



HAL
open science

Context-memory Aware Mapping for Energy Efficient Acceleration with CGRAs

Satyajit Das, Kevin Martin, Philippe Coussy

► **To cite this version:**

Satyajit Das, Kevin Martin, Philippe Coussy. Context-memory Aware Mapping for Energy Efficient Acceleration with CGRAs. Design, Automation and Test in Europe Conference (DATE), Mar 2019, Florence, Italy. hal-02086145

HAL Id: hal-02086145

<https://hal.science/hal-02086145v1>

Submitted on 1 Apr 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Context-memory Aware Mapping for Energy Efficient Acceleration with CGRAs

Satyajit Das, Kevin J. M. Martin, Philippe Coussy

Univ. Bretagne-Sud, CNRS UMR 6285, Lab-STICC

Lorient, France

satyajit.das@univ-ubs.fr, kevin.martin@univ-ubs.fr, philippe.coussy@univ-ubs.fr

Abstract—Coarse Grained Reconfigurable Arrays (CGRAs) are emerging as low power computing alternative providing a high grade of acceleration. However, the area and energy efficiency of these devices are bottlenecked by the configuration/context memory when they are made autonomous and loosely coupled with CPUs. The size of these context memories is of prime importance due to their high area and impact on the power consumption. For instance, a 64-word context memory typically represents 40% of a processing element area. In this context, since traditional mapping approaches do not take the size of the context memory into account, CGRAs often become oversized which strongly degrade their performance and interest. In this paper, we propose a context memory aware mapping for CGRAs to achieve better area and energy efficiency. This paper motivates the need of constraining the size of the context memory inside the processing element (PE) for ultra low power acceleration. It also describes the mapping approach which tries to find at least one mapping solution for a given set of constraints defined by the context memories of the PEs. Experiments show that our proposed solution achieves an average of $2.3\times$ energy gain (with a maximum of $3.1\times$ and a minimum of $1.4\times$) compared to the mapping approach without the memory constraints, while using $2\times$ less context memory. When compared to the CPU, the proposed mapping achieves an average of $14\times$ (with a maximum of $23\times$ and minimum of $5\times$) energy gain.

Index Terms—CGRA, Context memory, Ultra low power, accelerator

I. INTRODUCTION

The quest for a good trade-off between flexibility and efficiency leads to continuously explore new architectures and tools. The high flexibility offered by processors is alleviated by its poor performance efficiency. On the other extreme, the high performance efficiency of ASICs is achieved at high cost and no flexibility. Between these two, a wide range of architectures have emerged, mainly specialized for a given application domain. Even if studied for 25+ years, Coarse Grained Reconfigurable Arrays (CGRAs) are still in the race as low power computing solution providing a high grade of acceleration for a very wide range of application family. CGRAs themselves gather a broad type of reconfigurable architectures [6]. In this paper, we focus on CGRAs that can compute full applications based on multi-operations functional units (FU). The considered CGRA is loosely coupled with the CPU through a common data interconnection network, and is configured once for the full workload to be processed. The CGRA thus needs to store the contexts (or configurations, or instructions) in each FU (or tile). However, rather big memory capacities are needed to execute big kernels. The area and energy efficiency are then bottlenecked by the size

of the configuration/context memory in each tile. Hence, the compiler must be aware of the resources of the target CGRA. Taking into account the memory for data is already studied in [3], [5], [7]. In these papers, the work focuses on data usage, for a CGRA which is tightly coupled with a CPU and which is reconfigured cycle-by-cycle, with the goal to minimize the initial interval of the loop pipelining.

However, dedicated approaches are necessary to make the compiler aware about the size of the context memory, such that tiles do not get overflowed by the mapped operations (i.e. instructions to execute). In this paper we propose a mapping approach and associated tool that consider the context memory constraints for the targeted CGRA and tries to find a solution that fits the target. This gives the opportunity to minimize the context memory size for a target application domain to achieve better energy efficiency. In a nutshell following are the contributions of the paper:

- a formalization to optimistically define the size needed for the configurations;
- a study of traversal strategies when mapping the applications;
- a context-memory aware mapping approach that monitors the number of instructions used for a valid mapping to guide the mapping tool;
- experimental results on a wide range of kernels.

The rest of this paper is organized as follows. In Section II, the background is discussed. Section III describes the problem formulation and the proposed mapping flow. Section IV presents the experimental results. Finally the paper concludes in section V.

