Monomorphism-based CGRA Mapping via Space and Time Decoupling

Cristian Tirelli
Faculty of Informatics
Università della Svizzera italiana
Lugano, Switzerland
cristian.tirelli@usi.ch

Rodrigo Otoni
Faculty of Informatics
Università della Svizzera italiana
Lugano, Switzerland
otonir@usi.ch

Laura Pozzi
Faculty of Informatics
Università della Svizzera italiana
Lugano, Switzerland
laura.pozzi@usi.ch

***Abstract*—Coarse-Grain Reconfigurable Arrays (CGRAs) provide flexibility and energy efficiency in accelerating compute-intensive loops. Existing compilation techniques often struggle with scalability, unable to map code onto large CGRAs. To address this, we propose a novel approach to the mapping problem where the time and space dimensions are decoupled and explored separately. We leverage an SMT formulation to traverse the time dimension first, and then perform a monomorphism-based search to find a valid spatial solution. Experimental results show that our approach achieves the same mapping quality of state-of-the-art techniques while significantly reducing compilation time, with this reduction being particularly tangible when compiling for large CGRAs. We achieve approximately $10^5\times$ average compilation speedup for the benchmarks evaluated on a 20×20 CGRA.**

***Index Terms*—compilation, optimization, modulo scheduling, satisfiability modulo theories**

I. Introduction

As everyday applications require evermore computational power, there has been a growing need for high-performance and low-power architectures. Various solutions exist to efficiently perform compute-intensive tasks under tight power-resource constraints, some more effective than others. Application Specific Integrated Circuits (ASICs) accelerators, for example, provide excellent energy performance, but they are limited by their fixed functionality. Field Programmable Gate Arrays (FPGAs) are more flexible, since they can be reconfigured for different applications as needed, being ideal for prototyping and custom hardware implementations; this flexibility, however, comes at the cost of lower energy efficiency [1], [2]. Coarse-Grain Reconfigurable Arrays (CGRAs) offer a balanced compromise, with run-time reconfigurability at the instruction level and high computational efficiency [3], [4]. These characteristics are well suited to domains such as streaming and multimedia applications [5]–[9], but also in edge domains where they enable flexible hardware acceleration in resource-constrained scenarios. A CGRA is composed of a set of Processing Elements (PEs) usually organized in a 2D mesh near-neighbor topology. Every PE contains an Arithmetic Logic Unit (ALU) and a number of internal registers organized in a register file, as depicted in Fig. 1. Besides being connected to their neighbors according to a mesh topology, PEs also share a connection to an external memory, in order to load inputs and store outputs.

This work was supported by the Swiss National Science Foundation via project ADApprox (grant 200020_188613).

The main challenge of CGRA exploitation is the compilation process: translating high-level code onto the CGRA while taking advantage of the parallelism of the architecture. Compilers achieve this by following three steps: identify the code loops to be accelerated, extract their Data Flow Graphs (DFGs), and find valid *space-time* mappings for them. The latter notion refers to mapping DFG instructions to the right *place*, i.e., assigning them to PEs in a way that data dependencies are respected through the CGRA network, and at the right *time*, i.e., scheduling DFG instructions so that the data produced by every PE is consumed at the right time by other PEs. The quality of the space-time mapping directly affects performance, and existing mapping techniques rely on heuristics to schedule, place, and then route operations and data on the PEs. They suffer from limited scalability, however, which hampers their practical usage. To address this, we propose and evaluate a novel monomorphism-based approach for scalable CGRA mapping. Our idea is to decouple the spatial and temporal dimensions of the mapping process. First, we find a time solution that ensures spatial feasibility, and then we use a monomorphism search algorithm to efficiently find a solution in the spatial dimension. We provide a proof that a monomorphism is always present for time solutions under our constraints.

To summarize, our contributions are the following:

1) Decoupling of space and time for CGRA mapping.
2) Monomorphism-based mapping for CGRA compilation.
3) Proof of monomorphism presence given a time solution.
4) Evaluation showcasing the scalability of our approach.

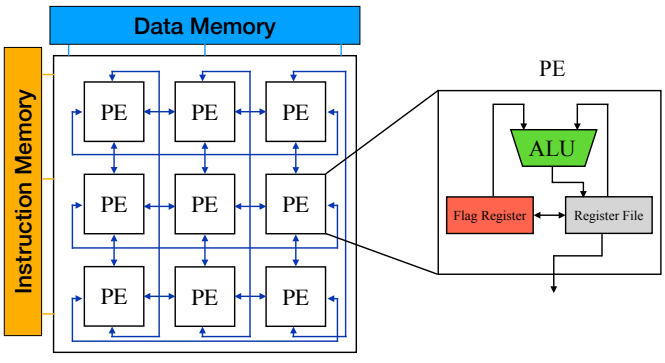

Fig. 1: 3×3 CGRA with internal view of a PE.

978-3-9826741-0-0/DATE25/© 2025 EDAA

II. Related Work

A recent survey [10] summarized the evolution of CGRA architectures and methodology advancements in the last thirty years. Here, we focus on the CGRA mapping problem, i.e., the mapping of application instructions onto PEs that is an integral part of the compilation of software source code onto a CGRA. Solutions proposed in the literature can be divided into two categories: heuristics and exact methods.

