Efficient Mapping of CDFG onto Coarse-Grained Reconfigurable Array Architectures

Satyajit Das, Kevin Martin, Philippe Coussy, Davide Rossi, Luca Benini

Efficient Mapping of CDFG onto Coarse-Grained Reconfigurable Array Architectures

Satyajit Das*†, Kevin J. M. Martin*, Philippe Coussy*, Davide Rossi†, and Luca Benini†‡

*Univ. Bretagne-Sud, UMR 6285, Lab-STICC, F-56100 Lorient, France,
[firstname].[lastname]@univ-ubs.fr
†Department of Electrical, Electronic and Information Engineering, University of Bologna, Italy,
[firstname].[lastname]@unibo.it
‡Integrated Systems Laboratory, ETH Zurich, Switzerland, [first-initial][last name]@iis.ee.ethz.ch

Abstract— In the approaching era of IoT, flexible and low power accelerators have become essential to meet aggressive energy efficiency targets. During the last few decades, Coarse Grain Reconfigurable Arrays (CGRA) have demonstrated high energy efficiency as accelerators, especially for high-performance streaming applications. While existing CGRAs mostly rely on partial and full predication techniques to support conditional branches, inefficient architecture and mapping support for handling control flow limits the use of CGRAs in accelerating either only inner loop bodies, or transformed loops specifically adapted to the target CGRA. This paper proposes a novel CGRA architecture with support for jump and conditional jump instructions and a lightweight global synchronization mechanism to enable complete Control Data Flow Graph (CDFG) mapping in an ultra-low-power environment. The architecture is coupled with a complete design flow that efficiently maps applications with heavy control flow starting from a generic C language description. The proposed mapping approach reduces the impact of wasteful instruction issues in the conventional approaches of predication providing an average energy improvement of 1.44x and 1.6x when compared to the state of the art partial and full predication techniques. Moreover, the proposed method achieves an average speed-up up to 21x and an energy improvement up to 50.42x while executing applications with heavy control flow with respect to sequential execution on a low-power embedded CPU, demonstrating its suitability for next generation IoT applications.

I. Introduction

During the last few decades, CGRA architectures have been mainly proposed for accelerating high-performance compute intensive and applications providing significant energy efficiency boost with respect to general-purpose processors. CGRAs are indeed an interesting trade-off between Field Programmable Gate Arrays (FPGAs) and many-core architectures thanks to their capability to exploit spatial computation typical of hardwired accelerators while maintaining high-level programmability typical of general-purpose processors [18]. However, in most state of the art models, CGRAs are coupled to a general purpose processor, where the processor executes the control parts and delegates data-flow parts to the CGRA [2, 17], precisely inner loop part. Hence, the success of such a partitioning is limited to applications where the execution time in the CGRA plus the communication overhead is less than the execution time on the processor only. This approach has demonstrated effectiveness in high performance environments, but it is much more challenging for deeply embedded applications where relatively small datasets do not allow CGRAs to fully exploit their streaming capabilities. Moreover, existing CGRAs mostly rely on partial and full predication techniques to support conditional branches [6] [7]. Although these techniques have be proven to be extremely effective for high-performance applications, as they can efficiently expose parallelism hence leading to faster execution, they also lead to a waste of resources and redundant instruction fetches, not suitable for a low- power environment where the primarily target is energy efficiency.

This paper proposes a framework to execute full control flow on CGRAs in an ultra-low-power (ULP) environment, completely releasing the host processor from performing loops control. This solution addresses standalone execution of all the levels of the loops, who is capable of mapping both loops and conditionals, and gives great flexibility to accommodate full application mapping on the CGRA. To enable lightweight support for control flow in CGRAs, we propose a novel CGRA architecture capable of executing jump and conditional jump instructions, and capable of synchronizing processing elements after executing jump or conditional jump. This approach also eases power management of unused tiles of the CGRA, that can be clock gated and power managed when waiting on global synchronization barriers, avoiding unnecessary waste of energy. When compared to state of the art methods to handle conditionals in CGRAs, namely partial and full predication, the proposed approach provides an average energy improvement of 1.44x and 1.6x, respectively, when executing a wide set of signal processing applications. When compared to a low-power OpenRISC CPU [10], a 4x4 CGRA architecture implemented in 28nm UTBB FD-SOI [13] technology achieves 100 MHz of operating frequency and power density of 1.78 μW/MHz at 0.6V, compared to 45 MHz and 3.54 μW/MHz achieved by the CPU, with an area overhead of 2.5x. When executing a wide range of signal processing applications our results show that the proposed CGRA architecture, along with the mapping approach, achieves an average speed-up of 21x, 4.8x and 4.8x with respect to a low power CPU, with -O0, -O2, -O3 optimization options, respectively, leading to a peak energy improvement of 50.42x.

