GRAMM: Fast CGRA Application Mapping Based on A Heuristic for Finding Graph Minors

Guanglei Zhou, Mirjana Stojilović†, Jason H. Anderson
Dept. of Electrical and Computer Engineering, University of Toronto, Toronto, Canada
†School of Computer and Communication Sciences, EPFL, Lausanne, Switzerland
guanglei.zhou@mail.utoronto.ca, mirjana.stojilovic@epfl.ch, janders@ece.utoronto.ca

***Abstract*—A graph H is a minor of a second graph G if G can be transformed into H by two operations: 1) deleting nodes and/or edges, or 2) contracting edges. Coarse-grained reconfigurable array (CGRA) application mapping is closely related to the graph minor problem, where H is the application's dataflow graph and G is the CGRA's device-model graph. A heuristic algorithm to find graph minors has proven to be practical for sparse graphs with hundreds of vertices in a quantum computing application. In this work, we adapt the heuristic to CGRA application mapping, where the graphs have directed edges, and the vertices have unique types (e.g., representing ALUs or interconnect). Additionally, we alter the original cost function, taking inspiration from PathFinder, an iterative negotiated-congestion routing algorithm. In an experimental study comparing with a CGRA mapper based on integer linear programming, we demonstrate a higher rate of successful mappings and from 80× to up to orders of magnitude lower runtime.**

I. Introduction

Coarse-grained reconfigurable arrays (CGRAs) are programmable hardware platforms with large ALU-like configurable processing elements (PEs) and word-wide configurable interconnect. From the perspectives of area, power, performance, and flexibility, CGRAs lie between custom Application-Specific Integrated Circuits (ASICs) and Field-Programmable Gate Arrays (FPGAs). As opposed to FPGAs, where one can configure interconnect and logic at a bit level, CGRAs offer configurability at word-wide level for both interconnect and compute. As Moore's law and Dennard scaling trends slow [1], CGRAs are emerging as a promising direction for application acceleration. To use a CGRA, an application needs to be successfully *mapped* onto it. CGRA mapping is a computer-aided design (CAD) task, which involves designating the PEs in the CGRA to perform target computations, deciding when such computations are performed (scheduling), as well as determining the routing paths among the PEs.

In CGRA CAD, the application is typically represented as a dataflow graph (DFG), where the vertices model operations (e.g., arithmetic operations, loads/stores, I/O operations), and the edges model data-dependencies between the operations. The CGRA device can *also* be modeled as a graph, where vertices represent the computational units (ALUs), memory ports, I/Os, and routing multiplexers. Edges in the device-model graph represent potential connections between vertices. A device-model graph example is shown in Fig. 1 for a simplified CGRA processing element. The multiplexer select signals and ALU op-code are driven by configuration bits (not shown), based on the application mapping results. CGRA mapping can be viewed as a constrained graph-embedding problem, where the application DFG is embedded within the device-model graph. Nodes of the DFG must be mapped to nodes of the device-model graph in a legal way, e.g., an I/O operation in the DFG must be mapped to a device-model graph vertex corresponding to an I/O. Similarly, edges in the DFG must be mapped to disjoint paths of interconnect resources within the device-model graph.

A unique aspect of CAD for CGRAs vs. CAD for ASICs and FPGAs is that in CGRAs, the scheduling, placement and routing are often formulated as a single problem instance, rather than separate steps. We speculate that the reason for this relates to the typically very limited routing flexibility of CGRAs: they often have nearest-neighbor word-wide interconnect between PEs, whereas FPGAs have large numbers of bit-level routing tracks between logic blocks. For CGRAs, the limited flexibility implies that if placement were decoupled from routing, it may be difficult to find a legal routing for a given placement.

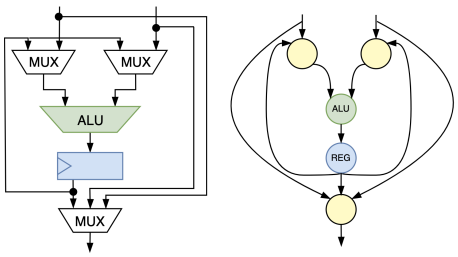


Fig. 1: CGRA PE snippet and its device-model graph.

