CLUMAP: Clustered Mapper for CGRAs with Predication

Omar Ragheb and Jason H. Anderson
Dept. of Electrical and Computer Engineering, University of Toronto
Toronto, ON M5S 3G4, Canada
omar.ragheb@mail.utoronto.ca, janders@eecg.toronto.edu

Abstract

Coarse-grained reconfigurable architectures (CGRAs) have gained popularity as accelerators for compute-intensive kernels. Complex CGRA architectures that support key features such as multi-context and predication are being developed to support a wider range of kernels. However, mapping applications on these complex architectures poses significant challenges. In this paper, we provide an architecture-agnostic clustered mapping technique and a new cost function tailored for simulated-annealing placement. The mapper simplifies placement and routing phases, demonstrating significant speedup for popular CGRA architectures: HyCUBE and ADRES. Additionally, our method demonstrates an increase in mapping success for the ADRES architecture.

Keywords

CGRAs, predication, mapping, CAD, multi-context

ACM Reference Format:

Omar Ragheb and Jason H. Anderson. 2024. CLUMAP: Clustered Mapper for CGRAs with Predication. In *61st ACM/IEEE Design Automation Conference (DAC '24), June 23–27, 2024, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3649329.3658269

1 Introduction

Different accelerator architectures offer different strategies for reducing an application's execution time compared to general-purpose CPUs. The use of graphics processing units (GPUs), where single instruction, multiple data (SIMD) engines give enhanced computing capability, has been extensively investigated for machine-learning applications. Another strategy involves creating application-specific integrated circuits (ASICs) specifically for compute-intensive tasks. However, reconfigurable architectures present another approach to realize application accelerators, as the hardware may be adapted to meet the requirements of the application and reconfigurable architectures are more energy efficient than GPUs. The performance and power efficiency of reconfigurable architectures is attractive compared to fixed-function ASICs.

Field-programmable gate arrays (FPGAs) and coarse-grained reconfigurable architectures (CGRAs) are the two primary reconfigurable architectures. CGRAs fall somewhere between custom ASICs and FPGAs in terms of configurability, performance, power, and area efficiency. CGRAs are programmable hardware fabrics that consist of a two-dimensional grid of processing elements (PEs) connected by programmable interconnect. Since the 1990s, extensive research has been conducted on CGRAs, and a wide range of designs have been proposed by the academic community (e.g. [4]), government (e.g. [15]) and industry (e.g. [14]). The primary ways in which CGRAs differ from each other include their logic, I/O capabilities, interconnect, and whether they support dynamic reconfigurability [10].

In this work, we consider CGRA mapping of *single-context* and *multi-context* CGRA architectures that support *predication*. *Multi-context* CGRAs allow multiple configuration bitstreams to be preloaded into the CGRA, which the CGRA cycles through in round-robin style (i.e. time multiplexing the CGRA resources). Each bitstream is referred to as a context, and a context completely defines the function of the CGRA's processing elements and interconnect. *Predication* is an approach to facilitate the implementation of applications with control flow on a CGRA. The predicates are true/false Boolean conditions that are used to determine whether an instruction should be executed or not, as well as to select data from one of two execution paths.

We present an architecture-agnostic CGRA mapper that supports multi-context, and predication architecture features, offering flexibility to target various architectures. The mapper extends prior work done by Ragheb et al. [13], incorporating two enhancements to accelerate the placement and routing processes within mapping. The first addition is clustering, which is used as a preprocessing step before placement and routing. Clustering, hence *CLUMAP*, ensures that interconnected operations are placed together, leading to a more compact placement. This allows simulated annealing to find solutions that utilize fewer CGRA PEs, freeing up more resources throughout the CGRA to be utilized. The clustering step eases the mapping process for intricate PE architectures, particularly those containing PEs that support predication. The second enhancement introduces a new cost function for simulated annealing placement that accounts for the operation scheduling. The cost function improves the utilization of the CGRA interconnect, leading to more efficient routing by balancing interconnect demand across contexts.