The paper is organized as follows. The next section discusses the background and related work. Section 3 presents the context and contribution of this paper. Subsequently the mapping method is described in section 4. Section 5 elaborates on the need for hardware support to handle full control flow of an application and also presents the CGRA architecture to support mapping of CDFG. In section 6 experimental set-up is presented and results are discussed. Finally the paper concludes in section 7.

II. Background

Several solutions have been proposed to map large applications with control flow onto CGRAs. The most commonly employed solution is to couple the CGRA with a host processor. Loosely coupled CGRAs like MorphoSys [17] or FLORA [11] employ shared registers or a special bus to connect the CPU to the CGRA. In Morphosys a tiny RISC processor, a fixed 8×8 16-bit Processing Element (PE) array, a data cache and DMA controller are linked together. The RISC processor handles control flow code, while the PE array accelerates data flow. ADRES [2] defines a template which allows application optimization, using two modes of operation (VLIW mode and CGRA mode) in mutually exclusive manner. VLIW mode handles the outer loops and CGRA mode concentrates on executing inner loop part. ADRES implements predicated operations for inner loop control flow.

Partial predication [6] maps instructions of both if-part and else-part on different PEs. If both the if-part and the else-part update the same variable, the final result is computed by selecting the output from the path that must have been executed based on the evaluation of the branch condition. Full predication [7] addresses the loss of performance due to the execution of both paths and the *select* operations. It does not require a select operation, instead, the operations that update the same variable are mapped to the same PE but at different times. The correct value of the output will be present in the PE after the maximum time. Since operations from one of the paths are executed at runtime and operations of the other path are squashed, the performance of full predication is degraded. Since the PE fetches the instructions on an unnecessary path, the condition can be checked only after fetching.

An upgrade of full predication was proposed in [8]. This scheme prevents the wasted instruction issues from false conditional path by introducing *sleep* and *awake* mechanisms, but it does not improve performance in terms of latency. Dual issue scheme in CGRA [6] improves the latency by issuing two instructions to a PE simultaneously, one from the if-path, another from the else-path. But this proposition is too restrictive, as far as imbalanced conditionals are concerned. All the approaches mentioned in this section do not map the outer loops of a nested loop application. Rather they focus on innermost loop only.

This paper implements all the conditionals efficiently inside the CGRA, improving energy consumption, whereas low power CGRA like [15] only focuses on DFG mapping. HPC Machines like RHyMe [4], ReNÉ [12] can execute loops independently. These architectures are based on REDEFINE [1] CGRA, which consists of a set of reconfigurable data-paths called HyperCells interconnected through a network-on-chip (NoC). While the data flow in these architectures rely on power hungry communication through switched network and NoC, the architecture proposed in this paper uses simple point to point data communication. Also the proposed CGRA architecture supports automated global synchronization of processing elements (PE) to orchestrate kernel execution. This makes it suitable for the ultra low power environment, as shown in the experiments.

III. Context and contribution

This paper focuses on mapping complete applications (including control structures) onto a CGRA. The high level abstraction of the application is directly taken as an input in the form of a CDFG. The other input is the CGRA model where the CDFG has to be mapped on. Formally, a CDFG is represented as $G = (V, E)$ where V is the set of basic blocks and $E \subseteq V \times V$ is the set of directed edges representing control flow. A Basic Block (BB) is represented as a data flow graph (DFG) or $BB = (D, O, A)$ where D is the set of data nodes, O is the set of operation nodes and A is the set of arcs representing data dependencies. A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end of basic block. The control flow at the end of basic block is supported with jump (*jmp*) and conditional jump (*cjmp*) instructions.

