EPIMap: Using Epimorphism to Map Applications on CGRAs

Mahdi Hamzeh, Aviral Shrivastava, and Sarma Vrudhula
School of Computing, Informatics, and Decision Systems Engineering
Arizona State University, Tempe, AZ
{mahdi, aviral.shrivastava, vrudhula}@asu.edu

ABSTRACT

Coarse-Grained Reconfigurable Architectures (CGRAs) are an attractive platform that promise simultaneous high-performance and high power-efficiency. One of the primary challenges in using CGRAs is to develop efficient compilers that can automatically and efficiently map applications to the CGRA. To this end, this paper makes several contributions: i) *Using Re-computation for Resource Limitations:* For the first time in CGRA compilers, we propose the use of re-computation as a solution for resource limitation problem. This extends the solutions space, and enables better mappings, ii) *General Problem Formulation:* A precise and general formulation of the application mapping problem on a CGRA is presented, and its computational complexity is established. iii) *Extracting an Efficient Heuristic:* Using the insights from the problem formulation, we design an effective global heuristic called EPIMap. EPIMap transforms the input specification (a directed graph) to an Epimorphic equivalent graph that satisfies the necessary conditions for mapping on to a CGRA, reducing the search space. Experimental results on 14 important kernels extracted from well known benchmark programs show that using EPIMap can improve the performance of the kernels on CGRA by more than 2.8X on average, as compared to one of the best existing mapping algorithm, EMS. EPIMap was able to achieve the theoretical best performance for 9 out of 14 benchmarks, while EMS could not achieve the theoretical best performance for any of the benchmarks. EPIMap achieves better mappings at acceptable increase in the compilation time.

Categories and Subject Descriptors

C.3 [**SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS**]: [Real-time and embedded systems]; D.3.4 [**Processors**]: [Code generation, Compilers, Optimization]

General Terms

Algorithms, Design, Performance

Keywords

Coarse-Grained Reconfigurable Architectures, Compilation, Modulo Scheduling

1 Introduction

The fundamental challenge faced by all segments of the microelectronics industry is the simultaneous demand for high-performance and higher power-efficiency. The next generation ultra high definition TVs and software defined radios require performance and power efficiencies of tens of Giga (10^9) operations per second per watt (Gops/W) [25]. On the other hand, experts believe that for Exascale computing (10^{18} ops/s = 10^9 Gops) to be really practical, power efficiencies of at least hundreds of Gops/W are necessary. Even under the most aggressive scaling scenarios and the most optimistic assumptions on the impact of high clock frequencies on power and thermal characteristics, fundamental innovations at the architecture level are also needed [5].

At the architecture level, accelerators are a promising approach to improve both the performance and power-efficiency of execution. Although special purpose or function specific hardware accelerators (e.g. for FFT) can be very power efficient, they are expensive, not programmable, and therefore limited in usage. Graphics Processing Units (GPUs) are becoming very popular; although programmable, they are limited to accelerating only "parallel loops." Field Programmable Gate Arrays are general-purpose, but they lose a lot of power-efficiency in managing the fine-grain reconfigurability they provide. Coarse-Grained Reconfigurable Architectures or CGRAs have been shown to be an excellent alternative as they not only have power efficiencies close to hardware accelerators, but can be utilized for a wide range of applications because they are *programmable*. For instance, the ADRES CGRA has been shown to achieve performance and power efficiency of 60 GOPS/W in 32 nm CMOS technology [6]. A catalog of existing CGRA designs and architectures is given in [13].

A CGRA is simply an array of processing elements (PEs) interconnected by a 2-D grid. A PE typically consists of an arithmetic logic unit (ALU) and a few registers. It is referred to as *coarse grained reconfigurable* because each PE can be programmed to execute different instructions at the cycle level granularity. CGRAs are completely statically scheduled. Computation is laid out on CGRA, with PEs operating on the output of their neighboring PEs. Very little power, other than the PE power is expended in performing an ALU operation; therefore CGRAs are very power-efficient.

Even though the possible performance and power-efficiency of a CGRA is very high, what can be achieved is critically limited by the compiler technology. A CGRA compiler is much more complex than a regular compiler, since in addition to the regular task of "expressing application in terms of machine instructions," a CGRA compiler must: i) perform explicit pipelining of operations (or software pipelining), ii) map operations to PEs, and iii) route data between PEs so as not to violate data dependencies. Since all of these are hard problems, existing CGRA compilers use "search

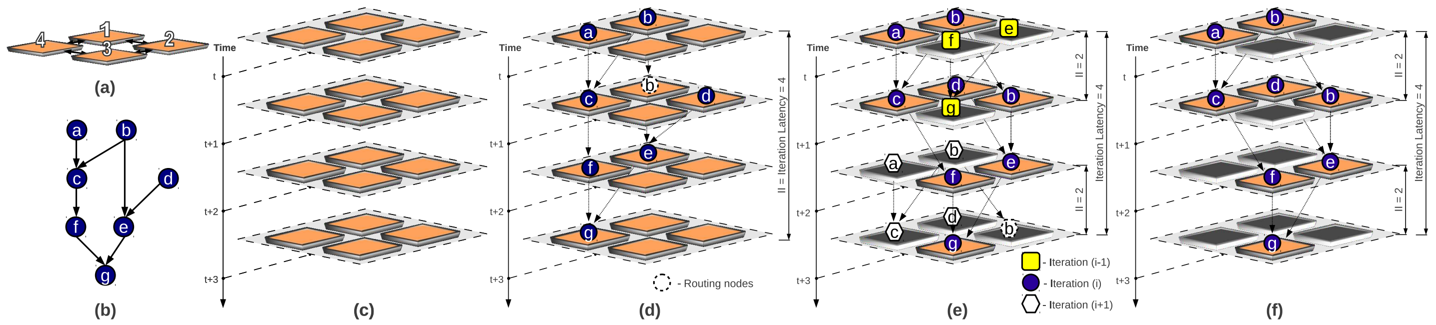


Figure 1: (a) a 2×2 CGRA, (b) an input DFG, (c) a time extended CGRA, (d) a valid mapping of the given DFG *(b)* on CGRA *(a)* with iteration latency $=II= 4$, (e) another mapping for the given DFG with iteration latency$= 4$ and $II = 2$, lower II is achieved because two iterations of the loops are executed simultaneously, Dark color PEs in *(e)* execute an operation of another iteration. (f) Only nodes from one iteration of the loop is shown for *(e)*. II is a key performance metric, with the lesser II the better throughput.

based" heuristics to map applications to the CGRA, but the quality of mapping is not good. Towards improving compilers for CGRA, this paper makes three important contributions:

1. Re-computation for limited resource problem: A major challenge to effective application mapping on CGRA is resource limitation. Traditional approach is to use routing to find a solution within the given resource limitation. However our formulation and method utilizes both routing and re-computation to find a solution within the resource limitation which often leads to better mappings.

