SAT-MapIt: A SAT-based Modulo Scheduling Mapper for Coarse Grain Reconfigurable Architectures

Cristian Tirelli
SYS Institute
Università della Svizzera italiana
Lugano, Switzerland
cristian.tirelli@usi.ch

Lorenzo Ferretti
Computer Science
University of California, Los Angeles
Los Angeles, United States
ferrelo@cs.ucla.edu

Laura Pozzi
SYS Institute
Università della Svizzera italiana
Lugano, Switzerland
laura.pozzi@usi.ch

***Abstract*—Coarse-Grain Reconfigurable Arrays (CGRAs) are emerging low-power architectures aimed at accelerating compute-intensive application loops. The acceleration that a CGRA can ultimately provide, however, heavily depends on the quality of the mapping, i.e. on how effectively the loop is compiled onto the given platform. State of the Art compilation techniques achieve mapping through modulo scheduling, a strategy which attempts to minimize the II (Iteration Interval) needed to execute a loop, and they do so usually through well known graph algorithms, such as Max-Clique Enumeration.**

We address the mapping problem through a SAT formulation, instead, and thus explore the solution space more effectively than current SoA tools. To formulate the SAT problem, we introduce an ad-hoc schedule called the *kernel mobility schedule* (KMS), which we use in conjunction with the data-flow graph and the architectural information of the CGRA in order to create a set of boolean statements that describe all constraints to be obeyed by the mapping for a given II. We then let the SAT solver efficiently navigate this complex space. As in other SoA techniques, the process is iterative: if a valid mapping does not exist for the given II, the II is increased and a new KMS and set of constraints is generated and solved.

Our experimental results show that SAT-MapIt obtains better results compared to SoA alternatives in 47.72% of the benchmarks explored: sometimes finding a lower II, and others even finding a valid mapping when none could previously be found.

***Index Terms*—CGRA, Mapping, SAT**

I. Introduction

The constant growth of computational requirements in everyday applications has increased the demand for high-performance and low-power architectures, able to perform compute-intensive tasks efficiently while at the same time dealing with tight power/resource constraints.

While Application Specific Integrated Circuits (ASICs) accelerators have been largely adopted in these scenarios due to their efficiency, they are limited by fixed functionality. due to their fine-grained structure.

Coarse-Grain Reconfigurable Arrays (CGRAs), which are programmable architectures able to achieve high efficiency with low power requirements [1] [2], provide a meet-in-the-middle approach given their coarse-grain reconfigurability, and hence have become popular in various domains such as streaming and multimedia applications [1]–[6], as well as medical ones [7]. A CGRA is a mesh of Processing Elements (PE) organized in a two-dimensional grid; each PE contains an ALU (Arithmetic-Logic Unit) and a number of internal registers, and is connected to neighbors PEs according to its mesh topology, and to main memory through memory lines. Figure 1 shows the generic CGRA architecture targeted by our methodology.

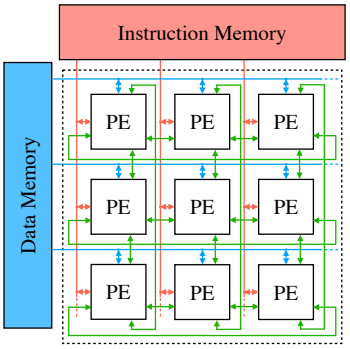

Fig. 1. Example 3×3 CGRA. Processing elements are connected in a 2D mesh with near-neighbour topology.

One of the fundamental challenges of CGRA exploitation is the compilation process, i.e. the translation of high level source code onto CGRA code, while taking advantage of the parallelism offered by the architecture. To do so, a technique called modulo scheduling is traditionally employed, which constructs a DFG from an application loop body, and then maps DFG nodes onto architecture PEs in an interleaved manner so that nodes from different iterations coexist in the same cycle, minimising the overall latency of each iteration.