II. BACKGROUND

CGRA considered in this paper is a grid of tiles (or Processing Elements) interconnected through a 2D-Mesh torus network. Each tile contains an ALU, a register file, and computes independently through its own context memory, decoder and controller, as presented in figure 1. Each context memory is loaded previously to computation by a global controller. Some tiles contain a load/store unit for reading and writing back data in the data memory through a logarithmic interconnect. The target CGRA is similar to the one in [1].

A straightforward design of such a CGRA is to define the same size of the context memory in each tile. However, when mapping an application on the CGRA, the compiler might not consider the size of the context memory, leading to oversizing this memory and finally losing interest and performance.

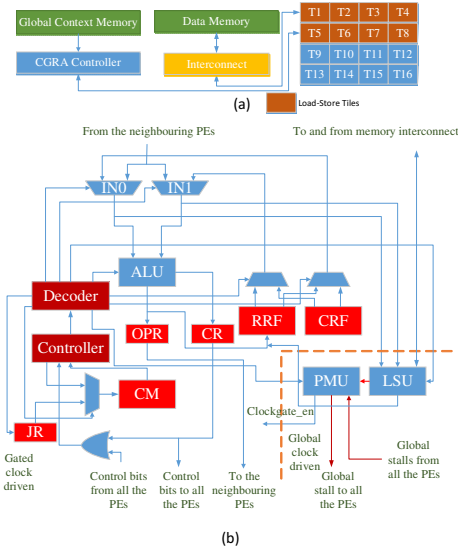


Fig. 1: (a) CGRA ecosystem; (b) Components of PE

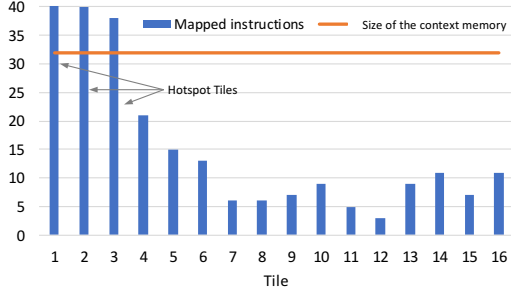


Fig. 2: Bad distribution of context words or instructions in mapping causes overfull usage

Although the program may theoretically fit in the CGRA, a bad distribution of the contexts may lead to hot-spot tiles or invalid solutions for overfull usage of the context memory as shown in Fig. 2, that presents mapping of matrix multiplication using a context unaware mapping proposed in [1].

The content of the context memory is made of three kinds of instructions: an operation (including control), a move, or a nop (no operation). Consecutive nops are gathered in one programmable nop (pnop) as presented in [1]. In this context, a mapping tool is thus facing two opposing goals: (1) introducing graph transformations in order to find a valid solution, thus increasing the number of operations, moves and nops, and (2) reducing the number of total instructions to make the contexts fit in the memory of each tile. Even if an analytical approach can be used to estimate the number of instructions needed for a given CDFG, the architectural constraints of the CGRA force to transform the data-flow graph [4].

The challenge of a CGRA compiler is to find a mapping between the application and the architecture. Typically, the application is modeled in an internal representation based on a CDFG (Control Data Flow Graph), where the control flow connects each other basic blocks (BBs), themselves composed of the data-flow part of the application. Since each tile of our CGRA contains some control logic, it is possible to map a CDFG onto this CGRA, independently of a CPU. A valid

mapping is thus a set of coherent mappings of each basic block. The mapping approach must travel each basic block, and for each block, it must find a valid mapping of the data-flow graph. The way the control flow and the data flow are traversed has an influence on the quality of the resulting mapping [1], [2]. When a valid mapping is found, the compiler generates the assembly code for each tile, which will be placed in the context memory.

III. CONTEXT MEMORY AWARE MAPPING

We first present the formalization of the general CGRA mapping problem with a basic solution. Next, we discuss its limitations according to context memory awareness with a goal of achieving higher energy efficiency. We then discuss the steps in detail to achieve the desired mapping of CDFG onto CGRA.

A. Basic mapping problem

Let $C = (V, E)$ be a CDFG representing the set of basic blocks $V(C)$ and control flow set $E(C)$. Each $b \in V(C)$ presents the data flow graph $b = (V_d, V_o, E)$, where $V_d(b)$ is the set of data nodes, $V_o(b)$ is the set of operation nodes and $E \in (V_o \times V_d)$ presents the data flow set.