2. General problem formulation and complexity analysis: We show that the mapping problem can be described as that of finding a subgraph in a minimally Time-Extended CGRA (TEC) graph that is *Epimorphic* to the input graph. This is important, because even though several application mapping heuristics exist for CGRA, the problem has not been formulated before, and we believe that this is the reason for the poor performance of existing heuristics.

3. Distilling an effective heuristic: Insights from the problem formulation enable us to identify the necessary conditions for a feasible solution, and distill an effective and theoretically justified heuristic, which we name EPIMap. EPIMap allows for a systematic search of the solution space which results in high quality mapping.

We compare the quality of mappings generated by EPIMap with the EMS [23] algorithm, which is one the best existing mapping algorithm both in terms of quality and compilation time [1]. Experimental results on 14 important kernels extracted from standard benchmarks, including SPEC2006 show that i) EPIMap generates mappings that have on average 2.85X better performance than EMS; ii) In 9 out of 14 benchmarks, EPIMap finds the optimum mapping while EMS could not find the optimum mapping for any of these benchmarks, and iii) the improvement in mapping quality comes at acceptable (one time) cost of increase in compile time.

2 Background and Related Works

A CGRA is a 2-D mesh of PEs, with each PE having an ALU and a register file (Figure 2). Each PE is connected to its neighbors, and the output of a PE at cycle t is accessible to its neighboring PEs in the next cycle. In addition, a common data bus from the data memory provides data to all the PEs in a row. The earlier work on CGRAs include XPP [4], PADDI [7], PipeRench [11], KressArray [14], Morphosys [18], MATRIX [21], and REMARC [22].

Since many applications spend most of the time on loops [24], this paper focuses on the problem of mapping the innermost loops on a CGRA. A single iteration of a loop is represented as a Data Flow Graph (DFG), which is a directed graph $D = (V_d, E_d)$ where nodes represent operations and arc $(a, b) \in E_d$ iff the output of operation a is an input of operation b. $(a, b) \in E_d$ implies that operation b can only be executed after operation a has been completed.

The inputs to the problem are a DFG of a loop (extracted by the compiler) and a $M \times N$ CGRA. The output is a valid mapping of the nodes in the DFG to the PEs in the CGRA. The goal is to minimize the total execution time of the entire loop. Figure 1 illustrates all the aspects of the problem. Figure 1(a) shows a 2×2 CGRA, and Figure 1(b) shows a DFG of a loop. To map the loop on the CGRA, first the CGRA must be extended in time. Figure 1(c) shows the CGRA extended 4 steps in time. The loop DFG must be mapped on TEC. Figure 1(d) shows a valid mapping of the loop DFG onto the TEC. The mapping is valid, because data dependencies between nodes are preserved. For example, node a at time t and c at time $t+1$ are mapped onto the PE_4. Since the result of node a is stored in PE_4, the output of a (PE_4 at time t) will be available for c (PE_4 at $t+1$). Nodes b and e are mapped on PE_1 at times t and $t+2$ respectively. The output of b must be retained in PE_1 until e is executed at $t+2$. Shown by a dashed node, this represents **routing** from PE_1 at time t to itself at time $t+2$.

It is important to note that the execution on the CGRA is pipelined. Pipelining essentially means that we can execute instructions of different iterations of the loop at the same time. This explicit pipelining of schedule is referred to as software pipelining and modulo scheduling [24] is one of the most popular techniques for the same. The performance metric here is *initiation interval* (II), rather than the schedule length. II is the number of cycles between the start of two consecutive iterations of a loop. Figure 1(e) shows another way to execute the same loop DFG with instructions from other iterations also executing simultaneously. The dark circle nodes in the diagram represent the operations of the i^{th} iteration, while the light square nodes represent the operations of the $(i-1)^{th}$ iteration, and the light hexagon nodes represent the operations of the $(i+1)^{th}$ iteration. Note that the schedule length is 4 cycles, but the II is only 2 cycles. This is because the dark circle nodes representing the i^{th} iteration span from time t to $t+3$, but the operations of the next iteration, i.e., the light hexagon nodes start from time $t+2$. This mapping of one iteration is then shown in Figure 1(f), where we have removed the nodes from other iterations for clarity and easier comparison with Figure 1(d). Note that PEs that are shaded

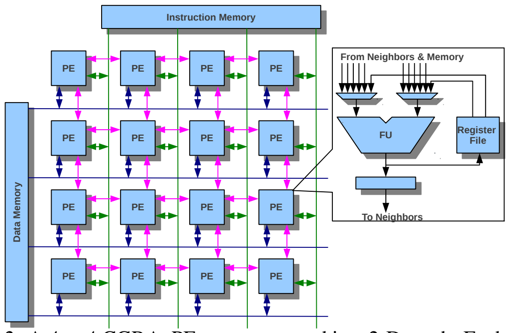


Figure 2: A 4×4 CGRA. PEs are connected in a 2-D mesh. Each PE is an ALU plus a local register file.

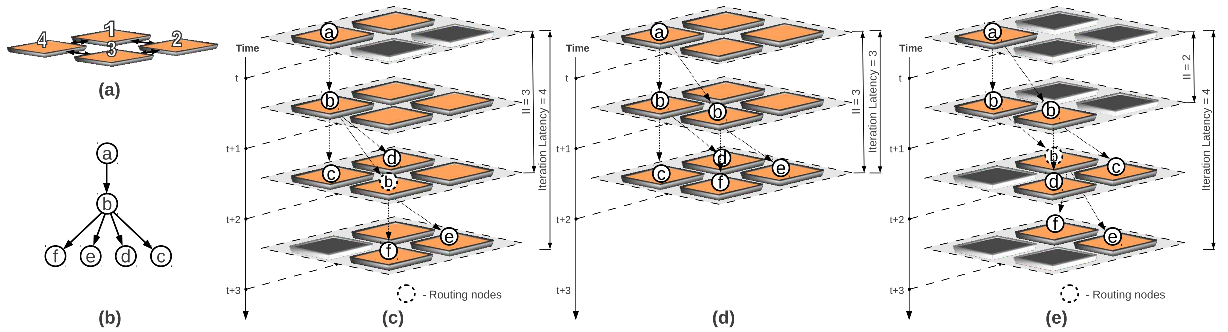


Figure 3: (a) a 2×2 CGRA, (b) input DFG where node b has out-degree of 4. (c) a valid mapping of the input DFG using routing, achieves $II = 3$. (d) shows the mapping of the input DFG using re-computation, achieves $II = 3$. (d) another mapping using both routing and re-computation, achieves $II = 2$.

black indicate their use in other iterations. As compared to Figure 1(d), since a new iteration starts every 2 cycles in Figure 1(f), the throughput and performance improve by a factor of 2.

