simultaneously propagates the source nodes, which are the parents and children of the cluster U (line 8). Then, Rewire summarizes the propagation information and uses the intersection to generate the placement candidates for each node $v \in U$ (line 9). Rewire uses data dependencies as constraints to prune invalid cluster placement candidates during building the placement of cluster U (line 10). The propagation and mapping generation will be detailed in Section IV-C and Section IV-D.

It is possible that the original U is not mappable due to the limited routing resource and complex data dependencies. Hence, we progressively add the connected node to U (line 13) until we map these nodes or reach the size limit of U (line 7). To append, we select the node that has the least depth-fist search (DFS) distance to the cluster U. Reaching the size limit means we cannot find a valid mapping with current II and need to increase II by one.

C. Routing Propagation

Rewire conducts forward propagation of values from the parents of U ($Parents(U)$) and backward propagation from the children of U ($Children(U)$) across the network. $Parents(U)$ is a set of DFG nodes that have been mapped and have a child node in U, but the node itself does not belong to U, denoted as $Parents(U) =$

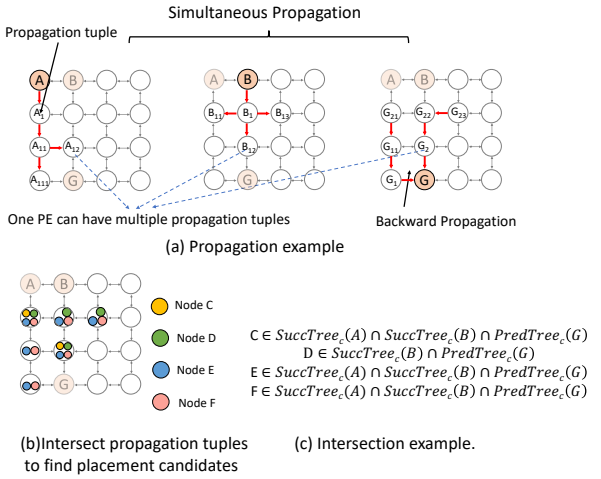


Fig. 4: Propagation and intersection example.

$v \in V \setminus U : \exists u \in U, (v,u) \in E$ and $\phi(v) \neq \emptyset$. Similarly, $Children(U) = v \in V \setminus U : \exists u \in U, (u,v) \in E$ and $\phi(v) \neq \emptyset$. Rewire propagates as long as the network path is not used by current mapping. Once reaching a PE, we will generate a propagation tuple. The propagation tuple contains four essential components: the source node, the propagation direction (forward/backward), the source PE, and the number of routing cycles. Thus, each tuple is associated with a specific PE in the network. The number of routing cycles is the cycle number from the source PE to the current PE.

The propagation tuple is a probe of network utilization that helps Rewire find suitable placement candidates. To keep the only necessary information, when a PE receives a propagation message from an adjacent PE, Rewire generates a new propagation tuple only if no existing tuple at that PE has the identical combination of source node, routing cycle count, and propagation direction. This mechanism prevents redundant exploration while covering all potential routing paths. To determine the number of propagation rounds, we heuristically use the maximum cycle difference between $Parents(U)$ and $Children(U)$ multiplied by three as the number of propagation rounds. When either $Parents(U)$ or $Children(U)$ is empty, we heuristically use the length of the longest path within U multiplied by five. This value allows the Rewire to generate sufficient propagation tuples.

Figure 4(a) provides an example of propagation for the motivating example in Section III. We note that we simultaneously propagate these sources nodes and we show them separately for simplicity. The IDs, like A_{12}, are to distinguish from each other and do not have any special meaning. The simultaneous propagation of multiple source nodes introduces resource contention in the network. Since the objective of propagation is to explore potential routing paths rather than perform final resource allocation, we continue propagation even when hardware resources have been traversed by other propagation tuples. These propagations generate a hierarchical structure where we define the successor tree $SuccTree(v)$ as the set of all propagation tuples originating from node v through forward propagation and the predecessor tree $PredTree(v)$ for backward propagation. We define $Tree(v)$ to represent all propagation tuples propagated from v.

