An efficient and flexible stochastic CGRA mapping approach

Satyajit Das, Kevin Martin, Thomas Peyret, Philippe Coussy

An Efficient and Flexible Stochastic CGRA Mapping Approach

SATYAJIT DAS, Univ. Bretagne-Sud, UMR 6285, Lab-STICC, France and Indian Institute of Technology Palakkad, India
KEVIN J. M. MARTIN, Univ. Bretagne-Sud, UMR 6285, Lab-STICC, France
THOMAS PEYRET, CEA, LIST, France
PHILIPPE COUSSY, Univ. Bretagne-Sud, UMR 6285, Lab-STICC, France

Coarse-Grained Reconfigurable Array (CGRA) architectures are promising high-performance and power-efficient platforms. However, mapping applications efficiently on CGRA is a challenging task. This is known to be an NP complete problem. Hence, finding good mapping solutions for a given CGRA architecture within a reasonable time is complex. Additionally, finding scalability in compilation time and memory footprint for large heterogeneous CGRAs is also a well known problem. In this paper, we present a stochastic mapping approach that can efficiently explore the architecture space and allows finding best of solutions while having limited and steady use of memory footprint. Experimental results show that our compilation flow allows to reach performances with low-complexity CGRA architectures that are as good as those obtained with more complex ones thanks to the better exploration of the mapping solution space. Parameters considered in our experiments are: number of tiles, Register File (RF) size, number of load/store units, network topologies, etc. Our results demonstrate that high-quality compilation for a wide range of applications is possible within reasonable run-times. Experiments with several DSP benchmarks show that the best CGRA configuration from the architectural exploration surpasses an ultra low-power DSP optimized RISC-V CPU to achieve up to 15.28× (with an average of 6× and minimum of 3.4×) performance gain and 29.7× (with an average of 13.5× and minimum of 6.3×) energy gain with an area overhead of 1.5× only.

CCS Concepts: • **Computer systems organization** → **Reconfigurable computing**; *Embedded systems*; *Multiple instruction, multiple data*; • **Software and its engineering** → **Retargetable compilers**.

Additional Key Words and Phrases: CGRA, mapping

Author version

This document is the author version of the paper "An Efficient and Flexible Stochastic CGRA Mapping Approach" by *Satyajit Das, Kevin J. M. Martin, Thomas Peyret, and Philippe Coussy*, accepted on July 2022 for publication in ACM Transactions on Embedded Computing Systems (TECS).
The original paper can be found here:
https://dl.acm.org/doi/10.1145/3550071

1 INTRODUCTION

Coarse-Grained Reconfigurable Array (CGRA) architectures are based on a set of interconnected processing elements (Fig. 1(a)). For the last 25+ years, CGRAs are studied for their interesting trade-off between flexibility and energy efficiency [22]. The literature is thus very rich in all kinds of architectures that fall in the family of CGRA [16, 24, 27]. One main category to classify CGRAs is their *structure* [27]. They distinguish themselves by the granularity of computing elements, homogeneity,

Authors' addresses: Satyajit Das, Univ. Bretagne-Sud, UMR 6285, Lab-STICC, F-56100, Lorient, France and Indian Institute of Technology Palakkad, Palakkad, Kerala, India, satyajitdas@iitpkd.ac.in; Kevin J. M. Martin, Univ. Bretagne-Sud, UMR 6285, Lab-STICC, F-56100, Lorient, France, kevin.martin@univ-ubs.fr; Thomas Peyret, CEA, LIST, Computing and Design Environment Laboratory, Gif-sur-Yvette, France, thomas.peyret@cea.fr; Philippe Coussy, Univ. Bretagne-Sud, UMR 6285, Lab-STICC, F-56100, Lorient, France, philippe.coussy@univ-ubs.fr.

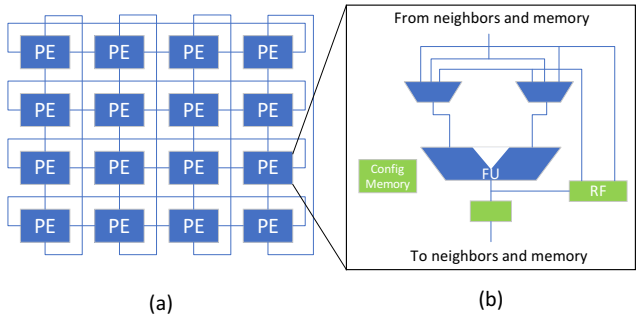


Fig. 1. A typical CGRA (a) grid of PEs, (b) internal architecture of PE

supported operations, network layers, and memory hierarchy. The Processing Element (PE), also named *tile*, is usually composed of a Functional Unit (FU), a register file, and an output register as shown in Fig. 1(b). The Samsung Reconfigurable Processor is certainly the main example of commercial success for the embedded world [6, 12], and Wave Computing relies on this kind of architecture for the topical deep learning domain [18]. As the architecture success depends heavily on the available tool support, the high adherence between the compilation tool and the target architecture further raises the challenge.

Compiling an application on a CGRA consists of finding a valid scheduling and binding of the application's operations on the resources of the CGRA, while respecting control and data dependencies, as well as architectural constraints. The result is a valid *mapping*. The high complexity of applications forces to automate the mapping process. However, scheduling and binding are inter-dependent and NP-complete problems. Hence, increasing the number of resources of the CGRA and the number of operations in the application results in exploding the solution space. More number of resources increase the chances to find a valid solution, but paradoxically, it makes difficult to find a solution. Intuitively, increasing the number of PEs and connections between them makes it easier to find binding solution. However, this increase escalates the computational complexity. While automatically finding a valid solution is a complex task, finding a valid mapping that optimizes some criteria (e.g. latency or area) becomes a real challenge. Efficient and scalable CGRA mapping approaches are thus needed.

While considering real, and thus complex, applications, a valid mapping is built step by step from a *partial* mapping to a new one, by increasing each time step, while guaranteeing the scheduling and binding constraints. The number of partial mappings increases dramatically at each step whereas only few of them can lead to a valid *full* mapping. As it is not possible to know if a partial mapping will lead to a valid solution, an idea is to keep enough partial solutions while controlling their number for scalability issue [4]. Finding the optimal architecture for a specific domain is very difficult and involves many compromises between architectural parameters like topology, RF sizes, CGRA sizes, load/store units, placements of heterogeneous PEs, etc. These parameters define the complexity of CGRAs. Naturally, more complex ones like complex interconnect topologies or bigger register files tend to give better performances than that of less complex ones.

This paper presents a scalable mapping approach with stochastic scheduling and pruning. The stochastic pruning approach proposed in this paper improves a basic stochastic based pruning proposed in [4] which is very limited in terms of the number of operations it can handle. The mapping approach proposed in this paper achieves better scalability, mapping quality and compilation time over the basic stochastic pruning. This paper also introduces stochastic based scheduling and

shows that only a smart stochastic scheduling can give the freedom to travel the mapping solution space efficiently. Indeed, the target architecture strongly constrains the mapping process which objective is to maximize timing performances. Hence, the stochasticity must be smartly managed. This paper shows that a apt exploration of the solution space brings out the best of less complex CGRAs with very negligible performance gap with complex CGRAs.

The contributions of this paper are:

- an improved stochastic pruning leveraging a basic approach proposed in [4] to achieve better scalability, mapping quality and compilation time,
- a new stochastic based scheduling for exploring the mapping solution space better to bring out the best of a less complex CGRA,
- a study of compilation time and the mapping quality of the proposed mapping approach comparing results with the state of the art non-stochastic based baseline mapping,
- architectural exploration of CGRAs to demonstrate efficiency of the proposed mapping approach in terms of scalability and quality of mappings,
- experimental results to show area and energy efficiency over a low power DSP optimized, highly energy efficient RISC-V based in-order processor.

The rest of this paper presents the baseline mapping approach with its limitations in section 2. The proposed mapping approach is detailed in section 3, and results are shown in section 4. The related works are discussed in section 5. Finally, section 6 concludes this paper.

2 THE BASELINE MAPPING APPROACH

2.1 Architecture, Application models and basic mapping problem

In our approach, a CGRA is modeled by a bipartite directed graph with two types of nodes: operator and register, in which temporal aspect is implicitly represented by connections from registers to operators. Fig. 2(a) presents a very simple 2-tile CGRA which model is illustrated in Fig. 2(b). Two subtypes of operator nodes are defined. The first one is conventional operator that represents the physical implementation of an operation. A conventional operator is usually able to compute different types of arithmetical/logical operations (e.g. +, -, &&, ||) and/or memory access (e.g. load/store). The second type of operator is memorization. It is associated to a register and represents the operation of keeping a value in a register explicitly. The connection between the output register and conventional operators depends on the interconnect network (e.g. in Fig. 2, only output registers can communicate with the operator of the other tile).

The application is modeled by a CDFG (Control Data Flow Graph). A CDFG is composed of a Control Flow Graph (CFG) and a set of basic blocks represented by Data Flow Graphs (DFGs). A DFG is a bipartite directed acyclic graph composed of data nodes (rectangles in Fig. 2(c) and Fig. 3), operation nodes (circles) and data dependencies (arcs). In the proposed approach, in addition to conventional operation nodes (+, ×, -), a particular operation node is introduced: *memorization*. The purpose of the memorization node is to make data dependencies explicit along cycles. For example, in the DFG of Fig. 2(c), node $2'$ is a memorization node that makes explicit the data dependency between nodes 2 and 4 over one clock cycle. Memorization nodes are added by graph transformations when necessary (i.e. when an operation has to be postponed). Three equivalences between DFG and CGRA graph models' nodes are defined: (a) data and register; (b) computation and conventional operator; (c) memorization operation and memorization operator. As a result, the two models are homomorphic.

Mapping of a DFG onto a CGRA is therefore a problem equivalent to finding a DFG in the CGRA graph. This problem is known as the maximum common sub-graph problem and can be solved, as in [9, 10, 21] by using Levi's algorithm [15].