2 Background

2.1 Dataflow Graph (DFG) and Device-Model Graph

When mapping an application to a CGRA, the graph representations of the application and the CGRA device (to which the application is mapped) are two key data structures. The mapping problem can be reduced to finding the subset of the architectural graph into which the application graph is embedded. The dataflow graph (DFG), a directed graph, is the graph representation of an application. The edges of the DFG indicate data transfer from the source (producer) operation to the sink (consumer) operation, while each node represents an operation to be executed. The input, output, load, and store

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
DAC '24, June 23–27, 2024, San Francisco, CA, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0601-1/24/06
https://doi.org/10.1145/3649329.3658269

instructions are also represented by nodes in the DFG. Loop-carried dependencies are represented by back-edges within the DFG – that is: cycles within the DFG.

For the device-model graph, a common approach is to use a directed modulo routing-resource graph (MRRG) [11]. Each node in the MRRG represents a hardware resource, such as an arithmetic logic unit (ALU), input/output (IO) ports, multiplexers, etc. The connections between the hardware resources are represented by the directed edges connecting the nodes. Regarding the graph representation's modulo feature, it is used to express multi-context CGRAs, wherein the CGRA's registers allow data to be transferred between contexts. For example, data produced in context i may be stored in a register and then consumed in context $i + 1$. Because contexts switch in a round-robin fashion, the registers in the last context have "wrap around" connections back to context 0.

2.2 Architectures Considered

2.2.1 ADRES The ADRES architecture [12] was one of the first CGRA architectures proposed as a template architecture. It features a top-row of PEs that share one distributed register file (DRF). Generally in ADRES, each PE is connected to its NSEW neighbors with no diagonal or wrap-around connections. The inputs of the PE could either be routed through a bypass multiplexer or routed to the inputs of the ALU. The ALU, or bypass multiplexer result, can be routed to a local register file (LRF) or a DRF to be saved and used within the next cycle.

2.2.2 HyCUBE The HyCUBE architecture [7] offers a flexible interconnect that features a crossbar switch within each PE. At each row/column intersection, there exists a crossbar switch made up of 6 inputs and 6 outputs. Four of the inputs and outputs are connected to the NSEW neighbors of the PE. The other 2 inputs come from the ALU in both registered and combinational forms. As for the 2 other outputs: they are the inputs to the ALU. The flexibility of the interconnect arises because multiple paths of the crossbar can be used concurrently. Lastly, the inputs of the PE also feature both a registered and a combinational path, allowing multi-hop connections to be pipelined.

2.3 CGRA-ME

CGRA-ME [3] is an open source tool for the modeling and exploration of CGRA architectures and the study of CGRA CAD tools. With the use of an API, CGRA-ME allows one to describe a CGRA architecture using a high-level language – *C++*. The tool enables mapping applications onto the modeled CGRA and the generation of Verilog for the CGRA from the architecture model. To verify the functional validity of the Verilog, the mapped application is represented in a configuration bitstream. This allows the Verilog to be simulated alongside the bitstream. Furthermore, the Verilog can be physically realized by ingesting into an ASIC standard-cell toolflow.

3 Overall Flow

The mapping flow introduced by Ragheb et. al. [13] forms the basis for the approach outlined in this paper, where the routing and scheduling algorithms remain unchanged. The mapping process follows a sequential flow that includes several key steps. The initial step involves scheduling, using an ASAP scheduler, followed by clustering. If clustering is successful, the next phases are placement and routing. After clustering, placement moves clusters as atomic units, instead of single DFG operational nodes. If a legal solution is achieved, the mapping process is considered complete. However, if a legal solution cannot be found, another round of placement and routing is initiated. The mapping process continues until either a designated time-out limit is reached or a predetermined threshold number of iterations has been completed. At that point, the mapping process has failed.

4 Clustering

CGRA architectures have experienced an increase in complexity to support a broader range of applications, incorporating intricate features such as predication networks, the integration of multiple ALUs within a single PE, and the introduction of heterogeneous PEs. Additionally, routing within a PE maybe constrained causing the mapping process to be more challenging. To address this, we developed and implemented a clustering algorithm, allowing the grouping of operations within a single PE. The primitives within a cluster are then moved as a unit during placement. Moreover, during the placement phase, which utilizes the simulated annealing algorithm, clusters have the capability to be placed together on the same PE if they use different functional primitives within a PE.

The clustering algorithm begins by traversing the device-model graph, generating PE templates for each distinct PE structure. Its primary objective is to accommodate as many DFG operational nodes as possible within a PE template, while ensuring the routability of dataflow edges.

