PANORAMA: Divide-and-Conquer Approach for Mapping Complex Loop Kernels on CGRA

Dhananjaya Wijerathne, Zhaoying Li, Thilini Kaushalya Bandara, and Tulika Mitra
{dmd,zhaoying,thilini,tulika}@comp.nus.edu.sg
National University of Singapore

ABSTRACT

CGRAs are well-suited as hardware accelerators due to power efficiency and reconfigurability. However, their potential is limited by the inability of the compiler to map complex loop kernels onto the architectures effectively. We propose PANORAMA, a fast and scalable compiler based on a divide-and-conquer approach to generate quality mapping for complex dataflow graphs (DFG) representing loop bodies onto larger CGRAs. PANORAMA improves the throughput of the mapped loops by up to 2.6x with 8.7x faster compilation time compared to the state-of-the-art techniques.

1 INTRODUCTION

The interest for flexible and power-efficient accelerators is ever increasing with the advent of IoT and edge computing technology in the global market. Coarse-Grained Reconfigurable Arrays (CGRAs) have the potential to become a prominent hardware accelerator in this domain since they can offer a good balance between flexibility, power efficiency, and performance [1–13]. CGRA is a spatial accelerator consisting of many reconfigurable processing elements (PE) concurrently executing a single application or application kernel. FPGAs are the closest contender but have poor power and area efficiency due to the bit level reconfigurability overhead [1].

CGRA consists of a PE array where the compute and interconnect elements are runtime reconfigurable according to a static schedule created at compile time. Figure 1 shows an example of CGRA architecture with a 4x4 PE array. PEs typically includes simple ALU as a Functional Unit (FU), register file (RF) as the local storage, and switches to communicate with the other PEs. All these elements are runtime reconfigurable through a predetermined sequence of configurations stored in the configuration memory. At runtime, this sequence of configurations is repeated cyclically. A subset of processing elements can access shared memory banks, which acts as the communication channel between the accelerator and the rest of the system (host processor, main memory).

As CGRAs repeatedly cycle through a small set of configurations, application loop kernels are the perfect candidates for acceleration. Application loop kernels are represented as Dataflow Graph (DFG), where the nodes represent operations, and the edges represent the dependencies between operations. Figure 2 shows snippets of DFGs. The compiler maps the DFG onto CGRA, i.e., assigns DFG nodes onto spatio-temporal PE coordinates while assuring valid

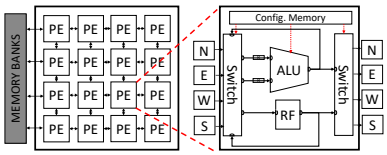
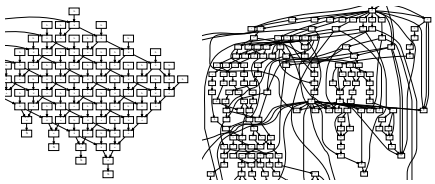


Figure 1: 4x4 CGRA Architecture.



(a) (b)

Figure 2: Example dataflow graphs.

data paths are available through PE interconnects to route the data dependencies. Therefore, the achievable performance and power efficiency of CGRA rely heavily on the quality of the compiler.

However, the compiler has become the main barrier preventing CGRA from reaching its full potential, particularly when scaling up to complex DFG and larger CGRA. Noticeably, most of the previous works on CGRA focused on either small CGRA sizes or small application kernels [2–11]. HiMap [16] has proposed a scalable CGRA compiler through a hierarchical mapping approach. However, they focus on kernels with regular (lattice-like) inter-iteration dependencies, as shown in figure 2a. They identify the repeatable patterns in the regular DFG and replicate the unique iteration mappings to reduce the compilation complexity. Therefore, the applicability of the compiler is limited to multi-dimensional kernels that form such regular DFGs. Figure 2b shows a complex loop kernel DFG with irregular dependencies, more prevalent in embedded applications. The scalability issue in the compiler has resulted in non-optimal performance and longer mapping time, hindering the scalability of CGRAs from supporting diverse, complex, and bigger workloads.

In this paper, we introduce a fast and scalable CGRA compiler, Panorama, for mapping complex loop kernels on CGRA. Panorama allows the mapper to see an all-encompassing global view, i.e., a Panoramic view of complex DFGs, and use a divide-and-conquer approach to generate quality mappings. The higher-level mapping partitions the DFG onto the CGRA clusters and guides the lower-level mapping, reducing overall complexity. Panorama is a portable solution that can be combined with existing low-level CGRA mappers to achieve enhanced performance in a shorter compilation time. We demonstrate the advantage of Panorama in combination with two state-of-the-art low-level mapping approaches, resulting in up to 2.6x performance increase with 8.7x faster compilation time. The framework is open-source and available from https://github.com/ecolab-nus/panorama.