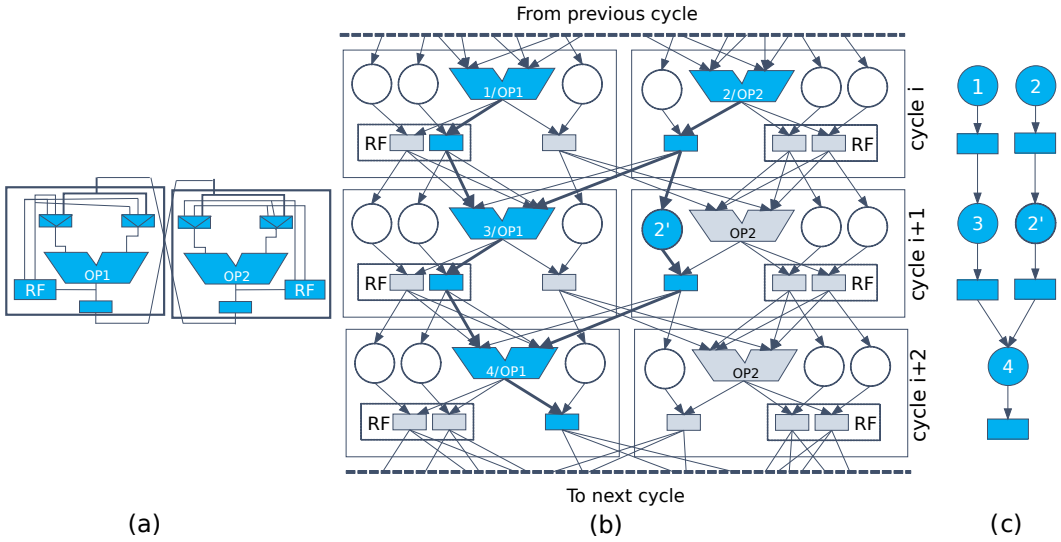


Fig. 2. (a) A 2×1 CGRA with 2 registers in RF, (b) Time extended CGRA on 3 cycles, (c) DFG model. In (b) a possible mapping of (c) is represented in blue. Circles are memorization nodes and rectangles are registers

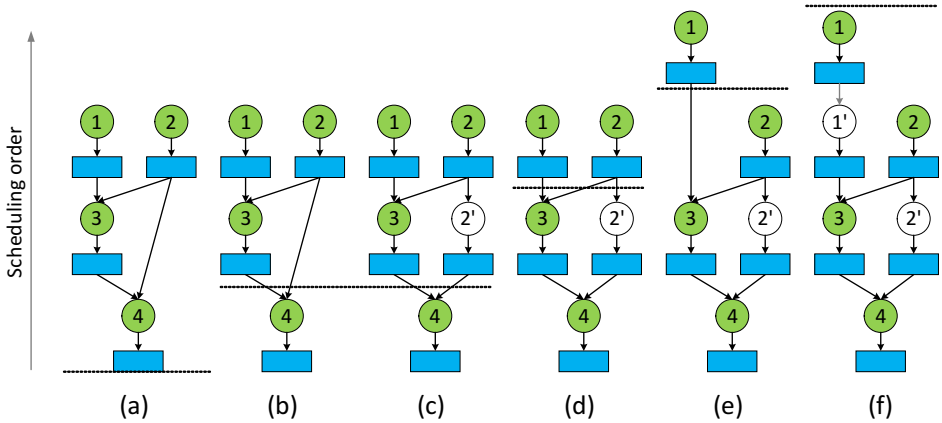


Fig. 3. Example of scheduled and transformed DFG on a one-tile CGRA. (a) Initial DFG, (b) after scheduling node 4, (c) after adding node 2', (d) after scheduling node 3 and 2', (e) after scheduling node 2, (f) Scheduled DFG after routing and scheduling node 1. Horizontal line shows the limit between scheduled and nonscheduled nodes. Memorization nodes are doted circles.

2.2 Baseline Mapping

The baseline mapping flow [20] as presented in the Fig. 4, is composed of four interdependent steps: scheduling, binding, graph transformation and redundant pruning, described as follows.

2.2.1 Scheduling. The scheduling step uses a backward traversal list scheduling algorithm to schedule nodes of each DFG. It relies on a heuristic in which the schedulable operations are listed by priority order. In backward traversal, a node is schedulable if and only if all its children are already scheduled (e.g. node 2, in Fig. 3(b), is not schedulable since node 3 is not yet scheduled; so,

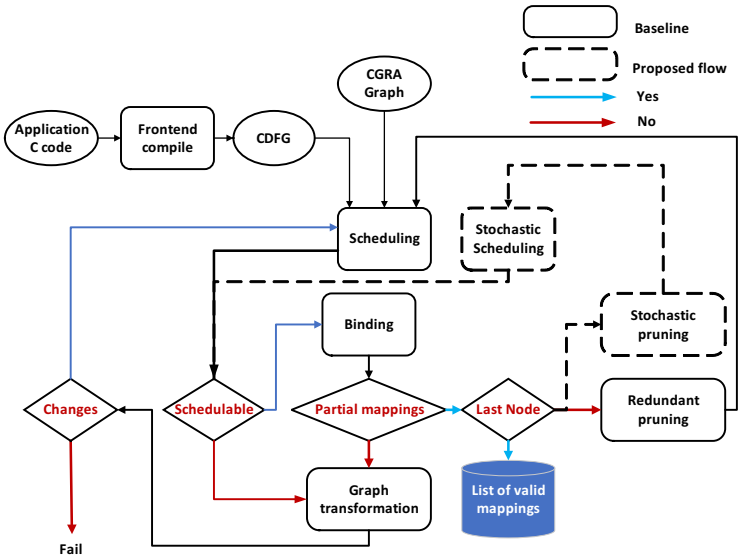


Fig. 4. Baseline and proposed mapping flow

it has to be routed or memorized to keep data dependency resulting in Fig. 3(c)). This approach reduces the number of graph transformations (transformation is performed when a node is not schedulable). The priority of nodes depends on their mobility and number of successors (fan-out). When several nodes have the same mobility, their respective number of successors is used as a second priority criterion. The higher the number of successors, the higher the priority. Indeed, a node with a higher number of successors is more difficult to map due to routing constraint coming from the limited amount of connections between tiles. Thus, scheduling these nodes at first usually allows for reducing the application's latency (e.g. node 2 in Fig. 3(d) has a higher priority than node 1).

As soon as nodes are prioritized and ordered for the current cycle, the baseline approach tries to find a binding solution for each node. The first node is selected from the ordered list and the algorithm searches for at least one binding solution. If there is no free resource for the selected operation, the graph is transformed. Details of the graph transformation are discussed in section 2.2.3.

2.2.2 Binding. The binding algorithm uses an incremental version of Levi's algorithm [15] which finds maximum common subgraph between two graphs. The algorithm adds the newly scheduled operation node and its associated data node to the sub-graph composed of the already scheduled and bound nodes. Only the previous set of solutions that have been kept by the *pruning* step are used to find every possibility to add this couple of nodes without considering the non-yet scheduled nodes (e.g. from the partial bindings obtained in Fig 6(c), the algorithm finds every possible binding with node 2). If no solution is found, there is absolutely no possibility to bind this couple in all the previous partial solutions because Levi's algorithm is an exact method. In that case, graph transformation is required. When the binding step finds possible solutions for the current operation node and associated data node, the partial mappings are provided to the redundant pruning step. For the last operation node, after finding the binding solution, the mapping flow generates a list of valid mappings.

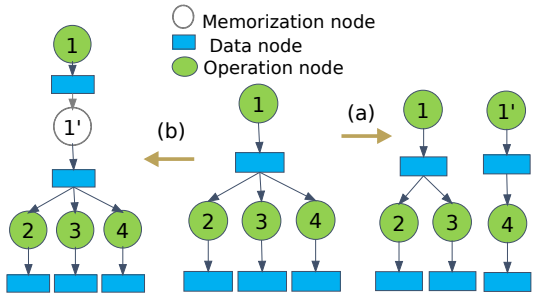


Fig. 5. DFG transformations: (a) Recomputing, (b) Rerouting

Fig. 6(c) describes binding of each operation in the DFG presented in the Fig. 6(a) onto the 2×1 CGRA presented in Fig. 6(b). In this figure, the possible solutions for each operation binding is presented where the currently bound operation and data node is added to the previous sub-graph.

2.2.3 Graph transformation. The DFG is transformed dynamically when a node is not schedulable or when the binding algorithm can not find any solution for the current node. Two graph transformations are available: (a) *recomputing* duplicates an operation node by keeping its same inputs and distributing output edges to reduce the number of successors of the original operation node (see Fig. 5(a)); (b) *Rerouting* adds a memorization node and its associated data node to delay one operation and to preserve data dependencies (see Fig. 5(b)).

There are two situations where a graph transformation is required. The first one occurs when one of the parent operations of a previously scheduled node is not schedulable (e.g. Node 2 in Fig. 3(b) which is one of the parents of node 3). In that case both *Recomputing* and *Rerouting* are available. The choice is made by using a cost function which takes the number of parents, the free resources and the number of successors (e.g. in Fig. 3(b), *Rerouting* is used since the number of free resources is equal to the number of schedulable operations) as inputs and performs a weighted sum. The other situation occurs when the binding algorithm does not find any solution with a couple "operation-data". Two reasons are possible: either no more free operators left in the CGRA or the produced data cannot concurrently reach the already bound successors operations through the interconnection network. In the first case, *Rerouting* is the only available transformation. In the other case, the two transformations are possible. The choice is realized by using the cost function.

2.2.4 Redundant pruning. To reduce the impact of exact binding approach, the baseline mapping flow introduces redundant pruning. The idea is to reduce the exponentially increasing number of partial mappings by removing redundant partial mappings. It removes redundant partial mappings as soon as no more node can be scheduled in the current clock cycle. A partial mapping is redundant when it uses exactly the same operators to make the same operations than another partial mapping at the current scheduling cycle. This step allows to keep only the differential mappings and preserves from an exhaustive search.