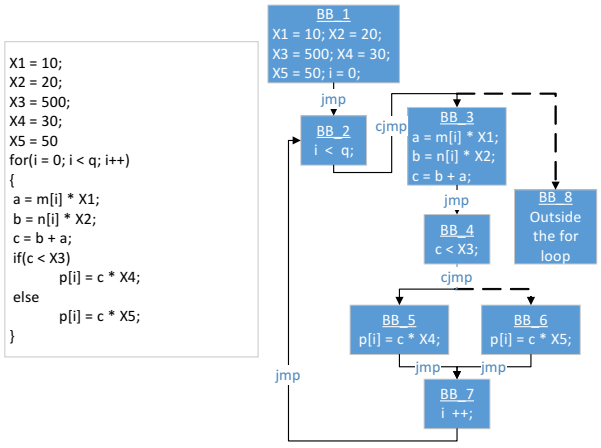

Fig. 1. Sample program and corresponding CDFG

TABLE I
THEORETICAL EFFICIENCY OF THE PROPOSED APPROACH

#instructions fetched			#instructions executed		
Partial pred	Full pred	Proposed approach	Partial pred	Full Pred	Proposed approach
$3n$	$2n$	n	$3n$	n	n

Fig. 1 represents a sample program and the corresponding CDFG. In this figure basic blocks are represented as blue rectangles. The flow from one basic block to another basic block is represented by black arrows and managed by *jmp* operation. The true and false path of a conditional which are managed by *cjmp* are shown by solid and dashed arrows respectively. Considering the execution flow of the above mentioned CDFG: $BB\_1 \rightarrow BB\_2 \rightarrow$ (either BB_3 or BB_8) if $BB_3 \rightarrow BB\_4$

$\rightarrow$ (either BB_5 or BB_6) $\rightarrow BB\_7 \rightarrow BB\_2 \cdots$ in CGRA, it becomes imperative to synchronize all the tiles to the execution of the same basic block. When the execution flow jumps from one basic block to another, all the tiles in CGRA must be synchronized to the current basic block execution. This is necessary as all the tiles can be used concurrently or sequentially to execute a single basic block. Hence, several basic block can use same PE in different time.

Table I presents a theoretical study of the efficiency of this synchronization based approach over partial and full predication. Here we have considered a conditional that has n operations in each path. All the n outputs of n operations from each path will be used in the rest of the program. Hence, partial predication needs to execute $3n$ instruction ($2n$ from both the paths and n select instructions). Full predication requires to fetch all the $2n$ instructions, but it can squash execution of n instructions from the false path depending on the predicate value. The proposed approach in one hand reduces the wasteful execution of the partial predication and instruction fetches in the full predication, decreasing overall energy consumption while executing kernels. The results show that the proposed approach achieves an average of $1.44\times$ and $1.6\times$ energy improvement over partial and full predication.

In summary the contributions of this work are: (a) A mapping support for full control flow of an application onto the CGRA. (b) A novel CGRA architecture to support execution of CDFGs, which is implemented using STMicroelectronics UTBB FD-SOI 28nm technology. (c) The efficiency of control flow execution over cpu performance. We show that, we are able to execute CDFGs with much more parallelism than a single cpu, leading to significant speed-up and energy efficiency boost, even with very moderate assumptions on the available bandwidth to memory. In addition, the area overhead of CGRA with different dimensions is given with respect to a cpu. (d) The energy improvement for execution of several kernels using the proposed mapping approach, compared to the state of the art mechanisms to handle control flow.

IV. Proposed method

A. Problem formulation

As the basic blocks in a CDFG are represented as DFGs, any of the known DFG mapping onto CGRA [5, 16, 3, 9] can be applied to map the basic blocks. Mapping of the basic blocks is done independently, since execution of basic blocks are mutually exclusive. So each basic block can use the maximum available resources, increasing the flexibility of the mapping. Problem arises for the variables which are used in several basic blocks. As an instance, variable c in the CDFG (Fig. 1) is used as result in BB_3 and as operand, in BB_4, BB_5 and BB_6. So the same register allocation must be used in BB_4, BB_5 and BB_6 where c was written in BB_3. The same goes for $X1$, $X2$, $X3$, $X4$, $X5$, i, a and b. These variables will be referred to as *symbol variables* in this paper. Allocated registers for these variables will be denoted with an overline, as $\overline{variable\_name}$. In our sample program m, n are the input arrays and p is the output array. The inputs and outputs are dealt with normal load and store operations in CGRA.