Existing techniques for modulo scheduling, described in Section II, rely mainly on heuristics to schedule, place and then route instructions and data on the PEs. Herein, we propose to address the mapping problem using SAT-MapIt, a SAT-based formulation where data dependency, architectural constraints,

This work was supported by the Swiss National Science Foundation under Grants 200020-182009 and 200020-188613, and in part by the National Science Foundation under Grants P2TIP2_199735.

and schedule are expressed as Boolean constraints. A valid mapping is then identified by determining if an assignment of the variables exists to satisfy the constructed boolean formula. We show that this technique is able to efficiently explore the space of possible mappings better than SoA techniques, hence producing high-performance schedules.

II. Related Work

Existing methodologies for the CGRA mapping problem can be divided in two main categories: the first using heuristics, the second using exact solutions. Early works in the first category include the method proposed by Mei et al. [8], which formulates scheduling, placement, and routing problems altogether, and proposes to solve it using simulated annealing. Alternatively, in [9] an edge-centric approach to modulo-scheduling was presented, where a schedule is generated by first routing each edge in the dataflow graph, and placement is addressed as a by-product of routing. In [10], EPImap proposed to use both routing and re-computation to find a valid mapping, by adopting an epimorphic (time-extended) graph formulation.

Epimap's performance was later improved by GraphMinor [11] and REGIMap [12], by reducing the mapping problem to the graph minor and max clique problem. In turn, [13] RAMP authors have further refined REGIMap by explicitly modeling and exploring various routing strategies and choosing the best one for each given loop kernel. CRIMSON [14] then proposed a randomized iterative modulo scheduling algorithm that explored the scheduling space more efficiently, and PathSeeker [15] improved on [14] by analyzing mapping failures and performing local adjustments to the schedule to obtain a lower compilation time and a better quality of the solution.

In our experiments we compare our results to those obtained by both RAMP [13] and PathSeeker [15]. These two works had shown superior performance with respect to the earlier methodologies mentioned above, and therefore represent the current SoA of the modulo scheduling mapping problem. We quantitatively compare our work to them, and show that SAT-MapIt can better explore the scheduling space and get smaller II (Iteration Interval) by using custom scheduling tables with a SAT formulation.

A second category of approaches addresses the mapping problem with Integer Linear Programming (ILP) or Boolean Satisfiability formulations. In [16] the authors propose an ILP formulation approach and prove the feasibility of mapping in the given number of cycles. Similarly, [17] propose to use a SAT solver instead of an ILP solver to identify a valid solution. The work proposed in [17] represents a first effort towards exact formulations using a SAT solver; however, it is not capable of achieving a modulo scheduled solution. Our SAT formulation is, to the best of our knowledge, the first to propose an exact solution to the Modulo Scheduling problem, and yet scale to Data Flow Graph sizes that were previously only tackled via heuristics.

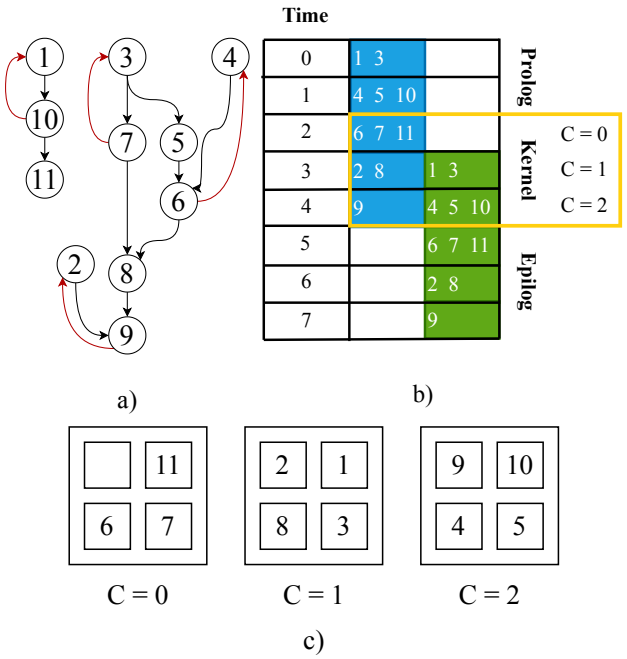

Fig. 2. a) Example Data Flow graph of a loop. b) Modulo scheduling of the DFG on the left, highlighting the division among Prolog, Kernel, and Epilog. The loop is unrolled once; the first iteration is in blue, and the second, shifted by II, is in green. c) Mapping example on a 2×2 CGRA

III. Background

In this section we provide the background needed to present our methodology, and we illustrate it wherever appropriate through a running example.