Let another graph $T = (V, E)$ representing the time extended directed graph (TEDG) with cycle t [4]. Each node $y \in V(T)$ has attributes (r, t) , where $r \in (FU \cup RF)$ refers to the resource and t is the cycle. Each $FU = (P, I)$ represents the functional unit with an instruction set P and instruction register file set I in the context memory, and each RF represents the register file set. Let $e = (x, y) \in E(T)$ be an edge where $x = (r_1, t)$ and $y = (r_2, t + 1)$. Then the edge e represents a connection from resource r_1 in cycle t to resource r_2 in cycle $t + 1$.

We are looking for a mapping from the DFG b to the target graph T . All the data and control dependencies in the CDFG can be mapped to direct edges in the TEDG. The *basic mapping* problem is that for any edge $e = (x, y) \in E(b)$, there is an edge $e = (f(x), f(y)) \in E(T)$, where f represents the operation mapping function from the DFG to the TEDG.

B. Solution for the basic mapping problem

The article [1] describes an efficient solution for the *basic mapping* problem. The basic approach takes the CGRA TEDG and CDFG as the inputs. To select one basic block from the CDFG, the basic mapping traverses the CDFG forward. The flow uses a backward traversal list *scheduling* algorithm to schedule the DFG of the current basic block. It relies on a heuristic in which the schedulable operations are listed by priority order, which is defined by their mobility and number of fan-outs. As soon as the highest priority node has been determined, the compiler tries to find a binding solution. The *binding* uses an incremental version of sub graph match finding. The algorithm adds the newly scheduled operation node and its associated data node to the sub-graph composed of already scheduled and placed nodes. *Location constraints* are used to find every possibility to add this couple of nodes without considering the non-yet scheduled nodes. The location constraints are added by the data which carry dependency in several basic blocks. These are defined as *symbol variables*

(i.e. variable i in the CDFG presented in 3(b)). The symbol variables are always placed into the register file rather than spilling into the memory for energy efficiency reasons. However, the forced use of locations may lead to additional routing (or *move* operations) which is taken care of in the graph transformation step. Since the binding is an exact approach, if no solution is found, there is absolutely no possibility to bind this couple in all the previous partial solutions. In that case, *graph transformation* is required. The basic approach uses re-computing and re-routing as graph transformation. The exactness of the binding approach leads to very large number of partial mappings. It grows exponentially if not pruned. Hence, a *stochastic* based pruning approach is applied after each operation binding, which discards partial mappings depending on a threshold function. Once all the nodes of the basic block have been scheduled and bound, the compiler selects one mapping among the several mappings generated, and selects the next basic block to be mapped. A forward CDFG traversal is used for this selection. After mapping of all the basic blocks a complete mapping is generated and the assembler generates the binary for each tile.

C. Context memory aware mapping problem

With the basic mapping approach, more than 50% of the Context Memory (CM) is unused for most of the tiles, as shown in Fig. 2, except the tiles with load-store units. For the load-store unit tiles, it drops down to 15%. This kind of uneven distribution of the instructions is due to the mapping of compute intensive kernels, where most computations are performed on the data loaded from memory and the results are stored into the memory. The load-store nodes are essentially the *hot-spots*. However, the waste of context memory in the rest of the tiles leads to larger area and energy consumption.

Hence, in the context of energy efficient mapping, it is necessary to take the context memory size into account. In this paper, we extend the *basic mapping* problem such that $\sum_{V_o \in D} n(V_o) + \sum_{T_o \in D} n(T_o) + \sum_{pnop \in F} n(pnop) \leq \sum_{I \in F} n(I)$ and $n(M_o) + n(pnop) \leq n(I)$.

Where $n(V_o)$ is the number of operations in each DFG, $n(T_o)$ is the number of transformed operations while mapping DFG, $n(pnop)$ is the number of programmable nops introduced while mapping, $n(I)$ is the number of instruction registers in the CM in each functional unit and $n(M_o)$ is the number of mapped operation in the functional unit. Hence, the number of operations mapped onto each tile and the *pnops* introduced must not exceed the size of the context memory.

To illustrate the above notations, we present a basic mapping in Fig. 3(d). In this figure, the CDFG (3(b)) of a sample program presented in Fig. 3(a) is mapped onto a 3×1 CGRA shown in the Fig. 3(c).