Improving the runtime and success rate of CGRA mapping is key to boosting designer productivity and reducing non-recurring engineering costs. In this work, we introduce *GRAMM*, a fast CGRA mapper based on a graph-minor heuristic. GRAMM is an acronym for GRAph Minor Mapping. Informally, a graph H is called a *graph minor* of another graph G if G can be transformed into H by deleting edges and vertices, as well as contracting edges. With this definition of graph minor, one can begin to see the relationship to CGRA mapping. Imagine the application DFG to be H and the device-model graph to be G: we wish to determine whether H is a minor of G, albeit with some constraints on the types of nodes and directions of edges. Until recently, it appears that most graph-minor-finding algorithms were quite compute intensive. In fact, an optimal solution to the graph-minor problem is NP-hard, as if H and G have the same number of vertices and edges, the graph-minor problem reduces to graph isomorphism [2]. Identification of graph minors also has applications in quantum computing, leading to the publication of a heuristic graph minor approach by D-Wave in 2014 [3]. In this work, we adapt the heuristic for CGRA application mapping, demonstrating up to orders of magnitude runtime speedup for our proposed mapper vs. a mapper based on integer linear programming. Furthermore, on average, mappings generated by GRAMM use at least 30% fewer PEs than the prior work.

II. Graph Minor Heuristic

We begin by reviewing D-Wave's graph minor heuristic [3], which works as follows: H is the minor graph, and G is the full target graph. For each node $x_i \in H$, we use the symbol $\phi(x_i)$ to denote a subgraph of vertices in G that x_i "maps" onto in the minor embedding. D-Wave refers to this subgraph as the *vertex model* of x_i [3]. $\phi(x_i)$ must be a connected subgraph, and as such, it is possible to collapse the nodes of $\phi(x_i)$ into a single vertex through edge contractions.

A legal minor embedding of H into G involves finding a non-empty $\phi(x_i)$ for all vertices $x_i \in H$ such that if two vertices, x_i and x_j, are connected in H, then the two vertex models $\phi(x_i)$ and $\phi(x_j)$ should also have at least one edge connecting them (i.e., at least one vertex in $\phi(x_i)$ has an edge to a vertex in $\phi(x_j)$). Moreover, a legal minor requires that all ϕ subgraphs are disjoint—there are no common vertices among any two of them.

At a high level, the algorithm works by iteratively finding a vertex model for each node y of H by considering the vertex models of y's neighbors in H. The algorithm allows *illegal intermediate states*, where some vertices in G are assigned to *multiple* vertex models. The algorithm then continues to refine vertex models for vertices in H until a legal minor is identified, or an exhaustion threshold is crossed. Interestingly, this approach resembles the PathFinder negotiated congestion routing algorithm for FPGAs, which allows temporary overuse of routing resources (i.e., illegal intermediate states), resulting in shorts between the routing of different signals [4]. The graph minor search algorithm also incorporates nondeterminism by randomizing the *order* in which the nodes of H are considered for finding their vertex models.

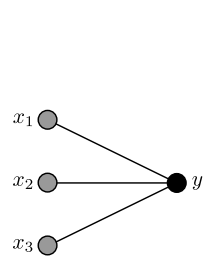
To illustrate the main action of the algorithm, we walk through an example. Consider Fig. 2a which shows a portion of graph H having a node y connected to three neighbors: x_1, x_2 and x_3. Let us assume we have already found vertex models for x_1, x_2 and x_3 in graph G (Fig. 2b), and we aim to compute $\phi(y)$, the vertex model for y. Notice that the vertex model for x_1, $\phi(x_1)$, has three vertices; the vertex models for x_2 and x_3 have one and two vertices, respectively. Other nodes of G are shown as black dots. For clarity, the other edges of G are not shown.

To find $\phi(y)$, the heuristic considers each node $g \in G$, and finds the lowest-cost weighted paths from g to $\phi(x_1)$, $\phi(x_2)$ and $\phi(x_3)$, respectively. The sum of the costs of the three paths is the *total cost* of paths from g. The specific costs assigned to vertices depend on whether they are used in multiple vertex models, detailed below. Fig. 2c shows a node g_i, and the identified three lowest-cost paths to the three vertex models. Fig. 2d shows a different node g_j and its corresponding lowest-cost paths to the three vertex models. The low-cost path-finding process is performed for all nodes in G.

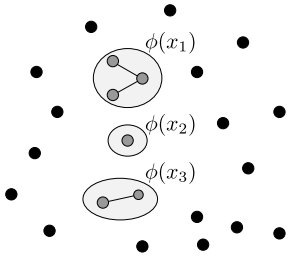
Having found the set of low-cost paths for each node in G, the algorithm then finds the node in G having the smallest total cost, i.e., paths to the vertex models, $\phi(x_1)$, $\phi(x_2)$ and $\phi(x_3)$ with lowest cumulative cost. This specific node, and the nodes on its lowest-cost paths, are chosen as the vertex model for y. Carrying on with the example, let us assume that g_j is the node of G having the lowest total cost; the selected vertex model for y, $\phi(y)$, is shown in Fig. 2e.

After finding vertex models using the above approach for every node in H, the algorithm checks if the vertex models correspond to a legal graph-minor solution. If yes, then the algorithm terminates with success. If not, the entire process is repeated, and new (refined) vertex models are identified for each node in H.