2.3 Limitations of the baseline approach

In order to highlight the limitations of the baseline approach, we present the impact of the exact binding approach which uses the sub graph matching algorithm [15]. In Fig. 6, the comprehensiveness of the binding approach is presented. As described earlier, the binding step finds all possible placement solutions for each operation and the associated data. The Fig. 6 shows the increase of solution space after binding of each operation, while mapping the simple DFG in Fig. 6 (a) onto a

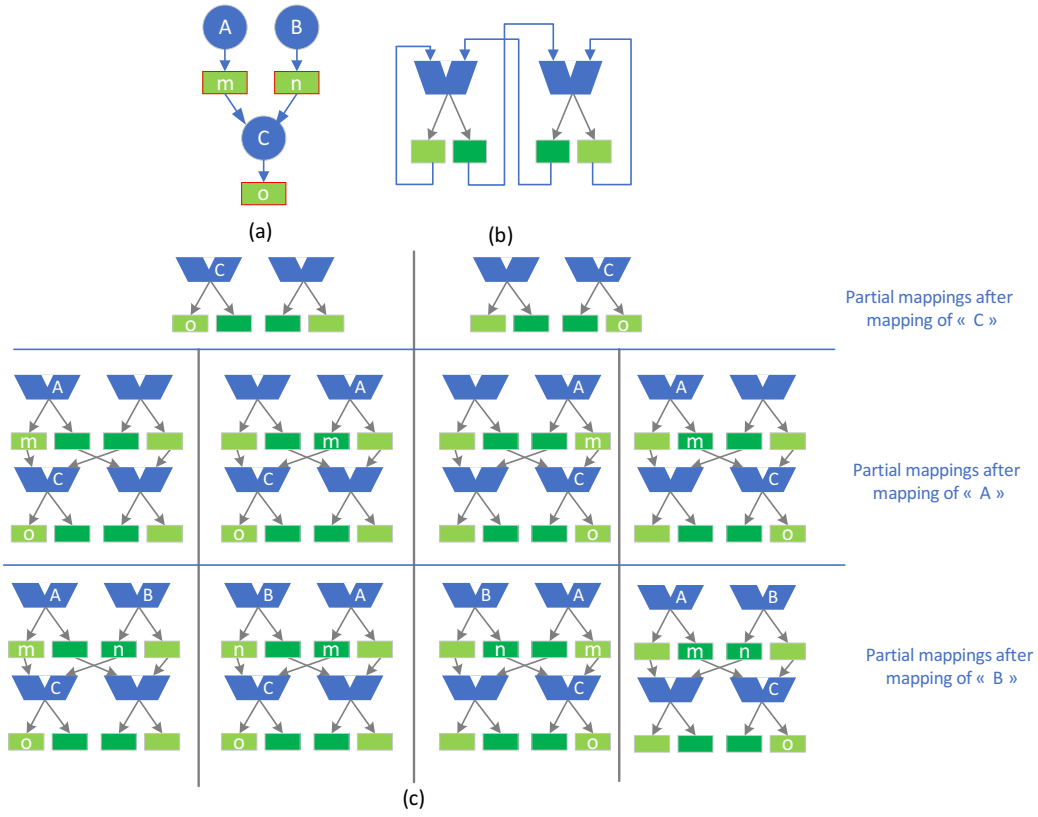


Fig. 6. (a) Sample DFG; (b) 2x1 CGRA with one register in the RF and one output register; (c) Partial mappings after mapping of each operation node.

1×2 CGRA (Fig. 6(b)). For this simple DFG and CGRA the solution space doubled after mapping two operations. In DFGs with higher number of nodes mapped on a bigger CGRA, the solution space increases exponentially.

The increasing number of partial mappings after mapping of each operation badly affects the memory footprint and the compilation time. Fig. 6 simply illustrates the problem through operation locations. In Fig. 7 the memory footprint is presented for mapping a matrix multiplication kernel onto a 4×4 CGRA without any pruning step involved (orange line). Note that the compilation tool implemented in java uses JVM with a maximum 4GB of heap. The figure shows that the exponential use of heap memory by the tool due to the exact binding approach drives the memory usage out of limit after mapping only 7 operation nodes.

The baseline mapping [20] introduces the redundant pruning step to cut off the impact of the exact binding approach which minimizes the memory footprint as shown in the Fig. 7 (blue line). However, for large DFGs, it eventually runs out of memory while mapping. In the Fig. 7, the limit is reached after mapping 33 operation nodes. In this paper, the goal is to go further by not only removing the redundant partial mappings, but also removing stochastically some partial mappings, while keeping a high diversity of these partial mappings to keep a chance that one can lead to a valid full mapping.

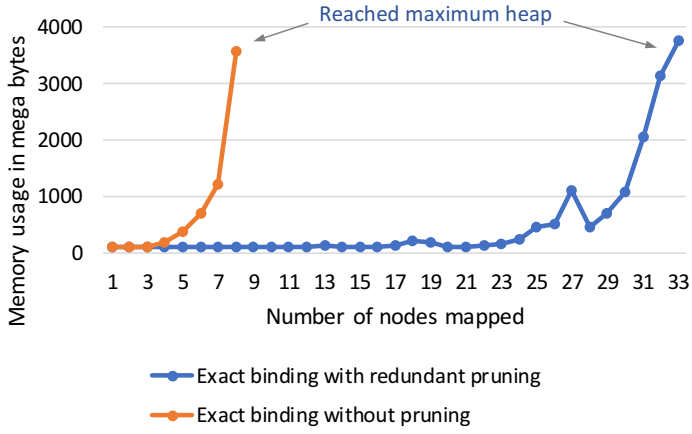


Fig. 7. Memory footprint while mapping with exact binding

3 PROPOSED MAPPING APPROACH

With the increased application complexity, the mapping approach must be scalable in terms of compilation time and memory usage while providing good quality mapping results. In this section, we present a smart stochastic based approach to provide better scalability and capability to efficiently explore the solution space for finding best mapping solutions.

Algorithm 1 presents the proposed mapping algorithm. In the algorithm, operation nodes of the input DFG are stored into *listofOperationNodes*. The stochastic scheduling proposed in the mapping flow takes this list and prepares an ordered list (*orderedList*) for the current cycle. The algorithm then takes the first item in the list and finds all the possible binding solutions in the CGRA graph. The solutions are stored in the *listofValidMappings*, and pruned next using the proposed stochastic pruning step. If no binding solution is found for the operation node, the algorithm transforms the DFG. The *orderedList* is then updated and the *listofOperationNodes* is prepared for the next cycle until all the operation nodes in the DFG are mapped. Finally, the *listofValidMappings* contains a list of valid mappings for the complete DFG. Details of the proposed stochastic scheduling and stochastic pruning is discussed below.

3.1 Stochastic Scheduling

In the baseline mapping approach, the priorities of the schedulable nodes are derived depending on two criteria: (a) the mobility of the nodes, (b) the number of outgoing arcs for the nodes having the same mobility. Despite these two types of criterion, it is possible that several nodes have the same priority (typically, those with same mobility and only one outgoing edge). We propose to randomize the scheduling of the nodes with same mobility and number of successors. This approach facilitates the mapping tool to travel the mapping solution space differently each time it tries to map the DFG, since scheduling order of the same priority nodes are different each time. Thus, stochasticity in scheduling process (Fig. 4) results in a better coverage of the underlying micro-architecture. The ability of this approach to efficiently travel the solution space is shown in the results section (4.2.2).

A *full* mapping is built up cycle by cycle. At each cycle, there are several nodes to be mapped. They are stored in the *listofOperationNodes*. The number of nodes in this list might be greater than the number of available resources at the current cycle. First, priority is given to nodes with a high fanout, assuming that they are more difficult to bind, then the mobility is considered. For

```
Result: listofValidMappings
dfg : Current DFG under mapping;
cgra : CGRA architectural model;
cycles = 1;
listofOperationNodes : operationNodes(dfg);
while listofOperationNodes.isNotEmpty() do
    orderedList = stochasticScheduling(listofOperationNodes);
    while mapped(orderedList.isNotEmpty()) do
        op = FirstNodePriority(orderedList);
        if doBinding(op, cgra, listofValidMappings) then
            stochasticPruning(λ);
        else
            graphTransformation(dfg);
        end
        update(orderedList);
    end
    cycles++;
    update(listofOperationNodes);
end
```

Algorithm 1: Proposed mapping algorithm

nodes with equal priority, the nodes in the *listofOperationNodes* appear in a given order without any scientific reason for this order else than an implementation reason (order of creation of the nodes in the compiler). In other words, considering different orders will allow to open new solution spaces that can then be explored. This is well known and described in CRIMSON paper [1]. Each order (new space) will not always lead to better solutions and can even lead to worse solutions, as shown in Fig. 14. However, at the end, several valid full mappings are found and the best ones are kept and the bad ones discarded. The stochastic scheduling helps to find better quality mappings. After selecting the operation node *op* by the stochastic scheduling, the mapping approach binds the operation using an incremental version of Levi's algorithm as discussed in the baseline mapping approach.

3.2 Stochastic Pruning

The comprehensive nature of binding step usually leads to a high number of partial mappings (depending on the data dependencies and architectural constraints). This prevents to use complex DFGs or CGRAs with large number of tiles or register files. To reduce the number of partial mappings (*nbMappings*) generated by the binding step, we introduce stochastic pruning step. For each partial mapping in the set {*nbMappings*}, the selection step generates a random number between 0 and 1 which is compared to a **threshold**. If the generated number is less than or equal to the threshold value, the partial mapping is kept otherwise the solution is discarded. Since the number of partial mappings (*nbMappings*) depends on the current step, and grows exponentially, choosing a fixed threshold value is not an option. Typically, there are only few partial mappings after the first step, so we must keep most of them, but there are quickly thousands of them after few steps, and many can be discarded. So, the threshold must adapt to the current number of partial mappings. As proposed in [4], a possible threshold is an exponential function which is widely applied in simulated annealing based algorithms. The threshold is defined in Equation 1, where the number