On the heuristics category, one of the first approaches proposed used simulated annealing for scheduling, placement, and routing altogether [11]. Addressing all three tasks simultaneously, however, leads to long execution times, low-quality solutions, and limited scalability. As an alternative, an edge-centric approach was proposed [12]. This method generates a mapping by prioritizing the routing of each edge, with placement being handled as a consequence of the routing process. Subsequently, leveraging graph-based techniques to solve the mapping problem became increasingly prevalent in the literature. For instance, EPImap [13] proposed an epimorphic mapping method, where a solution is found by searching for a mapping of the DFG onto the Modulo Routing Resource Graph (MRRG) [14], which is a graph representation of the architecture's evolution through time. Epimap's performance was later improved by GraphMinor [15] and REGIMap [16], by reducing the mapping problem to the graph minor and maximum clique problem. REGIMap was extended to also be aware of the internal register of the PEs [17]. More recently, CRIMSON [18] proposed a randomized iterative MS algorithm that explores the scheduling space more efficiently. An improved version of it, PathSeeker [19], is able to analyze the mapping failures and perform some local adjustments to the schedule to obtain shorter compilation time and better solutions. All these approaches attempt to first guess a schedule and then search for a spatial solution. They, however, do not guarantee that a spatial mapping can be found for a given time solution. In our work, we prove that if a schedule with some specific properties is identified, a valid space mapping always exists.

Exact methods try to abstract and optimally solve the mapping problem. One of the earlier works in this category proposes an integer linear programming (ILP) formulation and proves the feasibility of mapping for a given Iteration Interval (II) [20]. Alternatively, the usage of a Boolean satisfiability (SAT) formulation instead of an ILP one was later proposed [21]. This work was the first effort aimed at exploiting the power of modern SAT solvers in this context, but it has a critical limitation: it is not capable of modeling loop-carried dependencies. This limitation was recently overcome by the formulation of SAT-MapIt [22], which was shown to outperform comparable approaches in a hardware-agnostic context [23].

EPImap [13] uses epimorphism to find a mapping, and in this sense it is related to our monomorphism method. The main differences are that it addresses the space and time dimensions simultaneously, and that it adds routing nodes to the DFG to solve the mapping problem, leading to increased II. An initial approach to time and space decoupling, with a temporal search similar to EPImap's method, was presented in [24]. By adding routing nodes to the DFG, however, it leads to increased II. A last approach of note is HiMap [25], which uses hierarchical mapping for high scalability, but targets multidimensional kernels and is outside the scope of our work.

III. Background

A. Compilation

The choice of which compute-intensive loop should be accelerated is the first step of the compilation process. It can be done automatically [26] or manually via pragma-annotations, with the latter being the method used in our approach. After the loop has been chosen, we convert it into the LLVM intermediate representation [27]. From there, we extract a DFG, whose nodes represent instructions and edges represent data dependencies and loop-carried dependencies. Fig. 2a depicts a DFG, which we will use as running example. The next step is the mapping phase, where each node of the DFG is assigned to a specific PE at a given cycle. This is where our approach is applied.

B. Modulo Scheduling

To exploit the CGRA's architectural capabilities, we make use of Modulo Scheduling (MS) [28], an optimization technique aimed at reducing the number of cycles needed to execute a loop by interleaving multiple iterations of it. Once applied, the chosen loop is divided into three stages: prologue, kernel, and epilogue. The prologue and epilogue are executed only once: the former prepares the data for pipelining, while the latter finalizes the data produced from the pipeline. The kernel is repeated multiple times and includes the instructions to be parallelized through pipelining; its length is the II. A high quality mapping is associated with a low II, with the focus of mapping methods being to find the lowest possible II for a given DFG. An example of a legal mapping for the DFG in our running example is shown in Fig. 2b.

C. Mapping Problem

We tackle the mapping problem by dividing it into two distinct phases: time and space. First, we focus on finding a valid time solution, meaning that we aid to find a schedule that correctly resolves all the data dependencies in the DFG. Once the time solution is found, we move to the spatial phase, where all the placement of operations and routing of data is done accordingly to the CGRA size and topology. Fig. 2c illustrates valid and invalid solutions in time and space. On the figure's right, we have an invalid schedule for nodes 2 and 8 from the running example, with both nodes being scheduled at the same time step despite a dependency between them. An invalid spatial mapping is also shown, in which node 4 is placed on PE0 and node 7 on PE3. Due to the topology of the CGRA, the routing between PE0 and PE3 is not possible, causing the loop-carried dependency to be violated.

IV. Methodology

Decoupling space and time allows the search for a CGRA mapping to be done in two phases. Thus, it is possible to perform an independent search in one dimension first and then find an associated solution in the other one. Our approach first