Another problem arises while ordering the basic blocks for mapping, i.e. traversing the CDFG. To illustrate this problem let's consider a scenario, *scenario_1* where BB_6 is mapped first, BB_3 is mapped next and so on. When mapping BB_6, variables c and $X5$ are placed at $\overline{c}$ and $\overline{X5}$. When mapping BB_3, $\overline{c}$ and $\overline{X5}$ which are already mapped in BB_6, must be considered because $\overline{c}$ will be used to map c in BB_3. In other words, the placement of the variables in the memory elements must be respected. Also $\overline{a}$, $\overline{b}$, $\overline{X1}$ and $\overline{X2}$ must not reuse $\overline{X5}$. Otherwise, $X5$ will have wrong value when executing BB_6. Let's consider *scenario_2* with another order, like first BB_3 and then BB_6 and so on, then it is necessary to pass $\overline{c}$ and $\overline{X5}$ from BB_3 to BB_6 mapping. To keep c and $X5$ alive in BB_6, $\overline{c}$ and $\overline{X5}$ both must be used in mapping of BB_6. The placement or binding information which are passed from the mapping of one basic block to mapping of the other basic block is referred as *constraints* (e.g. *scenario_1*: $\overline{c}$ and $\overline{X5}$ passed from BB_6 to BB_3). There are two kinds of constraints. Constraints related to data that are used within a basic block mapping phase (e.g. *scenario_1*: $\overline{c}$ in BB_3 mapping) are called as *target location constraints*. Some of the data related to the constraints may not be used in the basic blocks but the constraints must be respected in the mapping of basic blocks. In order to keep these variables alive it is necessary to reserve the memory elements. Hence, while mapping the basic block, these resources must not override (e.g. *scenario_1*: $\overline{X5}$ in BB_3 mapping). These are called *reserved location constraints*. If the number of reserved location constraints is high, mapping a basic block becomes more complex. The number of reserved location constraints varies with the traversal of the CDFG. The effect becomes more prominent for larger applications with large number of data, larger number of basic blocks and complex control flow. Hence, an appropriate traversal is necessary to map the full CDFG efficiently.

The basic solution to deal with the symbol variables is to load and store the variables when necessary. The symbol variables will be stored where they are used as results and will be loaded when used as operands. This basic solution reduces the complexity of the mapping as there is no constraints in the basic block mapping. On the other hand it requires a huge memory bandwidth, significantly reducing the energy efficiency of the system. In the rest of the paper this basic solution is referred to as *systematic load-store based approach*.

B. Proposed approach

The proposed solution is named as *register allocation based approach*. The idea is to map the symbol variables in the register file of the processing element (PE). The symbol variables will be written in register file, where they are used as results and be retrieved from the registers where used as operands. In this approach, the effects of constraints in mapping are inevitable. Reserved location constraints restrict the use of some resources. Target location constraints force to reuse some resources. If there is only a single target location constraint in a basic block mapping, it becomes easier to start mapping from the known place. But as a matter of fact situations often arise where there appear several target location constraints while mapping a basic block. Forced placements by these constraints induce extra routing effort. To illustrate this, lets consider a scenario where BB_1 and BB_4 in Fig. 1 are already mapped (variables $X1$, $X2$, $X3$, $X4$, $X5$, c, i already mapped). The mapping of BB_3 must be done with target location constraints

$\overline{c}$, $\overline{X1}$ and $\overline{X2}$ and reserved location constrains $\overline{X5}$, $\overline{X3}$, $\overline{X4}$ and $\overline{i}$. $\overline{a}$ and $\overline{b}$ will be mapped in the respective PEs where $\overline{X1}$ and $\overline{X2}$ are allocated. Extra routing effort may be necessary to bring a and b to the PE where $\overline{c}$ is allocated. The mapping can be done because the addition operation attached to c must produce it in $\overline{c}$ which is a register in the register file (RF) of the corresponding PE. A graphical view of this issue is presented in Fig. 2, where BB_3 is being mapped onto a 3×1 CGRA with 4 registers in the RFs of each PE (R0 is the output register, see Fig. 4). In this CGRA, we consider the register files are local to the PEs. Connection between PEs are from output register (R0) to the inputs of neighbouring PE.