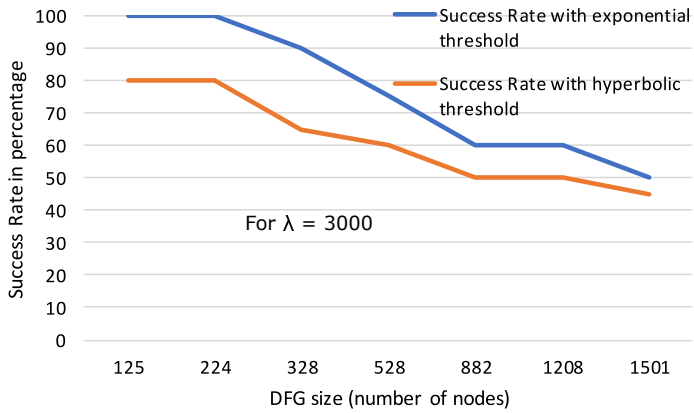


Fig. 8. success rate decreasing with the increase of DFG size

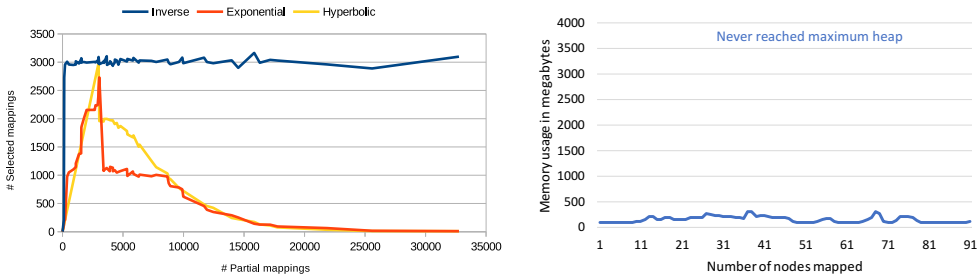
of partial mappings (*nbM*) is normalized with a user defined value λ. This way the function can be tuned from outside while *nbM* will also have a control over the final value of the threshold.

$$Threshold(nbM, \lambda) = exp(\frac{-nbM}{\lambda}) \tag{1}$$

$$Threshold(nbM, \lambda) = \frac{exp(2 * (\frac{nbM}{\lambda})) - 1}{exp(2 * (\frac{nbM}{\lambda})) + 1} \tag{2}$$

However, there may be several other choices of threshold based on different other functions such as hyperbolic, inverse, etc. The ultimate goal of defining the threshold is to have control over the number of partial mappings selected or passed. This number should be low enough to scale up and high enough to allow finding at least one valid solution as very few of the selected partial mappings result in a valid mapping. In other words, success rate highly depends on the choice of the threshold function.

Test results show that while mapping large DFGs with different threshold functions, exponential and hyperbolic based thresholds perform poorly. With the increase of the partial mappings available, the number of selected mappings gets decreased after a certain value in these two thresholds. Fig. 9a presents the decreasing phenomenon of the exponential threshold (equation 1) and hyperbolic threshold (equation 2) function. This figure reports the average number of selected partial mappings (*nbCurrentMappings*) for ten runs with average number of original partial mappings (*nbMappings*) for a λ of value 3000 (the same trend is experienced with several other values of λ). Though the decreasing number of the selected partial mappings in the exponential (equation 1) and hyperbolic (equation 2) threshold helps to reduce the compilation time, the quality of mappings and success rate to find a valid mapping get deteriorated. Fig. 8 presents the decreasing success rate for the mapping with exponential and hyperbolic threshold functions according to the increasing DFG size. The challenge now is to find a suitable threshold function which assists scaling the compilation time and enhances both the quality of mappings and the success rate at the same time. In order to meet all these goals, we propose to use an *inverse* function based threshold which is presented in



(a) (b)

Fig. 9. (a) Performance of threshold functions (we choose inverse function for its stable selection capability), (b)Memory footprint while mapping with stochastic pruning

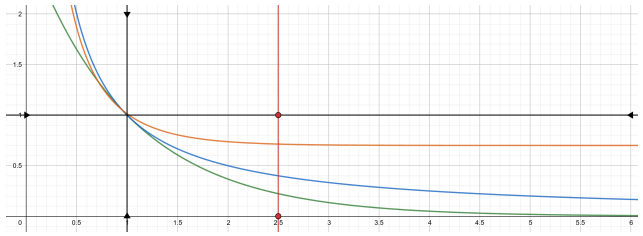


Fig. 10. Plot of the three decreasing functions considered. X axis is nbM/λ. Y axis is the theoretical probability to keep a mapping. Green: inverse exponential, orange: hyperbolic cotangent, blue: inverse function

Equation 3.

$$Threshold(nbM, \lambda) = \begin{cases} (\lambda/nbM) & if\, nbM > \lambda \\ 1 & if\, nbM \leq \lambda \end{cases} \tag{3}$$

The performance graph in Fig. 9a demonstrates the adaptivity of the inverse threshold. As explained earlier, while we experience exponential decay in selected number of mappings in case of exponential and hyperbolic function, the inverse function offers steadiness. The success rate with this threshold also was 100% in all of our experiments.

Notably, with the proposed stochastic pruning approach the memory footprint decreases drastically compared to the baseline approach and non stochastic based approach. Fig. 9b shows the memory footprint while mapping the matrix multiplication after introducing the proposed stochastic pruning. As shown in this figure, the memory usage never reaches the maximum heap size, and the memory usage stays at a reasonable level throughout the mapping.

In an attempt to try to understand why the inverse function works better than the two other candidates, we simply plotted the three mathematical functions. Figure 10 shows the plots of the three functions. The variable x is nbM/λ. The line at $x = 1$ indicates when $nbM = \lambda$. On the left part of this line, all mappings are kept, so the threshold functions do not hold. The interesting part is on the right part of this line. The line at $y = 1$ indicates the theoretical probability to keep a mapping. Considering a uniform distribution, a random number is compared to the threshold function. The mapping is kept if the number is less than the value of the function. Note that this figure is a representation of what happens in the tool and not the exact implementation. The theoretical trends show that as the number of mappings increases, the hyperbolic function tends to the probability to

keep most of them, the exponential function tends to the probability to discard most of them, and the inverse function stands between the two, with a smooth decreasing probability to discard the mappings. The red line at $x = 2.5$ shows an example of such a behavior, where the probability is around 70% for the hyperbolic function, 22% for the exponential, and 40% for the inverse function. As a conclusion, the hyperbolic function leads to high memory footprint as it keeps too many mappings, the exponential function discards too many mappings, reducing thus the chances to keep the partial mappings that will lead to a valid mapping. The inverse function presents the good trade-off between these two.

Finding a valid mapping solution depends on the number of selected mappings. Although the inverse threshold helps to control memory footprint by reducing the number of selected mappings, some heuristics is necessary to ensure a good number of selected mappings is maintained throughout the mapping. We propose to introduce bounds as control mechanisms: *LB (Lower Bound)* and *UB (Upper Bound)*. While randomly selecting the partial mappings from a set, it might so happen that the pruning function did not not select any one of the partial solutions failing to find any valid solution. Hence, it is absolutely necessary to set a minimum number (lower bound) which the pruning function must select from the solution space. This gives an opportunity to find a valid mapping in the end. If the selected partial solutions does not reach to the lower bound, the pruning function will iterate through the solution space until the number is reached. In this iterative selection process, the function might select huge number of solutions which will increase the compilation time. To get the compilation time scalable and success rate high, we introduce both *upper* and *lower* bound. However, the upper bound may impact the quality of mapping by over-constraining the selection process. In this article, we investigate the impacts of the different bound based pruning approaches for the best trade-off between quality of mapping and compilation time.

Through a set of experiments in the following section we select one variant between the two bounds based on their capability to find best quality mappings and compilation times for different benchmarks.

LB&UB. sets a lower bound and upper bound on *nbCurrentMappings* as presented in Equation 4 and 5. In this method also, a random number is generated between 0 and 1, which is compared to the threshold value. If the random number is less than or equal to the threshold or the lower bound is not satisfied, then it selects the partial solution from *nbMappings* and stores into *nbCurrentMappings*, otherwise the solution is discarded. If *nbCurrentMappings* exceeds the upper bound, then it stops selection of partial mappings (and all the remaining partial solutions are discarded).

$$\max nbCM = \lceil nbM/3 \rceil \tag{4}$$

$$\min nbCM = \begin{cases} \lfloor nbM/\lambda \rfloor & \text{if } nbM > \lambda \\ \lceil nbM/3 \rceil & \text{if } nbM \leq \lambda \end{cases} \tag{5}$$

LB only. generates a random number between 0 and 1 which is compared to the *threshold*. If the random number is less than or equal to the threshold then it selects the partial solution from *nbMappings* and stores into *nbCurrentMappings*, otherwise the solution is discarded. The solution space *nbMappings* is traversed again and again until *nbCurrentMappings* reaches the minimum bound as presented in the Equation 6.

$$\min nbCM = \lceil nbM/\lambda \rceil \tag{6}$$

The efficiency of the inverse function in terms of mapping quality and compilation time is presented in the next section.

4 EXPERIMENTS AND RESULTS

4.1 Experimental setup

The proposed mapping flow has been fully automated through a software tool implemented by using Java and Eclipse Modeling Framework (EMF). GCC 4.8 is used to generate CDFGs from applications described in C language. We developed a GCC plugin to parse the intermediate representation (IR) of GCC and produce an equivalent IR in the Java world. Nine applications from signal processing domain have been used for our experiments and are detailed in Table 1. A workstation with an Intel Xeon cpu @ 3.50 GHz × 4 and 8 GB of RAM has been used for the experiments.

We have considered the following methods for comparison. The methods are categorized into two classes, one which maps the application without pruning, and the other which uses pruning. In our comparison, methods **WP1** and **WP2** represent methods without pruning (WP). The (a) WP1 [14] solves the scheduling and binding problem separately. DFGs are transformed in the scheduling process by applying "simple route" transformation. A forward list scheduling algorithm and the original Levi's binding algorithm are used in the mapping. (b) WP2 [9] traverses the DFG forwardly, schedules nodes by applying a-priori transformation and binds using original Levi's algorithm. All the methods which employ pruning in the mapping process perform backward graph traversal, list scheduling, exact binding with dynamic graph transformations and pruning. They differ from each other by the way the pruning step is handled. (c) *RED* (**Redundant** deletion) [21] removes redundant partial mappings to prune the solution space in the baseline approach. (d) *SNoB* (**Stochastic** selection with **No Bounds**) uses stochastic method without any bounds to prune the solution space. (e) *SLUB* (**Stochastic** selection with **Lower** and **Upper Bounds** (LB&UB)) uses stochastic method with upper and lower bounds for pruning the solution space. (f) *SLoB* (**Stochastic** selection with **Lower only Bound** (LB)) uses stochastic method with only lower bound to prune the solution space. (g) *SE* (**S**tochastic selection with **E**xponential threshold) uses stochastic method with exponential threshold function [4]. The comparison is based on the latency of the mapped DFG (quality of mapping) and compilation time. Also, since all these methods are based on stochasticity, we present the best results based on ten executions.