DAC '22, July 10–14, 2022, San Francisco, CA, USA

ACM ISBN 978-1-4503-9142-9/22/07.
https://doi.org/10.1145/3489517.3530429

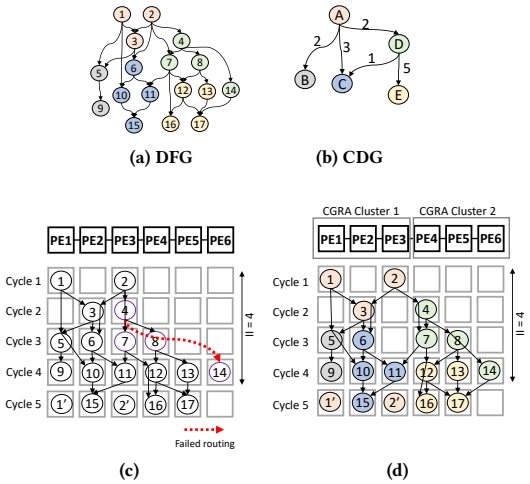


Figure 3: Mapping examples on a time-extended 6x1 CGRA. (c) A failed mapping instance generated by a conventional mapper. (d) A successful mapping result generated by Panorama approach.

2 MOTIVATING EXAMPLE

We first present the intuitions behind the Panorama mapping approach using a motivation example. Figure 3a shows the DFG corresponding to a loop body where the nodes represent the operations and edges represent the dependency between two operations. We compare the mappings of the DFG on a 6x1 linear CGRA, which only allows single-cycle single-hop data transfers. Mapping schedules are illustrated on time-extended CGRA where the CGRA resources are replicated along the time dimension. The target Initiation Interval (II) of both mapping schedules is 4, where the schedule repeats every four cycles, i.e., a new loop iteration can be initiated every four cycles. The conventional mapping algorithms place DFG nodes one by one on CGRA PEs, searching for the best placement for each node. Figure 3c shows a failed mapping instance generated by a mapper with such a narrow perspective. In this placement, it is not possible to find a valid route for node 14. Node 14 is placed three hops away from its parent (node 4), and data cannot reach in two cycles since CGRA only allows single-cycle single-hop data transfers. For the same reason, node 14 cannot be placed in any other available PEs. Thus it needs to remap the nodes to reach a valid mapping. The main reason behind such failures is the lack of a global view of the mapper. The local view of the mapper attempts to place nodes to closest PEs to reduce the routing cost of data dependencies. However, this may lead to requiring significant additional resources to route far-flung data dependencies that might be encountered later during the mapping. The narrow perspective also leads to closely packed nodes in a subset of CGRA PEs without fully exploiting the available resources. A closely packed schedule causes more routing congestions for the DFG nodes that are mapped later.

In contrast, we provide a global view of the DFG to the mapper by partitioning the DFG into clusters (a community of closely connected nodes). Node colors in the DFG represent the different clusters. Figure 3b shows the Cluster Dependency Graph (CDG) where nodes represent clusters of DFG nodes and edge weights represent the number of DFG edges between two clusters. The CDG consists of five nodes A, B, C, D, and E. We first map the CDG to CGRA clusters considering the structure of the CDG and CGRA clusters. Figure 3d shows a successful mapping result generated by our approach. Here the clusters A, B, and C are mapped onto CGRA cluster 1 {PE1, PE2, PE3}. The clusters D and E are mapped onto CGRA cluster 2 {PE3, PE4, PE5}. This ensures closely connected nodes are placed in the closely connected CGRA PEs. In this case, node 14 can find a placement with a valid route because nodes 4, 7, 8 have moved right to be closer to 14. This global view lets the mapper fully utilize the available CGRA resources and allows the mapper to reach a valid mapping faster.

3 PANORAMA COMPILER

Problem Formulation: 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. CDG is a directed acyclic graph $D^{'} = (\mathcal{V}, \mathcal{E})$ where the vertices $\mathcal{V}$ represent the cluster of nodes and the edges $\mathcal{E}$ represent the dependencies between clusters. Modulo Routing Resource Graph (MRRG) $H_{II} = (V_H, E_H)$ is a resource graph of the CGRA G that is time extended to II cycles [6]. V_H consists of two types of nodes: FUs (V_H^F) in each PE and ports (V_H^P) in interconnects and RFs. CGRA cluster graph is represented by $G^{'}_{R\times C} = (P, L)$ where elements of P are the physical CGRA clusters, and L represents the link between clusters. R and C denote the number of CGRA cluster rows and columns.