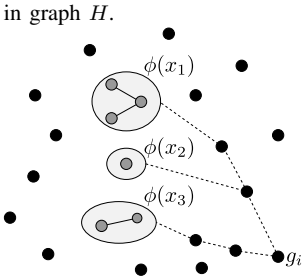
While the above represents the core ideas of the algorithm, there are some special cases to consider. First, at the beginning of the algorithm, it is possible that none of the vertex y's neighbors have a vertex model assigned yet. In this case, a random vertex of G is selected, and $\phi(y)$ is set to this random vertex. Another scenario is that only *some of* y's neighbors have a vertex model assigned so far. In this case, the path costs to y's neighbors having empty vertex models are set to 0. Finally, the heuristic also considers the case of finding the path cost of a vertex $g \in G$ to a vertex model $\phi(x_i)$, where g also happens to be part of $\phi(x_i)$. For this case, the path cost is set to the weight of vertex g.



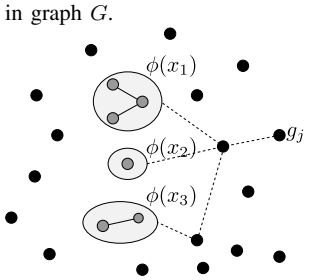
(a) A node y with three neighbors in graph H.



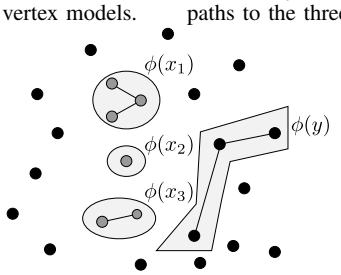
(b) Vertex models of y's neighbors in graph G.



(c) Node g_i with three lowest-cost paths to the three vertex models.



(d) Node g_j with three lowest-cost paths to the three vertex models.



(e) The selected vertex model $\phi(y)$ in G.

Fig. 2: A step-by-step example of graph minor embedding.

The last missing piece to discuss is the node weights. The weight (cost) of a vertex $g \in G$ is defined as:

$$w(g) = D^{|S(g)|} \tag{1}$$

where D is the *diameter* of G and $S(g)$ is the set of vertices in H that have g in the vertex model. That is, $S(g) = \{x_i : x_i \in H, g \in \phi(x_i)\}$. Intuitively, the weight of g is exponentially related to its *overuse* in vertex models.

Algorithm 1 formalizes the approach to find a vertex model for a node, given the vertex models of its neighbor nodes. Here, the x_j's represent the neighbors (in H) of the node for which we are finding a vertex model. Lines 1–2 check if the vertex models of the neighbors are empty, and if so, return a random vertex model. The main loop is on lines 4–12, which, for each vertex $g \in G$, finds a path from g to the neighboring vertex models. The checks on lines 5–8 handle edge cases, where *some* neighboring vertex models are empty or when g is part of one of the neighboring vertex models. The shortest-path distance algorithm is run on line 10. Finally, line 13 selects the node $g^* \in G$ having the lowest total cost.

Algorithm 2 is the top-level flow. At first, the vertex order is randomly shuffled (line 1). All vertex models are initialized to empty in line 2. The main work is in the loop on lines 4–9: the algorithm constructs the minor embedding by iteratively finding a vertex model for each node in H based on the shortest path distances and weights using Algorithm 1. Algorithm 2 will refine the vertex models until a valid minor is found, or it will report failure if improvement

Algorithm 1 FindMinimalVertexModel($G, w, \{\phi(x_j)\}$) from [3]

Input: Graph G with weights w, set of vertex-models $\{\phi(x_j)\}$;
Output: Vertex model $\phi(y)$ in G such that there is an edge between $\phi(y)$ and each $\phi(x_j)$;

if all $\phi(x_j)$ are empty **then**
  return random $g^* \in G$
end if
for all g $\in G$ and all x_j **do**
  if $\phi(x_j)$ is empty **then**
    $c(g, x_j) \leftarrow 0$
  else if $g \in \phi(x_j)$ **then**
    $c(g, x_j) \leftarrow w(g)$
  else
    $c(g, x_j) \leftarrow$ weighted shortest-path distance (g,$\phi(x_j)$) excluding $w(\phi(x_j))$
  end if
end for
$g^* \leftarrow argmin_g \sum_{x_j} c(g, x_j)$
return $g^* \cup$ paths from g^* to each $\phi(x_j)$

Algorithm 2 FindMinorEmbedding adapted from [3]

Input: Graph H with vertices $x_1, \ldots, x_n$, graph G;
Output: A valid H-minor in G, or failure;

randomize the vertex order $x_1, ..., x_n$
initialize all the $\phi(x)$ to empty
while stopping criteria not met **do**
  for $i = 1, 2, \ldots, n$ **do**
    for $g \in G$ **do**
      $w(g) \leftarrow D^{|S(g)|}$,
      where $S(g) = \{x_i : x_i \in H, g \in \phi(x_i)\}$
    end for
    $\phi(x_i)$ = FindMinimalVertexModel($G, w, \{\phi(x_j) : x_j$ connects to $x_i\}$)
  end for
  if $\phi(x_i), \ldots, \phi(x_n)$ represent a legal minor **then**
    return $\phi(x_i), \ldots, \phi(x_n)$ ("success")
  end if
end while
return "failure"

(i.e., fewer overlaps among vertex models) is not realized after a fixed number of iterations.