To study the impact of the proposed stochastic mapping approach on different CGRA architectures we performed a comprehensive architectural exploration for CGRAs. The goal of this study is to show that the stochastic mapping can find the best quality mapping for a given architecture, which in turn proves its flexibility. As architectural exploration of CGRA involves many trade-offs, choosing the right combination of parameters is hard. Moreover, an architecture instance might work well for one set of applications but not for others. Therefore, finding the optimal CGRA architecture instance

Table 1. Benchmarks considered and characteristics

Benchmark	nodes	ASAP	Parallelism
2D Discrete Cosine Transform (DCT-2D)	711	81	32
matrix product	504	98	32
Fast Fourier Transform (FFT)	1348	37	64
Trapezoidal (Trapez) filter	332	59	32
Exponential Moving Average Filter (EMA)	412	99	38
Moving Window De-convolution (MWD)	440	112	32
Unsharp Mask	91	27	16
Elliptic Filter	130	31	16
DC Filter	507	96	32

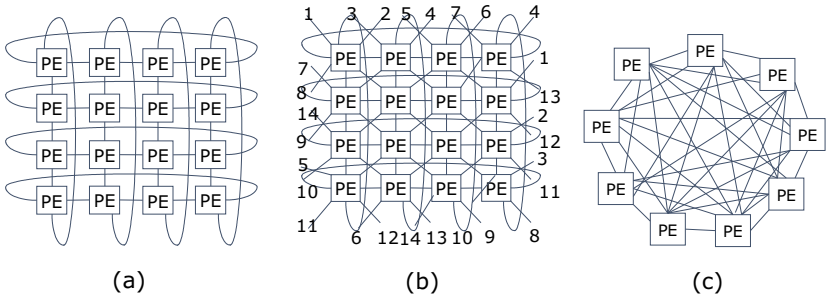


Fig. 11. Topologies: (a) mesh torus, (b) mesh-x-torus, (c) fully-connected

from the wide design space is very difficult. Instead, we have tried to explore the effects of varying important architectural parameters like: size of CGRA, RF size, interconnect topology, number of load/store units, placement of load/store units and placement of multipliers. This approach gives wide variety of trends involving several architectural parameters and directions to design an optimal CGRA architecture. In the experiments we assume the latency for all operations (operators and load/store operation) to be one. The topologies we have considered in our experiments are mesh-torus (Fig. 11(a)), mesh-x-torus (Fig. 11(b)) and fully-connected (Fig. 11(c)) (for clarity we have presented the fully connected topology for a 3×3 CGRA).

Finally, we explore the area overhead, performance and energy gain for the best suitable architecture found from the architectural exploration, comparing with a low power RISC-V based processor [7] which is highly optimized for DSP applications. The chosen RISC-V processor is an in-order 4-stage RISC-V CPU supporting SIMD extensions, custom DSP instructions, and misaligned load support. These features extensively reduce the bandwidth requirements for data memory and increase the computational efficiency. The core is highly optimized for DSP benchmarks, making it a good candidate for a fair comparison with the CGRA architectures.

4.2 Results

4.2.1 Comparison of the different methods. For this comparison we have considered 3x3 homogeneous CGRA with RF size 24. The CGRA is assumed to have infinite memory band-width. In other words, all the processing elements in the CGRA incorporate load-store unit and have access to the data memory without access contention. As the trends are similar for different sizes of CGRA and RF, we have presented results for only 3x3 CGRA with RF 24. Fig. 12 and Fig. 13 compare mapping latencies and compilation time respectively. In the comparison of mapping latencies we have normalized the resulting values with the ASAP (*as soon as possible*) length of the DFGs. Since the ASAP length of a DFG represents the minimum schedule length or minimum possible latency, this comparison helps to realize mapping quality of the methods. Results in Fig.12 shows that the methods without pruning (WP1 and WP2) are unable to find solutions for DFGs with higher number of nodes like fully unrolled FFT, DCT, Matrix Multiplication. WP1, that solves scheduling and binding totally independently performs the worst. WP2, which transforms the graph a priori, proves to be better than WP1. However, the results in the stochastic pruning based methods are far better than these two methods. Among the pruning based methods, *SLUB* approach performs the worst. This is due to over-constraining the solution space. The *SloB* approach performs the best. This proves that eliminating all the redundant partial mapping in *RED* approach is not always good, since the partial mappings may be using different registers of the same operator which can produce a good mapping. In the *SloB* approach, it is evident that introducing randomness in the selection

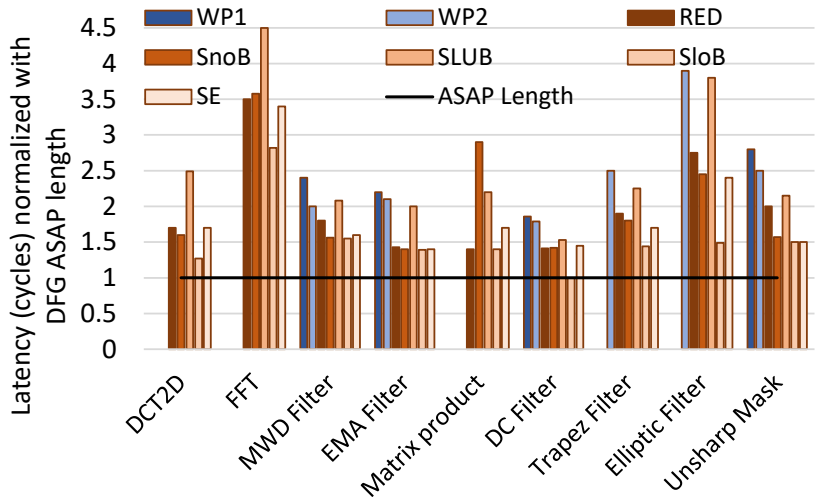


Fig. 12. Mapping latency comparison for 3x3 CGRA - RF24

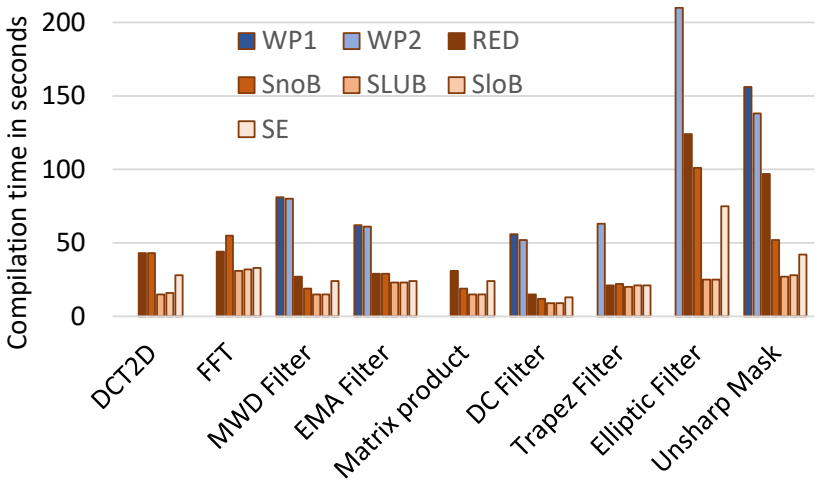


Fig. 13. Compilation time comparison for 3x3 CGRA - RF24

process yields better results. In [21], *RED* claims to obtain better results than EPIMAP [9] and REGIMAP [10]. To realize the gain in compilation time for the stochastic pruning based approaches, we have presented the absolute compilation time for different methods in Fig. 13.

Fig. 12 shows the ability of a method to find quality mappings. The closer the latency value to the ASAP line, the better the mapping is. The number of operation nodes present in the DFGs are shown in parenthesis for each kernel. What strikes at first is that the methods are unable to find good solution for the FFT kernel. It is because the DFG of FFT possesses huge parallelism which can only be satisfied with huge amount of resources. We have carried out further experiments for the FFT and needed a CGRA of up to 6x6 tiles with RF 16 to reach the ASAP. The latency comparison

in Fig. 12 depicts that *SLoB* provides the best quality of mappings whereas *SLUB* produces the worst latencies. The compilation time comparison in Fig. 13 shows that *SLUB* and *SLoB* achieved best scaling. Hence, we choose *SLoB* for the next set of experiments.

4.2.2 Integrating stochastic scheduling. In this section we present performance improvement after integrating stochasticity in the scheduling step of the mapping approach. We introduce *SLoBS* which integrates ***stochastic scheduling*** in *SLoB*. As a result, **SLoBS** improves latency up to 22.75% for FFT, while never leading worse latency and without impact on runtime. Stochastic scheduling along with stochastic pruning helps better architecture exploration. Fig. 14 exhibits this argument. It presents the results for ten runs of the DCT benchmark on different CGRA configurations with three different methods, *RED, SLoB and SLoBS*. As the trend is the same for all other kernels, the results for only one kernel is presented here for the sake of clarity and better understanding. Each point in the Fig. 14 corresponds to one run by a method on the corresponding CGRA configuration. The x axis of the graph represents latency normalized to ASAP length and the y axis represents the number of transformed nodes normalized to the number of operation nodes in the original graph. So, each point in the graph is basically the outcome latency and number of transformed nodes of each run by a certain method. The points corresponding to the method *RED* and *SLoB* show that they found similar latencies with almost similar number of transformations. The wide varieties of latencies and wide varieties of transformations of method *SLoBS* prove that this method can better explore the solution space. The nodes with similar priority are scheduled in different order for each run in the case of SloBS. This helps to explore different possibilities in the mapping latency and DFG transformations which are already existing. Hence, the method *SLoBS* found the best latency with least number of transformations. We have used the *SLoBS* approach for the next set of experiments targeting architectural exploration.

4.2.3 Scalability of the approach. We present the average compilation time comparison for different CGRAs between our previous work (SE) and the proposed work (SLoBS). In these experiments, the average compilation time is computed by taking the mean of compilation time for the kernels with the corresponding approach. Fig. 15 shows that the proposed approach **SloBs** has lower average compilation time compared to the baseline approach. With SLoBS, the mapping time slowly increases in a log scale due to less DFG transformations as discussed in the earlier section. On the contrary, the SE mapping time increases stiffly after 4×4 CGRA.