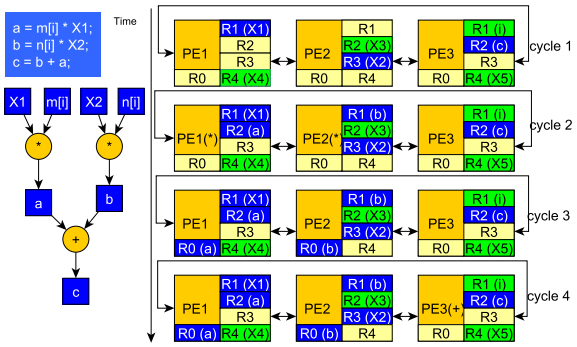

Fig. 2. Extra routing due to constraints in DFG mapping

As we can see in Fig. 2, execution starts with the target location constraints $\overline{c}$ $(PE3 - R2)$, $\overline{X1}$ $(PE1 - R1)$ and $\overline{X2}$ $(PE2 - R3)$, and the reserved location constraints $\overline{X3}$ $(PE2 - R2)$, $\overline{X4}$ $(PE1 - R4)$, $\overline{X5}$ $(PE3 - R4)$ and $\overline{i}$ $(PE3 - R1)$. The target location constraints force to map a, b in $PE1$ and $PE2$. Let's say they are mapped in $PE1 - R2$ and $PE2 - R1$, as they will be produced in cycle 1. In cycle 2, a and b can not be accessed to produce c. That is why extra routing is necessary to map the operation attached to c. The routing of a, b from the register file to the output registers is done in cycle 2 and in the next cycle c is generated in $\overline{c}$ which is $(PE3 - R2)$. Hence, mapping of the operation attached to c in this case, will experience longer schedule due to the constraints.

As mentioned earlier CDFG traversal has an impact on the number of reserved location constraints during the basic block mapping. Hence, it is necessary to select the basic blocks wisely for mapping. Four traversal strategies have been studied: 1) Forward breadth first traversal, 2) backward breadth first traversal, 3) depth first traversal and 4) random traversal. It appears that forward breadth first traversal induces the least number of reserved location constraints in the mapping of basic blocks[1]. Due to the data-flow between the basic blocks, generated symbol variables are most likely to be used in the successor basic blocks. In the forward breadth first traversal, the basic blocks with a greater number of symbol nodes are mapped earlier which also helps to reduce the number of target location constraints. More precisely the modified forward breadth first traversal is used to order the basic blocks while mapping. Fig. 3 shows the full compilation flow. The compiler takes application C code and the CGRA model as inputs. The C code is compiled to produce a CDFG. Each basic block in this CDFG is then mapped following the modified forward breadth first manner. At last the compiler produces the binary for each PE.

[1]the comparaison results are not given due to space limitation

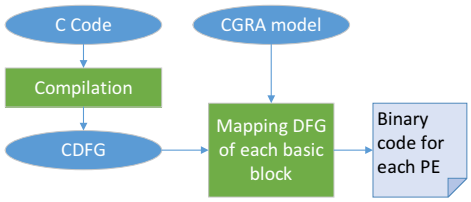

Fig. 3. The proposed mapping flow

V. CGRA ARCHITECTURE

As the mapping depends on control flow, CGRA which is being targeted by this approach must support two additional instructions jmp and cjmp. Every processing element in the CGRA executes the same jmp or cjmp instruction after execution of corresponding basic block, to synchronize to the next basic block execution. Every basic block has a starting address which is referred in jmp or cjmp instruction. Each processing element (PE) has its own instruction memory (IM), so that different instructions can be stored at the same address of each IM. Therefore, each PE can operate differently according to the instructions stored in the local IM. This makes it possible to achieve Multiple Instruction Stream, Multiple Data Stream (MIMD) operation in the processor array. The PEs are organized as an array using a mesh torus topology (Fig. 4), giving the opportunity to connect directly to four neighbouring PEs.

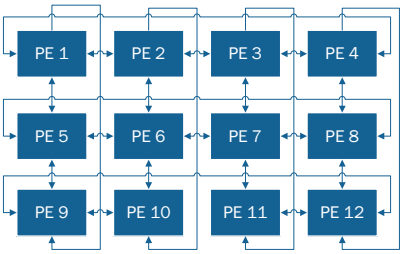

Fig. 4. 3×4 CGRA with mesh torus topology

A. Architecture of PE

Fig. 5 shows the system level architecture of a PE. The internal blocks are explained as follows. **Input mux:** Two input muxes are there for selecting two operands (OpA and OpB). The sources are the neighbouring PEs and the register file. 32-bits **ALU+multiplier (16x16 bits)** executes arithmetic and logical operations. The **shifter** performs shift operation on OpA by OpB bits. **Load store unit (LSU)** is optional for the PEs. **Controller** is responsible for selecting the address of next instruction to be executed. For the CGRAs coupled with host, this unit has to carry extra burden of managing stalls. **Regular Register File (RRF)** and Output Register (OR) are used to store temporary variables. **Constant register file (CRF)** is

dedicated to store constants. **Cond Register** is one bit register, which contains 0 for all the normal operations and true conditions. The value is set to 1 on false condition. The boolean OR of all the control bits from all the PEs gives the indication that one PE has executed false condition. Consequently in the next cycle, offset address of the basic block from false path is fetched. **Jump Register** contains the address to be jumped. **Instruction Memory** holds 32, 20-bits instruction.