Problem Definition: Given a DFG D and a CGRA G, the problem is to construct a minimally time extended MRRG H_{II} with a mapping $\phi = (\phi_V, \phi_E)$ from $D = (V_D, E_D)$ to H_{II}. ϕ_V denotes the operation to FU mapping (placement), where each operation $v \in V_D$ should have one-to-one mapping to an FU $\phi_V(v) \in V_H^F$. ϕ_E denotes the data dependency mapping (routing), where each data dependency edge $(v_p, v_q) \in E_D$ should map to a set of ports $S \subset V_H^P$ connecting $\phi_V(v_p)$ and $\phi_V(v_q)$. We decompose the problem into three sub-problems: clustering DFG D to CDG $D^{'}$, finding a mapping $\phi^{'}$ from $D^{'}$ to CGRA $G^{'}_{R\times C}$, and finally mapping DFG nodes within CDG nodes to PEs within the CGRA cluster. $\phi^{'}$ denotes the many-to-many CDG to CGRA cluster mapping, where each cluster $v^{'} \in \mathcal{V}$ should have a mapping to physical CGRA cluster $\phi^{'}(v^{'}) \in P$.

Mapping Algorithm: In this section, we present the details of each algorithm step. The input for the algorithm is the DFG of the target loop kernel and the architecture description of the CGRA. First, we partition the DFG to form the CDG. Then CDG is mapped onto the CGRA clusters using a graph drawing-based mapping technique. Finally, in the lower-level mapping, the DFG nodes within each CDG node are mapped to the CGRA PEs satisfying the conditions for a valid mapping ϕ. Algorithm 1 presents the overview of the Panorama mapping. We use Spectral Clustering [15] to partition the DFG, i.e., divide the DFG nodes into non-overlapping clusters such that the number of edges across clusters is minimized. We explore clustering solutions with different numbers of clusters to find the most balanced partitions with near equal cluster sizes (number of DFG nodes in a cluster)(lines 2-4).

However, it is possible to have different cluster sizes even in the most balanced clustering solution. Figure 4 shows an example of cluster mapping with such an imbalanced clustering solution.

Algorithm 1: Panorama Mapping

Input: DFG, Arch: CGRA Architecture, R: CGRA cluster rows, C: CGRA cluster columns, m: maxDFGClusters
Output: DFG mapped on minimally unrolled MRRG

```
1  k = R
2  while k < m do
3  |   CDG_k = SpectralClustering(DFG, k);
4  |   k++;
5  for each CDG in Top3BalancedPartitions(CDG_r, .., CDG_m) do
6  |   Success = False, ζ1 = 1, ζ2 = 1
7  |   while !Success do
8  |   |   Success, ClusMap = ClusterMapping(CDG, r, c, ζ1, ζ2)
9  |   |   ζ1++; ζ2++;
10 CGRA_Mapping(DFG, ClusMap_withMinζ1ζ2, Arch);
```

A DFG cluster with a comparatively bigger cluster size should be allocated more CGRA resources to fully utilize the CGRA (Eg. cluster D is assigned to two CGRA clusters). Similarly, DFG clusters with comparatively smaller cluster sizes should be able to share a single CGRA cluster (Eg. cluster A and B share the same CGRA cluster). Therefore, we propose many to many CDG to the CGRA cluster mapping algorithm. Cluster mapping is done with the top 3 balanced partitions (clustering solutions) (lines 5-9).

Finally, the cluster mapping solution with the least inter-cluster edge routing complexity is used to guide the lower-level CGRA mapping. Cluster mapping solution with minimum ζ_1 and ζ_2 values have the least inter-cluster edge routing complexity (explained in following sections). Lower-level CGRA mapping assigns the DFG nodes to FUs in CGRA PEs (within the assigned cluster) and routes the data dependencies within and across the clusters using the ports in interconnects and RFs. Panorama higher-level mapping can be used to guide any existing lower-level mapping technique. The details of the main algorithms are given in the following sections.

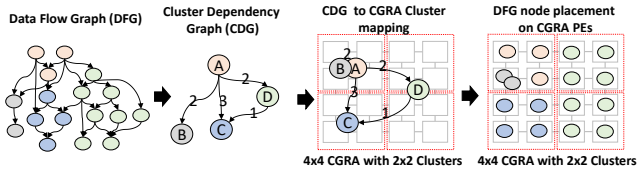


Figure 4: Cluster mapping with imbalanced clusters.

3.1 DFG Clustering

The objective of the clustering algorithm is to group the closely connected community of DFG nodes so that they can be placed and routed in closely connected PEs. We employ spectral clustering to partition the DFG. Spectral clustering has become a prominent clustering algorithm since it outperforms many other traditional clustering algorithms [15]. Spectral clustering uses the spectrum (eigenvalues) of the data's similarity graph, which needs to be clustered. In our case, the DFG is the similarity graph as it represents the relationship between nodes.

The inputs for the spectral clustering algorithm are the DFG and the number k of clusters to construct. The algorithm first computes the Laplacian matrix L of the adjacency matrix of the DFG. Then the first k eigenvectors of L are computed to form a matrix U using eigenvectors as columns. The row i of the matrix U defines the features of the DFG node i. Then the graph nodes are clustered based on these features using the k-means algorithm.