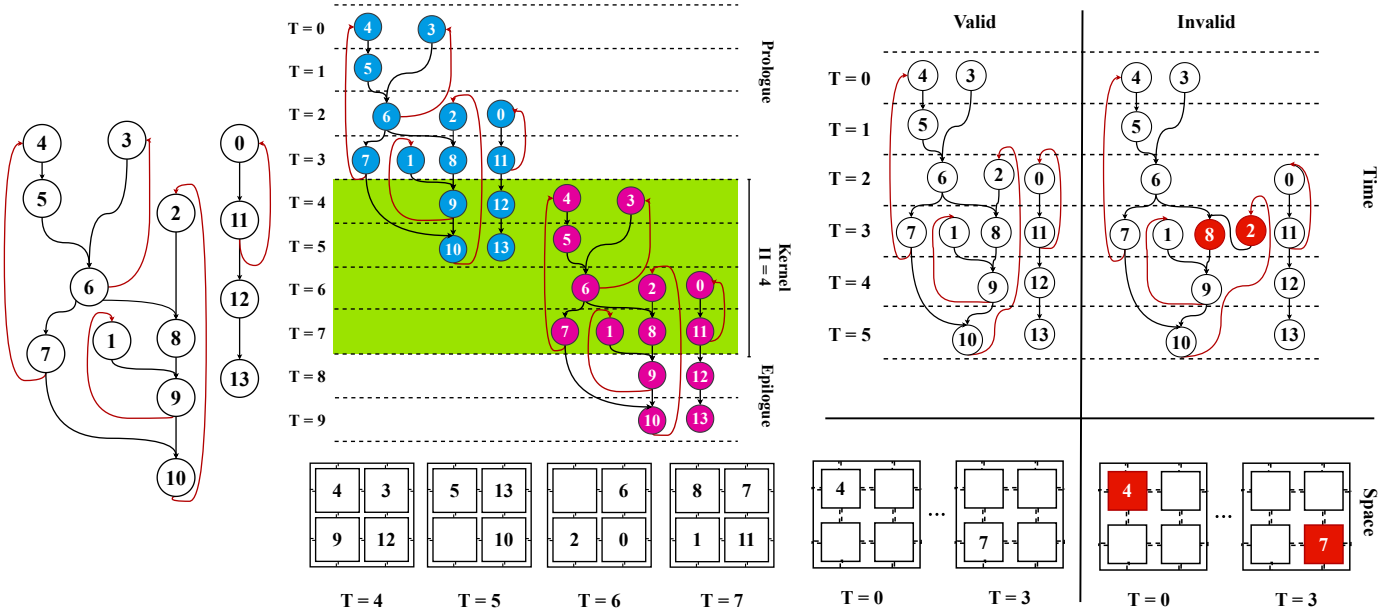

Fig. 2: Running example. a) A DFG, where black edges are data dependencies and red edges are loop-carried dependencies. b) Mapping of the DFG onto a 2×2 CGRA, on the bottom, with the division between prologue, kernel, and epilogue highlighted. c) Valid time and space solutions on the left, invalid solutions on the right; the erroneous allocations are shown in red.

finds a solution in time, via a modified version of SAT-MapIt's formulation [22], and then finds a solution in space, using monomorphism [29], [30]. We start by defining the compilation components, in Section IV-A, and then describe how a solution in time can be found, in Section IV-B, and how this solution can be extended to a solution in space, in Section IV-C, lastly, in Section IV-D, we provide a proof that a time solution under our constraints always implies the existence of a space solution.

A. Formal Definitions

We define a DFG as an undirected graph $\mathcal{G} = (\mathcal{V}_\mathcal{G}, \mathcal{E}_\mathcal{G}, l_\mathcal{G})$, where $\mathcal{V}_\mathcal{G}$ is the set of vertices, $\mathcal{E}_\mathcal{G} \subseteq \{\{u, v\} : u, v \in \mathcal{V}_\mathcal{G}\}$ is the set of edges, and $l_\mathcal{G} : \mathcal{V}_\mathcal{G} \to \mathcal{L}$ is a labeling function with $\mathcal{L} = \{0, ..., II - 1\}$ and $II \in \mathbb{N}^+$. This is the structure that we wish to compile to the CGRA.

To represent the allocation of instructions to the CGRA through time, we need to define an MRRG, whose structure consists of II stacked copies of the CGRA architecture linked via all valid routes allowed by the CGRA's topology. Fig. 3 shows an example of MRRG for a 2×2 CGRA with $II = 4$. Each one of the II architecture copies is defined as $\mathcal{M}^i = (\mathcal{V}_{\mathcal{M}^i}, \mathcal{E}_{\mathcal{M}^i})$, where $\mathcal{V}_{\mathcal{M}^i}$ is the set of vertices representing the CGRA's PEs at time step i and $\mathcal{E}_{\mathcal{M}^i} \subseteq \{\{u, v\} : u, v \in \mathcal{V}^i_\mathcal{M}\}$ is the set of edges representing the interconnections among the PEs of the CGRA. In Fig. 3, the vertices at $T = 0$ would be in the set $\mathcal{M}^0$ and all the black edges at that time step would be in $\mathcal{E}_{\mathcal{M}^0}$.

We define the MRRG as $\mathcal{M} = (\mathcal{V}_\mathcal{M}, \mathcal{E}_\mathcal{M}, l_\mathcal{M})$, where $\mathcal{V}_\mathcal{M}$ has the CGRA's PEs for all available time steps and $\mathcal{E}_\mathcal{M}$ has all interconnections between PEs at every time step, as follows:

$$\mathcal{V}_\mathcal{M} = \bigcup_{i \in \mathcal{L}} \mathcal{V}_{\mathcal{M}^i} \quad \mathcal{E}_\mathcal{M} = \bigcup_{i \in \mathcal{L}} \mathcal{E}_{\mathcal{M}^i} \cup \bigcup_{i,j \in \mathcal{L}} \left\{ e : \wedge \begin{array}{c} e \in \mathcal{E}_{\mathcal{M}^i, \mathcal{M}^j} \\ i = j - 1 \end{array} \right\}$$

with $\mathcal{E}_{\mathcal{M}^i, \mathcal{M}^j} \subseteq \{\{u, v\} : u \in \mathcal{V}_{\mathcal{M}^i}, v \in \mathcal{V}_{\mathcal{M}^j}\}$. The set $\mathcal{E}_{\mathcal{M}^0, \mathcal{M}^1}$ is partially depicted in green in Fig. 3 for the vertex associate to PE0. The function $l_\mathcal{M} : \mathcal{V}_\mathcal{M} \to \mathcal{L}$ assigns a fixed label to every vertex, reflecting the fact that $\mathcal{V}_{\mathcal{M}^i}$ represents the CGRA architecture at time step i. In Fig. 3 all vertices at $T = 0$ have label 0, while the ones at $T = 1$ have label 1, and so on. Concretely, we have that $\forall i \in \mathcal{L} \,.\, \forall v \in \mathcal{V}_{\mathcal{M}^i} \,.\, l_\mathcal{M}(v) = i$.