4.1 Definitions

Several key definitions will be employed in the paper to describe the clustering algorithm. The term **Ops** refers to the list of operation nodes in the dataflow graph, each denoted as **O**. **F** represents the functional nodes within the device model. The set **Neigh(O)** pertains to the input/output operation nodes connected to a specific operation **O**. The set **PEs(O)** refers to the PEs containing functional nodes capable of mapping the operation **O**. Within a PE, the set **FUs(O, PE)** denotes the functional nodes **F** capable of mapping a particular operation **O**. While the set **FUs(clust, PE)** denotes a set of placements **P** of the cluster within a PE. Additionally, **OpVisited** signifies the set of operations that have been visited during the algorithmic process and **OpQueue** signifies a queue of operations. While **validPlacements** represents a set of valid placements of operations wihtin a cluster to functional nodes within a PE and **FUUsed** represents a set of functional nodes that are utilized.

4.2 Main Algorithm

The clustering algorithm is outlined in Algorithm 1. It takes as input a list of all DFG operational nodes `Ops` and the different types of `PE` templates. The algorithm starts by adding operations with no inputs (i.e. starting nodes `0`) within the DFG to `OpQueue`. Algorithm 1 iteratively dequeues operations, identifies PE templates capable of mapping the operation, `PEs(O)`, and iterates over each PE `p` within `PEs(O)`. The function `generateCluster`, discussed in Section 4.3, is invoked for each PE with arguments that include operation `O` and `tempClust`. The function populates `tempClust` with a cluster that fits within PE `p`. On Line 13, the algorithm checks if the generated

Algorithm 1: Clustering

```
Data: Ops, List of template PEs
Result: List of all Clusters
1  Init OpQueue, OpVisited, Clusters;
2  foreach operation O with no inputs do
3      Enqueue O into OpQueue;
4  while (OpQueue is not empty) do
5      Dequeue the front operation O from OpQueue;
6      if (O is not in OpVisited) then
7          PEs(O) ← getPEsFor(O);
8          if (PEs(O) is empty) then
9              ERROR("No PE could map operation");
10         Init clust, tempClust, PE;
11         foreach p in PEs(O) do
12             generateCluster({O}, tempClust, {∅},
                OpVisited, p);
13             MAX({clust, PE}, {tempClust, p});
14         if (validate(clust, PE) returns false) then
15             Enqueue O back into OpQueue;
16         else
17             foreach O in clust do
18                 Add O to OpVisited;
19                 Enqueue Neigh(O) into OpQueue;
20             Add clust to clusters;
21 return Clusters;
```

cluster contains more operations than the current cluster `clust`. If true, it sets the generated cluster to the current cluster and the generated PE to the current PE. Otherwise, it discards the generated cluster.

Next, the function on Line 14 executes a series of routing validation checks, as discussed in Section 4.4, to ensure that the routing within the PE is sufficient to host the cluster. If the validation fails, the operation `O` is re-enqueued for future processing. To prevent the recurrence of the same cluster, the validation algorithm imposes constraints on the operations of the cluster not to be grouped together. Otherwise, the algorithm marks the operations within the current cluster as visited by adding them to `OpVisited`, enqueues the neighboring operations `Neigh(O)` to the operations within the cluster to the `OpQueue`, and adds the cluster to the list of clusters.

This process continues until the `OpQueue` is empty, and the algorithm returns a Boolean indicating the success of the clustering operation.

4.3 Generate Cluster

Algorithm 2 presents the `generateCluster` function, invoked on line 12 of Algorithm 1. This function creates a cluster for a given PE template by applying the following steps:

- Create an empty list `nextOps` that will hold neighboring operations `Neigh(O)` of clustered operation `O`.
- For each unvisited and unclustered operation `O` within the list `Ops`, iterate through the unused functions `F` within `PE`. If there exists a function `F` that can map operation `O`, add operation `O` to the cluster, mark the function `F` as used by adding it to `FUUsed`, and push the neighbouring operations `Neigh(O)` to `nextOps`.
- If `nextOps` is not empty, recursively call the algorithm with the next set operations while keeping the other arguments the same, to find additional operations that can fit within the same cluster.
- The algorithm concludes when there are no more operations being clustered, causing `nextOps` to be empty.