D. Multi-node Mapping Generation

Propagation Information Intersection: After propagation, we first intersect propagation tuples to find placement candidates for DFG nodes in target cluster U. Now, each PE V_H virtually owns multiple propagation tuples, which are from different source nodes and arrive at different cycles. Figure 4(b) shows an example of intersecting these propagation tuples to find placement candidates for the unmapped DFG nodes in Figure 2(a). For a node v in U, PE V_H can be a placement candidate if it has the required propagation tuples for v. The requirement is, for each edge (v,u) (or (u,v)), or for each parent and child, we need to have a corresponding tuple, and these corresponding tuples must arrive at the same cycle to ensure the "right" execution of node v_i. If a parent or child node of v in U is not the source node of propagation, we use DFS to find a source node to represent this parent or child. For example, for the DFG in Figure 2(a), F is not a propagation source node, and we use DFS to find G as the propagation source node. In this example, three edges are connected to C; therefore, a PE can be a placement candidate of C only if the PE has all three corresponding tuples to ensure routing paths. In other words, we need to have three tuples originating from A, B, and G, and these tuples must arrive at the same cycle.

Hence, we formalize the intersection rule as below:

$$PCandidates(v_i) = \{V_H^F | \exists c \in \mathbb{N} : V_H^F \in (\bigcap_{v_j \in Parents(v_i)} SuccTree_c(source(v_j))) \cap (\bigcap_{v_k \in Children(v_i)} PredTree_c(source(v_k)))\} \quad (1)$$

where $SuccTree_c(v)$ represents the successor tree from node v at cycle c, $PredTree_c(v)$ represents the predecessor tree from node v at cycle c, and c is the common arrival cycle for all required tuples. For example, for node C, we can obtain the candidates with:

$$PCandidates(C) = \{V_H^F | \exists c \in \mathbb{N} : V_H^F \in (SuccTree_c(A) \cap SuccTree_c(B) \cap PredTree_c(G))\} \quad (2)$$

Figure 4(c) provides a few simplified examples using intersection to find the placement candidates for unmapped nodes. With the propagation and intersection example, we can find we only need several rounds of propagation to let these DFG nodes find potential placement candidates, even though the process has not yielded the final multi-node mapping yet. In contrast, conventional approaches usually need to invoke the path routing algorithm multiple times to check multiple candidates for only one DFG node placement.

Algorithm 2: Generate multi-node mapping

Input: DFG, CGRA, cluster U, $PCandidates(v_i)$ for each $v_i \in U$
Output: Mapping

1 Sort U by topological order $U = \{v_i | i = 0, ..., m\}$;
2 **foreach** $v_i \in U$ **do**
3 | Sort placement candidates by available execution cycles;
4 **while** *not exceed the time limitation* **do**
| // generate $Placement(U)$
5 | **foreach** v_i ($0 \leq i \leq m$) **do**
6 | | **foreach** v_j ($0 \leq j \leq i-1$) **do**
7 | | | Generate time constraints related to current node v_i;
8 | | select a candidate obeying execution cycle constraints;
9 | **if** *routing verified* **then**
10 | | return the $Placement(U)$;

Multi-node Mapping Generation: Algorithm 2 shows how to generate multi-node mapping. After the intersection of propagation tuples, we generate placement candidates $PCandidates(v_i)$ for each node $v_i \in U$, where placement candidates can execute at different cycles (decided by the arriving cycles of propagation tuples). We utilize the execution cycle information to filter invalid placement for U. We first sort nodes in cluster U by topological order (line 1) and sort placement candidates for each DFG node v_i by their execution cycles (lines 2 and 3). Then, we build placement candidates for U (represented as $Placement(U)$), which need to obey the execution order among nodes in U. For example, for the DFG in