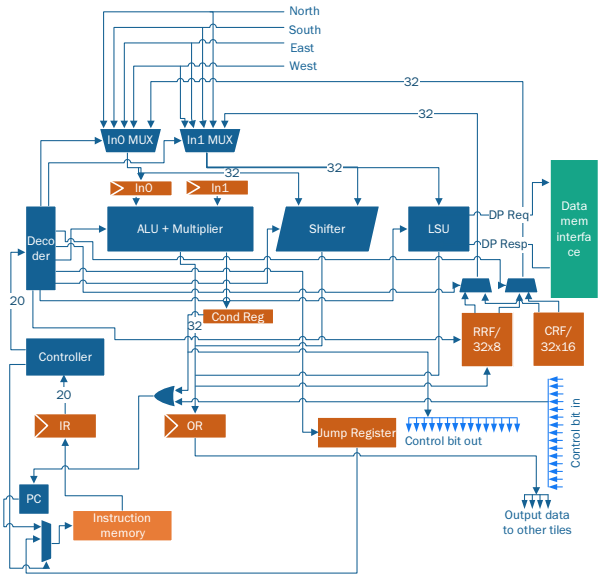

Fig. 5. Design of processing element

VI. Experiments & Results

A. 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. A cycle accurate model of the CGRA architecture presented in this paper has been developed in C++. The binary generated by the CGRA compiler is fed into this simulation model. The latency of all operations in CGRA is assumed one cycle, without loss of generality. Load and store operations requires two CGRA operations, one for address bus transaction and the other for data bus transaction. Firstly, the (a) proposed register allocation based approach and the (b) basic solution systematic load store based approach are compared with the performance of a CPU in terms of number of cycles. The low-power cpu chosen is a Or1K [10]. The results are obtained with OVPsim [14]. The binary is obtained with the toolchain provided by OVPsim. The simulator is instruction-accurate, and we assume one cycle per instruction to be in line with one cycle per operation in the CGRA. The performance comparison of standalone execution in CGRA and CPU, is done for -O0, -O1, -O2 and -O3 optimizations for CPU. Secondly, we compare area and energy consumption of the post synthesis implementation of CGRA with that of CPU in STMicroelectronics 28nm UTBB FD-SOI technology. They were synthesized using Synopsys design compiler, and Synopsys PrimePower was used for

TABLE II
Area information

	area in μm^2			Gate count
	Combinational	Sequential	Total	
cpu			50,000	113,636
CGRA 2x2	21,938	16,911	38,849	88,293
CGRA 3x3	41,326	38,035	79,361	180,366
CGRA 4x4	66,532	67,553	134,085	304,739

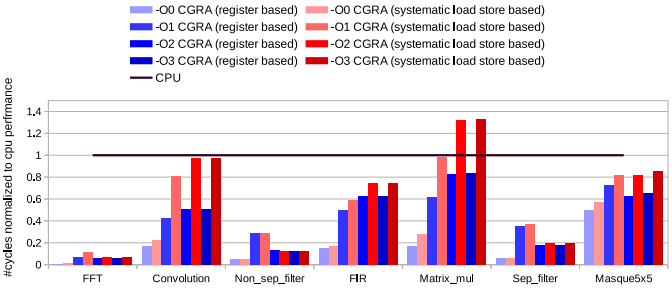

Fig. 6. Performance of proposed mapping approach in a CGRA with infinite band-width

power analysis. Table II refers the area of the information including 1kB of instruction memory for the CPU and 32 20-bit words of instruction memory per tile for CGRA. We performed timing analysis and power analysis at 0.6V supply voltage at the temperature of 25°C. In this operating point, the processor can achieve 45 MHz, with a power density of 3.54 μW/MHz, while a 4x4 CGRA achieves 100 MHz and a power density of 1.78 μW/MHz thanks to the much simpler structure of the tiles architecture with respect to a CPU.

B. Results

We have carried out a first set of experiments on the two approaches presented in this work: (a) systematic load store and (b) register allocation. We have first considered a 4x4 CGRA with "infinite bandwidth". Each PE in this case has a LSU and the number of load store per cycle is not limited, since this represents the theoretical worst case for the register allocation approach and the best case for the systematic load store approach we want to compare with. Fig. 6 shows the performance of the proposed register based approach and the basic load-store based approach normalized to cpu execution, for different optimization options. For all the cases, register based mapping approach achieves best performances. Systematic load store based approach sometimes (for matrix multiplication and masque) performed even worse than cpu.