III. GRAMM: Adaptation for CGRA Application Mapping

We now discuss implementation details regarding how we adapted the above heuristic to CGRA mapping, and how we further enhanced it to improve mapping success.

A. Core Changes

1) Directed Graph: Both inputs (application DFG and device-model graph) to mapping are directed graphs, whereas the original D-Wave heuristic was for undirected graphs. We modified the algorithm to work for directed graphs when finding the weighted shortest paths from a vertex in G to vertex models. Paths may not exist between all vertices of G once edge directionality is accounted for.

We use Dijkstra's algorithm to compute the shortest paths and distances from neighboring vertex models to the vertices in graph G. Dijkstra's algorithm normally applies to an edge-weighted graph, not a vertex-weighted graph. We assign the weight of the sink node for an edge as its edge weight.

When finding a vertex model for a vertex $y \in H$, via selecting the best "root" vertex in G (i.e., the vertex having the lowest cumulative cost), we must ensure the vertex has a valid path in the directed graph to all of y's neighbors' vertex models.

2) Vertex Attributes: Node attributes are introduced to ensure a DFG node is mapped to the correct type of vertex in the device graph. For example, we must map an ALU-operation node in the DFG to an ALU vertex in the device graph, and likewise map memory operations in the DFG to memory-port vertices. For each vertex model, it should contain at least one vertex of the correct type, with the remaining vertices being routing-related vertices.

When a random vertex must be selected as a vertex model, we ensure the random vertex is of a type that can accommodate the operation of the node in H for which we are finding a vertex model.

3) Legal Merging of Paths: When finding a vertex model for a node $y \in H$ using its neighbors' vertex models, the original heuristic allows path merging. From the circuit perspective, this is only legal for fanouts of y, not for fanins. The fanins of y are separate input dependencies that must be routed using entirely disjoint CGRA paths.

With these modifications applied to the heuristic, we now have an algorithm suitable for CGRA mapping. Searching for graph minors results in simultaneous placement and routing of the application DFG onto the device-model graph, where infeasible intermediate states (resource overuse) are temporarily permitted, and iteratively resolved through costing, re-placement, and re-routing.

B. Enhancements to Improve Mapping Success

1) Randomizing the Selection of Min-Cost Vertex Model: We observed that when finding the minimum-cost vertex model for a node in H, there often exist several vertex models having the *same* minimum cost. Our initial implementation selected the first such model discovered. As an alternative approach, we explored random selection among the minimum-cost models. In the latter approach, as each new minimum-cost model is discovered, we flip a coin to determine whether to select it as the best, thus injecting more non-determinism into the graph-minor heuristic. We refer to this approach as `RAND_BEST`.

2) PathFinder-like Vertex Costing: Noting the similarity between the iterative legalization approach in the graph-minor heuristic and the PathFinder, negotiated-congestion FPGA router [4], we explored using a PathFinder-like cost function for nodes of G as an alternative to Eqn. (1). Like the graph-minor heuristic, PathFinder costs a vertex (CGRA resource) based on the present demand for the resource (i.e., its overuse). However, PathFinder's costing also includes a *history* term, which reflects the overuse of a vertex in previous iterations of the algorithm (e.g., previous iterations of the `while` loop in Algorithm 2). The history term is helpful to prevent oscillations between the intermediate solutions, where overuse of certain vertices is transferred to overuse of other vertices, and then back again.

Taking inspiration from PathFinder's cost function, we consider the following approach to costing the vertices $g \in G$:

$$w_{\text{PathFinder}}(g) = (1 + |S(g)|) \times P \times (1 + H(g)), \qquad (2)$$

where $S(g)$ is as defined previously: the set of vertices of H that use g in their vertex model. P is a scalar constant reflecting the penalty for overuse. We set P to one initially, and increase it geometrically by 1.1 in each iteration (determined empirically). $H(g)$ is the history term, initialized to zero for each vertex. After each iteration of the `while` loop in Algorithm 2, we increase $H(g)$ by one for each overused vertex $g \in G$ (i.e., where $|S(g)| > 1$). We refer to this approach as `PATHFINDER`.