D. Proposed memory aware mapping

To satisfy the equilibrium between the number of mapped operations for a tile and the size of context memory, the key is to sort the partial mappings and explore the design space better. As opposed to the *basic mapping* approach, where the partial mapping solution space is reduced without any constraints to achieve better compilation time, the partial mapping solutions

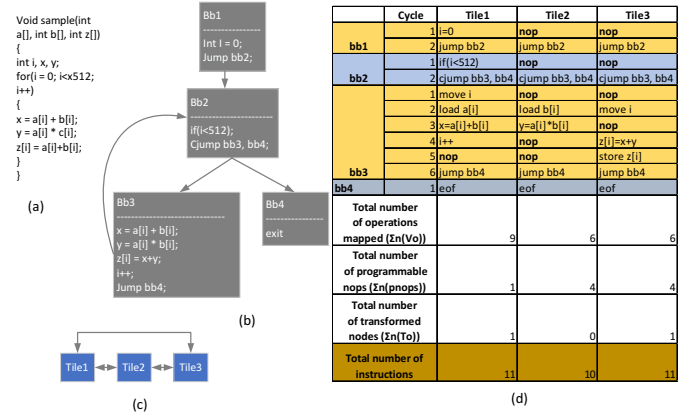


Fig. 3: (a) Sample program; (b) CDFG representation of the program in (a); (c) a 3×1 CGRA; (d) Illustration of total number of operations mapped with $n(V_o)$, $n(T_o)$ and $n(pnop)$ using the mapping of the CDFG in (b) onto the CGRA in (c)

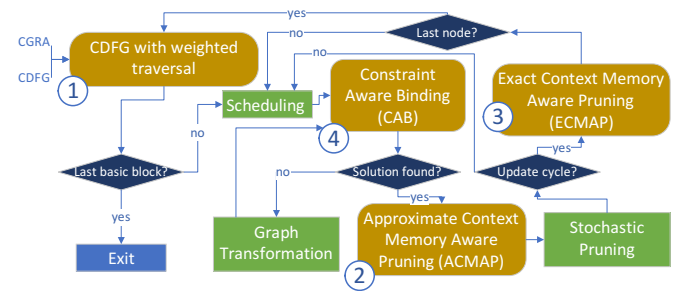


Fig. 4: Context memory aware mapping flow based on basic approach

in the context memory aware approach must be pruned smartly to satisfy the memory size. We introduce several steps in the basic mapping approach (in Fig. 4) to achieve the memory constraint aware mapping for a given CGRA architecture with a very small compromise in the compilation time.

1) *CDFG with weighted traversal*: As discussed earlier, the location constraints for the *symbol variables* introduces additional routing increasing the number of *move* operations where the symbol variable is stored and consequently number of *pnops* in the other tiles. Hence, for a context memory aware mapping, managing the symbol variables is important. Mapping first the basic blocks which contain the highest number of symbol variables gives the possibility to introduce less number of moves. Hence, we first compute the weight of each basic block as the function $W_{bb} = n(s) + \sum_{i=0}^{n(s)} f_s$, where $n(s)$ denotes number of symbol variables present in the basic block and f_s denotes the number of fan-outs of each symbol variable. The CDFG is traversed (see Fig. 4) according to the descending order of the basic blocks' weight.

Fig. 5 represents the comparison in the number of *pnops* and *moves* between the proposed weighted traversal and the forward CDFG traversal in the basic mapping flow. Fig. 5 presents the results for an FFT kernel, but the trend is similar for all other kernels. In this case the weighted traversal resulted around 42% reduction in *moves* and 24% reduction in *pnops* compared to the forward traversal. This new traversal helps in reducing the number of *move* and *pnop* instructions, but does

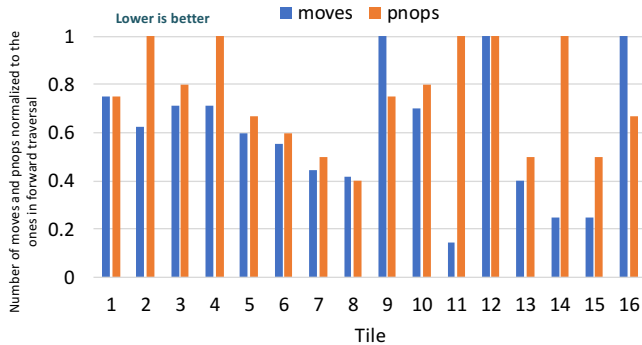


Fig. 5: Number of *plops* and *moves* in weighted traversal normalized to forward traversal

not guarantee a limited number of instructions.