A. Compilation

In order to accelerate an application onto a CGRA, a compute-intensive loop is identified in the application; the identification can either be performed automatically via techniques such as for example [18], or manually by the programmer via pragma-annotations, as done in this work. Then, the loop needs to be compiled, in order to be translated onto CGRA instructions.

The first step in this process is to generate a semantically-equivalent version in Intermediate Representation (LLVM IR in our case) and from there to Data Flow Graph (DFG) depicted in Figure 2a. DFGs are directed graphs in which nodes represent instructions, edges represent dependency relations between instructions, and back-edges correspond to loop-carried dependencies.

In a second phase, the generated DFG is mapped onto the CGRA, by assigning each of its nodes to a given PE at a given cycle. In order to perform such mapping, a technique called modulo scheduling is employed.

B. Modulo Scheduling

Modulo Scheduling (MS) is a compilation technique that enables efficient execution of a loop body, by executing multiple iterations of it in an interleaved manner. As exemplified in Figure 2b, a modulo-scheduled loop is divided into three

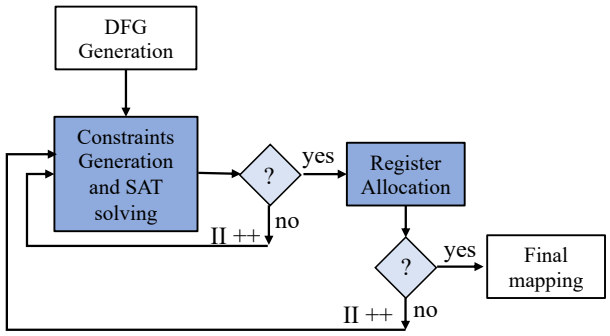

Fig. 3. SAT-MapIt searches for mappings for a given II, iteratively increasing II in case the SAT solver returns UNSAT, or register allocation fails to colour the model returned by the solver.

stages: prologue, kernel, and epilogue. Prologue and epilogue are one-time executed stages: the former is used to prepare the data to feed the pipeline, the latter to reorganize them at the end. The kernel is instead repeated multiple times and includes the instructions to be parallelized through pipelining. Goal of MS is to pipeline as effectively as possible the execution of kernel instructions, and this corresponds to minimizing the Iteration Interval (II), i.e. the length of the kernel stage, which is 3 cycles in our example. The mapping problem hence consists in finding a legal Modulo Schedule for a loop, performing the placement and routing of instructions in a constrained 3D space represented by the PE dimensions and by time. An example of legal mapping for the DFG in the running example is shown in Figure 2c.

IV. Methodology

A. Overview

Our methodology addresses mapping by solving a SAT problem where data dependency, schedule and CGRA architecture are expressed as Boolean constraints in a conjunctive normal form (CNF). Our toolchain, depicted in Figure 3, takes as input the C code of the application, converts it into LLVM IR, extracts the designated loop structures and, through a custom LLVM pass, generates its DFG.

The information retrieved from the LLVM pass is then used to build a set of schedules, in turn translated into a set of constraints, which get finally fed to a SAT solver. If the answer of the solver is SAT, the next step is to verify that there are enough registers available in the PEs to store the generated data for the whole liveness duration, and this is determined via Register Allocation. If this step succeeds, then a valid mapping has been identified. Otherwise, the current Initiation Interval is increased and the process is iteratively repeated.

Herein, we detail the various steps of our methodology: schedule creation, SAT formulation, and register allocation.