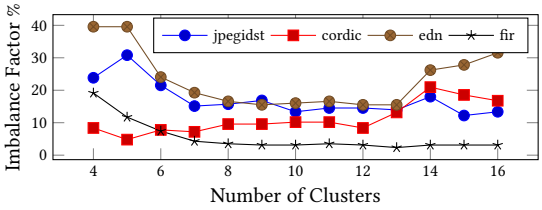


Figure 5: Imbalance factor variation with number of clusters.

Choosing the number of clusters is a crucial step as it defines the complexity of the final mapping. Spectral clustering gives the near-optimal clustering solution minimizing the inter-cluster edges for a given k number of clusters. However, minimizing the inter-cluster edges is not sufficient as the clustering solutions can reach two extremes, with almost all the DFG nodes in one cluster or many clusters with small cluster sizes. Both these extremes diminish the effect of higher-level mapping and are against our goal of simplifying the mapping problem. Thus, we choose the number of clusters based on the size imbalance factor (IF). The IF is the relative difference between cluster sizes in the smallest and largest clusters (relative to total DFG nodes). Figure 5 shows the variation of IF of four DFG kernels against the number of clusters. A low IF denotes more balanced cluster sizes. We set the minimum number of clusters to the number of CGRA cluster rows (R) since the column-wise scattering step in the cluster mapping algorithm requires at least R number of clusters (explained in the next section). We explore clustering solutions between R and m number of clusters where m is an input to the algorithm. For all the evaluated kernels, clustering solutions exist with a IF of less than 20%, proving the effectiveness of spectral clustering in partitioning the DFGs.

3.2 Cluster Mapping

The ultimate goal of the cluster mapping step is to equally distribute the DFG nodes on CGRA, reducing inter-cluster edge distance so that lower-level mapping would be less complex. Our cluster mapping algorithm is inspired by the graph drawing technique called split & push algorithm [14]. We note that SPKM [11] has also used split & push algorithm for kernel mapping on spatial-only CGRAs. However, SPKM is a one-to-one mapping from DFG nodes to PEs. Our problem requires many-to-many mapping between the CDG nodes and CGRA clusters. Thus, our objectives are fundamentally different and hence require very different formulations.

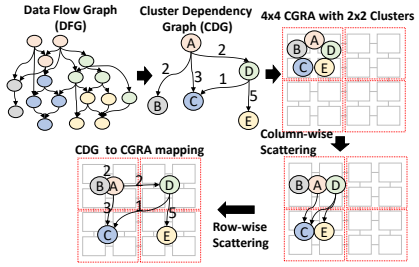


Figure 6: Illustration of cluster mapping algorithm.

Figure 6 shows the flow of the cluster mapping algorithm. The algorithm has two main steps called column-wise scattering and row-wise scattering. It starts by putting all the CDG nodes in a

single CGRA cluster at cluster coordinate (1,1). Then it splits the CDG nodes into two groups (B, A, D and C, E) and pushes one of the groups (C, E) into the adjacent CGRA cluster in the same column. This step is repeated until all the CGRA clusters in the first column are filled with nodes (column-wise scattering). Then the nodes allocated in the first column are scattered row-wise to obtain the final mapping (row-wise scattering). The crucial step of the algorithm is how it splits the nodes into two groups in column-wise scattering. If the split-step cuts adjacent edges (edges that share a common node), it will result in diagonal edges. For example, since column-wise scattering cuts $A-C$ and $D-C$ edges, either $A-C$ or $D-C$ edge becomes a diagonal edge in the final mapping. Diagonal edges would increase the routing complexity in the lower-level mapping as they would require more routing resources than direct edges. Diagonal edges can be minimized by finding a matching cut. The matching cut is defined as a set of non-adjacent edges (edges without common node) whose removal makes the graph disconnected.

Both column-wise scattering and row-wise scattering are formulated as ILP problems, as explained in the following sections. The ILP formulates the problem of finding a many-to-many mapping with the following conditions: 1) distribute cluster nodes across the CGRA clusters proportionate to cluster sizes, 2) minimize diagonal edges, and 3) minimize the distance between dependent clusters based on the number of inter-cluster DFG edges.

3.2.1 ***Column-wise Scattering.*** In this step, we distribute the CDG nodes across the CGRA clusters in a single column. It starts with all the nodes in one cluster, and in each repetition, the nodes are separated and pushed to the next CGRA cluster. The ILP formulation aims to find a matching cut to separate the nodes into two sets.

Boolean Decision Variable: v_{irc} is 1 if i-th CDG node $v_i \in \mathcal{V}$ is not pushed onto the CGRA cluster $p_{(r+1)c}$ from p_{rc}, 0 otherwise ($p_{(r+1)c}, p_{rc} \in P$). r and c are CGRA cluster row and column ids.

Objective Function: $Minimize|\sum_{v_i \in \mathcal{V}} v_{ir1} * |v_i| - (|V_D|/R)|$

where $|v_i|$ denotes cluster size of v_i, $|V_D|$ denotes the total number of DFG nodes. The objective function allows the cluster nodes to distribute across the CGRA cluster rows proportionate to the cluster size. Thus the DFG nodes are distributed evenly across the CGRA cluster rows.