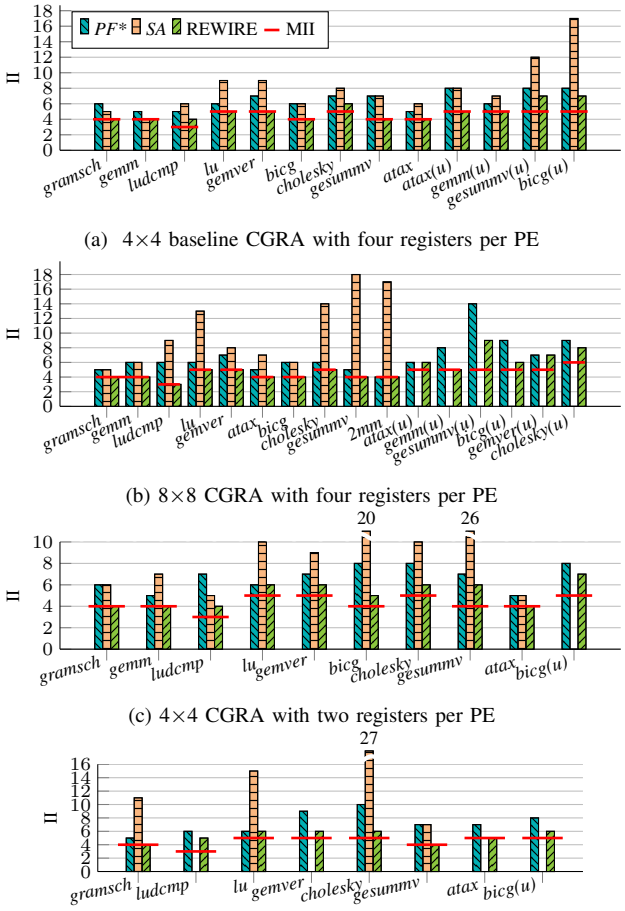
Figure 2(a), the execution cycle of E's placement candidate must be larger than the cycle of C's with respect to the data dependency. As $PCandidates(v_i)$ can have different execution cycles, we need to remove the invalid combinations during $Placement(U)$ generation. Hence, we iterate from v_0 to v_m to build $Placement(U)$ (lines 5-8) and check the execution cycle constraints (lines 6-8) during this process. Moreover, we use an index vector to track these placement candidates and iterate them to ensure we do not explore the same $Placement(U)$. When we add a placement candidate for v_i, we check the cycle execution constraints from v_0 to v_{i-1} if it has any data dependency with v_i. We find that this method is quite effective in pruning invalid $Placement(U)$.

As we mentioned in propagation, the propagation tuple may not always be valid, as we do not check resource contention among tuples during propagation. Hence, we must verify the $Placement(U)$ through routing (lines 9 and 10). If not successfully verified, we keep finding valid $Placement(U)$ (lines 4-10). After propagation, only part of PE can be the placement candidate of the DFG node, while the cycle execution constraints prune most invalid $Placement(U)$, meaning we only need to check a small portion of $Placement(U)$. Our evaluations show that generated $Placement(U)$ has a very high success rate of around 95% for verification. The reason is that the execution cycle constraints have successfully pruned many invalid $Placement(U)$, and the propagation mechanism has already performed a preliminary evaluation of routability.

V. Evaluation

In this section, we evaluate Rewire against two recent CGRA mapping techniques in terms of application performance (iteration interval II) and compilation time. We compare Rewire against two mapping approaches: SA and PF*. Simulated Annealing (SA) has been widely used in different CGRA compilation frameworks [10], [29], [32]. PathFinder is a popular algorithm used by several CGRA mappers [2], [22], [39], [40]. PathFinder and SA are two representative examples of the first type and the second type of mapper as discussed in Section II, respectively. We refer to our fine-tuned implementation of PathFinder with 3K lines of C++ code as PF*. The idea of PF* is to generate an initial mapping by selecting the placement with the minimal routing cost for the edges and then amend the mapping through multiple remapping iterations until a feasible solution is reached. We use the initial mapping of PF* as the initial mapping for Rewire, and then use our multi-node mapping to reach a feasible solution from the initial infeasible mapping.