B. Schedule creation

We first create As-Soon-As-Possible (ASAP) and As-Late-As-Possible (ALAP) schedules, for the input DFG. This corresponds to detecting how early and how late each node can be scheduled. We then generate the Mobility Schedule (MS), which expresses the mobility of each node from its $ASAP$ to its $ALAP$ time position. Figure 4 shows these schedules for our running example.

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

Fig. 4. ASAP, ALAP, and Mobility Schedule

At this point we create the Kernel Mobility Schedule (KMS): a custom structure which SAT-MapIt uses to then formulate the mapping problem. The KMS can be seen as *a superset of all possible kernels*, and is the product of iteratively folding MS by an amount equal to II: every time MS is folded by II into KMS, each node receives a label that refers to the iteration number it belongs to. The more folding we have, the more iterations our kernel will execute at the steady-state. This process is depicted in Figure 5, again for our running example. Given an II of 3, the MS is folded twice ($\lceil 5/3 \rceil = 2$) and hence the KMS contains two iterations. The first is depicted in blue, and the second in green.

Together, DFG and KMS are used to generate all the statements of the CNF formulation of the mapping problem, which is the subject of the next subsection.

MS

Time	Nodes
0	1 2 3 4
1	1 2 4 5 7 10
2	1 2 6 7 10 11
3	2 8 10 11
4	9 11

KMS

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

Fig. 5. Kernel Mobility Schedule creation.
In blue iteration 0, in green iteration 1

C. SAT formulation

We create a CNF formula using literals in the form: $x_{n,p,c,it}$, where n denotes the node identifier in the DFG, p denotes a PE on the CGRA, c represents at which cycle a node is scheduled, and it to which iteration the node refers to.

Our problem formulation can be described at a high level by partitioning all statements into three main sets of clauses that assure the following:

- C1: Every node is associated with a set of literals, and for each one of those sets, one and only one literal must be set to *True*.
- C2: At most one node should be assigned to a PE at a given cycle, since two or more nodes cannot be scheduled simultaneously on the same PE.
- C3: Each node's predecessor and/or successor must be assigned to a neighbor or on the same PE.

To provide a formal description of the above constraints, we introduce additional definitions. Let $\mathcal{L}$ be the set of all literals, then $\mathcal{L}(n)$ is the set of all literals associated to node n.

To make the notation more compact and easy to read, we also associate each literal in the form $x_{n,p,c,it}$ to a literal written as v_i. For example $\mathcal{L}(n_3)$ would be written as:

$$\mathcal{L}(n_3) = \{v_0, v_1, v_2, v_3\}$$

where $v_0 = x_{3,0,1,1}$, $v_1 = x_{3,1,1,1}$, $v_2 = x_{3,2,1,1}$ and $v_3 = x_{3,3,1,1}$. This represents the fact that node 3 appears only at time 1 and only at iteration 1 in the KMS, as shown in Figure 5, and that it can be mapped onto any PE: 0,1,2,3. Now we can start the description of the three sets of constraints. The first set, C1, ensures that all nodes are mapped on the CGRA, and can be encoded formally with:

$$\begin{aligned} \phi(n) &= \bigvee_{v_i \in \mathcal{L}(n)} v_i \\ \xi(n) &= \bigwedge_{(v_i,v_j) \in \mathcal{M}(n)} \neg(v_i \wedge v_j) \\ \zeta(n) &= \phi(n) \wedge \xi(n) \end{aligned} \tag{1}$$

where n is one of the nodes id in the DFG and $\mathcal{M}(n)$ is defined as follows:

$$\mathcal{M}(n) = \{(v_i, v_j) : v_i \prec v_j, (v_i, v_j) \in \mathcal{L}(n) \times \mathcal{L}(n)\}$$

where $v_i = x_{n,p_1,c_1,it_1}$ and $v_j = x_{n,p_2,c_2,it_2}$ with $p_1 \neq p_2$, $c_1 \neq c_2$ and $it_1 \neq it_2$. Furthermore the symbol $\prec$ represents the *lexicographically smaller-than* relation between two literals. For example $x_{3,0,1,0}$ is lexicographically smaller than $x_{3,1,0,0}$, while $x_{3,1,0,1}$ is not.