Constraints: Following two constraints are used to reduce the number of diagonal edges by finding a matching cut. These constraints are derived from the fork minimization algorithm in [11, 14] and applied to multi-degree nodes (nodes with degrees higher than one). The key idea is to minimize the number of adjacent edges (of a multi-degree node) cut by the matching cut.

$\sum_{v_j \in adj(v_i^m)} (v_{jr1} + v_{ir1}^m) \leq \zeta_1 + \eta * v_{ir1}^m$

$\sum_{v_j \in adj(c_i^m)} (v_{jr1} + v_{ir1}^m) \geq 2 * deg(v_{ir1}^m) - \zeta_2 - \eta * (1 - v_{ir1}^m)$

where v_i^m is the multi degree node. $adj(v_i^m)$ are the set of nodes adjacent to v_i^m and $deg(v_i^m)$ is the degree of v_i^m. η is a large constant value used to linearize the equations. ζ_1 and ζ_2 are integer variables used to control the number of diagonal edges allowed. If $\zeta_1 = \zeta_2 = 1$, no diagonal edges would be allowed. We increase the ζ_1 and ζ_2 values until we find a feasible solution to ILP.

3.2.2 ***Row-wise Scattering.*** Column-wise scattering assigns each CDG node to a CGRA cluster row. The row-wise scattering distributes those assigned CDG nodes across the row to set the final cluster coordinate for each CDG node. Row-wise scattering is also formulated as an ILP.

Boolean Decision Variable: v_{irc} is 1 if ith CDG node $v_i \in \mathcal{V}$ is mapped onto CGRA cluster column c at the row r fixed in the column-wise scattering.

Objective Function:

$Minimize|\sum_{(v_i,v_j) \in \mathcal{E}} \sum_{c=1}^{C} w(v_i, v_j) \times c \times (v_{irc} - v_{jrc})|$

where $w(v_i, v_j)$ is the number of inter cluster DFG edges between CDG nodes v_i and v_j. The objective function minimizes the distance between dependent clusters allowing dependent CDG nodes with more inter-cluster edges to be placed in closer CGRA clusters.

Constraints:

$\forall i \in \mathcal{V}, \sum_{c=1}^{C} v_{irc} = |v_i|/(|V_D|/R \times C), \quad \sum_{\forall v_i \in \mathcal{V}} v_{irc} \geq 1$

The first constraint allows one to many mappings, i.e., one CDG node, to map onto multiple columns based on the cluster size. ($|V_D|/R \times C$) gives the average number of DFG nodes each CGRA cluster should have when the DFG nodes are equally distributed. The second constraint allows many-to-one mapping, i.e., multiple CDG nodes can be placed on the same CGRA cluster.

Algorithm 2: CGRA Mapping

```
   Input: DFG, Arch, Cluster Mapping
   Output: DFG mapped on minimally unrolled MRRG
1  minII = Minimum(recurrenceMII, resourceMII)
2  while iterate do
3      MRRG = UnrollGraph(Arch, minII);
4      for each node in orderedDFGNodes do
5          for each unmapped FU in MRRG do
6              if Cluster(node) is mapped to Cluster (FU)) then
7                  EstimateLeastCostPlacement();
8          ScheduleAndPlaceNode();
9      while currentTemp > minTemp do
10         overuse= PathFinderRouting();
11         if overuse == 0 then
12             mapSuccess = true;
13         else
14             simAnnealingPlacement();
15             currentTemp = updateTemperature();
16     minII++; iterate = (!mapSuccess && minII < maxII);
```

3.3 CGRA Mapping

Panorama is a portable higher-level mapper which can be combined with any lower-level CGRA mapper. This section explains how Panorama guides a CGRA mapper implemented based on SPR (Schedule, Place, and Route) [2]. The algorithm 2 shows the main body of the mapping process. The inputs are the DFG of the target kernel, cluster mapping result, and detailed architecture description of the target architecture. The minimum II (minII) is found based on the DFG and CGRA characteristics (line 1) [22]. We incrementally increase the minII if the subsequent steps fail to find a valid mapping. For each minII, MRRG is created by unrolling the Architecture resource graph into II cycles (line 3).