Algorithm 2: Generate Cluster (generateCluster)

```
Data: Ops, clust, FUUsed, OpVisited, PE
1  Init nextOps;
2  foreach O in Ops do
3      if (O is not in clust and O is not in OpVisited) then
4          foreach F in PE do
5              if (F is not in FUUsed and F can map O) then
6                  addConnOps(&nextOps, Neigh(O));
7                  FUUsed.add(F);
8                  clust.add(O);
9                  break;
10 if (nextOps is not empty) then
11     generateCluster(nextOps, ...);
12 return clust;
```

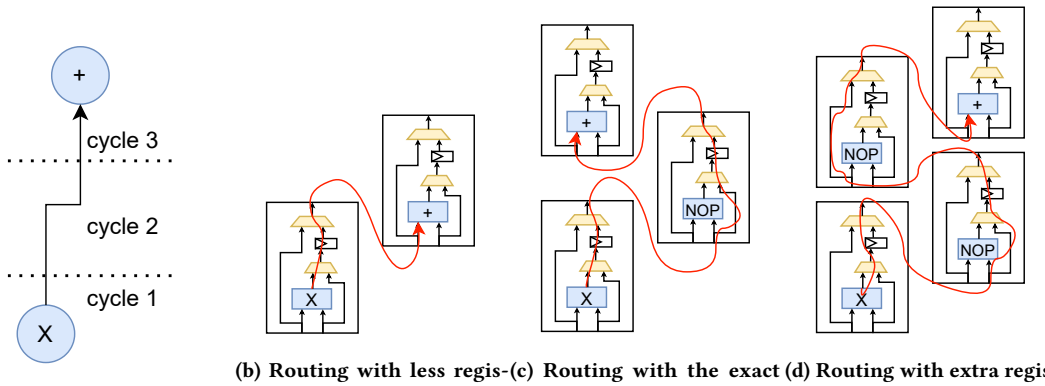
This iterative and recursive approach ensures that the algorithm explores a wide range of combinations of operations that could be clustered for a given PE.

4.4 Validate Cluster

Algorithm 3: Validate Cluster

```
Data: clust, PE
Result: Boolean indicating validity
1  Init validPlacements;
2  foreach O in clust do
3      FUs(O, PE) ← getFUSetForOp(O, PE);
4  FUs(clust, PE) ← enumPlace(FUs(O, PE));
5  foreach P set in FUs(clust, PE) do
6      if validateRoutes(P) is true then
7          validPlacements.add(P);
8  foreach O in clust do
9      foreach F in FUs(O,PE) do
10         if F is not found wihtin validPlacements then
11             FUs(O,PE).remove(F);
12     if No valid functions F for operation O then
13         return false;
14 return true;
```

Algorithm 3 validates the compatibility of a given cluster with a PE template. It starts by populating the map `FUs(O, PE)`, a key component defined within the overall mapping algorithm. It populates the map by looping over each operation `O` within the cluster `clust` and calls the function `getFUSetForOP(O, PE)` on line 3, which returns a set functional units that is able to map operation `O` within the template `PE`.



(a) DFG (b) Routing with less registers on the path (c) Routing with the exact number of registers (d) Routing with extra register on the path

Figure 1: Motivation of the utilizing the latency cost

Then Algorithm 3 enumerates all possible placements of the cluster `clust` within the PE template, by invoking the function `enumPlace(FUs(O, PE)` on line 4 and stores the set of placements into `FUs(clust, PE)`. This enumeration is feasible due to the limited number of functional units within the PE. Each placement `P` within `FUs(clust, PE)` undergoes validation to ensure the existence of sufficient routes for each function unit `F` that maps operation `O` within the cluster. This validation is done by calling the function `validateRoutes(P)` on the placement `P`, on line 6. The function applies two sets of checks: firstly, it verifies whether the PE has enough routes to connect each utilized functional unit within the placement `P`; secondly, it checks if the PE possesses sufficient routes to the global routing of the CGRA, enabling all operations within the cluster to connect to operations outside of the cluster `clust`. If a placement `P` is deemed valid, it is included in a set of valid placements `validPlacements`.