A property of interest for compilation is the connectivity degree of vertices in the MRRG. For every vertex $v \in \mathcal{V}_{\mathcal{M}^i}$ and time step $i \in \mathcal{L}$, we define the connectivity degree of v as $\mathcal{D}^v_{\mathcal{M}^i} = |\{\{u, v\} \in \mathcal{E}_{\mathcal{M}^i}\}|$. Since all the vertices of $\mathcal{M}$ have the same degree, we have $\forall i \in \mathcal{L} . \forall v \in \mathcal{V}_{\mathcal{M}^i} . \mathcal{D}_\mathcal{M} = \mathcal{D}_{\mathcal{M}^i} = \mathcal{D}^v_{\mathcal{M}^i}$. An MRRG with connectivity degree 3 can be seen in Fig. 3; self-loops are in the MRRG but are omitted for clarity.

The final element we need to define is monomorphism, which allows us to find a solution in space given a solution in time. A monomorphism from a graph $\mathcal{G}$ to a graph $\mathcal{M}$ is a function $f : \mathcal{V}_\mathcal{G} \to \mathcal{V}_\mathcal{M}$ that respects three properties:

$$f \text{ is injective} \qquad \text{(mono1)}$$
$$\forall v \in \mathcal{V}_\mathcal{G} \,.\, l_\mathcal{G}(v) = l_\mathcal{M}(f(v)) \qquad \text{(mono2)}$$
$$\forall \{u, v\} \in \mathcal{E}_\mathcal{G} \,.\, \{f(u), f(v)\} \in \mathcal{E}_\mathcal{M} \qquad \text{(mono3)}$$

Property mono1 ensures that one vertex of the MRRG is the target of at most one vertex of the DFG, since one PE can execute only one operation at any given time step. Property mono2

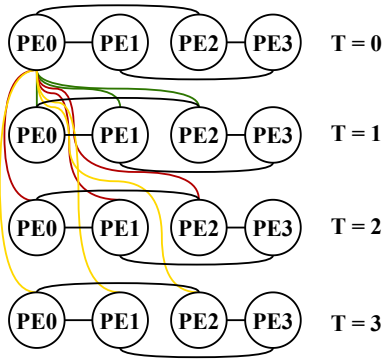

Fig. 3: MRRG for a 2×2 CGRA and $II = 4$. Black edges represent CGRA adjacencies, while green, red, and yellow edges represent time adjacencies from PE0 at $T = 0$. Time adjacencies of the other PEs, as well as the self-loops inherent to every PE, are omitted for clarity.

ensures that every vertex is executed at the correct time step. Property mono3 ensures that an MRRG edge satisfies the data dependencies. An example can be seen in Fig. 4.

B. Time Solution

In our formulation, we define the DFG as an undirected graph. This is possible because we initially find a schedule with the directed version of the DFG. Once a schedule is found, each vertex is labeled with its respective timing information and the directionality of the edges becomes redundant and is removed.

To find a suitable time schedule, we start by creating the as soon as possible (ASAP) and as late as possible (ALAP) schedules for the input DFG, to derive the range of possible scheduling time steps of every vertex. This is used in the Mobility Schedule (MobS), which expresses the mobility of each vertex. Tab. I shows these schedules for the DFG in Fig. 2.

Similar to the approach of Tirelli et al. [22], the MobS is the base structure used, together with the II, to create the Kernel Mobility Schedule (KMS) needed to formulate the scheduling problem. The KMS is the *superset of all possible schedules* for a given II, and is the result of iteratively folding the MobS by an amount equal to II. After each folding happens, every vertex copied into the KMS is assigned a label that refers to the iteration number. The number of loop iterations interleaved and executed at the same time is $\lceil \frac{MobS_{length}}{II} \rceil$. In our example, we have $\lceil 6/4 \rceil = 2$, thus every execution of the kernel computes

TABLE I: ASAP, ALAP, and MobS for the DFG in Fig. 2

	Nodes		
Time	**ASAP**	**ALAP**	**MobS**
0	0 1 2 3 4	4	0 1 2 3 4
1	5 11	3 5	0 1 2 3 5 11
2	6 12	0 2 6	0 1 2 6 11 12
3	7 8 13	1 8 11	1 7 8 11 12 13
4	9	7 9 12	7 9 12 13
5	10	10 13	10 13

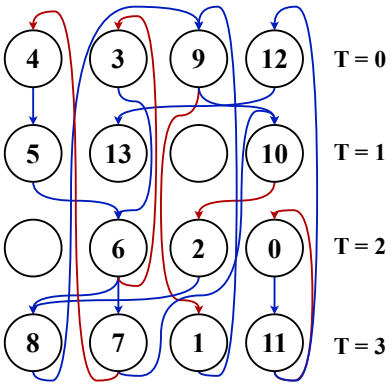

Fig. 4: Monomorphism between the DFG shown in Fig. 2 and the MRRG shown in Fig. 3; data dependencies are in blue and loop-carried dependencies routed in the MRRG are in red.

data from two different loop iterations. Tab. II shows the KMS for the DFG in Fig. 2.

The search starts from the mII, defined in [28] as follows:

$$mII = max(ResII, RecII)$$

$$ResII = \left\lceil \frac{|\mathcal{V}_\mathcal{G}|}{|\mathcal{V}_{\mathcal{M}^1}|} \right\rceil \qquad RecII = \max_{l \in DFG} \left\lceil \frac{length(l)}{distance(l)} \right\rceil$$

where $ResII$ represents the minimum number of resources required by the architecture to execute all the instructions in the DFG, and $RecII$ represents the longest cycle length in the DFG. In our example, we have $ResII = \lceil \frac{14}{2 \cdot 2} \rceil = 4$ and $RecII = 4$. Consequentially, $mII = max(4, 4) = 4$ is our starting point, meaning that no solution below $II = 4$ is valid.