We evaluate these mappers on the following CGRA architectures: (a) 4×4 CGRA with four registers per PE; (b) 8×8 CGRA with four registers per PE; (c) 4×4 CGRA with two registers per PE; (d) 4×4 CGRA with one register per PE. All the 4×4 CGRAs have two local memory banks, and PEs on the left-most column can access the memory. The 8×8 CGRA have eight local memory banks, and the left-most and right-most PE columns can access the memory, making 16 PEs can access the memory. Registers play a crucial role in buffering data during routing, particularly when the producer and consumer are distant nodes along spatial or temporal dimensions. We evaluate with different numbers of registers to evaluate Rewire's capability to handle different routing resources. 4×4 CGRA with one register per PE is not practical as it has very limited routing resources. Yet, we use it to evaluate the capability to handle extreme cases. We use the PolyBench [41], MachSuite [42] and MiBench [43] for our applications. We also used unrolled versions (unroll factor of 2) of some of these benchmarks to stress the compiler, specially on 8×8 CGRA. The number of DFG nodes varies from 26 to 51 and the average number of DFG nodes is 38. We measure the compilation time on the Intel Xeon Gold CPU (2.60GHz).



(a) 4×4 baseline CGRA with four registers per PE

(b) 8×8 CGRA with four registers per PE

(c) 4×4 CGRA with two registers per PE

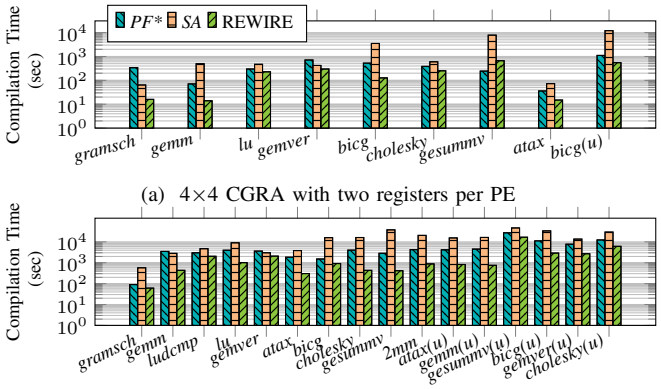
(d) 4×4 CGRA with one register per PE

Fig. 5: Comparison on the mapping quality of Rewire with PF* and SA on different benchmarks and CGRA configurations. The lower the II value, the higher the performance. *u* next to a benchmark represents the unrolled version with a factor 2. MII is the theoretically minimal II that may not be achievable in practice. Missing bars for SA for some benchmarks indicate that SA was not successful in mapping those benchmark-architecture configurations. We did not include a benchmark-architecture configuration if none of the three approaches can map it due to inadequate availability of resources, e.g., unrolled loops on architectures with few registers per PE.

A. Quality of Mapping

Figure 5 shows the comparison of the mapping quality for the different approaches (Rewire, PF*, and SA) in terms of the II values on four different CGRA configurations. Recall that II is the number of cycles between starting the execution of consecutive iterations of a loop kernel. The lower II value indicates lower execution time and hence higher performance. MII is the theoretically minimal II which is calculated by resource and inter-iteration data dependencies. However, achieving the MII may not always be possible as complex data dependency constraints are not considered in the theoretical calculation. Nevertheless, we consider a mapping optimal if it achieves the MII. If the difference between the II achieved and MII is one, we call the mapping near-optimal.