This mapping and scheduling of the loop DFG on the CGRAs is done by the compiler. Recognizing the significant influence of the compiler on the achieved performance, much recent research has focused on efficient application mapping schemes [2, 3, 8, 9, 12, 15, 17, 20, 23, 26]. Each of these schemes impose their own restrictions and employ different, albeit intuitively justifiable, heuristic local decisions. For instance, many of these earlier works partition the problem in to three subproblems, namely, scheduling (when to perform an operation), placement (on which PE to perform an operation) and routing (how to route data between PEs), and solve these independently using a generic search method such as simulated annealing [15, 20], or use techniques developed in high-level hardware synthesis [9]. Since the key metric used in all the methods is II, the existing methods select an II, attempt to solve the three problems (possibly in different order), and if a feasible solution is not found, increase the II and repeat.

One of the major drawbacks of compiler research in this area is the lack of a precise and general formulation of the problem. Proper problem formulation allows us to systematically solve the mapping problem. Next, we first explain the concept of re-computation and why it is needed, then we show our problem formulation which allows for both *routing* and *re-computation*, and finally present our EPIMap heuristic before showing the experimental results.

3 What is Re-computation and Why?

Resource limitation is a major problem when trying to map a loop DFG onto a CGRA. One kind of resource limitation that is seen very often is the *out-degree problem*. Figure 3 illustrates the out-degree problem, and its solutions. The node b in the DFG in Figure 3(b) has an out-degree of 4 whereas the maximum out-degree of PEs in the TEC is 3 (if b is mapped on PE_4 at time $t+1$, then a dependent operations can only be mapped on PE_4, PE_1, or PE_3 at time $t+2$). Normally, if a node has out-degree greater than the maximum out-degree of the TEC, then it cannot be mapped onto the CGRA. In this example, only three of the dependent operations (among c, d, e, and f) can be mapped at time $t+2$, and the fourth will have to be mapped at time $t+3$; but if we do that, the fourth operation cannot receive the value of b.

Figure 3(c) shows how routing can be used to solve this out-degree problem. Operation b is performed on PE_4 at time $t+1$. At time $t+2$, PE_3, being adjacent to PE_4, copies the result of b to its output register. At time $t+3$, PE_3 uses its own output (which has b) to perform operation f, and PE_2, which is adjacent to PE_3, uses the result in PE_3's output register (result of b) to perform operation e. Thus PE_3 just routes the output of operation b at time $t+2$, to make it available at time $t+3$. Another, slightly non-intuitive way to solve the out-degree problem is **re-computation**. Figure 3(d) shows that operation b is performed on both PE_4 and PE_3 at time $t+1$. Then their results are used by dependent nodes c, d, e and f in time $t+2$.

Note that this is not routing, but re-computation because there is no path between two PEs on which b is mapped. In this example, routing, and re-computation both enable an II of 3. Another example of re-computation, explained in detail in Appendix A illustrates a case, where re-computation leads to better II than possible by routing. But, even in this example, Figure 3(e) shows that performing both routing and re-computation, the input graph can be mapped with an II of 2. Using only one of them, the best achievable II is 3. Consequently, the problem formulation must uniformly account for routing and re-computation.

4 Problem and Complexity

Let $D = (V_d, E_d)$ be the input DFG, and $C = (V_c, E_c)$ the TEC, extended to some number k time steps[1]. Let $C^* = (V_{c^*}, E_{c^*})$ be a subset of C where $V_{c^*} \subseteq V_c$ and $E_{c^*} \subseteq E_c$. In Appendix B, we give a formal definition of a *valid* mapping of operations in D to PEs in C, but intuitively, a valid mapping is one that allows correct execution of the DFG by satisfying all the data dependencies. Of course, valid mappings must allow overlapping operations in different time steps, and resolve out-degree and resource conflict problems by routing and/or re-computation. We define valid mappings in terms of **Epimorphisms**, which forms the basis of our algorithm and heuristic.

DEFINITION 1. *Let G and H be two digraphs. A mapping* $f : V(G) \to V(H)$ *is a Homomorphism if* $(f(u), f(v)) \in E(H) \Rightarrow (u, v) \in E(G)$. *It is called an Epimorphism if it is node surjective, i.e., every node in H is an image of some node in G. Note that node surjective implies arc surjective [16].*

Simply stated, for every valid mapping there exists a smallest k for which there is an Epimorphic map $M : C^* \to D$ that satisfies certain conditions. Conversely, an Epimorphic map from $M : C^* \to D$, for some k, that satisfies the same conditions corresponds to a valid mapping. Thus the optimization problem is to construct an Epimorphism $M : C^* \to D$, where C^* is a subgraph of a minimally extended TEC C. The problem formulation and the NP-completeness proof is explained in detail in Appendix C.

Now we see how the formulation works. Consider a node (i.e. a PE) $i \in V_{c^*}$. Let $i' = M(i) \in V_d$. i' be the operation that is mapped to PE i. For example, if the node i is PE_4 at time t in Figure 1(d), then, in the mapping shown, $i' = M(PE_{4,t}) = a$. Similarly, let $j \in V_{c^*} : \exists (j, i) \in E_{c^*}$, and let j' be the operation that is mapped to PE j. For example, j is the PE_4 at time $t+1$, and $j' = M(PE_{4,t+1}) = c$. Then epimorphism requires that if there is an arc between a and c, then there must be an arc between PE_4 at time t, and PE_4 at time $t+1$. This example illustrates how epimorphism ensures that data dependencies are preserved.

[1]To avoid cumbersome notation, we don't show the explicit dependence on k, which is to be determined and minimized.

Our problem formulation seamlessly captures routing and re-computation. Whenever we use routing/re-computation, a set of PEs in the TEC map to one operation in the DFG. For example, in Figure 3(c), in which the out-degree problem is resolved using routing, PE_4 of time $t+1$, and PE_3 of time $t+2$ are mapped to operation b. Since operation a has an arc to operation b, epimorphism requires that there be at least one arc between the set of PEs that are mapped to a, and the set of PEs that are mapped to b. This is true, since there is an arc from PE_4 at time t (where operation a is mapped), to PE_4 at time $t+1$ (where operation b is mapped). Similarly the data dependencies with operations c, d, e, and f are satisfied. In Figure 3(d), in which the out-degree problem is resolved using re-computation, a set of two PEs, PE_4 of time $t+1$, and PE_3 of time $t+1$ are mapped to operation b. In this case also, the data dependencies with the other operations are satisfied.

Again, to use routing or re-computation, a set of PEs in the TEC map to one operation in the DFG. The only difference is the presence/absence of arc between the PEs in the set. In Figure 3(c), PE_4 of time $t+1$, and PE_3 of time $t+2$ have an arc between them, so they transfer data through routing. On the other hands, in Figure 3(d), PE_4 of time $t+1$, and PE_3 of time $t+1$ do not have an arc between them, therefore the operation b has to be recomputed.