Inspired by the idea used in [22] to express the solution space of the mapping problem, we design our own form of expressing the solution space which focus solely on the time dimension. We use the KMS to describe the solution space of the time dimension as a Satisfiability Modulo Theories (SMT) formula and include additional constraints, not present in [22], to ensure that the schedule obtained in our search leads to a monomorphism of the DFG in the MRRG. The constraints of our formulation are divided in three sets: modulo scheduling, capacity, and connectivity; the latter two are our additions.

1) Modulo scheduling constraints: The labeling of the vertices corresponds to their scheduling. With the following constraints, we ensure that the correct order of execution of all the DFG vertices is respected. Data dependencies and loop-carried dependencies are respectively encoded as follows:

$$\begin{cases} t_d \leq t_s \text{ if } it_s - it_d = 1 \\ t_d > t_s \text{ if } it_s = it_d \end{cases} \qquad \begin{cases} t_d \leq t_s \text{ if } it_s = it_d \\ t_d > t_s \text{ if } it_s - it_d = 1 \end{cases}$$

TABLE II: KMS for the MobS in Tab. I and an II of 4.

Time	**Nodes**	
0	$0_0 1_0 2_0 6_0 11_0 12_0$	
1	$1_0 7_0 8_0 11_0 12_0 13_0$	
2	$7_0 9_0 12_0 13_0$	$0_1 1_1 2_1 3_1 4_1$
3	$10_0 13_0$	$0_1 1_1 2_1 3_1 5_1 11_1$

where t_d and t_s are the times at which the destination and source vertices can be scheduled, and it_s and it_d refer to the subscripts in the KMS that indicate how many foldings have been done on the MobS. In the KMS there are many possibilities, not all of them are valid. With our SMT formulation, we can explore the scheduling space to find all valid solutions.

2) Capacity constraints: These constraints ensure that, once a valid schedule is found in the time dimension, it can be effectively mapped onto the physical resources of the CGRA. Concretely, we need to ensure that the capacity of the CGRA is not exceeded at any time step. Thus, we require that $\forall i \in \mathcal{L} \,.\, C_i \leq |\mathcal{V}_{\mathcal{M}^i}|$, where $C_i = |\{u \in \mathcal{V}_\mathcal{G} : l_\mathcal{G}(u) = i\}|$.

3) Connectivity constraints: We also need to ensure that the connectivity degree of the PEs is never exceeded. Let S_v^i be the set of neighbors of $v \in \mathcal{V}_\mathcal{G}$ at time step $i \in \mathcal{L}$, defined as $S_v^i = \{\{u, v\} \in \mathcal{E}_\mathcal{G} : l_\mathcal{G}(u) = i\}$. Having $\mathcal{D}_\mathcal{M}$ as the CGRA connectivity degree, e.g., $\mathcal{D}_\mathcal{M} = 3$ in a 2×2 architecture and $\mathcal{D}_\mathcal{M} = 5$ in 3×3 and larger architectures, we must require that $\forall v \in \mathcal{V}_\mathcal{G} \,.\, \forall i \in \mathcal{L} \,.\, |S_v^i| \leq \mathcal{D}_\mathcal{M}$. This allows for the correct routing of resources for every PE, since the number of successors of a DFG vertex scheduled on the same time step will be at most the number of neighbors of a PE.

C. Space Solution

Many techniques found in the literature can be reused to navigate the spatial dimension. We, however, follow a novel approach based on monomorphism extraction. Algorithms for such extraction [29], [30] have demonstrated their ability to handle large graphs with low computational overhead, which is corroborated by our experimental results.

D. Proof of Monomorphism Existence

We want to prove that, for all $\mathcal{G}$ and $\mathcal{M}$ satisfying the above constraints, there exists a monomorphism f from $\mathcal{G}$ to $\mathcal{M}$. Concretely, we need to show that there exists a function that respects properties mono1, mono2, and mono3.

1) Function f is injective: To establish that there exists an injective interpretation for the function $f : \mathcal{V}_\mathcal{G} \to \mathcal{V}_\mathcal{M}$ we must show that its codomain it at least as large as its domain. This is guaranteed by two following disequalities:

$$|\mathcal{V}_\mathcal{G}| = \sum_{i \in \mathcal{L}} C_i \leq \sum_{i \in \mathcal{L}} |\mathcal{V}_{\mathcal{M}^i}| = |\mathcal{V}_\mathcal{M}| \qquad \forall i \in \mathcal{L} \,.\, C_i \leq |\mathcal{V}_{\mathcal{M}^i}|$$

2) Function f maps vertices: To establish that a vertex mapping is possible we must show that the number of vertices labeled with a specific $i \in \mathcal{L}$, i.e., C_i, is at most the capacity of the CGRA, i.e., $|\mathcal{V}_{\mathcal{M}^i}|$. This is guaranteed by the capacity constraints described in Section IV-B2.

3) Function f maps edges: To establish that an edge mapping is possible we must show that the cardinality of the set of neighbors of every vertex $v \in \mathcal{V}_\mathcal{G}$ labeled with a specific $i \in \mathcal{L}$, i.e., $|S_v^i|$, is at most the connectivity degree of the CGRA, i.e., $\mathcal{D}_\mathcal{M}$. This is guaranteed by the connectivity constraints described in Section IV-B3.

V. Experiments

We evaluated our methodology on four different CGRA configurations: 2×2, 5×5, 10×10, and 20×20. Each PE register file in the CGRA could be accessed by neighboring PEs. We choose all the innermost loops from the MiBench [31] and Rodina [32] benchmarks suites, excluding those with function calls or conditional statements inside the loop body, resulting in a total of 17 benchmarks. We compared our approach against that of [22], since it is compatible with our target architecture and was shown to consistently yield better results than other comparable state-of-the-art methodologies. The metrics chosen for our evaluation were II value and compilation time. We used the Z3 solver [33] to solve the SMT formulas generated. All experiments were performed on a Linux Machine with a 3.30 GHz Intel Core i9 CPU and 256 GB of RAM.