The next steps introduce memory size constraints in the mapping.

2) *Approximate Context memory Aware Pruning (ACMAP)*: The exact approach in the binding stage (see section III-B) generates all the possible placement solutions for the current node. Over time this causes bottleneck due to exponential growth in the number of partial mapping solutions. The stochastic based pruning approach helps to reduce the number of partial solutions and better scale the compilation time. Since this pruning process is random in nature, it is possible to discard the valid solutions which are compliant with the memory constraints of the tiles and keep the solutions which violate the constraints.

Hence, it is necessary to add an additional pruning step just before the stochastic pruning (see Fig. 4), which filters the partial solutions according to the memory constraints. For every partial mapping, the approximate context memory aware pruning computes the number of operations mapped and the approximate number of *plops* for each functional unit. Since the actual number of the *plops* can only be obtained after each cycle of mapping, this step computes the upper limit of *plop* possible for each functional unit (a pessimistic estimation). On the one hand, approximate number of *plops* computed in the method aids to keeping solutions which may not actually fit in the context memory of the tiles. On the other hand, not computing the possible higher number of *plops* in this stage allows discarding solutions which may fit the memory constraints. Hence, we introduce an exact memory aware pruning.

3) *Exact Context memory Aware Pruning (ECMAP)*: The exact context memory aware pruning is introduced before moving to the next cycle (see Fig. 4). For each partial mapping, the exact memory aware pruning computes the number of operations mapped and the exact number of *plops* for each functional unit. This filter helps to keep only the valid partial mappings according to the memory constraints. However, the binding approach does not take into account which tiles are already full and will be discarded anyway in the memory aware pruning step. For better exploration of the design space, we have added the constraint awareness in the exact binding step, which is discussed in the following.

TABLE I: Different configurations for context memory

Config	Load-Store tiles	Tiles with CM 64	Tiles with CM 32	Tiles with CM 16	Total
HOM64	1-8	1-16			1024
HOM32	1-8		1-16		512
HET1	1-8	1-4	5-8, 13-16	9-12	576
HET2	1-8	1-4	5-8	9-16	512

4) *Constraint Aware Binding (CAB)*: For each partial mapping in the exact pruning stage we characterize each tile depending on the number of operations mapped and *plops*. The tiles which cannot take any further instruction are tagged as the *blacklisted tiles* for the partial mapping. While the compiler binds the next operation for this partial mapping, the *blacklisted tiles* are discarded from the binding possibilities. This helps to create new partial mappings only for the valid tiles, giving the opportunity for better exploring the solution space.

We have introduced four dedicated steps in the basic mapping approach to achieve context aware mapping. In the next section we evaluate the performance of the proposed approach.

IV. EXPERIMENTAL RESULTS

This section analyses the performance of context memory aware mapping compared to the basic mapping approach. It also discusses implementation results for the target CGRA providing area, and energy consumption while running several compute intensive signal processing kernels.

A. Experimental setup

To analyze the mapping quality of the context aware mapping, we present latency results for different CM sizes. Four different configurations as shown in the table I are considered for these experiments.

In HOM64 and HOM32, all the tiles have CM of size 64 and 32 respectively. In the HET1 configuration, only the 4 (tile 1, 2, 3, 4) out of 8 load-store tiles have CM of size 64. The tiles in the two rows which are in the nearest neighbor of the tiles 1, 2, 3 and 4 (tile 5, 6, 7, 8 and 13, 14, 15, 16) have CM of size 32. The rest of the 4 tiles (tile 9, 10, 11, 12) have CM of size 16. In the configuration HET2, tiles in the two rows of load-stores (tile 1, 2, 3, 4 and tile 5, 6, 7, 8) have CM-64 and CM-32 respectively. All the other tiles in this configuration have CM of size 16.

B. Comparing different approaches

Since, the context memory aware mapping upgrades the basic approach by adding several smart pruning steps, we profile mapping performance of each additional step. Mappings generated by the basic mapping approach is run on the HOM64 configured CGRA (since the basic mapping generates maximum of 64 word context in the tiles while mapping this specific set of signal processing kernels), whereas the mappings generated by the additional steps are run on the HOM32, HET1 and HET2.