We now explain the conditions. Essentially, the condition is just to ensure that the PE at which computation, or re-computation happens, receives all the input operands. Thus, if we consider a node (i.e. a PE) $i \in V_{c^*}$. Let $i' = M(i) \in V_d$. i' is the operation that is mapped to PE i. Let $j \in V_{c^*} : \exists(j,i) \in E_{c^*}$. Then if there is no PE that is connected to i onto which operation i' is mapped, (i.e. if $(j,i) \in V_{c^*}$ and $M(j) \neq i'$), then for any $k' : \exists(k',i') \in V_d$, there must exist a $k \in V_{c*} : \exists(k,i) \in E_{c^*}$ and $M(k) = k'$.

5 EPIMap

In this section, we describe our heuristic algorithm called EPIMap. EPIMap reduces search space because of three main reasons: i) it changes the input DFG to ensure that the graph meets necessary mapping conditions; ii) it determines a more accurate lower bound on II, and iii) when a placement is impossible, it changes nodes (re-computation or routing) that are left unmapped and attempts a new placement. A combination of routing and re-computation can be achieved when a node cannot be placed multiple times.

5.1 Overview

The EPIMap algorithm is presented in Algorithm 1. EPIMap initially changes the input DFG to hold the necessary mapping conditions (line 1-4). In the prepared DFG, it determines the minimum II (line 5). Then it attempts to find a valid mapping for this II. EPIMap achieves this in the `While` loop shown in line 6. If such a mapping cannot be found, it collects the set of nodes left unmapped and changes their input through routing or re-computation (line 24). In addition, it stores the number of unmapped nodes in the previous attempt (line 25). In the next attempt, if the number of unmapped nodes increases, it avoids further attempt for the prepared DFG, restores the original DFG, and decreases the number of operations at each cycle (line 21, 22) and retries again. If at any point of time MII increases and becomes greater than II (line 12), EPIMap restores the original DFG, increases II, and attempts for a new mapping (line 12-16). EPIMap repeats these steps until a valid mapping can be achieved.

5.2 Necessary Mapping Conditions

We first characterize the properties of a DFG so that it is mappable.
1. Out-degree of each operation must be less than the out-degree of PEs in the TEC. If there is a node u with out-degree larger than that of PEs in the TEC, then a feasible mapping cannot be found. This can be fixed by either routing or re-computation. To perform routing, EPIMap adds a new node v and moves some outgoing arcs to v. Then it adds the arc (u,v). EPIMap performs re-computation by creating a new node v and connecting all nodes that have an outgoing arc to node u, to v, i.e. $\forall r \in V_d : (r,u) \in E_d$, add a new arc (r,v). Function *Constraint_Outdegree* in EPIMap is called to perform this step. From definition 1, we can immediately conclude that the modified graph is epimorphic to the input graph.

2. DFG must be balanced. A linear ordering of the nodes can be obtained by topological sorting. EPIMap schedules operations using this order. If there is an arc (i,j) between nodes i (scheduled at time t_i) and j (scheduled at time t_j) where $t_i - t_j > 1$, then DFG is not balanced. The formal definition of a balanced graph is given in the Appendix D. When such an arc is found, EPIMap adds extra nodes and balances the graph. From Definition 1, we can conclude that the balanced graph is epimorphic to the input graph. For example, in Figure 1(b), there is a distance of 2 between orders of node b and e. To overcome this problem, another b node is inserted between those nodes as illustrated in Figure 1(d). It should be noted that EPIMap removes every cycle in the DFG because topological sorting is impossible for cyclic graphs. This step partially schedules the operations while maintaining the routing. EPIMap calls *Balance* function for this phase.

3. The number of nodes at each level must be less than the number of PEs in the CGRA (non-time extended). If a level violates this constraint, EPIMap finds a set of nodes that can be moved to the other levels (previous or next) with the minimum cost. Here, the cost is the number of extra nodes that should be added to the DFG to preserve the data dependency of nodes. Again, from definition 1, this graph is also epimorphic to the input DFG. This step completes scheduling while maintaining routing. At the end of this step, cycles in the DFG are restored.

5.3 Determining Minimum II

While result of previous steps schedules the operations based on the necessary conditions on the DFG, this step finds a modulo schedule for the operations. Let MII denote the minimum II. This can be expressed as $MII = Max(ResMII, RecMII)$, where $ResMII = \lceil \frac{n}{M \times N} \rceil$ is the resource constrained minimum II and n is the number of operations in the DFG, and $RecMII$ is recurrence-constrained MII. Recurrence-constrained indicates inter-iteration dependency of operations in a loop. When such a dependency exists, the next iteration cannot start until the results from the previous iteration becomes available [24]. When MII is found, EPIMap folds (modulo schedules) the DFG (after preprocessing). To create such a graph, called MDFG, EPIMap assigns a level to each node. The level of each node in MDFG is the level of that node in the preprocessed DFG modulo MII. In MDFG, if the number of nodes at each level is less than the number of PEs in the CGRA, MII is feasible, otherwise, EPIMap moves operations to next or previous levels to satisfy this constraint. If satisfying this constraint is impossible, EPIMap increases MII until such a graph can be achieved. This phase is completed by *DetermineMII* and *UpdateMII* functions. It should be noted that EPIMap determines II accurately without any attempt to actually place the operations onto the CGRA.

5.4 Placement

EPIMap places operations (assigns nodes in the DFG to the nodes in the TEC) by finding the maximum common subgraph (MCS) (line 17) between the TEC and MDFG using Levi's algorithm [19]. If MCS is isomorphic to DFG, the mapping is completed. Otherwise, DFG must be changed. Since MCS is an NP-Complete

Algorithm 1 EPIMap(Input D, Input C)

```
1: $D_p \leftarrow$ Constraint_Outdegree(D);
2: $M \leftarrow |V_c|$;
3: $D_p \leftarrow$ Balance($D_p$);
4: $D_p \leftarrow$ Constraint_Levels($D_p$, M);
5: $II \leftarrow$ DetermineMII($D_p$); $D_i \leftarrow D_p$;
6: while Mapping is not found do
7:     N=$\infty$;
8:     while true do
9:         $D_i \leftarrow$ Balance($D_i$);
10:        $D_i \leftarrow$ Constraint_Levels($D_i$, M);
11:        $D_{mp}, MII \leftarrow$ UpdateMII($D_i$, M);
12:        if $MII > II$ then
13:            $II \leftarrow II + 1$;
14:            $M \leftarrow |V_c|$;
15:            $D_i \leftarrow D_p$; break;
16:        end if
17:        $CS \leftarrow$ MCS ($D_{mp}, II, C$);
18:        if $V_{CS}$ = $V_{D_{mp}}$ then
19:            return $CS$;
20:        else
21:            if $V_{D_i} - V_{CS} > N$ then
22:                $M \leftarrow M - 1$; $D_i \leftarrow D_p$; break;
23:            else
24:                $D_i \leftarrow$ ChangeInput($V_{D_i} - V_{CS}$, $D_i$);
25:                $N \leftarrow |V_{D_i} - V_{CS}|$;
26:            end if
27:        end if
28:    end while
29: end while
```

problem [10], we have modified Levi's algorithm to keep track of the number of attempts that did not increase the number of nodes in the common subgraph. When the number of unsuccessful attempts reaches a threshold value, EPIMap avoids further attempts.