1) Iteration Interval comparison: Tab. III shows the II values obtained in our experiments, with solutions being found by our methodology in 62 out of 68 cases. Our approach achieves the same II as [22] in 57 cases. In 5 cases, we find a solution, but the tool from [22] times out, which demonstrates our ability to obtain quality mappings. In only one case, out of all our 68 experiments, our space search reaches a timeout but the compared tool does not.

2) Compilation time comparison: Tab. III also shows the compilation times obtained in our experiments. For all benchmarks, we observe a significant reduction in compilation time, with average speedups of $30.85\times$, $103.76\times$, $887.84\times$, and $10288.89\times$, for 2×2, 5×5, 10×10 and 20×20 CGRAs sizes. For an accurate comparison, we excluded from the average speedup calculation the cases for which one of the tools had a timeout. These results show how scalability is enhanced with our approach. The speedup of $10288.89\times$ stands out, with [22] having to search across the entire mapping space, while our methodology benefits from its compositionality. By examining individual benchmarks, we find that our approach achieves a compilation speedup in almost every cases, with the exception of one case of a timeout and two cases of minimal difference.

Scalability is heavily influenced by the number of PEs in the grid, even for DFGs with as few as 23 nodes, e.g., aes benchmark. Fig. 5 shows how CGRA size impacts each approach. While the compilation times of SAT-MapIt [22] increase with the CGRA size, the compilation times of our

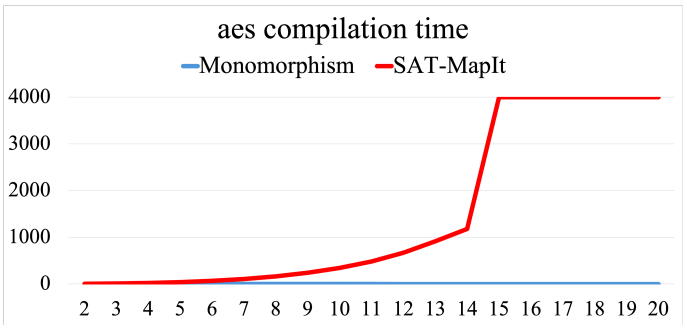

Fig. 5: Compilation time (y-axis), in seconds, in relation to CGRA sizes (x-axis) for the aes benchmark.

Benchmark	DFG Nodes	2 × 2 CGRA							5 × 5 CGRA						
		Compilation Time			ΔT	CTR	*II*	*mII*	Compilation Time			ΔT	CTR	*II*	*mII*
		Monomorphism		SAT-MapIt					Monomorphism		SAT-MapIt				
		Time	Space						Time	Space					
aes	23	0.40	0.02	2.57	-2.15	6.12	16	14	0.47	0.04	39.07	-38.56	76.16	16	14
backprop	34	0.44	0.03	110.01	-109.54	233.07	10	9	0.12	0.29	9.98	-9.56	23.82	5	5
basicmath	21	0.32	0.11	0.42	0.01	0.98	7	7	0.13	0.31	7.82	-7.38	17.77	7	7
bitcount	7	0.038	∼0.01	0.06	-0.02	1.73	3	3	0.39	∼0.01	1.15	-0.76	2.95	3	3
cfd	51	TO	-	TO	-	-	-	13	0.07	TO	23.59	-	-	3	3
crc32	24	0.20	∼0.01	3.85	-3.64	18.34	11	8	0.30	∼0.01	75.75	-75.45	250.00	11	8
fft	20	0.09	∼0.01	0.46	-0.37	5.05	7	7	0.14	∼0.01	8.22	-8.08	57.89	7	7
gsm	24	0.06	∼0.01	0.43	-0.36	6.30	6	6	0.11	∼0.01	15.49	-15.36	122.94	5	4
heartwall	35	0.14	∼0.01	1.31	-1.17	9.21	9	9	0.16	∼0.01	45.18	-45.01	272.17	3	3
hotspot3D	57	1.13	0.09	223.51	-222.29	182.46	17	15	0.54	0.01	209.87	-209.32	378.14	6	3
lud	26	0.07	∼0.01	0.45	-0.37	5.54	7	7	0.07	∼0.01	7.95	-7.88	107.72	3	3
nw	33	0.18	∼0.01	2.48	-2.29	13.03	9	9	0.05	1.16	5.39	-4.17	4.43	2	2
particlefilter	38	0.12	∼0.01	1.67	-1.55	13.47	10	10	0.34	∼0.01	28.08	-27.73	81.16	9	9
sha1	21	0.05	0.43	0.27	0.21	0.56	6	6	0.11	0.09	15.44	-15.24	77.20	4	2
sha2	25	0.07	∼0.01	0.60	-0.52	7.29	7	6	0.16	4.07	9.22	-4.99	2.18	7	7
stringsearch	28	0.10	∼0.01	1.04	-0.94	9.90	7	7	0.10	1.09	17.01	-15.82	14.29	3	3
susan	21	0.09	∼0.01	0.97	-0.88	10.34	6	6	0.08	∼0.01	15.94	-15.85	171.40	2	2
Average	-	0.22	0.042	21.88	**-21.61**	**30.85**	-	-	0.20	0.44	31.97	**-31.32**	**103.76**	-	-