3) Ordering of Nodes in H when Finding Vertex Models: The original heuristic suggested ordering the nodes in H randomly and then proceeding with finding a vertex model for each. As an alternative to this, we explored sorting the nodes of H according to the size (number of vertices) of their vertex model in the previous iteration of the `while` loop of Algorithm 2. For nodes of H where this quantity is equal, we break ties randomly. The intuition here is that there may be an advantage in prioritizing the DFG nodes (and their corresponding connections) that require many CGRA resources. We incorporated the sorting in addition to the PathFinder costing, and refer to this approach as `PATHFINDER_SORT`.

We also experimented with sorting the nodes of H according to their topological distance from a CGRA I/O or memory port; however, we found that this technique did not improve results beyond the ordering by the size of the vertex models.

IV. Experimental Study

We evaluate the proposed mapper using the CGRA-ME [5], an open-source framework for modeling and exploring CGRA architectures and CAD. In CGRA-ME, the target CGRA architecture is specified using a `C++` API. The primitives through which one may compose a CGRA include interconnect (e.g., multiplexers, crossbars) and PEs (e.g., ALUs, memory ports, I/Os). To evaluate GRAMM, we use fifteen benchmarks from the CGRA-ME framework and three additional circuits (2exp_6, 3mandelbrot, 2mandelbrot), which we handcrafted. The first is a parallel 6-degree Taylor expansion for e^x, and the second two perform iterated Mandelbrot set computations.

As a test vehicle for our mapper evaluation, we use the popular ADRES CGRA [6]. Fig. 3 shows an ADRES CGRA architecture with four rows and four columns of PEs. Each PE can perform ALU operations, including addition, subtraction, multiplication, shifting, and logical operations. Each PE contains a local register file (LRF). A unique memory port is assigned to each row and connected to all the PEs in that row to perform memory load and store. As for the connections between PEs, each PE has South/West/North/East connections to the neighboring PEs. Also, PEs in the top row have an additional toroidal connection to PEs in the bottom row. Likewise, the left-most column PEs and right-most column PEs are also connected to one another via toroidal connections. A bypass route is available within each PE, but it comes at the cost of leaving the ALU idle in that PE. Even though we chose ADRES for the experiments, the proposed mapper is in no way tied to a specific CGRA architecture.

A. Impact of Optimizations

To evaluate the impact of the proposed mapper enhancements, we target a 6×6 ADRES CGRA and run the mapper 100 times

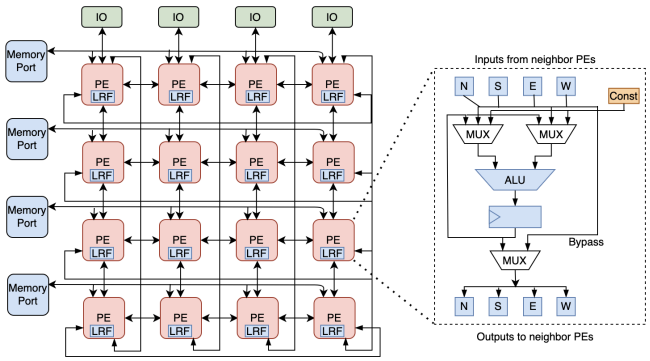


Fig. 3: 4×4 ADRES CGRA architecture.

TABLE I: Impact of algorithm enhancements: the number of successful mappings out of 100 attempts targeting a 6×6 ADRES CGRA.

Benchmark	BASELINE	RAND BEST	PATHFINDER	PATHFINDER SORT
accumulate	16	28	100	100
cap	1	19	99	98
conv2	51	61	100	100
conv3	9	17	100	100
mac	61	64	100	100
mac2	3	7	98	98
matrixmultiply	11	31	100	100
mults1	0	2	70	79
mults2	2	6	92	91
nomem1	91	99	100	100
nomem2	96	96	100	100
simple	20	26	100	100
simple2	5	14	100	100
sum	89	96	100	100
2exp_6	3	5	99	99
2mandelbrot	2	4	99	100
Average	**29**	**36**	**97**	**98**

with different random seeds. We record the number of successful mappings out of 100 runs. Table I shows the results. There are four columns for each benchmark, reflecting scenarios described above: the baseline graph-minor heuristic (`BASELINE`), random selection among min-cost vertex models (`RAND_BEST`), PathFinder-style costing (`PATHFINDER`), and PathFinder costing coupled with sorting the nodes of H (`PATHFINDER_SORT`). The table excludes one of the benchmarks, 3mandelbrot, which cannot be mapped on a 6×6 CGRA due to a lack of resources.

As shown in Table I, as the optimizations are layered upon one another, the number of successful mappings increases. On average, the baseline produces successful mappings 29% of the time. The success rate increases to 36% when randomizing the min-cost model selection. PathFinder-style costing improves success dramatically to 97%. Sorting further improves mapping success to 98%. In the following section, we compare the most successful mapping approach, `PATHFINDER_SORT`, with another CGRA application mapper.