This evaluation uses 47 different DFG and architecture combinations. Rewire can achieve superior performance than PF* and SA if not the same on every application and CGRA combination. Both Rewire and PF* successfully map all the combinations, whereas SA



(a) 4×4 CGRA with two registers per PE

(b) 8×8 CGRA with four registers per PE

Fig. 6: Compilation time comparison of Rewire with PF* and SA on 4×4 CGRA with four registers per PE and with two registers per PE.

fails to map 12 combinations. Among the 47 combinations, Rewire delivers optimal or near-optimal mapping in 38 cases. In comparison, PF* and SA achieve optimal or near-optimal mapping for only 15 and 3 cases, respectively. Overall, Rewire delivers an average performance speedup of 1.3x and 2.1x compared to PF* and SA, respectively. The mapping quality difference shows Rewire's ability to handle complex data dependencies across application kernels on different CGRAs.

Rewire is also versatile and scalable to support various CGRA configurations. Rewire consistently achieves 1.33x, 1.31x, 1.29x, and 1.32x performance improvement compared to PF* on 4×4 baseline CGRA with four registers per PE, 4×4 CGRA with one register per PE, 4×4 CGRA with two registers per PE, and 8×8 CGRA with four registers per PE, respectively. This consistency underlines Rewire's ability to achieve superior performance across varied architectures. Moreover, compared to SA, Rewire can achieve 1.58x, 3.25x, 2.10x, and 2.21x performance improvement on these architectures, respectively. The most significant improvement comes from the 4×4 CGRA with one register. This architecture has very limited routing resources to evaluate the capability to handle extreme cases, as mentioned earlier, and a few DFGs can be mapped on the architecture. This improvement also highlights Rewire's capability to handle various architectures. The key factor behind the success of Rewire is its ability to coordinate the mapping of multiple DFG nodes, while PF* and SA make localized and myopic decision, making them struggle to handle intricate data dependencies.

Rewire achieves significant performance improvement from 4×4 CGRA with 4 registers per PE to 8×8 CGRA with 4 registers per PE. Rewire generates optimal mapping for *cholesky*, *ludcmp*, and *gesumv(u)* for the 8×8 CGRA, while fails on 4×4 CGRA to generate mappings due to insufficient hardware resources. This demonstrates the scalability of Rewire to handle larger search space by effectively pruning away infeasible candidates to avoid unnecessary searches.

B. Compilation Time

Figure 6 shows the compilation time comparison among these mappers on 8×8 CGRA with four registers per PE and 4×4 CGRA with two registers per PE. Note that the Y-axis is in log scale. Mappers can explore for a maximum of one hour per II, and can terminate early at each II due to the backtracking limitation (PF*) or no mapping cost improvement after 100 iterations (SA). As mentioned earlier, SA cannot map several benchmarks even after numerous iterations. In those cases, we choose the termination time as the compilation time. We note that the propagation time usually takes less than one second in Rewire, and the most time-consuming step happens at pruning during multi-node mapping generation.

TABLE I: Number of single-node remapping iterations for PF* and SA on 4×4 CGRA with four registers per PE and with one register per PE.

4×4 CGRA with 1 register per PE			4×4 CGRA with 4 registers per PE		
Benchmark	**PF***	**SA**	**Benchmark**	**PF***	**SA**
gramsch	242	1335	gramsch	323	1084
ludcmp	362	1422	ludcmp	315	1494
lu	233	1231	lu	233	1440
gemver	339	1428	gemver	293	1208
cholesky	332	1390	cholesky	332	1116
gesummv	935	1440	gesummv	228	1259
atax	232	1451	atax	242	920
bicg(u)	442	1425	bicg(u)	423	1341

On 4×4 CGRA with two registers per PE, Rewire achieves 4.2x and 13.5x compilation time reduction compared to PF* and SA, respectively. On 8×8 CGRA with four registers per PE, Rewire achieves 2.47x and 6.97x compilation time reduction compared to PF* and SA, respectively. Overall, Rewire achieves significant compilation time reduction compared to other approaches. The main reason is that Rewire maps multiple nodes in one go and does not need numerous iterations to map each node individually and backtrack (remap) if it fails to map future nodes. Moreover, the compilation time on 8×8 CGRA is significantly higher than 4×4 CGRA due to the larger search space, but Rewire can still achieve substantial compilation time reduction compared to the other two approaches, demonstrating the efficiency of the pruning algorithm.