To guide the CGRA mapping with Panorama higher-level mapping result, we restrict DFG nodes to all functional units within the CGRA cluster decided from the cluster mapping algorithm (line 6). To obtain the initial mapping, we find the least cost placement for each DFG node out of the designated functional units respecting latency constraints arising from recurrence relationship, i.e., inter-iteration dependencies (lines 4-8) [22]. Then, we use PathFinder-Routing() to find a valid route for all the edges between DFG nodes,

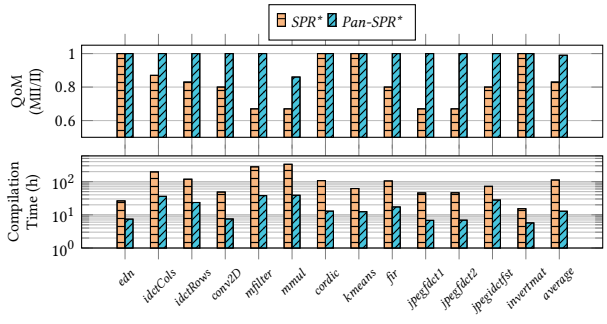


Figure 7: Comparison with SPR*.

including inter-cluster edges (line 10) [24] . The inter-cluster edges and back edges (representing inter-iteration edges) are prioritized for using the inter-cluster communication links (if available). We allow routing resources to be overused when establishing valid routes. If the PathFinderRouting() finds routes with resource overuse, the placement is changed using the simulated annealing-based cooling schedule (line 14) [23]. The simulated annealing-based placement is repeated until a PathfinderRouting() comes up with zero resource overuse routing. The process is terminated if the cooling temperature is less than the minimum temperature allowed.

4 EXPERIMENTAL EVALUATION

We implemented Panorama using a diverse set of tools. DFGs are extracted from annotated C kernels using a DFG generator written in LLVM 10.0 [19]. We implemented DFG clustering and cluster mapping using python libraries, Scikit-Learn [17] for spectral clustering, and gurobipy [18] as an ILP solver. The CGRA lower-level mappers are implemented in C++. The representative loops (Table 1a) are selected from standard DSP benchmark suite mediabench [20] and modern embedded benchmark suite embench [21]. The loop kernels are unrolled to take advantage of larger CGRA and have an average of 432 nodes. To provide an idea of the relative DFG complexity maximum degree of the DFG nodes is also listed. We map kernels onto 16x16 CGRA with 4x4 clusters. Each CGRA cluster has a 4x4 PE array, local memory bank, and PEs in the left-most PE column that can access the memory. We measure the runtime of compilers on the Intel Xeon Gold CPU (2.60GHz).

DFG Clustering and Cluster Mapping: Table 1a summarizes the results of DFG clustering, cluster mapping, and compilation time. K, Inter-E, Intra-E, and STD denote the number of clusters, dependencies across clusters, dependencies within clusters, and standard deviation of cluster size, respectively. The number of dependencies within clusters (Intra-E) is significantly higher than the number of dependencies across clusters (Inter-E), showing the clustering algorithm's effectiveness. Cluster mapping results show how many DFG clusters (CDG nodes) are mapped to each CGRA cluster. The clustering solutions with higher STD results in many-to-many mappings. The average compilation time for clustering plus cluster mapping is 9.23 seconds.

Comparison with Architecture Adaptive Compiler: Architecture adaptive compilers can support a variety of CGRAs, given the architecture description as the input. Table 1b shows the summary of prominent architecture adaptive CGRA compilers proposed in the literature [2, 5–9, 11]. SPR [2] is the most scalable compiler

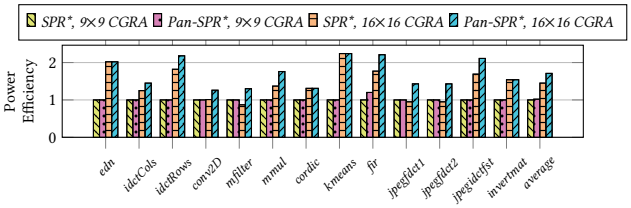


Figure 8: Power efficiency comparison.

evaluated on benchmark kernels with an average of 263 nodes and 16x16 CGRA. We refer to our implementation of SPR as SPR*. Note that SPR* compilation time is comparable with other works for smaller DFG and CGRA sizes. With SPR*, we use a detailed CGRA architecture description, where each PE has RF with eight registers and four read/write ports. All PEs have a neighbor to neighbor connections. Also, six inter-cluster links connect PEs between neighboring clusters. We compare the SPR* with Pan-SPR*. In Pan-SPR*, Panorama higher-level mapping guides the SPR* lower-level mapping, as explained in section 3.3. Figure 7 shows the Quality of Mapping (QoM = Minimum possible II (MII)/ Mapped II) and the compilation time in log scale. Pan-SPR* can generate better mappings (lower or equal II) for all the benchmarks compared to SPR*. Pan-SPR* achieves MII for all benchmarks except *mmul* where SPR* only achieves MII for four benchmarks. *mmul* has high fanout nodes (high max. degree), causing more routing congestions, consequently failing to achieve MII. Pan-SPR* brings down SPR*'s on average 112 hours long compilation time down to a more manageable 12 hours. On average, Pan-SPR* achieves 22% better mapping quality with 8.7x faster compilation time than SPR*. This shows that Panorama can substantially enhance both the performance and compilation time of architecture adaptive compiler.