We present the latency and compilation performance of SLoBS for different kernels in Fig. 16 (a) and (b) respectively. We have considered 3×3, 4×4, 5×5, and 6×6 with RF8 in this comparison. Since we focus on small sized CGRAs for ultra-low power embedded applications, we considered the CGRAs sizes up to 6×6. The latency and compilation time are chosen from the best out of 10 runs for each kernel presented. The primary y axis represents latency in clock cycles and secondary y axis depicts compilation time. For all the kernels, the proposed approach achieves ASAP schedule for 6×6 CGRA with almost similar compilation time for 4×4 and 5×5 CGRAs. In our experiments, we show mapping of wide range of filter application which performs in the target CGRA with high energy efficiency. In the literature [22], most of the traditional CGRAs used in embedded domain uses less than 64 PEs. CGRAs targeting HPC application domain offer higher number of PEs.

SLoBS finds the solutions for different CGRA configurations except for FFT in 3×3 CGRA. This is due to FFT kernel's complexity as to DFG size and parallelism available which could not fit in the target small CGRA. However, the method finds the mapping solution for FFT on bigger CGRAs and finally it achieves the ASAP length on 6×6 CGRA. It is interesting to notice that for almost all the kernels (except FFT) on 4×4 CGRA, the method finds solutions closer to the ASAP length. Hence, there is negligible latency improvement in bigger CGRAs.

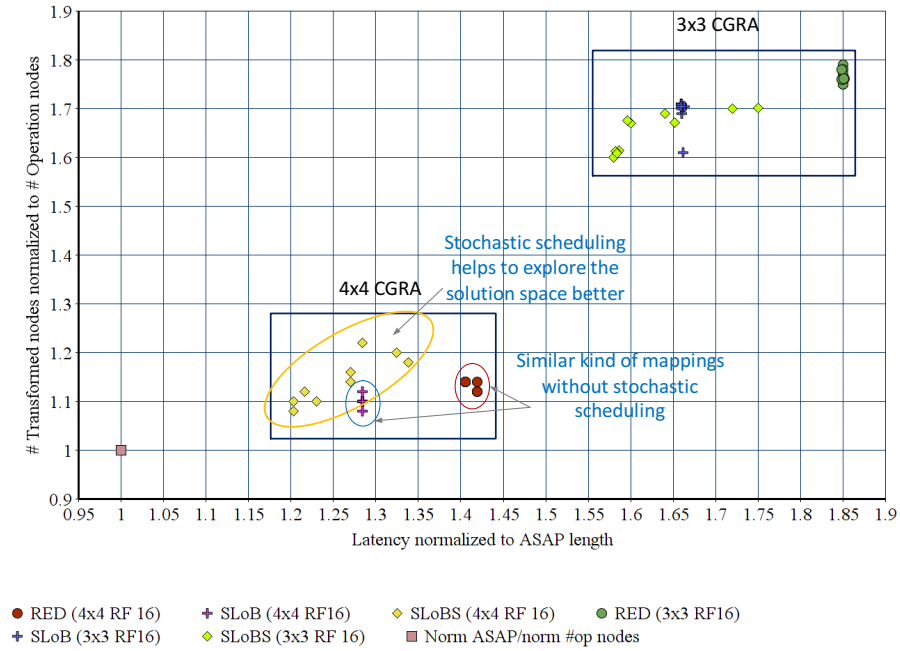


Fig. 14. Architectural coverage between methods for ten runs of the DCT benchmark on different CGRA configurations

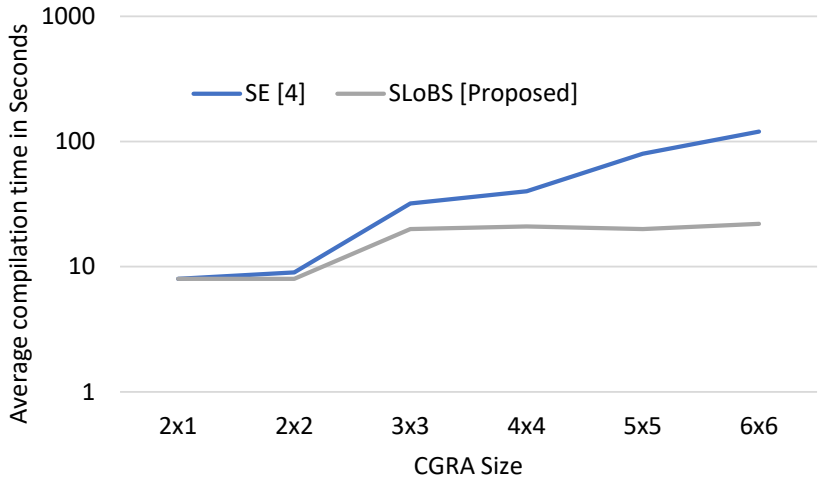


Fig. 15. Average compilation time comparison for different CGRA sizes

4.2.4 Experiments with topology, CGRA size and RF size. Interconnection topology and RF size are important dimensions in CGRA architecture because schedulability largely depends on these two parameters. Intuitively it is easier to map kernels on a CGRA with richer interconnections and big

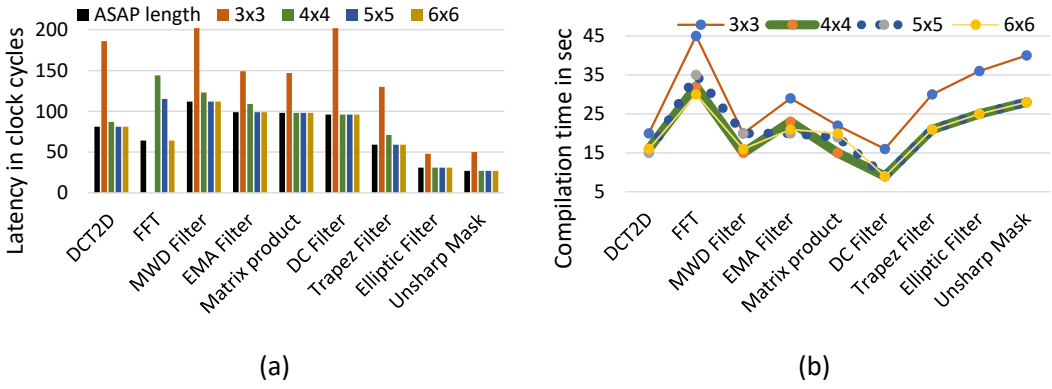


Fig. 16. (a) Latency achieved and (b) compilation time for different CGRA sizes, each having RF8

RF sizes. However, through our experiments we show that with the proposed mapping approach less complex interconnection with less RF pressure can have significant performance achievement. To exhibit this, we have considered a 4x4 homogeneous CGRA with infinite bandwidth (CGRAs with different sizes have similar trends). Fig. 17 represents latencies for the respective CGRA configurations normalized to ASAP length. The distance from the ASAP line in this graph devolves the performance of the corresponding configuration. The latencies closer to the ASAP length represent better performance. A latency of 0.5 means that the specific configuration is unable to find a mapping solution. Fig. 17 shows that increasing the RF size does not lead to big performance gain. After a threshold in the RF size (which depends on the application), the performance is almost similar. With the increase of interconnection complexity, performance is increased but the small performance gains encourage to use less complex topology. We have also found that the difference between the number of transformed operation nodes after mapping for different CGRA size is very less. As an instance, mapping on 4×4 CGRA results an average of 1.09% transformed operation nodes, for 5×5 CGRA this number is 1.08% and for 3×3, 1.18%. The dynamic transformation and stochastic selection in our mapping approach ensure maximum use of resources resulting in very small performance gain. The ability of the tool to find good solutions without assuming huge connectivity and huge RF sizes says that the CGRA architecture scales pretty well.

4.2.5 Experiments with the number of load/store (LS) units. Memory bandwidth is an important metric for a real implementation of CGRAs. In the previous set of experiments, we assumed that all CGRA instances had infinite bandwidth (i.e. all the PEs are capable of load/store operations). Here we limit the number of LS units in the architecture. In our experiments for 4×4 CGRA the number varies from 4 (top row) to 8 (top two rows). All the CGRAs have an RF of size 24. Fig. 18 presents latencies normalized to the latency of CGRAs with infinite bandwidth. We have experienced that for smaller CGRAs (i.e. 3×3) increasing the number of LS units does not help improving performance. Larger CGRAs (Fig. 18) in contrast have significant performance gain with increased number of LS units. So, if we go for a trade-off between the size of CGRA and the number of ls units, large CGRAs with smaller number of LS units have better performance than small CGRAs with large number of LS units. And this is an interesting information, because it indicates that huge aggregate bandwidth is not necessarily needed at the interface of the CGRA.