In the final step, Algorithm 3 loops over each operation `O` within the cluster `clust`, ensuring that for each functional unit `F` able to map operation `O` within PE template `PE`, valid placements could be found. If the functional unit is not found then it is removed from the global map `FUs(O, PE)`. During the placement phase, only functional units within this map can place operation `O`. Finally, if there are no functional units within the set `FUs(O, PE)` that passed the validity checks for mapping operation `O`, Algorithm 3 returns false; otherwise, it returns true.

5 Scheduling-Aware Cost function

The placement algorithm presented in the mapping algorithm detailed in [13] utilizes simulated annealing and the common half-perimeter bounding-box wirelength (HPWL) cost function. However, this cost function is designed for FPGA architectures, which are significantly different CGRAs in terms of granularity and routing resources. Here, we propose an extended cost function that is aware of the scheduling of operations. The extended schedule-aware cost function (SC) function is utilized in conjunction with the HPWL cost function, as shown in Equation 1

$$TotalCost = \alpha * HPWL + (1 - \alpha) * SC \quad (1)$$

Here, α represents a weight parameter, used to flexibly balance the total cost between the HPWL and SC.

The motivation behind the cost function can be seen in Fig. 1. Given a multiply and add operation, scheduled two cycles apart as shown in Fig. 1a, the ideal placement of the two operations would be two PEs apart as shown in Fig. 1c. However, if the placements of the operations are one PE apart, as illustrated in 1b, then the routing will struggle to find a correct solution. On the other hand, if the operations are placed more than two PEs apart as shown in Fig. 1d, although the routing would be valid, each extra PE would be used as a bypass, thereby wasting the PE and rendering it unusable for useful computational work. The SC cost function employed to overcome these drawbacks is a scheduling-aware cost function:

$$SC = \sum_{e=1}^{Eedges} (|M_d - C|) * C \quad (2)$$

In equation 2, C denotes the number of cycles required between these two operations, and M_d denotes the Manhattan distance between the two placed operations in terms of height, width, and depth (i.e. context index difference). The absolute value of the difference between M_d and C, is taken ensuring that the cost is zero if the operations are placed at the optimal distance and non-zero otherwise. Additionally, the cost is scaled by the number of cycles C, aiming to prioritize edges with more cycles at the ideal distance. Lastly, equation 2 sums up the scaled version of the distance difference between M_d and C for each application dataflow graph edge.

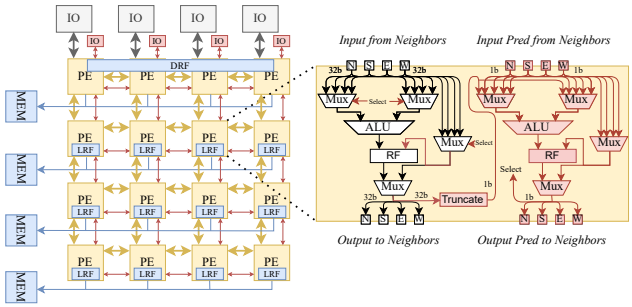


Figure 2: ADRES predication architecture.

6 Predication

Predication involves associating data or an operation with a predicate (true/false value) to manage its execution. Consider for example, an if-then-else (ITE) structure, where the "if" statement's operations are chosen to be executed when the predicate is true. Conversely, operations linked to the "else" statements are chosen to be carried out if the condition is false (i.e. the predicate is false). ITE statements are commonly used in applications, which makes them a barrier to the broadening the applications that can be mapped onto CGRAs. ITE statements can be realized through predication

Table 1: Runtime for CLUMAP, compared to previous work (TO is timeout of 2 hours).

	Benchmark	nomem1	sum	mac	simple	nomem2	conv2	MM	accum	conv3
HyCUBE	CLUMAP	0.05	0.30	0.11	1.78	0.07	1.97	0.24	3.07	3.74
	ILPMap	10.00	133.01	114.04	202.85	12.48	970.90	1286.61	4222.69	TO
	HeuMap	23.29	28.56	35.45	36.60	24.19	26.11	39.83	34.53	TO
ADRES	CLUMAP	0.03	0.28	0.11	31.35	0.06	1.64	6.45	23.69	276.04
	ILPMap	9.79	5.65	133.36	197.26	5.01	112.69	357.61	442.25	1758.79
	HeuMap	17.60	28.91	31.39	TO	18.95	TO	TO	TO	TO

techniques in both hardware and software, making predication a crucial component that CGRAs require to execute applications that contain control flow.