Figure 8 shows the power efficiency (MOPS/mW) comparison between 9x9 CGRA and 16x16 CGRA (normalized with power efficiency of SPR* mappings on 9x9 CGRA). We implemented two CGRA architectures in RTL and synthesized them onto a commercial 40nm process using Synopsys toolchain to obtain the power numbers at 100MHz frequency. The power efficiency of 16x16 CGRA is 68% higher than the 9x9 CGRA, which shows the benefit of scaling up the CGRA size. The Pan-SPR* achieves 16% power efficiency improvement over SPR* compilation on 16×16 CGRA.

Comparison with Architecture Specific Compiler: Ultra-Fast [3] is a recently published compiler specifically designed for HyCUBE CGRA [4] and claims orders of magnitude improvement in compilation time. In Pan-UltraFast, Panorama higher-level mapping guides the Ultra-Fast lower-level mapping. Our goal is to demonstrate the versatility and effectiveness of our approach for different architectures and easy integration with diverse low-level mapping techniques in the literature.

However, we note that Ultra-Fast assumes extreme single-cycle multi-hop connection, allowing single-cycle communication between any two PEs in the entire array. This assumption simplifies the 3D mapping problem into a 2D mapping problem and substantially reduces the mapping time. It also uses a completely abstract representation of HyCUBE with unlimited registers per PE and again reduces the compilation complexity. These broadly simplifying architectural assumptions can dramatically reduce the compilation time and are expected to reduce II values. Surprisingly, we

(a) Summary of DFG characteristics, clustering, and cluster mapping results

Kernel	DFG Characteristics			Clustering Results				Cluster Mapping Result	Compilation Time (s)	
	Nodes	Edges	Max Deg.	K	Inter-E	Intra-E	STD	(CDG nodes per CGRA cluster)	Clustering	Clus Map
edn	507	633	25	10	76	557	22.6	[2,2,1,1],[2,1,1,2],[2,1,1,1],[2,1,2,1]	7.6	1.15
idctcols	403	580	23	29	85	495	17.5	[3,3,2,3],[1,2,1,3],[3,1,2,1],[1,3,5,1]	7.25	1.07
idctrows	427	694	40	10	61	633	33	[1,1,1,2],[2,1,1,2],[1,1,1,2],[2,1,1,2]	7.11	1.16
2-D convolution	512	666	36	12	98	568	15.6	[1,1,2,1],[1,1,1,1],[1,1,1,1],[2,2,1,4]	8.92	1.18
matched filter	501	572	75	16	85	487	17.7	[1,1,1,2],[1,1,1,1],[1,1,1,1],[1,1,1,4]	10.26	1.12
matrix multiply	503	609	53	16	116	493	11.1	[2,1,1,2],[2,1,1,2],[2,2,1,1],[1,1,1,1]	7.88	1.1
cordic	294	491	14	22	81	410	10.5	[3,3,1,2],[3,1,1,2],[3,1,1,1],[3,1,2,2]	8.21	1.18
k-means clust.	461	545	42	16	96	449	12.5	[1,1,1,2],[1,1,1,1],[1,1,1,3],[1,1,1,1]	10.4	1.19
fir	256	310	49	16	62	248	2.5	[1,1,1,1],[1,1,1,1],[1,1,1,1],[1,1,1,1]	7.75	1.1
jpegfdct	440	593	35	10	54	539	37.4	[1,1,1,3],[2,1,1,2],[1,1,1,2],[1,1,1,2]	7.46	1.1
jpegidctfst	486	626	27	15	86	540	32.7	[1,1,1,2],[1,1,1,2],[1,1,1,1],[1,1,1,6]	7.16	1.17
invertmat	389	610	37	24	103	507	9.9	[2,1,1,4],[1,1,1,3],[1,1,1,1],[3,2,2,7]	7.23	1.09
averege	**432**	**578**	**38**	**16**	**83**	**494**	**18.5**		**8.1**	**1.13**

(b) Summary of previous works

	DFG Nodes	CGRA Size	Compilation Time
CGRA-ME [7]	12	4x4	NA
SPKM [11]	16	4x4	~1s
G-Minor [5]	35	4x4, 16x16	0.2s, 7s
EPIMAP [8]	35	4x4, 16x16	54s, 23min
DRESC [6]	56	4x4	~15min
EMS [9]	4~142	4x4	~37min
SPR [2]	263	16x16	NA
SPR*	30	4x4	30s

Table 1

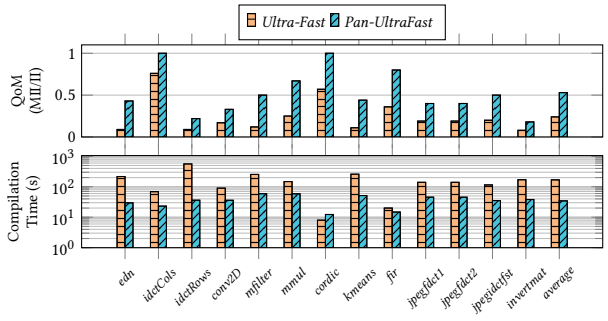


Figure 9: Comparison with Ultra-Fast.

observe that compared to SPR* compilation for a detailed and realistic architecture, Ultra-Fast compilation for abstract (and potentially unrealizable) architecture still generates significantly higher II values ranging from 1.1x to 13x across all kernels (i.e., much lower throughput) due to its greedy placement strategy (compare QoM metric in Figures 7 and 9).