6 Experimental Results

We have modified GCC and defined a new C pragma directive to specify the loops in the source code. We have taken loops from different applications including SPEC2006 benchmarks programs.

To evaluate the effectiveness of EPIMap, we compare the average II achieved using EPIMap with that achieved by EMS [23]. EMS uses a node selection scheme to map operations. To show that EPIMap generates better mappings than EMS even if EMS uses a better node selection scheme, we created 500 random node selection orders and selected the mapping that EMS generates with the best II. The mapping results for this case are labeled **BCEMS**. The original EMS results are labeled **EMS**.

We assume a 4×4 homogeneous CGRA, and PEs are capable of performing fixed-point and logical operations. Access to the memory as well as other operations have latency of 1 cycle. We assume that there is enough memory to hold the instructions and variables of loops where PEs have 2 local registers. In addition, we assume that all PEs have access to the data memory but the data bus is shared among PEs in a row and is mutually exclusive. For load and store operations, two instructions are executed, one for address generation and one that generates/loads data.

6.1 EPIMap Generates Better Mapping

Figure 4(a) shows the average achieved II of different benchmarks using different mapping techniques. The first observation is that on average, EPIMap achieves a lower II than both EMS and BCEMS in all applications. We also note that even with better node selection scheme, EMS results is worse than EPIMap.

In the best case, EPIMap-II $= 0.19 \times$ EMS-II (*Swim_calc2*), and EPIMap-II $= 0.28 \times$ BCEMS-II (*LowPass*). This means that in the best case, EPIMap accelerates *Swim_calc2* by a factor of 5.25X more than EMS and by 3.5X more than BCEMS, as the performance is inversely proportional to II. On average, EPIMap finds mappings with II of 0.35X and 0.44X of that EMS and BCEMS, which translates to a 2.26X and 2.84X better performance, respectively.

6.2 EPIMap's II is close to Minimum II

In Figure 4(b), the relative II (in percentage) is calculated by dividing MII by resulting II, i.e. $\frac{MII}{II}$; thus, the higher value implies a mapping that is closer to an optimal II.

We observe that EPIMap achieves the lower bound (MII) in 9 out of 14 benchmarks. In the five cases where EPIMap could not achieve the MII, it was either because there was no mapping that can achieve MII or the threshold value was reached before a valid mapping was found.

The average relative II gets worse for the applications with complex memory access patterns including *Bzip2* which has many memory related store instructions (bus is shared and mutually exclusive). In addition, there are many loops in *Bzip2* with the number of nodes nearly equal to the number of nodes in the TEC. When EPIMap reduces the number of operations at each level of the preprocessed DFG, some nodes will be rescheduled. To maintain data dependency for those nodes, extra nodes need to be added to the DFG. Therefore, $ResMII$ increases.

We also note that EMS and BCEMS could not achieve MII for any benchmark. The average relative II of EMS is 45% of the average relative II of BCEMS, and never exceeds 60% for average relative II of BCEMS. However, on the average, $\frac{MII}{EPIMap-II} \approx 0.92$, whereas $\frac{MII}{EMS-II} \approx 0.30$ and $\frac{MII}{BCEMS-II} \approx 0.47$. EPIMap produces mapping that achieve II which is much closer to MII than either EMS or BCEMS.

6.3 DFG Modification Reduces Search Space Significantly

Experimental results show that EPIMap generates valid mappings in 70% of loops in 6 benchmarks including: *Sor, Sobel, Lowpass, Laplace, wavelet, and Sjeng* in the first try before any attempt is made to reduce the number of operations at each level (second heuristic). However, EMS tries on average 7.87 unsuccessful IIs to find a valid mapping for these benchmarks. For each unsuccessful II, EMS searches for different node mappings and routing. This shows that changing input graph to ensure that it satisfies the necessary mapping conditions reduces search space substantially.

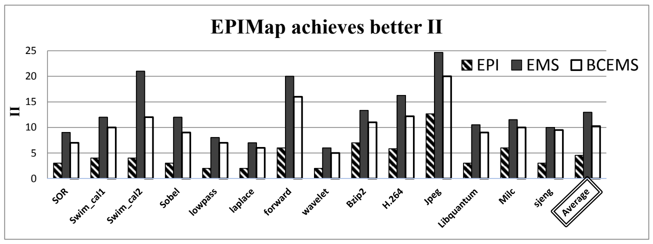
6.4 Counter-Intuitive Minimum IIs after Limiting Number of Nodes at Each Cycle

In the first glance, it seems that reducing number of nodes at each cycle increases II. However, for 50% of loops in *Swim_call, Swim_cal2, H.264, Jpeg, Libquantum and Sjecg*, this constraint is used on average 3 times but it did not increase the achieved II.

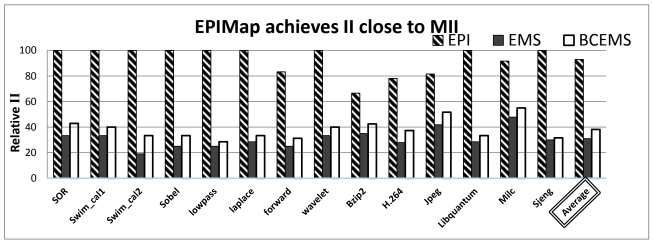
This is because the number of nodes in the modified DFG of these loops is less than the number of nodes in the TEC. This difference allows EPIMap to add more nodes to the DFG without increasing $ResMII$. Hence, we conclude that level constraint does not reduce mapping quality unless the number of nodes in the extended CGRA and modified DFG are close.

6.5 Reasonable Compilation Time

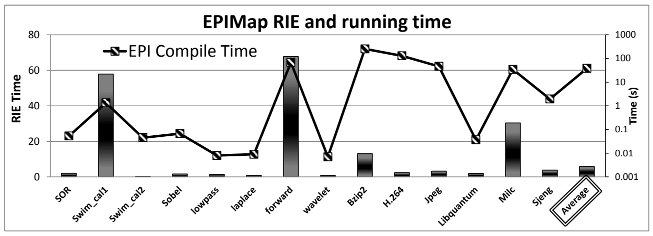
We measured the running time of EPIMap and EMS on an Intel Core2 machine with CPU frequency of 2.66 GHz. Figure 4(c)



(a) EPIMap achieves mapping with average II of 0.35X lower than EMS.



(b) In 9 out of 14 applications, EPIMap generates mapping with $II = MII$.