Table I shows the number of iterations for PF* and SA on 4×4 CGRA with four registers per PE and 4×4 CGRA with one register per PE. In each iteration, both methodologies select one node to unmap. As the mappers need to generate a new initial mapping for each explored II value, we use the average number of remapping iterations from the start II to the final mapped II. Both PF* and SA need a significant number of single-node mapping iterations, while Rewire only needs to construct the right cluster of multiple nodes and amend their invalid mappings in one shot. SA needs much more remapping iterations than PF*. The reason is that PF* evaluates all the placement candidates for each single-node remapping and selects the best one, while SA selects one candidate randomly. Thus, PF* has a better chance of success. Moreover, PF* and SA generally need more remapping iterations on CGRA with one register per PE compared to CGRA with four registers per PE. The reason is that the former has fewer routing resources and needs more iterations to find a mapping. Rewire might need multiple iterations to build U as we progressively append the node to U. However, U cannot be bigger than the size of DFG, and we have limited the size to 15, as mentioned previously. Hence, Rewire takes significantly fewer iterations than both methodologies.

In summary, for almost all applications and CGRA combinations, Rewire substantially outperforms PF* and SA in mapping quality (application performance) and compilation time. With the consolidated routing paradigm, Rewire scales with CGRA size and complex DFGs and well handles architectures with limited routing resources.

VI. Conclusion

Existing mapping approaches place and route nodes individually, limiting the ability to comprehensively coordinate multi-node mapping. Rewire presents a consolidated routing paradigm for CGRA mapping that effectively handles complex multi-node dependencies. By mapping multiple nodes in one shot through propagation and consolidation of routing information, our approach achieves significant improvements in both performance and compilation time compared to two popular mappers.

REFERENCES

[1] Zhaoying Li et al. Coarse-grained reconfigurable array (cgra). *Handbook of Computer Architecture*, pages 1–41, 2022.

[2] Manupa Karunaratne et al. Hycube: A cgra with reconfigurable single-cycle multi-hop interconnect. In *DAC*, 2017.

[3] Raghu Prabhakar et al. Plasticine: A reconfigurable architecture for parallel patterns. In *ISCA*, pages 389–402. IEEE, 2017.

[4] Dhananjaya Wijerathne et al. Cascade: High throughput data streaming via decoupled access-execute cgra. *TECS*, 2019.

[5] Yaqi Zhang et al. Scalable interconnects for reconfigurable spatial architectures. In *ISCA*. IEEE, 2019.

[6] Manupa Karunaratne et al. 4d-cgra: Introducing branch dimension to spatio-temporal application mapping on cgras. In *ICCAD*. IEEE, 2019.

[7] Ankita Nayak et al. A framework for adding low-overhead, fine-grained power domains to cgras. In *DATE*. IEEE, 2020.

[8] Michael Pellauer et al. Efficient control and communication paradigms for coarse-grained spatial architectures. *TOCS*, 2015.

[9] Rick Bahr et al. Creating an agile hardware design flow. In *DAC*. IEEE, 2020.

[10] Jian Weng et al. Dsagen: Synthesizing programmable spatial accelerators. In *ISCA*. IEEE, 2020.

[11] Tan Cheng et al. Aurora: Automated refinement of coarse-grained reconfigurable accelerators. In *DATE*. IEEE, 2021.

[12] Kaushalya Bandara Thilini et al. Revamp: A systematic framework for heterogeneous cgra realization. In *ASPLOS*, 2022.

[13] Tony Nowatzki et al. Stream-dataflow acceleration. In *ISCA*, 2017.

[14] Karthikeyan Sankaralingam et al. Exploiting ilp, tlp, and dlp with the polymorphous trips architecture. In *ISCA*, 2003.

[15] Graham Gobieski et al. Riptide: A programmable, energy-minimal dataflow compiler and architecture. In *MICRO*, 2022.