Benchmark	DFG Nodes	10 × 10 CGRA							20 × 20 CGRA						
		Compilation Time			ΔT	CTR	*II*	*mII*	Compilation Time			ΔT	CTR	*II*	*mII*
		Monomorphism		SAT-MapIt					Monomorphism		SAT-MapIt				
		Time	Space						Time	Space					
aes	23	0.48	∼0.01	342.11	-341.63	705.38	16	14	0.48	0.013	TO	-	-	16	14
backprop	34	0.13	0.11	112.80	-112.56	470.00	5	5	0.14	0.024	TO	-	-	5	5
basicmath	21	0.14	∼0.01	102.83	-102.69	711.63	7	7	0.19	0.086	1362.58	-1362.30	4936.88	7	7
bitcount	7	0.039	∼0.01	14.73	-14.69	371.97	3	3	0.062	∼0.01	223.88	-223.82	3492.67	3	3
cfd	51	0.12	TO	TO	-	-	-	2	0.14	TO	TO	-	-	-	2
crc32	24	0.31	∼0.01	262.82	-262.51	834.88	11	8	0.33	0.012	3867.11	-3866.77	11307.34	11	8
fft	20	0.14	∼0.01	101.34	-101.20	711.66	7	7	0.23	∼0.01	1485.63	-1485.39	6289.71	7	7
gsm	24	0.11	∼0.01	191.03	-190.91	1603.95	5	4	0.14	∼0.01	2799.07	-2798.71	18722.88	5	4
heartwall	35	0.17	∼0.01	571.87	-571.69	3124.97	3	3	0.28	∼0.01	TO	-	-	3	3
hotspot3D	57	0.71	TO	TO	-	-	-	2	0.83	TO	TO	-	-	-	2
lud	26	0.08	∼0.01	89.75	-89.66	1048.48	3	3	0.086	∼0.01	1321.66	-1321.56	13216.60	3	3
nw	33	0.06	10.25	61.55	-51.23	5.97	2	2	0.068	0.15	981.69	-981.47	4503.17	2	2
particlefilter	38	0.37	70.34	451.48	-380.77	6.38	9	9	0.37	141.54	TO	-	-	9	9
sha1	21	0.14	0.03	195.86	-195.69	1119.20	4	2	0.12	0.036	TO	-	-	4	2
sha2	25	0.17	10.21	107.51	-97.13	10.36	7	7	0.17	2.02	1585.18	-1582.99	723.83	7	7
stringsearch	28	0.11	0.73	203.88	-203.04	242.71	3	3	0.11	0.61	3108.92	-3108.20	4317.94	3	3
susan	21	0.09	∼0.01	213.63	-213.54	2350.17	2	2	0.09	∼0.01	3314.91	-3314.82	35377.91	2	2
Average	-	0.17	6.11	201.54	**-195.26**	**887.84**	-	-	0.14	0.29	2006.06	**-2004.62**	**10288.89**	-	-

TABLE III: Experimental results for four CGRAs; ΔT and CTR (Compilation Time Ratio) are the difference and ratio of compilation time between the compared approaches. Each experiment had a 4000 seconds timeout. The average excludes benchmarks for which one of the tools had a timeout.

approach remain consistently small regardless of the grid size. This speedup comes from our decoupling of time and space, which enables independent searches over each dimension that are smaller and thus easier to navigate than the full mapping space. Furthermore, the search over the spatial dimension is accelerated by leveraging information obtained from the temporal solution.

3) Limitations of our work: Currently, our approach only targets architectures in which every PE can read the internal register of neighboring PEs. This makes the decoupling of space and time easier, but increases the complexity of the hardware design. Future work will focus on overcoming this restriction.

VI. Conclusion

We propose a scalable CGRA mapping approach that effectively decouples the space and time dimensions and explores them in isolation. It first finds a time solution suitable for the target CGRA, by utilizing an SMT formulation and solver, and then explores the spatial dimension to find a space solution by utilizing a monomorphism search. Our experimental results showcase that our approach suffers no reduction in the quality of results achieved, finding mappings of quality similar to those found by state-of-the-art methods, while providing significantly better scalability, with an average speedup of $10288.89\times$ when compiling for a 20×20 CGRA.

REFERENCES

[1] T.-J. Lin, W. Zhang, and N. K. Jha, "A Fine-Grain Dynamically Reconfigurable Architecture Aimed at Reducing the FPGA-ASIC Gaps," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 12, pp. 2607–2620, 2014.

[2] I. Kuon and J. Rose, "Measuring the Gap Between FPGAs and ASICs," in *Proceedings of the 14th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2006, pp. 21–30.

[3] Z. Li, D. Wijerathne, X. Chen, A. Pathania, and T. Mitra, "ChordMap: Automated Mapping of Streaming Applications onto CGRA," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 2, pp. 306–319, 2022.

[4] M. Karunaratne, A. K. Mohite, T. Mitra, and L.-S. Peh, "HyCUBE: A CGRA with Reconfigurable Single-Cycle Multi-Hop Interconnect," in *Proceedings of the 54th ACM/IEEE Design Automation Conference*, 2017, pp. 1–6.

[5] O. Akbari, M. Kamal, A. Afzali-Kusha, M. Pedram, and M. Shafique, "PX-CGRA: Polymorphic Approximate Coarse-Grained Reconfigurable Architecture," in *Proceedings of the 21st Design, Automation and Test in Europe Conference*, 2018, pp. 413–418.

[6] T. Oh, B. Egger, H. Park, and S. Mahlke, "Recurrence Cycle Aware Modulo Scheduling for Coarse-Grained Reconfigurable Architectures," in *Proceedings of the 10th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, 2009, pp. 21–30.

[7] H. Lee, D. Nguyen, and J. Lee, "Optimizing Stream Program Performance on CGRA-Based Systems," in *Proceedings of the 52nd ACM/IEEE Design Automation Conference*, 2015, pp. 1–6.

[8] D. Wijerathne, Z. Li, M. Karunarathne, A. Pathania, and T. Mitra, "CASCADE: High Throughput Data Streaming via Decoupled Access-Execute CGRA," *ACM Transactions on Embedded Computing Systems*, vol. 18, no. 5s, pp. 1–26, 2019.