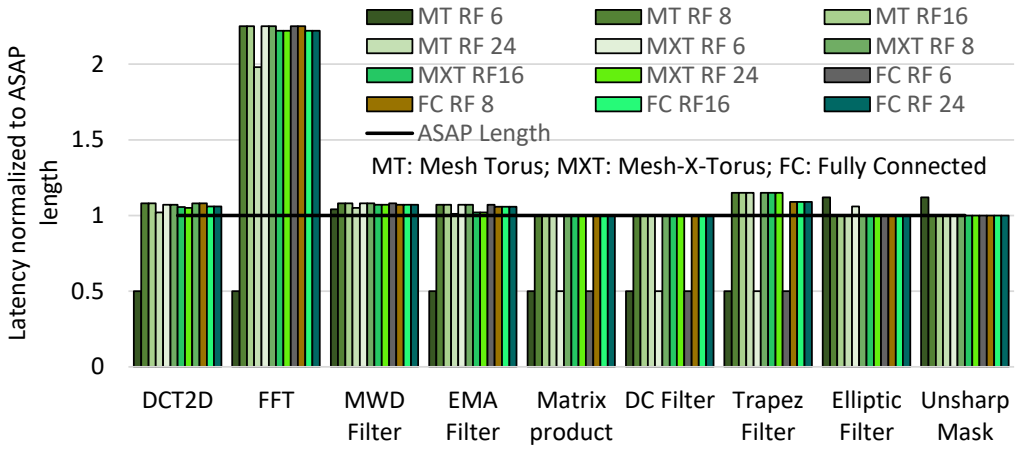


Fig. 17. Latency comparison for 4x4 CGRA

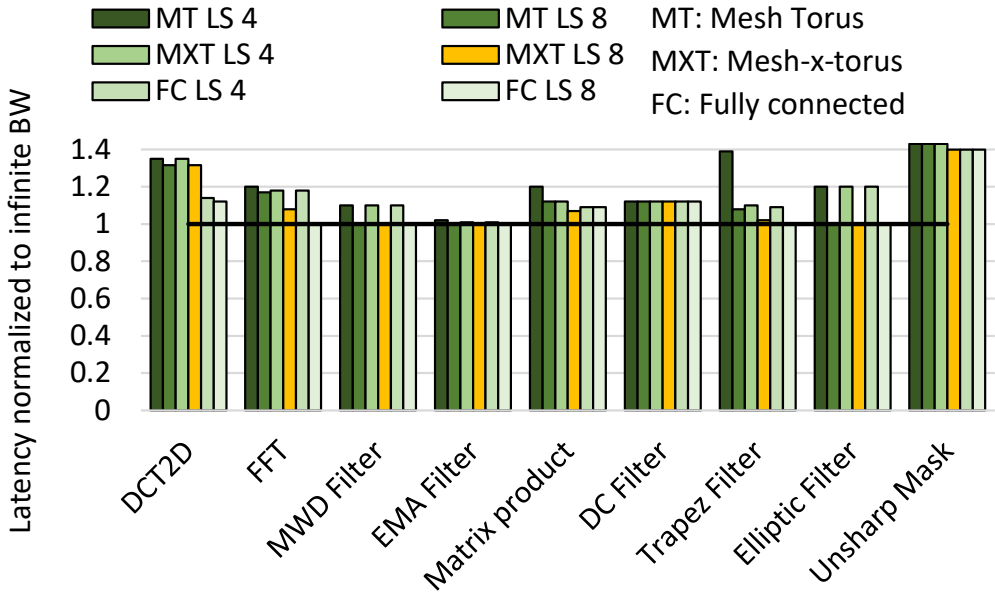


Fig. 18. Latency comparison for 4x4 CGRA with limited bandwidth

4.2.6 Experiments with the position of load/store (LS) units. In CGRAs with limited bandwidth placement of LS units plays an important role in improving schedulability. Better reachability of LS units to other PEs helps improving performance. In this set of experiments, we have considered three arrangements of LS units: row wise placement (Fig. 19(a)), diagonal placement (Fig. 19(b)) and zigzag placement (Fig. 19(c)).

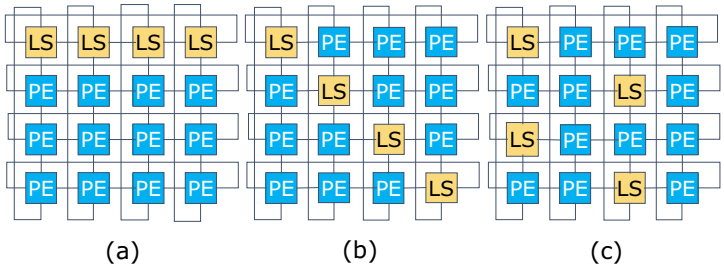


Fig. 19. Different placements of load/store nodes

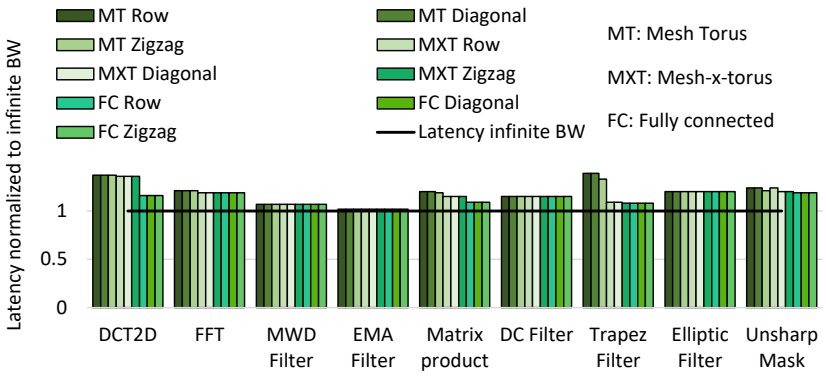


Fig. 20. Latency comparison between different placement of load/store units

Fig. 20 presents the latency comparison between several placements of LS units for a 4x4 CGRA with RF8 and mesh torus topology (trends are similar for other configurations). In most of the cases, zigzag placement of load store nodes performed well for all the topologies and in few cases zigzag arrangement in less connected topology reached the performance of fully connected ones. This is solely because the zigzag placement of the load/store units gives better reachability to all the PEs.

4.2.7 Area, performance and energy comparison with CPU. This section describes the implementation results for the designed CGRA. The choices are: 4×4 PE array, RF of size 8 in each tile, mesh torus connection, 4 load-store units arranged in zigzag manner. To present the area overhead we considered RISC-V CPU [7]. This core is highly optimized for DSP benchmarks and features SIMD extensions, including dot-product and shuffle instructions, and misaligned load support that greatly reduce the load-store traffic to data memory while maximizing computational efficiency. In [7], it is reported that in a low power 28 nm FD-SOI process a peak energy efficiency of 193 MOPS/mW (40MHz, 1 mW) can be achieved by the core. In our experiments, both the CGRA and CPU core were synthesized with Synopsys design compiler 2014.09-SP4 using STMicroelectronics 28nm UTBB FD-SOI technology libraries. Synopsys PrimePower 2013.12-SP3 was used for timing and power analysis at the supply of 0.6V, 25°C temperature, in typical process conditions. The cycle information was achieved simulating the RTL with Mentor Questa Sim-64 10.5c. For area comparison, the CPU includes 16 kB of data memory, and 1 kB of instruction cache, which is equivalent to the 16 KB of data memory and 4 KB of instruction memory used for the CGRA in the experiments. As a result, the CGRA has an area overhead of 1.5× compared to that of the

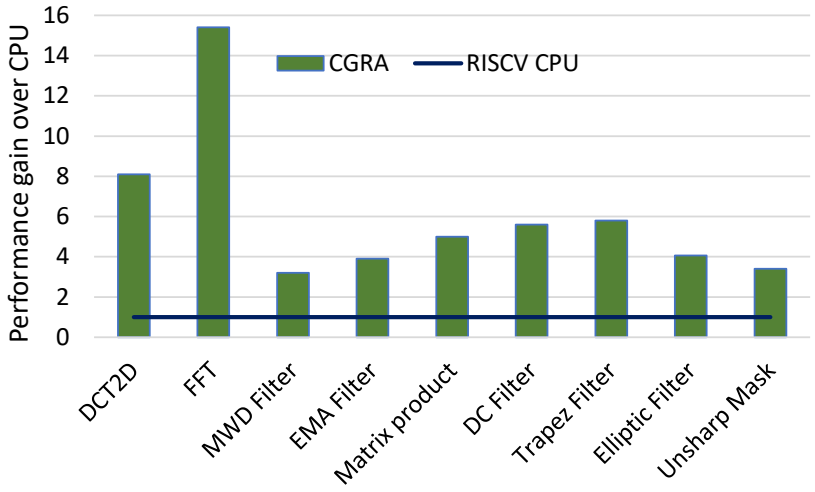


Fig. 21. Performance comparison with cpu

Table 2. Energy consumption (nano Joule) comparison with cpu

Kernels	CGRA	RISC-V CPU	Energy gain
DCT	0.74	11.89	16.1x
Matrix_mul	1.12	10.91	9.7x
FFT	1.12	33.28	29.7x
Trapez Filter	0.66	7.09	10.7x
EMA filter	0.74	5.42	7.3x
MWD	0.8	5.01	6.3x
Unsharp mask	0.27	1.78	6.6x
Elliptic filter	0.39	3.19	8.2x
Average	**0.73**	**9.82**	**13.5x**

CPU. Fig. 21 presents the execution time comparison in cycles for different kernels normalized to CPU performance. Since the DFG of FFT kernel exposes highest parallelism, a maximum of 15.28× performance gain is achieved in the CGRA. However, the CGRA achieves an average of 6× performance gain compared to the RISC-V CPU.

Table 2 compares the energy consumption in nano Joule (nJ) while running different kernels in the CGRA and CPU. The CGRA consumes an average of 13.5× (maximum of 29.7× and minimum of 6.3×) less energy compared to the execution in CPU.

5 RELATED WORK

As CGRAs are classically used to speed up loop kernels, early methods in the literature focused on modulo scheduling and minimizing the initiation interval of a loop to maximize the throughput [5, 14, 17, 19, 28]. AURORA [23] relies on HyCUBE [11], which is a heuristic for modulo scheduling. In AURORA, the stochastic process is used to explore the architecture, not to map the application. This paper proposes a solution for temporal mapping in general, not for modulo scheduling only.

Some mapping approaches concern *spatial* mapping only, because the PEs are not time-shared [8, 25]. The methods then need to solve the binding problem only, i.e. assigning a PE to a given operation. The simulated annealing is one of the most used technique [17, 25] in that context.

When the PEs of the CGRA are time-shared, the mapping methods can be roughly divided into two main categories which differ on the way scheduling and binding steps are realized, i.e. sequentially or simultaneously.

The first category simplifies the mapping problem by solving scheduling and binding sequentially with heuristics and/or meta-heuristics [14, 19] or exact methods [9, 10]. In [14], iterative modulo scheduling heuristic is used for scheduling. In [19] an edge-based binding heuristic is used, instead of classical node-based approaches, to reduce the number of fails. The method presented in [14] starts by finding a solution on a simplified problem with heuristic-based methods for both scheduling and binding and then tries to improve the initial solution with a quantum-inspired evolutionary algorithm (QEA). In [9, 10], authors solve the scheduling and the binding problems sequentially by using respectively a heuristic and an exact method. Scheduling is made implicitly by integrating both architectural constraints and timing aspects into the DFG by statically transforming it. Two transformations are proposed: (1) *recomputing* that duplicates computation nodes and (2) *rerouting* that duplicates data nodes to make explicit the conservation of a result. These transformations are performed in the hope of facilitating the mapping of the application. The binding is performed by finding the common sub-graph between the transformed DFGs and a time extended CGRA with Levi's algorithm [15]. The transformations allow for finding a better mapping than [19]. However, since the graph transformations are done a priori, it is very difficult to know which transformation is relevant at a given time. This reduces the ability of the method to efficiently explore the solution space since the problem is over-constrained. In [28], transformations are performed in the polyhedral domain to optimize the initiation interval.