The predication scheme we choose to support is known as partial predication, in which predicates are applied to a partial set of operations that decide the correct result of the application. The support of partial predication entails the following modifications:

- **Hyperblock generation**: Utilizing an intermediate language such as LLVM or MLIR for the generation of hyperblocks, similar to Liolli et al. [9].
- **Architecture extension:** Extending the architecture to include a separate 1-bit wide interconnect network, highlighted in red in Figure 2.
- **1-bit ALU Addition:** Adding a 1-bit ALU, as shown in Fig. 2, capable of performing bitwise AND, OR, and XOR operations, which are used to merge predicates.
- **Multiplexer modification:** Modifying multiplexers to implement the SELECT and PHI instructions. This involves loading two configurations into the multiplexer, with the predicate determining which configuration to utilize. When the multiplexer implements neither a PHI nor a SELECT operation, both configurations are loaded with the same value, causing the predicate to be ignored.
- **Mapping support:** Incorporating mapping support for predicated SELECT/PHI instructions, along with multi-bitwidth interconnect.

All the aforementioned changes have been implemented within the CGRA-ME tool for both the ADRES and HyCUBE architectures. These adaptations enable the tool to effectively handle partial predication.

7 Experimental Results

We conducted two experiments to demonstrate the capabilities of CLUMAP. The first compares CLUMAP with earlier mappers on a set of 11 DFGs with no predication. The second experiment examines how the mapper is affected by the optimizations in both single-context and multi-context, predicated architectures. In the second set of experiments, a set of 14 DFGs with both PHI and SELECT instructions are mapped to ADRES and HyCUBE architectures with predication support.

7.1 Comparison with previous mappers

We compared this work to earlier mappers within CGRA-ME in terms of runtime. We used the ILP-based mapper (ILPMap) from [2] and heuristic based mapper (HeuMap) presented in [16]. Benchmarks with various DFG sizes are mapped to a CGRA size of 6×6 for both HyCUBE and ADRES architectures. Note that the experiments are conducted on DFGs without PHI or SELECT instructions. Any PHI operation present within the DFGs are replaced with back-edges into the operation that feeds the PHI. Additionally, the architectures do not include the predicated architecture changes mentioned in Section 6. This is because previous mappers do not support predicated architectures.

The results obtained are summarized in Table 1. As the results show, this work brings about multiple orders of magnitude speedup over previous mappers for all benchmarks. Furthermore, both the ILP-Mapper and heuristic mapper times out (TO) (TO means that the mapper failed to find a valid mapping within a 2 hour time limit) for the conv3 mapping to the HyCUBE architecture. For the ADRES architecture, the heuristic mapper TOs on 5 benchmarks, while ILP amd CLUMAP successfully map all benchmarks. CLUMAP demonstrates improved runtime compared to the existing mappers in CGRA-ME. Note also that previous mappers do not guarantee a latency-balanced mapping, while CLUMAP does guarantee it.

7.2 Optimization Study

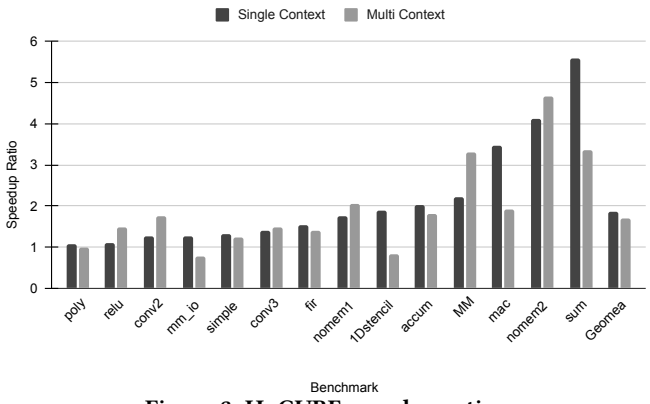


Figure 3: HyCUBE speedup ratio.