B. Comparison to Other Mapping Technique:

We compare `PATHFINDER_SORT`, our best-performing mapping technique, with the exact integer linear programming-based (ILP) mapper [7] in CGRA-ME, set to have a two-hour time-out. Gurobi ILP solver is used [8]. For each benchmark, we target 6×6 and 8×8 CGRAs and report results for both array sizes and the three mappers. The only exception is the FFT benchmark, dimensioned for a 10×10 CGRA (and, hence, impossible to map on a smaller array). GRAMM (this work) was run 100 times on each benchmark, and we report the number of successful mapping attempts, as well as the average and standard deviation of runtime across all 100 runs (including the failed mappings). The experiments were run on a 10-core Intel Xeon CPU. As with other works on CGRA mapping, our emphasis is on finding legal mappings with low runtime. Note, however, that minimum-cost placement and routing with Eqn. (2) will naturally minimize the number of used CGRA resources.

Table II shows the runtime comparison for two CGRA sizes. The first two columns list the names of the 18 benchmarks and the corresponding DFG size. The latter concern all DFG nodes, including I/Os, constants, loads/stores, and arithmetic/logical operations. The subsequent two columns report the number of successful mappings of our new mapper, out of the 100 runs (equivalent to a success rate in %). Then, we report the average runtime (in seconds) and the standard deviation across all the runs. The following columns report

TABLE II: Run-time & resource use comparison between GRAMM and ILP-based mapper [7]. *TO*: timeout of 120 minutes; *U*: unmappable.

		GRAMM (This Work)				ILP Mapper		Speedup (×)		GRAMM		ILP Mapper	
	DFG	Mapping Success (%)		Runtime (s) avg (dev)		Runtime (s)		GRAMM vs. ILP		Routing Use avg (dev)		Routing Use	
Benchmark	**size**	**6×6**	**8×8**	**6×6**	**8×8**	**6×6**	**8×8**	**6×6**	**8×8**	**6×6**	**8×8**	**6×6**	**8×8**
accumulate	24	100	100	0.1 (0.05)	0.29 (0.15)	274.9	474.4	2749.0	1635.9	20.6 (2.3)	23.4 (2.8)	26	47
cap	30	98	100	0.24 (0.14)	0.5 (0.22)	2913	TO	12137.5	-	27.6 (2.3)	32.6 (3.55)	33	U
conv2	22	100	100	0.06 (0.03)	0.16 (0.09)	19	720.5	316.7	4503.1	16.5 (2.1)	18.87 (3.09)	32	33
conv3	32	100	100	0.18 (0.09)	0.39 (0.2)	36.3	85.7	201.7	219.7	24.2 (2.3)	29.2 (3.5)	33	58
mac	15	100	100	0.03 (0.01)	0.08 (0.04)	60	19.3	2000.0	241.3	11.4 (1.7)	12.92 (2.32)	9	36
mac2	32	98	100	0.27 (0.17)	1.12 (0.49)	598.4	4220	2216.3	3767.9	27.1 (2.8)	35.26 (4.27)	31	49
matrixmultiply	23	100	100	0.08 (0.05)	0.19 (0.11)	67.1	58.8	838.8	309.5	19.8 (2.6)	22.75 (3.65)	29	58
mults1	39	79	100	0.6 (0.31)	0.86 (0.4)	TO	TO	-	-	29.6 (2.5)	36.6 (4.14)	U	U
mults2	33	91	100	0.41 (0.23)	0.74 (0.31)	TO	5893.1	-	7963.6	29.1 (2.6)	34.2 (3.46)	U	48
nomem1	6	100	100	0.01 (0)	0.02 (0)	0.8	1.8	80.0	90.0	4.1 (1)	4.8 (1.26)	36	64
nomem2	8	100	100	0.01 (0)	0.03 (0.01)	1.34	39.5	134.0	1316.7	5.4 (1)	6.14 (1.66)	14	10
simple	18	100	100	0.05 (0.03)	0.15 (0.08)	14.8	225.9	296.0	1506.0	18.1 (2.3)	20 (3.54)	22	18
simple2	20	100	100	0.08 (0.04)	0.2 (0.09)	15.4	262	192.5	1310.0	21.9 (2.3)	23.99 (2.41)	30	36
sum	9	100	100	0.01 (0)	0.03 (0.01)	1.5	5.4	150.0	180.0	5.8 (1.2)	6.59 (1.6)	12	11
2exp_6	21	99	100	0.33 (0.15)	1.05 (0.47)	TO	TO	-	-	27.8 (2.3)	35.6 (3.97)	U	U
3mandelbrot	42	0	97	1.03 (0.03)	1.36 (0.68)	TO	TO	-	-	U (-)	44.24 (4.82)	U	U
2mandelbrot	28	100	100	0.21 (0.1)	0.47 (0.17)	TO	TO	-	-	25.5 (2.9)	29.43 (4.66)	U	U
FFT (10×10)	43	7		9.3 (1.12)		TO		-		79.14 (7.22)		U	

the runtime of the exact ILP mapper. The next two columns give the achieved average speedup against the ILP mapper.