Fig. 6 shows the mapping results after adding the approximate context memory aware pruning (ACMAP) step in the basic approach. The latency results are normalized with the performance of the basic approach for HOM64 configuration. The zero values in the chart refer to no mapping solution. For

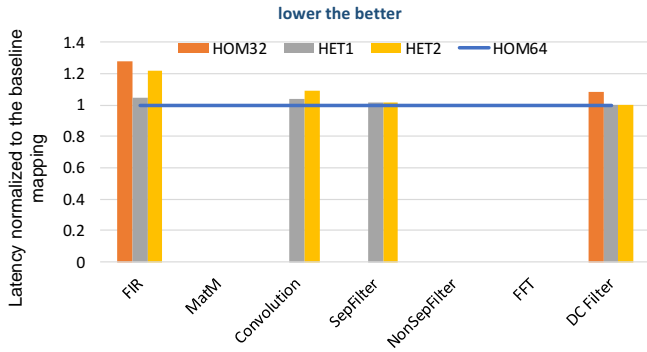


Fig. 6: Latency comparison in different configuration with basic + ACMAP

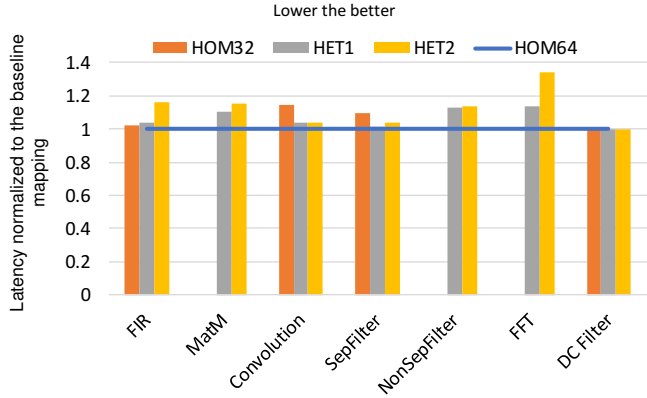


Fig. 7: Latency comparison in different configuration with basic + ACMAP + ECMAP

example in the matrix multiplication, non separable filter and FFT, the approximate pruning could not find any solution for the HOM32, HET1, HET2 configuration. For the convolution and separable filter, it was able to find the solution for the HOM32 and HET1 but could not find the solution for HET2 configuration. The abundance of invalid mappings in this step is the evidence of finding less solutions.

In Fig. 7, we present latency results after introducing the exact context memory aware pruning (ECMAP) step in the basic + ACMAP version. It is obvious that the mapping performance is improved, since the addition of the exact pruning helps to find mapping solutions for most of the times except for the kernels matrix multiplication, FFT and non separable filter. One interesting observation is that, when the tiles are over constrained the penalty in latency performance is very small. We have investigated that the configuration where the exact pruning could not find the mapping solution is HOM32, where all the load store tiles are over constrained.

Fig. 8 presents the latency results after adding CAB. The figure shows that the latency performance for the configuration HET2 improves compared to the previous version.

Since adding the pruning steps require additional compilation time, we compared the penalty after each additional step. Fig. 9 shows the average compilation time for the context aware mapping with each addition normalized with the basic approach. The results show that context memory aware mapping performs with an average of $1.8\times$ increased

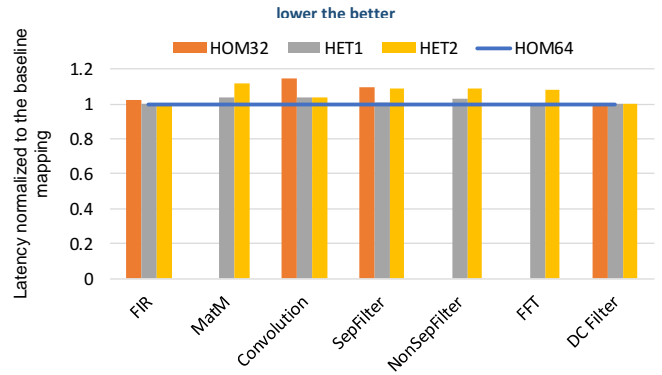


Fig. 8: Latency comparison in different configuration with basic + ACMAP + ECMAP + CAB