Fig. 6 sums up the performance gain for different applications. The speed-up decreases with the compilation option as very aggressive compilation optimizations on loops are performed on the cpu with -03. Even if we do not implement such aggressive passes in our flow, still we are able to accelerate.

In Fig. 7, to show the effect of a more realistic number of load-store units on the mapping approaches we limit the number of load-store units to 4 (top row) [2], and we compare the results with respect to a CPU. Systematic load-store based approach performs not up to the mark, whereas register based approach gets much less effected by the limit of aggregated bandwidth. The speed-up with respect to a general-purpose

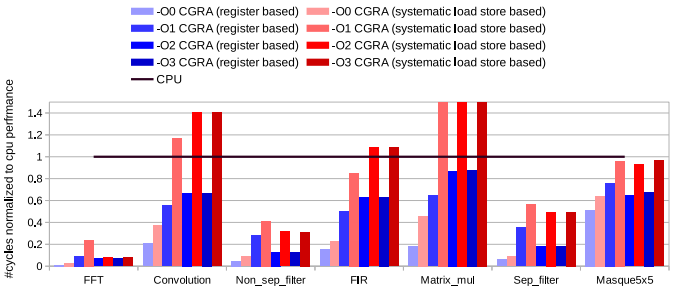

Fig. 7. Performance of proposed mapping approach in a CGRA with limited band-width

TABLE III
ENERGY CONSUMPTION IN μJ, IN CPU AND CGRA

	Opt -O0		opt -O1		opt -O2		Opt -03	
kernels	**CGRA**	**CPU**	**CGRA**	**CPU**	**CGRA**	**CPU**	**CGRA**	**CPU**
FFT	0.021	5.016	0.021	0.589	0.018	0.582	0.018	0.582
Convolution	1.256	14.493	0.556	2.632	0.556	2.184	0.556	2.183
Non_sep_filter	6.373	269.817	5.275	36.713	2.061	31.89	2.061	32.158
FIR	0.103	1.358	0.051	0.206	0.051	0.162	0.051	0.162
Matrix_mul	0.002	0.022	0.001	0.004	0.001	0.003	0.001	0.003
Sep_filter	5.083	166.419	4.318	24.129	1.932	21.179	1.931	21.179
Masque5x5	0.003	0.01	0.001	0.002	0.001	0.002	0.001	0.002

processor decreases with the compilation option, due to strong CPU- dependent optimization passes performed by GCC. Table III presents energy consumption for several kernel execution in a 4x4 CGRA and a CPU, showing that the proposed approach outperforms the sequential execution on a CPU by up to 21x and 50.42x in terms of performance and energy efficiency, respectively. The results show that with the proposed approach we can execute CDFGs with much more parallelism than a single CPU, leading to significant speedup and energy efficiency boost, even with very moderate assumptions on the available bandwidth to memory. In addition, the area overhead with respect to a CPU is moderate, leading to higher silicon area efficiency ($\sim$20x).

When we compare the proposed approach with state of the art, full predication and partial predication approaches, we achieve an average of 1.44x and 1.6x energy improvement (Table IV), respectively. As we did not implement any power management techniques, the energy results include dynamic, static and clock tree power of the idle PEs for all the presented techniques. This demonstrates that even if theoretically, predication can expose more parallelism and leads to faster execution, in practice it leads to waste of resources and inefficiency compared to synchronized execution, making the latter more suitable for low-power, deeply embedded applications where the primary target is energy efficiency.

VII. CONCLUSION

Control flow in the applications is a significant bottleneck in CGRA architectures, avoiding to exploit their full potential. In this paper we have presented a CGRA architecture and mapping approach to implement full control flow onto a CGRA in an ultra-low-power environment. The proposed approach overcomes limitations and inefficiencies of state of the art predications methods, achieving 1.44x and 1.6x energy gain over partial and full predication techniques, respectively. The proposed approach also achieves a speed-up up to 21x and an energy improvement up to 50x with respect to an embedded CPU with an area overhead of 2.7x, demonstrating it suitability for next generation IoT applications.

TABLE IV
ENERGY CONSUMPTION IN μJ FOR DIFFERENT APPROACHES

kernels	Proposed approach	Partial Pred	Full Pred
FFT	0.021	0.051	0.057
Convolution	1.256	1.396	1.467
Non_sep_filter	6.373	8.877	9.338
FIR	0.103	0.13	0.136
ManhDist	0.046	0.058	0.065
Matrix_mul	0.002	0.003	0.004
Sep_filter	5.083	6.542	7.349
Masque5x5	0.003	0.003	0.004

REFERENCES

[1] M. Alle, K. Varadarajan, A. Fell, R. R. C., N. Joseph, S. Das, P. Biswas, J. Chetia, A. Rao, S. K. Nandy, and R. Narayan. Redefine: Runtime reconfigurable polymorphic asic. *ACM Trans. Embed. Comput. Syst.*, 9(2):11:1–11:48, Oct. 2009.