Comparing the mapping success of GRAMM versus ILP on an 8×8 CGRA, we observe that GRAMM found a successful mapping in all 100 runs and all benchmarks except 3mandelbrot (97 successful attempts out of 100). At the same time, ILP mapper timed out. Furthermore, ILP timed out while finding a mapping for five benchmarks. Regarding the speed of mapping, if we compute the geometric mean of the speedup of GRAMM versus ILP across all the benchmarks, we find that GRAMM outperformed the ILP mapper by ∼855×. In seven runs and 9.3 seconds (on average), GRAM found successful mappings for the FFT benchmark (targeting a 10×10 CGRA), while the ILP mapper timed out. As we will later show, the FFT benchmark represents a very hard mapping problem because it requires almost 80% of the PEs to be mapped.

When targeting a 6×6 CGRA, with fewer resources and thus an even more challenging mapping task, GRAMM was 100% successful for 10 circuits and at least 90% successful on 15 of them. Of the few difficult-to-map benchmarks, 3mandelbrot failed to map (due to a lack of resources), and mults1 was successfully mapped in 79 attempts. Conversely, the ILP mapper timed out and failed to find a mapping for five benchmarks, including 3mandelbrot and mults1. GRAMM took at most 1.03 seconds, on average, to find a successful mapping. Considering the geometric mean of the speedup, we find that GRAMM outperformed the ILP mapper by 547.5×.

To assess mapping quality, the right-most columns of Table II show the resource usage of the GRAMM and ILP mapping techniques for both CGRA sizes. As a usage metric, we count the total number PE output multiplexers used in the mapping, as shown in the right side of Fig. 3. This metric reflects the aggregate number of PEs used—either for an ALU operation or as a route-through. The metric is indicative of area and power consumption. For GRAMM, the average and standard deviation of the resource usage across all successful mappings are reported. On average, GRAMM mappings use 69% (resp. 57%) of resources compared to ILP on 6×6 (resp., 8×8 CGRA). GRAMM mappings are superior in all but two cases: 1) to map the mac benchmark on a 6×6 CGRA, GRAMM used 26.6% more resources than ILP, though at the advantage of ∼2,000× faster mapping time, and 2) to map the simple benchmark on a 8×8 CGRA, GRAMM used 11% more resources than ILP. Looking at the standard deviations of resource usage, we find they are fairly low, within 8–24% on a 6×6 CGRAs and 10–27% on a 8×8 CGRA; hence, GRAMM quality of results scale well with the increased device and benchmark graph size. Finally, for the `FFT` benchmark targeted to a 10×10 CGRA, resource usage is 79%, on average, reflecting that a large fraction of PEs is needed for the mapping and explaining the low mapping success rate (7% in Table II).

V. Related Work

Aside from the mappers within the CGRA-ME framework, there are various other works on CGRA mapping; for example, a simulated annealing-based method (DRESC [9] and SPR [10]), a reinforcement learning-based method [11], a ZDD-based method [12], as well as other methods like PathSeeker [13] and RAMP [14]. [11] reports average runtimes of ∼100 seconds on DFGs of similar size to ours. [13] does not report DFG sizes, though reports runtimes close to 10 seconds on 4×4 CGRAs and shows [14] to have considerably longer runtimes. TAEM [15] is a mapper with emphasis on CGRAs having heterogeneous resources, based on a maximum-weighted clique approach; numerical run-times are not reported. An interesting recent work, GEML [16], applies graph neural networks to CGRA mapping. Although GEML is focused on multi-context CGRAs, it reports mapping runtimes ranging from 2.5 to 102 seconds for some single-context 8×8 CGRAs on similarly-sized DFGs to our own. The application of graph minor techniques in CGRA mapping is a niche direction; there is one prior work. Chen et al. [17] formulated the CGRA mapping problem with route sharing as a graph minor problem. Their solution can provide high-quality results with a relatively small runtime compared to simulated annealing-based mappers. However, [17] performs an *exhaustive* exploration when mapping the application DFG to the device model graph, with backtracking upon failure, which is impractical for most mapping problems.