To assess the impact of the optimizations, we mapped a set of 14 benchmarks on predicated HyCUBE and ADRES architectures. This set comprises the 9 benchmarks used in the previous study, now with predicated PHI instructions to observe the impact on mapping time, and an additional 5 benchmarks that include predication. For the mapping, we employed three mapping strategies:

- Baseline with no optimizations, presented in by Ragheb et al. [13].
- Baseline with clustering discussed in Section 4.
- Baseline with clustering and scheduling-aware cost function discussed in Section 5.

Because of space limitations, and for reasons given below, the first and the third mapping strategies were used to map benchmarks on

Table 2: Shows ADRES architecture results of runtime and speedup, for the clustering optimization and clustering along with the scheduling-aware cost function compared with baseline (TO is timeout of 2 hours).

	Baseline	Clust		Clust + Sched Cost	
Benchmark	Time (s)	Time (s)	Speedup	Time (s)	Speedup
nomem1	15.33	1.70	9.03	0.90	1.89
sum	20.93	4.00	5.23	2.83	1.42
mac	69.78	69.10	1.01	20.00	3.45
nomem2	13.25	3.84	3.45	1.97	1.95
conv2	67.17	35.14	1.91	9.04	3.89
conv3	TO	TO	TO	158.80	TO
fir	386.71	59.49	6.50	24.52	2.43
poly	TO	133.48	TO	315.01	0.42
relu	TO	582.03	TO	39.10	14.89
1Dstencil	TO	343.84	TO	248.35	1.38

a 6 × 6 single-context and 4 × 4 multi-context HyCUBE architecture. For the ADRES architecture, we applied the three mapping strategies to map benchmarks on 6 × 6 single-context architecture.

Fig. 3 shows the speedup ratio between the first and third mapping strategies. The dark grey bars represent the mapping of benchmarks on single context, while, the light grey bars depict the mapping on a 4 × 4 CGRA with a minimum of 2 contexts. For the single context, the optimization consistently leads to a speedup across all benchmarks, resulting in a geometric mean speedup of **1.89**. In the multi-context mapping, there is a degradation for 2 benchmarks, namely mm_io and 1DStencil. However, the other 12 benchmarks experience significant speedups, resulting in a geometric mean speedup of **1.69**. Note that results for the second mapping strategy are omitted for HyCUBE as the difference between it and the third mapping strategy is not significant. This is primarily due to the existence of a crossbar in HyCUBE offering sufficient routing, causing the effect of the scheduling-aware cost function to be negligible.

In Table 2, columns 2, 3, and 5 give the runtime for each of the three mapping strategies for single context ADRES architecture, respectively. Column 4 represents the speedup ratio between mapping strategies 1 and 2, and column 6 represents the speedup ratio between mapping strategies 2 and 3. The mapping of 4 out of the 14 benchmarks, namely mm_io, simple, accum, and MM (matrix multiply), failed on all strategies; hence, they are not shown in the table. By solely utilizing the clustering algorithm, 3 benchmarks (poly, relu, and 1DStencil) were successfully mapped, avoiding timeouts (TO), and there was a consistent speedup in terms of runtime. However, mapping strategy 3 leads to the successful mapping of the conv3 benchmark and all benchmarks experience significant speedup in terms of time, resulting in a geometric mean of 2× speedup across the 9 benchmarks that passed on both mapping strategies 2 and 3.

8 Related Work

Mapping on CGRAs has been extensively studied, as CAD mapping support is an essential requirement for any architecture. However, most of the mapping approaches that have been proposed are tied to a certain architecture. One of the flexible extensible approaches is SPR [5]. Although this approach considers multi-context CGRAs, it does not include mapping for single-context and predicated architectures. There have been multiple approaches that utilize ILP and Boolean SAT [1, 2, 16]. However, these approaches are usually time consuming since they are trying to find an exact legal optimal solution.

Han et al. [6] explores different predication approaches to reduce the power consumption of the CGRA. The different approaches include a state-based predication. Dual-issue single-execution (DISE) maps mutually exclusive instructions onto the same PE and partial predication decides which instruction is executed. Another interesting approach to predication was introduced in [8], called 4D-CGRA. However, all previous approaches offer predication techniques that are specific to the given architecture and cannot be extended to any CGRA architecture.