[2] F. Bouwens, M. Berekovic, A. Kanstein, and G. Gaydadjiev. Architectural exploration of the adres coarse-grained reconfigurable array. In *Proceedings of the 3rd International Conference on Reconfigurable Computing: Architectures, Tools and Applications*, ARC'07, pages 1–13, Berlin, Heidelberg, 2007. Springer-Verlag.

[3] L. Chen and T. Mitra. Graph minor approach for application mapping on cgras. *ACM Trans. Reconfigurable Technol. Syst.*, 7(3):21:1–21:25, Sept. 2014.

[4] S. Das, N. Sivanandan, K. T. Madhu, S. K. Nandy, and R. Narayan. Rhyme: Redefine hyper cell multicore for accelerating hpc kernels. In *2016 29th International Conference on VLSI Design and 2016 15th International Conference on Embedded Systems (VLSID)*, pages 601–602, Jan 2016.

[5] M. Hamzeh, A. Shrivastava, and S. Vrudhula. Regimap: Register-aware application mapping on coarse-grained reconfigurable architectures (cgras). In *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, pages 1–10, May 2013.

[6] K. Han, J. Ahn, and K. Choi. Power-efficient predication techniques for acceleration of control flow execution on cgra. *ACM Trans. Archit. Code Optim.*, 10(2):8:1–8:25, May 2013.

[7] K. Han, J. K. Paek, and K. Choi. Acceleration of control flow on cgra using advanced predicated execution. In *Field-Programmable Technology (FPT), 2010 International Conference on*, pages 429–432, Dec 2010.

[8] K. Han, S. Park, and K. Choi. State-based full predication for low power coarse-grained reconfigurable architecture. In *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1367–1372, March 2012.

[9] W. Kim, Y. Choi, and H. Park. Fast modulo scheduler utilizing patternized routes for coarse-grained reconfigurable architectures. *ACM Trans. Archit. Code Optim.*, 10(4):58:1–58:24, Dec. 2013.

[10] D. Lampret, C.-M. Chen, M. Mlinar, J. Rydberg, M. Ziv-Av, C. Ziomkowski, G. McGary, B. Gardner, R. Mathur, and M. Bolado. Openrisc 1000 architecture manual. *Description of assembler mnemonics and other for OR1200*, 2003.

[11] D. Lee, M. Jo, K. Han, and K. Choi. Flora: Coarse-grained reconfigurable architecture with floating-point operation capability. In *Field-Programmable Technology, 2009. FPT 2009. International Conference on*, pages 376–379, Dec 2009.

[12] K. T. Madhu, A. Rao, S. Das, K. C. Madhava, S. K. Nandy, and R. Narayan. Flexible resource allocation and management for application graphs on renÉ mpsoc. In *Proceedings of the 7th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and the 5th Workshop on Design Tools and Architectures For Multicore Embedded Computing Platforms*, PARMA-DITAM '16, pages 13–18, New York, NY, USA, 2016. ACM.

[13] P. Magarshack, P. Flatresse, and G. Cesana. Utbb fd-soi: A process/design symbiosis for breakthrough energy-efficiency. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 952–957. EDA Consortium, 2013.

[14] OVP. Open virtual platform.

[15] N. Ozaki, Y. Yasuda, M. Izawa, Y. Saito, D. Ikebuchi, H. Amano, H. Nakamura, K. Usami, M. Namiki, and M. Kondo. Cool mega-arrays: Ultralow-power reconfigurable accelerator chips. *IEEE Micro*, 31(6):6–18, Nov 2011.

[16] T. Peyret, G. Corre, M. Thevenin, K. Martin, and P. Coussy. An automated design approach to map applications on cgras. In *Proceedings of the 24th Edition of the Great Lakes Symposium on VLSI*, GLSVLSI '14, pages 229–230, New York, NY, USA, 2014. ACM.

[17] H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. C. Filho. Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Transactions on Computers*, 49(5):465–481, May 2000.

[18] M. Taylor. Is dark silicon useful? harnessing the four horsemen of the coming dark silicon apocalypse. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 1131–1136, June 2012.