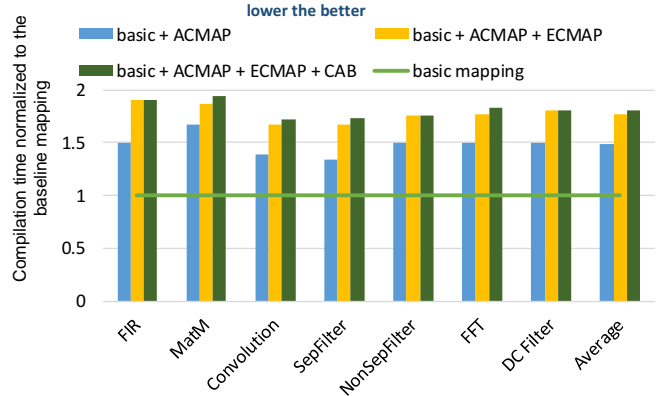


Fig. 9: Compilation time comparison after adding each step of memory aware mapping

compilation time. However, since the average compilation time for the basic mapping approach and context memory aware mapping averages around 17 and 30 seconds respectively, the penalty is acceptable.

Finally, we compared the performance of the basic mapping and the context memory aware mapping with the cpu performance. For comparison we consider or1k cpu. Fig. 10 presents the total execution time (clock cycles) of seven compute-intensive kernels. The execution time is normalized with respect to that of or1k processor, where the kernels are compiled with -O3 optimization flag. The performance of the context aware mapping is evaluated in the HET1 and HET2 configurations, which shows that it performed almost similarly as the basic mapping with decreased context memory size. As a result, the context aware mapping achieves an average of $10\times$ speed up with a maximum of $22\times$ for the HET1 and $19\times$ for HET2, and a minimum of $5\times$ for both configurations.

C. Area and Energy Results

This section describes the implementation results for the CGRA with different configurations to present the area efficiency achieved by the context aware mapping, providing a comparison with the or1k CPU. The designs 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

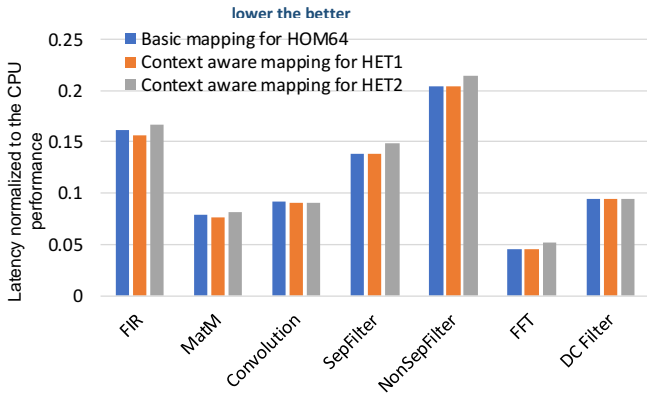


Fig. 10: Execution time comparison between context aware mapping and not context aware mapping normalized to CPU execution

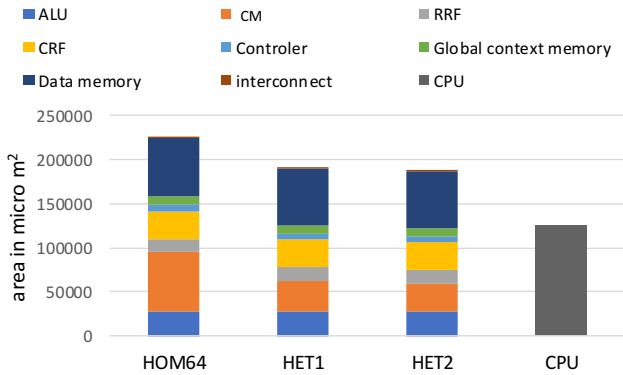


Fig. 11: Area comparison with the CPU

typical process conditions. The cycle information was achieved simulating the RTL with Mentor Questa Sim-64 10.5c. In the following, the experiments consider a CGRA of 4×4 array with 16 PEs, each one including 20×64 -bit CM for HOM64. For HET1 and HET2, the CM used are of 20×64 , 20×32 and 20×16 . The regular register files used are of 32×8 -bit and the constant register files are of 32×16 -bit. For area comparison, the CPU includes 32 kB of data memory, 4 kB of context memory, and 1 kB of instruction cache, which is equivalent to the design parameters of the CGRAs used in the experiments. Fig. 11 shows the area breakdown for the CGRAs with different configuration. The reduced area usage by the context memory in HET1 and HET2 helps in gaining energy efficiency which is shown in later set of experiments. Thanks to the context memory aware mapping, the compilation finds solutions for the HET1 and HET2. As a result, the total area overhead becomes $1.5 \times$ compared to $2 \times$ in the HOM64.