(c) Better mapping is achieved at the cost of increase in compilation time. However, the actual compilation time is usually negligible.

Figure 4: The comparison of achieved II of different applications using EPIMap, EMS, and BCEMS. On average, EPIMap achieves lesser II than other techniques. This better mapping quality comes at the cost of more compilation time of those applications.

shows the relative increase in execution (RIE) time of EPIMap and its actual execution time for different applications, where the threshold of EPIMap is 10^7. The bars show the RIE time of EPIMap to EMS, i.e. $\frac{T_{EPI}-T_{EMS}}{T_{EMS}}$ for different benchmarks. The actual EPIMap execution time is depicted by the solid line.

It can be observed that EPIMap achieves better average II at the cost of more execution time, the time it takes for EPIMap to generate a valid mapping or compilation time of application. In fact, the average RIE time of EPIMap to EMS is around 5.8. However, we can observe that the actual compilation time is fairly low, mostly less than 30 seconds on average.

7 Summary

CGRAs are promising structures that provide high-performance and high power-efficiency. However, achieving high power-efficiency is challenging primarily because compilation for CGRA is difficult. In this paper, we formulate the problem of mapping application onto a CGRA and establish its complexity. We also characterize the necessary conditions for application specification to find a feasible mapping. To tackle the mapping problem, we proposed a heuristic algorithm called EPIMap. EPIMap is different from the existing methods in the sense that it systematically searches the solution space to find a valid mapping. Our experimental results show that EPIMap generates mapping which leads to significant performance improvement compared to EMS.

8 Acknowledgments

This work was supported in part by NSF IUCRC Center for Embedded Systems under Grant DWS-0086, by the Science Foundation of Arizona (Grant SRG 0211-07), and by the Stardust Foundation.

9 References

[1] Aa, T. V., Raghavan, P., Mahlke, S. A., Sutter, B. D., Shrivastava, A., and Hannig, F. Compilation techniques for cgras: exploring all parallelization approaches. In *CODES+ISSS* (2010), pp. 185–186.

[2] Ahn, M., Yoon, J., Paek, Y., Kim, Y., Kiemb, M., and Choi, K. A spatial mapping algorithm for heterogeneous coarse-grained reconfigurable architectures. In *Proc. DATE* (2006), pp. 363–368.

[3] Bansal, N., Gupta, S., Dutt, N., Nicolau, A., and Gupta, R. Network topology exploration of mesh-based coarse-grain reconfigurable architectures. In *Proc. DATE* (2004), pp. 474–479.

[4] Becker, J., and Vorbach, M. Architecture, memory and interface technology integration of an industrial/ academic configurable system-on-chip (csoc). In *Proc. ISVLSI* (2003), pp. 107–112.

[5] Borkar, S. Design challenges of technology scaling. *IEEE Micro 19*, 4 (1999), 23–29.

[6] Bouwens, F., Berekovic, M., Sutter, B. D., and Gaydadjiev, G. Architecture enhancements for the adres coarse-grained reconfigurable array. In *Proc. HiPEAC* (2008), pp. 66–81.

[7] Chen, D. C. *Programmable arithmetic devices for high speed digital signal processing*. PhD thesis, University of California, Berkeley, 1992.

[8] Dimitroulakos, G., Galanis, M., and Goutis, C. Exploring the design space of an optimized compiler approach for mesh-like coarse-grained reconfigurable architectures. In *Proc. IPDPS* (2006), pp. 113–122.

[9] Friedman, S., Carroll, A., Van Essen, B., Ylvisaker, B., Ebeling, C., and Hauck, S. Spr: an architecture-adaptive cgra mapping tool. In *Proc. FPGA* (2009), pp. 191–200.

[10] Garey, M. R., and Johnson, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[11] Goldstein, S., Schmit, H., Moe, M., Budiu, M., Cadambi, S., Taylor, R., and Laufer, R. Piperench: a coprocessor for streaming multimedia acceleration. In *Proc. ISCA* (1999), pp. 28 –39.

[12] Guo, Y., Hoede, C., and Smit, G. A pattern selection algorithm for multi-pattern scheduling. In *Proc. PDPTA* (2006), pp. 198–205.

[13] Hartenstein, R. A decade of reconfigurable computing: a visionary retrospective. In *Proc. DATE* (2001), pp. 642–649.

[14] Hartenstein, R., and Kress, R. A datapath synthesis system for the reconfigurable datapath architecture. In *Proc. ASP-DAC* (1995), pp. 479 –484.

[15] Hatanaka, A., and Bagherzadeh, N. A modulo scheduling algorithm for a coarse-grain reconfigurable array template. In *Proc. IPDPS* (2007), pp. 1–8.

[16] Hell, P., and Nesetril, J. *Graphs and Homomorphisms*. Oxford University Press, New York, NY, USA, 2004.

[17] Lee, J.-E., Choi, K., and Dutt, N. D. Compilation approach for coarse-grained reconfigurable architectures. *IEEE Design and Test of Computers 20*, 1 (2003), 26–33.

[18] Lee, M.-H., Singh, H., Lu, G., Bagherzadeh, N., Kurdahi, F. J., Filho, E. M. C., and Alves, V. C. Design and implementation of the morphosys reconfigurable computingprocessor. *J. VLSI Signal Process. Syst. 24* (2000), 147–164.

[19] Levi, G. A note on the derivation of maximal common subgraphs of two directed or undirected graphs. *Calcolo 9* (Dec. 1973), 341–352.

[20] Mei, B., Vernalde, S., Verkest, D., and Lauwereins, R. Design methodology for a tightly coupled vliw/reconfigurable matrix architecture: a case study. In *Proc. DATE* (2004), pp. 1224–1229.

[21] Mirsky, E., and DeHon, A. Matrix: a reconfigurable computing architecture with configurable instruction distribution and deployable resources. In *Proc. FPGAs for Custom Computing Machines* (1996), pp. 157 –166.

[22] Miyamori, T., and Olukotun, K. Remarc: Reconfigurable multimedia array coprocessor. *IEICE Trans. on Information and Systems* (1998), 389–397.

[23] Park, H., Fan, K., Mahlke, S. A., Oh, T., Kim, H., and Kim, H.-s. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In *Proc. PACT* (2008), pp. 166–176.

[24] Rau, B. R. Iterative modulo scheduling: an algorithm for software pipelining loops. In *Proc. MICRO* (1994), pp. 63–74.

[25] Woh, M., Mahlke, S., Mudge, T., and Chakrabarti, C. Mobile supercomputers for the next-generation cell phone. *Computer 43*, 1 (2010), 81 –85.

[26] Yoon, J., Shrivastava, A., Park, S., Ahn, M., and Paek, Y. A graph drawing based spatial mapping algorithm for coarse-grained reconfigurable architectures. *IEEE Trans. on VLSI Systems 17*, 11 (2009), 1565–1578.