[16] Graham Gobieski et al. Snafu: an ultra-low-power, energy-minimal cgra-generation framework and architecture. In *ISCA*. IEEE, 2021.

[17] Yaqi Zhang et al. Sara: Scaling a reconfigurable dataflow accelerator. In *ISCA*. IEEE, 2021.

[18] Zhaoying Li et al. Enhancing cgra efficiency through aligned compute and communication provisioning. In *ASPLOS*, 2025.

[19] Miaomiao Jiang et al. Fhe-cgra: Enable efficient acceleration of fully homomorphic encryption on cgras. In *DAC*, 2024.

[20] Cheng Tan et al. Iced: An integrated cgra framework enabling dvfs-aware acceleration. In *MICRO*. IEEE, 2024.

[21] Zhaoying Li et al. Lisa: Graph neural network based portable mapping on spatial accelerators. In *HPCA*. IEEE, 2022.

[22] Stephen Friedman et al. Spr: an architecture-adaptive cgra mapping tool. In *FPGA*, 2009.

[23] Mahdi Hamzeh et al. Regimap: Register-aware application mapping on coarse-grained reconfigurable architectures (cgras). In *DAC*, 2013.

[24] Dhananjaya Wijerathne et al. Panorama: divide-and-conquer approach for mapping complex loop kernels on cgra. In *DAC*, 2022.

[25] Xingchen Man et al. Casmap: agile mapper for reconfigurable spatial architectures by automatically c lustering intermediate representations a nd s cattering mapping process. In *ISCA*, 2022.

[26] Mingyang Kou et al. Geml: Gnn-based efficient mapping method for large loop applications on cgra. In *DAC*, 2022.

[27] Shail Dave et al. Ramp: Resource-aware mapping for cgras. In *ISCA*, 2018.

[28] Hyunchul Park et al. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In *PACT*, 2008.

[29] S Alexander Chin et al. Cgra-me: A unified framework for cgra modelling and exploration. In *ASAP*, pages 184–189. IEEE, 2017.

[30] Zhaoying Li et al. Chordmap: Automated mapping of streaming applications onto cgra. *TCAD*, 2021.

[31] Matthew JP Walker et al. Generic connectivity-based cgra mapping via integer linear programming. In *FCCM*. IEEE, 2019.

[32] Dhananjaya Wijerathne et al. Morpher: An open-source integrated compilation and simulation framework for cgra. In *WOSET*, 2022.

[33] Cheng Tan et al. Opencgra: An open-source unified framework for modeling, testing, and evaluating cgras. In *ICCD*. IEEE, 2020.

[34] Yan Zhuang et al. Towards high-quality cgra mapping with graph neural networks and reinforcement learning. In *ICCAD*, 2022.

[35] Tony Nowatzki et al. A general constraint-centric scheduling framework for spatial architectures. *ACM SIGPLAN Notices*, 48, 2013.

[36] S Alexander Chin and Jason H Anderson. An architecture-agnostic integer linear programming approach to cgra mapping. In *DAC*, 2018.

[37] Bingfeng Mei et al. Dresc: A retargetable compiler for coarse-grained reconfigurable architectures. In *FPT*. IEEE, 2002.

[38] B Ramakrishna Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Micro*, pages 63–74, 1994.

[39] Frederick Tombs et al. Mocarabe: High-performance time-multiplexed overlays for fpgas. In *FCCM*, pages 115–123. IEEE, 2021.

[40] Su Zheng et al. Fastcgra: A modeling, evaluation, and exploration platform for large-scale coarse-grained reconfigurable arrays. In *ICFPT*. IEEE, 2021.

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

[42] Brandon Reagen et al. Machsuite: Benchmarks for accelerator design and customized architectures. In *IISWC*. IEEE, 2014.

[43] Matthew R Guthaus et al. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the fourth annual IEEE international workshop on workload characterization.* IEEE, 2001.