Table II compares the energy consumption in μ Joule while running different kernels in the CGRA and CPU. The mappings generated by the basic approach were run in the HOM64 version of the CGRA, while the HET1 and HET2 run the mappings generated by the context aware approach. Results show that the context aware mapping achieves an average of $2.3 \times$ energy gain (with a maximum of $3.1 \times$ in HET1 and $2.75 \times$ in HET2, and a minimum of $1.4 \times$ in both configuration) compared to the basic mapping. Comparing with the CPU, the

TABLE II: Energy consumption in μ Joule for not context aware and context aware mapping compared with CPU

Kernels	CPU	Basic mapping HOM64	Context aware mapping HET1	Context aware mapping HET2
FIR	0.132	0.022 6x	0.007 18x	0.008 17x
MatM	2.037	1.041 2x	0.456 4x	0.49 4x
Convolution	2.159	0.221 10x	0.157 14x	0.157 14x
SepFilter	20.94	4.246 5x	2.314 9x	2.482 8x
NonSepFilter	31.794	6.298 5x	2.22 14x	2.333 14x
FFT	0.576	0.033 17x	0.025 23x	0.028 21x
DC Filter	0.175	0.019 9x	0.01 18x	0.01 18x

context aware mapping achieves an average of $14 \times$ (with a maximum of $23 \times$ and minimum of $5 \times$) energy gain. In [1], it is reported that the basic mapping approach combined with the target CGRA with HOM64 configuration achieves leading-edge energy efficiency, surpassing by more than one order of magnitude other state of the art architectures.

V. CONCLUSION

This paper presents a novel context memory aware mapping to achieve better area and energy efficiency. The approach adds smart pruning steps into a basic mapping flow to satisfy memory constraints. As a result, the proposed approach is able to find mappings for CGRAs with less context memory. Experiments show that the proposed mapping uses almost half of the context memory compared to a homogeneous memory solution used in the basic mapping approach. Consequently, the proposed approach achieves an average of $2.3 \times$ energy gain (with a maximum of $3.1 \times$ in HET1 and $2.75 \times$ in HET2, and a minimum of $1.4 \times$ in both configuration) compared to the basic mapping. Compared to the CPU, the context aware mapping achieves an average of $14 \times$ (with a maximum of $23 \times$ and minimum of $5 \times$) energy gain.

REFERENCES

- [1] S. Das, K. J. Martin, D. Rossi, P. Coussy, and L. Benini. An energy-efficient integrated programmable array accelerator and compilation flow for near-sensor ultra-low power processing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.
- [2] S. Das, T. Peyret, K. Martin, G. Corre, M. Thevenin, and P. Coussy. A scalable design approach to efficiently map applications on cgras. In *VLSI (ISVLSI), 2016 IEEE Computer Society Annual Symposium on*, pages 655–660. IEEE, 2016.
- [3] S. Dave, M. Balasubramanian, and A. Shrivastava. RAMP: Resource-aware Mapping for CGRAs. In *Proceedings of the 55th Annual Design Automation Conference, DAC '18*, pages 127:1–127:6, New York, NY, USA, 2018. ACM.
- [4] M. Hamzeh, A. Shrivastava, and S. Vrudhula. EPIMap: using epimorphism to map applications on CGRAs. In *DAC*, pages 1284–1291. ACM, 2012.
- [5] Y. Kim, J. Lee, A. Shrivastava, J. Yoon, and Y. Paek. Memory-Aware Application Mapping on Coarse-Grained Reconfigurable Arrays. In Y. N. Patt, P. Foglia, E. Duesterwald, P. Faraboschi, and X. Martorell, editors, *High Performance Embedded Architectures and Compilers*, number 5952 in Lecture Notes in Computer Science, pages 171–185. Springer Berlin Heidelberg, Jan. 2010.
- [6] M. Wijnvliet, L. Waeijen, and H. Corporaal. 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)*, pages 235–244, July 2016.
- [7] S. Yin, X. Yao, D. Liu, L. Liu, and S. Wei. Memory-Aware Loop Mapping on Coarse-Grained Reconfigurable Architectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(5):1895–1908, May 2016.