APPENDIX

A Example of the Need for Re-computation

One form of resource limitation problem is resource conflict problem. If due to the modulo scheduling, the number of nodes using the same resource exceeds the number of available resources, we call it resource conflict problem. There is a resource conflict between nodes b, e and f in Figure 3(b) when it is mapped onto a 2×2 CGRA (Figure 3(a)). For such a mapping $MII = \lceil \frac{6}{4} \rceil = 2$. To meet data dependencies, node b has to be executed on a PE that is adjacent to PEs executing nodes c and d. In addition, nodes e and f should be mapped on PEs which are also adjacent to PEs executing nodes c and d. In mapping shown in Figure 3(c), nodes c and d are mapped onto PE_4 and PE_2 respectively. Therefore, there are two PEs adjacent to these PEs, PE_1 and PE_3. Due to the modulo scheduling, nodes b from iteration i, e, and f from iteration $i-1$ are executed at the same time. At this time, there are three nodes but two available resources. In Figure 3(c), node b is executed on two different PEs to overcome this problem. Without re-computation, it is impossible to find a valid mapping for $II = 2$.

B Formal Problem Definition

DEFINITION 2. *Valid Mapping: Let n be the number of nodes in V_d, i.e. $n = |V_d|$, and $C = (V_c, E_c)$ be TEC. Let $C^* = (V_{c^*}, E_{c^*})$ be a subset of C, i.e., $C^* \subseteq C$. Let $S = \{s_1, s_2, s_3, ..., s_n\}$ be a set of n disjoint subsets of V_{c^*} such that $1 \le \forall i \le n, |s_i| \ge 1$. A mapping function $f : V_d \to S$ is a valid mapping iff $\forall u, v \in V_d : (u, v) \in E_d$,*

$\forall v' \in f(v)$, there must be a path from a node $u' \in f(u)$ to v'. The nodes in this path must only be nodes in $f(v)$.

In this definition, the set of PEs in the TEC are partitioned into some sets s_i. The number of sets in this partition must be n to ensures that every node in the DFG will be mapped into a set. $s_i > 0$ ensures that every operations is executed at least once. If the number of PEs in a set s_i is more than one, it implies that the nodes mapped into s_i will be executed, and/or routed, and/or re-computed.

A mapping is valid if data dependencies between nodes in the DFG are preserved after mapping. When there is an arc between two nodes $(u, v) \in E_d$ in the DFG, every PE in the set s_v that executes node v, can receive the result of execution of u as input. Thus, for every PE in the set s_v, there must be a path from PEs in the set s_u where u is mapped into. More precisely, every PE in the set s_u either executes u or routes the result of execution of u.

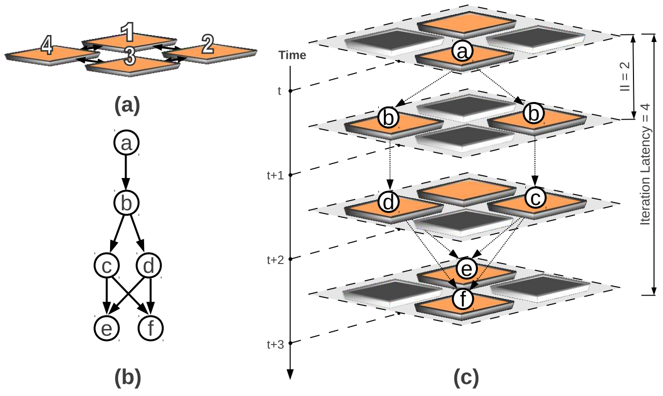


Figure 5: (a) An input DFG, (b) a 2×2 CGRA, (c) the best achievable mapping. There is resource conflict between nodes b and nodes e or f ($II = 2$). Re-computing node b allows us to avoid this conflict.

Therefore, the result of u can be routed from PEs in this set to the PEs executing node v that need u as an input.

C Problem Formulation

In this section, we formulate the problem of mapping a DFG onto a CGRA as finding an Epimorphic subgraph of TEC onto the input DFG. For simplicity, we assume that input DFG is acyclic.

To do this, Theorem 1 states that when a node a is mapped to a set of nodes in TEC, the nodes in this set are only connected to each other (routing) or they are connected to the nodes on which the adjacent nodes of a are mapped (re-computation is there if more than one node in that set). Then Theorem 2 states that every valid mapping implies an Epimorphic function from a subgraph of TEC to the input DFG. Using Theorem 1 and 2, we prove that every node a in the DFG whose in-degree is greater than 0 would be mapped onto a node in TEC which has input either from the nodes that a is mapped onto (routing) or from all the nodes that the input subgraph of a is mapped onto in Theorem 3. Then, Theorem 4, also proves the converse, that every Epimorphic function from a subgraph of TEC to the input DFG that holds the aforementioned properties implies a valid mapping. Consequently, they are equivalent.

Finally, since we prove the equivalence of problem of mapping application onto a CGRA with graph Epimorphism which is an NP-Complete problem [16], therefore CGRA application mapping is NP-Complete.

THEOREM 1. *In a valid mapping, every node $u \in V_d$ that is mapped onto a set $s_i \subset V_{c^*}$ where $|s_i| > 1$, $\forall g, h \in s_i$, if there is a path P between g and h then P only passes through some nodes in s_i.*

PROOF. Let's assume that there is a path P' between g and h which passes through some nodes not in s_i. Without loss of generality, let's assume that P' starts from g passes through at least one node $k \notin s_i$ and ends in h. Therefore, there is an arc between $f^{-1}(g)$ and $f^{-1}(k)$ and also an arc between $f^{-1}(k)$ and $f^{-1}(h)$ where $f^{-1}(h) = f^{-1}(g)$ in the DFG. This implies that there is a cycle in the input DFG which contradicts with our assumption that the DFG is acyclic. □

THEOREM 2. *Let $D(V_d, E_d)$ be the input DFG and $C_k(V_c, E_c)$ be the CGRA graph extended in time k. Every valid mapping implies an Epimorphic function $M : C^* \longrightarrow D$ where C^* is a subgraph of C.*

PROOF. Let V_{c^*} be the set of nodes in C^* and E_{c^*} be the set of arcs.

- m is function. A mapping is valid if each PE executes the maximum of one operation per cycle. Therefore, $\forall i \in V_{c^*}, M(i)$ is exactly one element in V_d.
- m is surjective. A valid mapping maps implies that every nodes in graph D must be mapped onto some PEs in TEC. Therefore, it is surjective.
- S is homomorphic to D. In a valid mapping, every node $i \in V_d$ is mapped to a set of nodes $s_i \subset V_{c^*}$. Therefore, $\forall a, b \in V_{c^*}$, if there is an arc $(a, b) \in E_{c^*}$, then either $a, b \in s_i$ or $a \in s_i, b \in s_j$ where $s_i \neq s_j$. The first case simply implies homomorphism according to definition. We claim that the second case also implies homomorphism. Let's assume node $j \in V_d$ is mapped to s_j. If there is an arc between nodes i and j then it implies a homomorphism. Otherwise, it is trivial to see that if there is no arc between i and j but between a and b, then we can remove (a, b) from E_s and mapping will be still valid.