9 Conclusion

We introduced a CGRA mapping approach that, while based on simulated annealing-based placement and negotiated congestion routing, incorporates two optimizations: pre-clustering of primitives into processing elements, and a schedule-aware cost function in placement. These optimizations prove to be valuable additions to the architecture-agnostic CGRA mapper presented by Ragheb et. al. [13]. The optimizations demonstrate significant speedup and improvements to mapping success for the HyCUBE and ADRES CGRA architectures.

References

[1] Samit Chaudhuri and Asmus Hetzel. 2017. SAT-based compilation to a non-von Neumann processor. In *IEEE/ACM ICCAD*. 675–682.

[2] S Alexander Chin and Jason H Anderson. 2018. An architecture-agnostic integer linear programming approach to CGRA mapping. In *ACM/IEEE DAC*.

[3] S. Alexander Chin, Noriaki Sakamoto, Allan Rui, Jim Zhao, Jin Hee Kim, Yuko Hara-Azumi, and Jason Anderson. 2017. CGRA-ME: A unified framework for CGRA modelling and exploration. In *IEEE ASAP*. 184–189.

[4] Carl Ebeling, Darren C. Cronquist, and Paul Franklin. 1996. RaPiD — Reconfigurable pipelined datapath. In *FPL*, Reiner W. Hartenstein and Manfred Glesner (Eds.). 126–135.

[5] Stephen Friedman, Allan Carroll, Brian Van Essen, Benjamin Ylvisaker, Carl Ebeling, and Scott Hauck. 2009. SPR: an architecture-adaptive CGRA mapping tool. In *ACM FPGA*. 191–200.

[6] Kyuseung Han, Junwhan Ahn, and Kiyoung Choi. 2013. Power-efficient predication techniques for acceleration of control flow execution on CGRA. *ACM TACO* 10, 2 (2013), 1–25.

[7] Manupa Karunaratne, Aditi Kulkarni Mohite, Tulika Mitra, and Li-Shiuan Peh. 2017. HyCUBE: A CGRA with reconfigurable single-cycle multi-hop interconnect. In *IEEE/ACM DAC*.

[8] Manupa Karunaratne, Dhananjaya Wijerathne, Tulika Mitra, and Li-Shiuan Peh. 2019. 4D-CGRA: Introducing branch dimension to spatio-temporal application mapping on CGRAs. In *ACM/IEEE ICCAD*.

[9] Austin Liolli, Omar Ragheb, and Jason Anderson. 2021. Profiling-Based Control-Flow Reduction in High-Level Synthesis. In *IEEE FPT*.

[10] Leibo Liu, Jianfeng Zhu, Zhaoshi Li, Yanan Lu, Yangdong Deng, Jie Han, Shouyi Yin, and Shaojun Wei. 2019. A Survey of Coarse-Grained Reconfigurable Architecture and Design: Taxonomy, Challenges, and Applications. *ACM Comput. Surv.* 52, 6, Article 118 (10 2019), 39 pages. https://doi.org/10.1145/3357375

[11] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. 2002. DRESC: A retargetable compiler for coarse-grained reconfigurable architectures. In *IEEE FPT*. 166–173.

[12] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. 2003. ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix. In *FPL*.

[13] Omar Ragheb, Rami Beidas, and Jason Anderson. 2023. Statically Scheduled vs. Elastic CGRA Architectures: Impact on Mapping Feasibility. In *IEEE IPDPS Workshops (CGRA4HPC)*. 468–475.

[14] M. Suzuki, Y. Hasegawa, Y. Yamada, N. Kaneko, K. Deguchi, H. Amano, K. Anjo, M. Motomura, K. Wakabayashi, T. Toi, and T. Awashima. 2004. Stream applications on the dynamically reconfigurable processor. In *IEEE FPT*. 137–144.

[15] Cheng Tan, Chenhao Xie, Ang Li, Kevin J. Barker, and Antonino Tumeo. 2020. OpenCGRA: An Open-Source Unified Framework for Modeling, Testing, and Evaluating CGRAs. In *IEEE ICCD*. 381–388. https://doi.org/10.1109/ICCD50377.2020.00070

[16] Matthew JP Walker and Jason H Anderson. 2019. Generic connectivity-based CGRA mapping via integer linear programming. In *IEEE FCCM*. 65–73.