Nonetheless, compared to Ultra-Fast, PAN-Ultrafast improves the quality of mapping by 2.6x and reduces the compilation time by 4.8x. This shows the portability and the effectiveness of Panorama higher-level mapping irrespective of the choice of the low-level mapper. Figure 9 shows the QoM and the compilation time in logscale for these two approaches.

5 RELATED WORK

Many works in FPGA synthesis use cluster-based mapping (also known as packing) to assign technology mapped netlist onto FPGA configurable logic blocks (CLB) [25]. The packing algorithms try to pack nodes in the netlist to each CLB to its full capacity to minimize the number of CLBs needed. The algorithms used to solve the packing problem are not applicable to CGRA because CGRA allows temporal mapping. The goal of the CGRA mapping is to distribute the nodes equally without trying to fully pack the part of resources. Few works in CGRA mapping also use DFG clustering to achieve different objectives [2, 9].

6 ACKNOWLEDGEMENTS

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

7 CONCLUSION

CGRAs are promising as accelerators due to the excellent balance between power efficiency, flexibility, and performance. However, the complexity of the mapping problem limits the scalability of CGRAs. Panorama is a fast, scalable, and portable compiler solution that enables existing compilers to support complex application kernels on larger CGRAs. We demonstrated the capability of Panorama in combination with two state-of-the-art low-level mapping approaches.

REFERENCES

[1] L. Liu et al., "A survey of coarse-grained reconfigurable architecture and design: Taxonomy, challenges, and applications," in CSUR'19
[2] S. Friedman et al., "SPR: an architecture-adaptive CGRA mapping tool," in FPGA'09
[3] J. Lee and T. E. Carlson , "Ultra-Fast CGRA scheduling to enable run time, programmable CGRAs," in DAC'21
[4] M. Karunaratne et al., "HyCUBE: A CGRA with reconfigurable single-cycle multi-hop interconnect," in DAC'17
[5] L. Chen and T. Mitra, "Graph minor approach for application mapping on CGRAs," in TRETS'14
[6] B. Mei et al., "DRESC: A retargetable compiler for coarse-grained reconfigurable architectures," in FPT'02
[7] S. A. Chin et al., "CGRA-ME: A unified framework for CGRA modelling and exploration," in ASAP'17
[8] M. Hamzeh et al., "EPIMap: Using epimorphism to map applications on CGRAs," in DAC'12
[9] H. Park et al., "Edge-centric modulo scheduling for coarse-grained reconfigurable architectures," in PACT'08
[10] Z. Li et al., "LISA: Graph neural network based portable mapping on spatial accelerators," in HPCA'22
[11] J. W. Yoon et al., "A graph drawing based spatial mapping algorithm for coarse-grained reconfigurable architectures," in VLSI'09
[12] T. K. Bandara et al., "REVAMP: a systematic framework for heterogeneous CGRA realization," in ASPLOS'22
[13] Z. Li et al., "Chordmap: Automated mapping of streaming applications onto CGRA," in TCAD'21
[14] G. Di Battista et al., "A split & push approach to 3D orthogonal drawing," in Graph Algorithms And Applications 2'04
[15] U. Von Luxburg, "A tutorial on spectral clustering," in Statistics and computing'07
[16] D. Wijerathne et al., "Himap: Fast and scalable high-quality mapping on CGRA via hierarchical abstraction," in DATE'21
[17] L. Buitinck et al., "API design for machine learning software: experiences from the scikit-learn project," in arXiv'13
[18] Gurobi optimizer reference manual, https://www.gurobi.com
[19] C. Lattner and V. Adve , "LLVM: A compilation framework for lifelong program analysis & transformation," in CGO'04
[20] C. Lee et al., "Mediabench: A tool for evaluating and synthesizing multimedia and communications systems," in MICRO'97
[21] Embench: A modern embedded benchmark suite, https://www.embench.org/
[22] B Ramakrishna Rau, "Iterative Modulo Scheduling: An algorithm for software pipelining loops," in MICRO'94
[23] S. Kirkpatrick et al., "Optimization by simulated annealing," in Science'83
[24] L. McMurchie and C. Ebeling, "Pathfinder: A negotiation-based performance-driven router for FPGAs," in Reconfigurable Computing'08
[25] A. Marquardt et al., "Using cluster-based logic blocks and timing-driven packing to improve FPGA speed and density," in FPGA'99