VI. Conclusions and Future Work

We described a new approach to CGRA mapping based on a graph-minor heuristic [3]. The heuristic was adapted to CGRA mapping, and enhanced to increase the success of finding a legal mapping, borrowing concepts from an FPGA negotiated-congestion routing algorithm. Our fast mapper provides orders of magnitude speedup over the ILP mapper in the openly-available CGRA-ME framework [18]. Additionally, the mappings it produces use, on average, approx. 30–40% fewer resources. Early mappability estimation and alternative cost functions are two of the many avenues for future work.

REFERENCES

[1] W. Haensch, "Scaling is over — What now?" in *Proceedings of the 75th Annual Device Research Conference*, South Bend, IN, USA, Jun. 2017, pp. 1–2.

[2] G. L. Miller, "Graph isomorphism, general remarks," in *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, Boulder, CA, USA, May 1977, pp. 143–50.

[3] J. Cai, W. G. Macready, and A. Roy, "A practical heuristic for finding graph minors," pp. 1–16, Jun. 2014, arXiv, 1406.2741.

[4] L. McMurchie and C. Ebeling, "PathFinder: A negotiation-based performance-driven router for FPGAs," in *Proceedings of the third International ACM Symposium on Field-Programmable Gate Arrays*, Napa Valley, CA, USA, Feb. 1995, pp. 111–17.

[5] S. A. Chin, N. Sakamoto, A. Rui, J. Zhao, J. H. Kim, Y. Hara-Azumi, and J. Anderson, "CGRA-ME: A unified framework for CGRA modelling and exploration," in *Proceedings of IEEE 28th International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, Sattle, WA, USA, Jul. 2017, pp. 184–89.

[6] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins, "ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix," in *Proceedings of the 13th International Conference on Field Programmable Logic and Applications (FPL)*, Lisbon, Portugal, Sep. 2003, pp. 61–70.

[7] S. A. Chin and J. H. Anderson, "An architecture-agnostic integer linear programming approach to CGRA mapping," in *Proceedings of the 55th ACM/IEEE Design Automation Conference (DAC)*, San Francisco, CA, USA, Jun. 2018, pp. 1–6.

[8] Gurobi Optimization, LLC, "Gurobi optimizer reference manual," 2022, accessed: 2023-06-02. [Online]. Available: https://www.gurobi.com

[9] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins, "DRESC: A retargetable compiler for coarse-grained reconfigurable architectures," in *Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT)*, Hong Kong, China, Dec. 2002, pp. 166–73.

[10] S. Friedman, A. Carroll, B. V. Essen, B. Ylvisaker, C. Ebeling, and S. Hauck, "SPR: An architecture-adaptive CGRA mapping tool," in *Proceedings of the 17th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Monterey, CA, USA, Feb. 2009, pp. 191–200.

[11] D. Liu, S. Yin, G. Luo, J. Shang, L. Liu, S. Wei, Y. Feng, and S. Zhou, "Data-flow graph mapping optimization for CGRA with deep reinforcement learning," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 12, pp. 2271–83, Dec. 2019.

[12] R. Beidas and J. H. Anderson, "CGRA mapping using zero-suppressed binary decision diagrams," in *Proceedings of the 27th Asia and South Pacific Design Automation Conference (ASP-DAC)*, Taipei, Taiwan, Jan. 2022, pp. 616–22.

[13] M. Balasubramanian and A. Shrivastava, "PathSeeker: A fast mapping algorithm for CGRAs," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE)*, Antwerp, Belgium, Mar. 2022, pp. 268–73.

[14] S. Dave, M. Balasubramanian, and A. Shrivastava, "RAMP:Resource-aware mapping for CGRAs," in *Proceedings of the 55th ACM/IEEE Design Automation Conference (DAC)*, San Francisco, CA, USA, Jun. 2018, pp. 1–6.

[15] M. Kou, J. Gu, S. Wei, H. Yao, and S. Yin, "TAEM: Fast transfer-aware effective loop mapping for heterogeneous resources on CGRA," in *Proceedings of the 57th ACM/IEEE Design Automation Conference (DAC)*, San Francisco, CA, USA, Jul. 2020, pp. 1–6.

[16] M. Kou, J. Zeng, B. Han, F. Xu, J. Gu, and H. Yao, "GEML: GNN-based efficient mapping method for large loop applications on CGRA," in *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC)*, San Francisco, CA, USA, Jul. 2022, pp. 337–42.

[17] L. Chen and T. Mitra, "Graph minor approach for application mapping on CGRAs," in *Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT)*, Seoul, South Korea, Dec. 2012, pp. 285–92.

[18] M. Suzuki, Y. Hasegawa, Y. Yamada, N. Kaneko, K. Deguchi, H. Amano, K. Anjo, M. Motomura, K. Wakabayashi, T. Toi, and T. Awashima, "Stream applications on the dynamically reconfigurable processor," in *Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT)*, Brisbane, QLD, Australia, Dec. 2004, pp. 137–44.