[9] L. Duch, S. Basu, R. Braojos, G. Ansaloni, L. Pozzi, and D. Atienza, "HEAL-WEAR: An Ultra-Low Power Heterogeneous System for Bio-Signal Analysis," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 64, no. 9, pp. 2448–2461, 2017.

[10] A. Podobas, K. Sano, and S. Matsuoka, "A Survey on Coarse-Grained Reconfigurable Architectures from a Performance Perspective," *IEEE Access*, vol. 8, pp. 146 719–146 743, 2020.

[11] B. Mei, M. Berekovic, and J.-Y. Mignolet, "ADRES & DRESC: Architecture and Compiler for Coarse-Grain Reconfigurable Processors," in *Fine- and Coarse-Grain Reconfigurable Computing*. Springer, 2007, pp. 255–297.

[12] H. Park, K. Fan, S. A. Mahlke, T. Oh, H. Kim, and H.-S. Kim, "Edge-Centric Modulo Scheduling for Coarse-Grained Reconfigurable Architectures," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, 2008, pp. 166–176.

[13] M. Hamzeh, A. Shrivastava, and S. Vrudhula, "EPIMap: Using Epimorphism to Map Applications on CGRAs," in *Proceedings of the 49th ACM/IEEE Design Automation Conference*, 2012, pp. 1284–1291.

[14] H. Park, K. Fan, M. Kudlur, and S. Mahlke, "Modulo Graph Embedding: Mapping Applications onto Coarse-Grained Reconfigurable Architectures," in *Proceedings of the 2006 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2006, pp. 136–146.

[15] L. Chen and T. Mitra, "Graph Minor Approach for Application Mapping on CGRAs," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 7, no. 3, pp. 1–25, 2014.

[16] M. Hamzeh, A. Shrivastava, and S. Vrudhula, "REGIMap: Register-Aware Application Mapping on Coarse-Grained Reconfigurable Architectures (CGRAs)," in *Proceedings of the 50th ACM/IEEE Design Automation Conference*, 2013, pp. 1–10.

[17] S. Dave, M. Balasubramanian, and A. Shrivastava, "RAMP: Resource-Aware Mapping for CGRAs," in *Proceedings of the 55th ACM/IEEE Design Automation Conference*, 2018, pp. 1–6.

[18] M. Balasubramanian and A. Shrivastava, "CRIMSON: Compute-Intensive Loop Acceleration by Randomized Iterative Modulo Scheduling and Optimized Mapping on CGRAs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 3300–3310, 2020.

[19] ——, "PathSeeker: A Fast Mapping Algorithm for CGRAs," *Proceedings of the 25th Design, Automation and Test in Europe Conference*, 2022.

[20] 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*, 2018, pp. 1–6.

[21] Y. Miyasaka, M. Fujita, A. Mishchenko, and J. Wawrzynek, "SAT-Based Mapping of Data-Flow Graphs onto Coarse-Grained Reconfigurable Arrays," in *Proceedings of the 28th IFIP/IEEE International Conference on Very Large Scale Integration - Systems-on-Chip*, 2020, pp. 113–131.

[22] C. Tirelli, L. Ferretti, and L. Pozzi, "SAT-MapIt: A SAT-based Modulo Scheduling Mapper for Coarse Grain Reconfigurable Architectures," in *Proceedings of the 26th Design, Automation and Test in Europe Conference*, 2023, pp. 1–6.

[23] C. Tirelli, J. Sapriza, R. Rodríguez Álvarez, L. Ferretti, B. Denkinger, G. Ansaloni, J. M. Calero, D. Atienza, and L. Pozzi, "SAT-based Exact Modulo Scheduling Mapping for Resource-Constrained CGRAs," *Journal on Emerging Technologies in Computing Systems*, vol. 20, no. 3, 2024.

[24] Z. Zhao, W. Sheng, Q. Wang, W. Yin, P. Ye, J. Li, and Z. Mao, "Towards Higher Performance and Robust Compilation for CGRA Modulo Scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 9, pp. 2201–2219, 2020.

[25] D. Wijerathne, Z. Li, A. Pathania, T. Mitra, and L. Thiele, "HiMap: Fast and Scalable High-Quality Mapping on CGRA via Hierarchical Abstraction," in *Proccedings of the 24th Design, Automation and Test in Europe Conference*, 2021, pp. 3290–3303.

[26] G. Zacharopoulos, L. Ferretti, E. Giaquinta, G. Ansaloni, and L. Pozzi, "RegionSeeker: Automatically Identifying and Selecting Accelerators from Application Source Code," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 4, pp. 741–754, 2018.

[27] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2nd ACM/IEEE International Symposium on Code Generation and Optimization*, 2004, pp. 75–86.

[28] B. R. Rau, "Iterative Modulo Scheduling," *International Journal of Parallel Programming*, vol. 24, no. 1, pp. 3–64, 1996.

[29] V. Bonnici, R. Giugno, A. Pulvirenti, D. Shasha, and A. Ferro, "A Subgraph Isomorphism Algorithm and its Application to Biochemical Data," *BMC bioinformatics*, vol. 14, no. S-7, pp. 1–13, 2013.

[30] V. Carletti, P. Foggia, A. Saggese, and M. Vento, "Challenging the Time Complexity of Exact Subgraph Isomorphism for Huge and Dense Graphs with VF3," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 40, no. 4, pp. 804–818, 2017.

[31] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," in *Proceedings of the 4th IEEE International Workshop on Workload Characterization*, 2001, pp. 3–14.

[32] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization*, 2009, pp. 44–54.

[33] L. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008, pp. 337–340.