Both graphs in Figure 3(c) and Figure 3(d) are Epimorphic to Figure 3(b). All arcs in the mapping correspond to an arc in the input DFG. However, there is an arc between PE_4 at time t and PE_3 at time $t+2$. The second one is used for routing. Figure 3(d) shows another mapping. In this mapping there is an arc between PE_4 time t and PE_4 at time $t+1$. This arc is mapped into the arc between nodes a and b in the input DFG. Similarly, there is an PE_4 time t and PE_3 at time $t+1$ which is also mapped into the same arc. This case, node b is executed two times (re-compuation).

□

DEFINITION 3. *Input subgraph: for every node i in a digraph $D(V_d, E_d)$, there exists a subgraph $G(V_g, E_g)$ such that V_g is the set of all nodes $j : (j,i) \in E_d$ in addition to node i; also E_g is the set of all arcs $(j,i) : (j,i) \in E_d$.*

DEFINITION 4. *An isomorphism from $G(V_g, E_g)$ onto $H(V_h, E_h)$ is defined as $f : V_g \longrightarrow V_h$ such that:*

1. $|E_g| = |E_h|$
2. $|V_g| = |V_h|$
3. $\forall u, v \in V_g : (u,v) \in E_g$ *iff* $(f(u), f(v)) \in E_h$ *[10].*

THEOREM 3. *Every valid mapping implies an Epimorphic function $M : C^* \longrightarrow D$ such that $\forall i \in V_{c^*}$, **input subgraph** of $M(i)$ $K(V_k, E_k)$, **input subgraph** of i $L(V_l, E_l)$: if $Indegree(M(i)) > 0 \Rightarrow$*

1. *$\forall j \in V_l : M(j) = M(i)$ or*
2. *K and L are isomorphic.*

PROOF.
- Theorem 2 proves that every valid mapping implies an Epimorphic function $M : C^* \longrightarrow D$.
- Let's assume $a \in V_d$ is mapped onto set $s_a \subset V_{c^*}$. $\forall i \in V_{c^*} : Indegree(i) > 0$ either:
  - all incoming arcs of i are from nodes $j \in s_a$ which implies $\forall j \in V_l : M(j) = M(i)$.
  - input arcs of i are from a combination of nodes $j \in s_a$ and nodes $k \notin s_a$. This case cannot happen according to Theorem 1.
  - all incoming arcs are from nodes $j \notin s_a$. In this case, we show that in order to have a valid mapping, K and L should be isomorphic. Let's assume that K and L are not isomorphic. Then either $V_k \neq V_l$ or $E_k \neq E_l$. If the number of nodes or arcs are different then the mapping cannot be valid. It should be noted that the only node in K with incoming arcs is $M(i)$ and i is the node in L with incoming arcs. Therefore, the connection topology cannot violate isomorphism constraint if the mapping is valid.

□

THEOREM 4. *Every Epimorphic function $M : C^* \longrightarrow D$ with following constraint implies a valid mapping. Constraint: $\forall i \in V_{c^*}$, **input subgraph** of $M(i)$ $K(V_k, E_k)$, **input subgraph** of i $L(V_l, E_l)$: if $Indegree(M(i)) > 0 \Rightarrow$*

1. *$\forall j \in V_l : M(j) = M(i)$ or*
2. *K and L are isomorphic.*

PROOF.
- Because the function is surjective, all nodes in D must be covered by at least one node in C^*.
- Since epimorphism preserves node adjacency [16] and the function is surjective, then all arcs in E_d are covered by an arc in E_{c^*} which implies, for all $u, v \in V_d : (u,v) \in E_d$ there is an arc between $(m^{-1}(u), m^{-1}(v))$.
- $\forall u \in V_d : indegree(u) > 0$ where $K(V_k, E_k)$ is input subgraph of u and $L(V_l, E_l)$ is input subgraph of $m^{-1}(u)$:
  - if $\forall j \in V_l : M(j) = u$, there must be a node $v \in V_{c^*}$ such that input subgraph of v is isomorphic to K. Because function is surjective, all arcs of E_k must be mapped. Therefore, according to the constraint, since the first case cannot happen, there exists a node whose input subgraph is isomorphic with K which implies u is mapped properly.
  - if K and L are isomorphic, then mapping of u is valid.

□

THEOREM 5. *Every valid mapping from an input DFG $D(V_d, E_d)$ onto CGRA graph extended in time k $C_k(V_c, E_c)$ is equivalent to an Epimorphic function $M : C^* \longrightarrow D$ such that $\forall i \in V_s$: $K(V_k, E_k)$ **input subgraph** of $M(i)$, $L(V_l, E_l)$ **input subgraph** of i : if $Indegree(M(i)) > 0 \Rightarrow$*

1. *$\forall j \in V_l : M(j) = M(i)$ or*
2. *K and L are isomorphic.*

where $C^*(V_{c^*}, E_{c^*}) : C^* \subseteq C$.

PROOF.
- From Theorem 3, every valid mapping implies an Epimorphic function $M : C^* \longrightarrow D$ with above-mentioned constraints.
- From Theorem 4, every Epimorphic function $M : C^* \longrightarrow D$ with above-mentioned constraints implies a valid mapping.

□

THEOREM 6. *The problem of mapping an input DFG onto a CGRA is NP-Complete.*

PROOF. Mapping an input DFG onto a CGRA is in NP. The decision problem of the above problem is: Given an input DFG D, a certificate which is a T, a given graph G' that is a subgraph of the extended in time graph of CGRA and a mapping from subsets of G' onto nodes in D, is mapping to G' a valid mapping? The yes-instance of this problem can be verified in polynomial time. For every arc in the G' we can verify the valid mapping conditions by checking the corresponding arc in the input DFG. Therefore, mapping is in NP. Finding minimum Epimorphism has been proved to be NP-Complete [16]. We have shown the equivalence of valid mapping and Epimorphism, therefore, we conclude that the problem of finding a valid mapping is NP-Complete. □

D Formal Definitions

DEFINITION 5. *Balanced graph $G(V_g, E_g)$: a graph is balanced if $\forall i, j \in V_g$ such that there are two paths $P_1(i,j)$ and $P_2(i,j)$ between them then the length of paths are equal. In addition, $\forall i, j \in V_g$: if $\exists k, l \in V_g$ such that $P_1(i,k), P_2(j,k), P_3(i,l)$, and $P_4(j,l)$ then $L(P_1) - L(P_2) = L(P_3) - L(P_4)$ where $L(P_i)$ is the length of path P_i.*