The second category of approaches solves the scheduling and binding problems as a whole. Hence, [2, 3] uses exact methods, e.g. ILP-based algorithms, to find optimal results. Unfortunately, these methods suffer from scalability issues as illustrated in [14]. A key feature of meta-heuristics is also a guided stochastic algorithm. The meta-heuristics can be divided in population-based approaches or local search-based approaches. Several population-based meta-heuristics have been used to solve the mapping problem, like Genetic Algorithms [13] or QEA [14]. Among local search techniques, mostly Simulated Annealing (SA) has been used, like the method presented in [17] and its extension that performs register allocation [5]. Another random-driven algorithm is presented in [1]. Randomness is used in the scheduling step to produce a valid schedule. Then the place and route step is performed. The authors also interestingly show that this approach allows to better explore the solution space.

In [21], graph transformations help again in finding better mappings. However, this approach is not scalable. To deal with scalability, an interesting approach presented in [26] detects the repetitive patterns in the graph, and maps in a hierarchical manner the complex loops on the CGRAs. This method is more efficient than a naive full loop unrolling technique that leads to unscalable and intractable large DFGs.

This work gets together the advantages of graph transformations and simultaneous scheduling and binding. The key novelty of the proposed mapping approach is to make use of a stochastic process (both for scheduling and binding) and graph transformations on-the-fly to better explore the solution space of the mapping problem. This helps to adapt the application to the architecture instead of enhancing the architecture to execute efficiently the application. This paper shows that an inverse function based approach performs better than an exponential function based approach typically used in simulated annealing. The scheduling relies on a random-based heuristic algorithm. The binding is based on an exact method. A stochastic-based pruning step then selects a limited

number of partial solutions. Transformations of the formal model of the application are performed dynamically when no partial solution is found. The combination of the whole makes the approach scalable while keeping good quality results.

6 CONCLUSION

This paper presents a stochastic based approach to map applications on a CGRA. The efficiency and scalability of the method was shown through experiments under several architectural parameters of a CGRA. The stochastic based pruning approach with inversion based threshold and lower only bound helped to achieve best results in terms of quality of mapping and compilation time. Furthermore, stochastic based scheduling approach helped to explore the architectural space better which in turn helped to find the best mapping solution. The experimental results showed that the method can find the best latencies in most of the cases, provided that enough parallelism is present in the benchmarks. Our approach applied functionally-invariant transformations to the application graph to better match the CGRA architecture. Results showed that we can reach same performance on simple interconnected CGRAs as complex ones. The experiments also revealed that the placement of the load/store units is more important than their number. Very small increase in the operation nodes after mapping adds another credit to the combined use of dynamic graph transformation and stochastic pruning. The efficient and flexible CGRA mapping approach presented in this paper helps to explore the CGRA as low power accelerators.

REFERENCES

[1] Mahesh Balasubramanian and Aviral Shrivastava. 2020. 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* 39, 11 (2020), 3300–3310. https://doi.org/10.1109/TCAD.2020.3022015

[2] Janina A Brenner, JC Van Der Veen, Sándor P Fekete, J Oliveira Filho, and Wolfgang Rosenstiel. 2006. Optimal Simultaneous Scheduling, Binding and Routing for Processor-Like Reconfigurable Architectures. In *FPL*. 1–6. https://doi.org/10.1109/FPL.2006.311262

[3] S. Alexander Chin and Jason H. Anderson. 2018. An Architecture-Agnostic Integer Linear Programming Approach to CGRA Mapping. In *Proceedings of the 55th Annual Design Automation Conference* (San Francisco, California) *(DAC '18)*. Association for Computing Machinery, New York, NY, USA, Article 128, 6 pages. https://doi.org/10.1145/3195970.3195986

[4] Satyajit Das, Thomas Peyret, Kevin J. M. Martin, Gwenolé Corre, Mathieu Thevenin, and Philippe Coussy. 2016. A Scalable Design Approach to Efficiently Map Applications on CGRAs. In *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 655–660. https://doi.org/10.1109/ISVLSI.2016.54

[5] Bjorn De Sutter, Paul Coene, Tom Vander Aa, and Bingfeng Mei. 2008. Placement-and-routing-based Register Allocation for Coarse-grained Reconfigurable Arrays. *SIGPLAN Not.* 43, 7 (June 2008), 151–160.

[6] Andrei Frumusanu. 2016. The Samsung Exynos 7420 Deep Dive - Inside A Modern 14nm SoC. http://www.anandtech.com/show/9330/exynos-7420-deep-dive

[7] Michael Gautschi, Pasquale Davide Schiavone, Andreas Traber, Igor Loi, Antonio Pullini, Davide Rossi, Eric Flamand, Frank K Gürkaynak, and Luca Benini. 2017. Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25, 10 (Oct 2017), 2700–2713. https://doi.org/10.1109/TVLSI.2017.2654506

[8] Graham Gobieski, Ahmet Oguz Atli, Kenneth Mai, Brandon Lucia, and Nathan Beckmann. 2021. Snafu: An Ultra-Low-Power, Energy-Minimal CGRA-Generation Framework and Architecture. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 1027–1040. https://doi.org/10.1109/ISCA52012.2021.00084

[9] Mahdi Hamzeh, Aviral Shrivastava, and Sarma Vrudhula. 2012. EPIMap: using epimorphism to map applications on CGRAs. In *DAC*. ACM, 1284–1291. https://doi.org/10.1145/2228360.2228600

[10] Mahdi Hamzeh, Aviral Shrivastava, and Sarma Vrudhula. 2013. REGIMap: register-aware application mapping on coarse-grained reconfigurable architectures (CGRAs). In *DAC*. ACM, 18:1–18:10. https://doi.org/10.1145/2463209.2488756

[11] Manupa Karunaratne, Aditi Kulkarni Mohite, Tulika Mitra, and Li-Shiuan Peh. 2017. HyCUBE: A CGRA with Reconfigurable Single-Cycle Multi-Hop Interconnect. In *Proceedings of the 54th Annual Design Automation Conference 2017* (Austin, TX, USA) *(DAC '17)*. Association for Computing Machinery, New York, NY, USA, Article 45, 6 pages. https://doi.org/10.1145/3061639.3062262

[12] Changmoo Kim, Mookyoung Chung, Yeongon Cho, M. Konijnenburg, Soojung Ryu, and Jeongwook Kim. 2012. ULP-SRP: Ultra low power Samsung Reconfigurable Processor for biomedical applications. In *FPT*. 329–334. https://doi.org/10.1109/FPT.2012.6412157

[13] Takuya Kojima, Nguyen Anh Vu Doan, and Hideharu Amano. 2020. GenMap: A Genetic Algorithmic Approach for Optimizing Spatial Mapping of Coarse-Grained Reconfigurable Architectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 28, 11 (2020), 2383–2396. https://doi.org/10.1109/TVLSI.2020.3009225

[14] Ganghee Lee, Kiyoung Choi, and N.D. Dutt. 2011. Mapping Multi-Domain Applications Onto Coarse-Grained Reconfigurable Architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30, 5 (May 2011), 637–650. https://doi.org/10.1109/TCAD.2010.2098571

[15] Giorgio Levi. 1973. A note on the derivation of maximal common subgraphs of two directed or undirected graphs. *CALCOLO* 9, 4 (Dec. 1973), 341–352. https://doi.org/10.1007/BF02575586

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

[17] Bingfeng Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. 2002. DRESC: a retargetable compiler for coarse-grained reconfigurable architectures. In *FPT*. 166–173. https://doi.org/10.1109/FPT.2002.1188678

[18] Chris Nicol. 2017. A coarse grain reconfigurable array (cgra) for statically scheduled data flow computing. *Wave Computing White Paper* (2017).

[19] Hyunchul Park, Kevin Fan, Scott A. Mahlke, Taewook Oh, Heeseok Kim, and Hong-seok Kim. 2008. Edge-centric Modulo Scheduling for Coarse-grained Reconfigurable Architectures. In *PACT*. ACM. https://doi.org/10.1145/1454115.1454140

[20] Thomas Peyret, Gwenolé Corre, Mathieu Thevenin, Kevin J. M. Martin, and Philippe Coussy. 2014. An automated design approach to map applications on CGRAs. In *Proceedings of the 24th Edition of the Great Lakes Symposium on VLSI*. ACM, 229–230.

[21] Thomas Peyret, Gwenolé Corre, Mathieu Thevenin, Kevin J. M. Martin, and Philippe Coussy. 2014. Efficient application mapping on CGRAs based on backward simultaneous scheduling/binding and dynamic graph transformations. In *ASAP*. 169–172. https://doi.org/10.1109/ASAP.2014.6868652

[22] A. Podobas, K. Sano, and S. Matsuoka. 2020. A Survey on Coarse-Grained Reconfigurable Architectures From a Performance Perspective. *IEEE Access* 8 (2020), 146719–146743. https://doi.org/10.1109/ACCESS.2020.3012084

[23] Cheng Tan, Chenhao Xie, Ang Li, Kevin J. Barker, and Antonino Tumeo. 2021. AURORA: Automated Refinement of Coarse-Grained Reconfigurable Accelerators. In *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1388–1393. https://doi.org/10.23919/DATE51398.2021.9473955

[24] Vaishali Tehre and Ravindra Kshirsagar. 2012. Survey on Coarse Grained Reconfigurable Architectures. *International Journal of Computer Applications* 48, 16 (June 2012), 1–7.

[25] Jian Weng, Sihao Liu, Vidushi Dadu, Zhengrong Wang, Preyas Shah, and Tony Nowatzki. 2020. DSAGEN: Synthesizing Programmable Spatial Accelerators. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 268–281. https://doi.org/10.1109/ISCA45697.2020.00032

[26] Dhananiaya Wijerathne, Zhaoying Li, Anuj Pathania, Tulika Mitra, and Lothar Thiele. 2021. HiMap: Fast and Scalable High-Quality Mapping on CGRA via Hierarchical Abstraction. In *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1192–1197. https://doi.org/10.23919/DATE51398.2021.9473916

[27] Mark Wijtvliet, Luc Waeijen, and Henk Corporaal. 2016. Coarse grained reconfigurable architectures in the past 25 years: Overview and classification. In *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*. 235–244. https://doi.org/10.1109/SAMOS.2016.7818353

[28] Shouyi Yin, Dajiang Liu, Leibo Liu, Shaojun Wei, and Yike Guo. 2015. Joint affine transformation and loop pipelining for mapping nested loop on CGRAs. In *DATE*. 115–120.