Equation 1 will be used on each node of the DFG and each ζ generated will be added to the SAT formulation.

The second set of constraints, C2, forbids the mapping of more than a single node on the same PE at the same time, and is encoded through:

$$\begin{aligned} M(n,m) &= \bigwedge_{(v_i,w_j) \in \mathcal{V}(n,m)} \neg(v_i \wedge w_j) \\ \zeta &= \bigwedge_{n}^{N-1} \bigwedge_{m=n+1}^{N} M(n,m) \end{aligned} \tag{2}$$

with $\mathcal{V}(n,m)$ defined as:

$$\mathcal{V}(n,m) = \{(v_i, w_j) : v_i \prec w_j, v_i \in \mathcal{L}(n), w_j \in \mathcal{L}(m)\}$$

The last set of constraints, C3, handles the dependencies in the DFG, and assures that 1) each data dependency is mapped on neighbors PE on the CGRA and that 2) data produced by a node in a given cycle is consumed before being overwritten by another node on a subsequent cycle. For each dependency, we consider literals that are at most one iteration apart in the KMS, that are on a neighbour PE and that respect one of the relations:

$$c_d \leq c_s \text{ if } it_s \neq it_d \quad or \quad c_d > c_s \text{ if } it_s = it_d \tag{3}$$

where c_d is the cycle at which the destination node is scheduled, and c_s is the cycle at which the source node is scheduled. This constraint ensures that a node consumes the value produced by the predecessor in the proper order, avoiding overlapping of the same dependencies among kernel and prologue/epilogue stages.

The CNF in this case is composed of two main terms; one that handles the case in which the output of the source node is delivered to the destination node through the registers on the PE, and the other in which the data created by the source node is written in the output register of the PE, instead, and hence it must not be overwritten in subsequent cycles.

Dependencies that use internal registers can be encoded through a set of CNF of this form:

$$\zeta_1 = v_i \wedge w_j \tag{4}$$

where $v_i = x_{n_s,p_1,c_1,it_1}$ and $w_j = x_{n_d,p_2,c_2,it_2}$ and p_1 is neighbour of p_2. On the other hand dependencies that do not use internal registers need to be encoded with a slightly different formulation. In particular we expand Equation 4 by appending another term. The CNF in this case become:

$$\zeta_2 = v_i \wedge w_j \wedge \neg \left(\bigvee z_k \right) \tag{5}$$

where $z_k = x_{n_l,p1,c_l,it_l}$ iterate over all literals in all subsequent cycles, with the same PE as source node. This assures that the data is properly consumed by the destination node without being overwritten. Equation 4 and 5 are repeated for each dependency in the KMS and every CNF generated is added to the formulation after an *or* operation among all the terms, so that the solver is free to choose the option that makes the mapping problem satisfiable.

D. Register allocation

Once the solver returns a satisfying mapping, a last phase is needed in order to validate such mapping in terms of register usage: Register Allocation – implemented in SAT-MapIt as a separate phase subsequent to SAT solving. It is solved optimally by exploiting the SSA format of the input code [19]. For each PE in the CGRA, we generate an interference graph that we proceed to color. If the coloring succeeds, no further action is needed, and the coloring hence generated concludes the mapping, by adding information on which register should hold the output of each PE at each cycle. If it fails, however, we split the overlapping intervals that make the interference graph uncolorable with the given number of colors, by adding load and stores, and hence additional cycles.

V. Experimental Results

Experimental Setup. We evaluate the effectiveness of SAT-MapIt on a set of loop kernels from MiBench and Rodinia benchmark suites. We compare the obtained II, and the time to find it, with respect to two techniques of the SoA: RAMP [13] and PathSeeker [15]. For these two SoA techniques we use the original code publicly released by the authors. In the target CGRA architecture that we consider in our experiments, each PE is connected to the four nearest neighbors, as in Figure

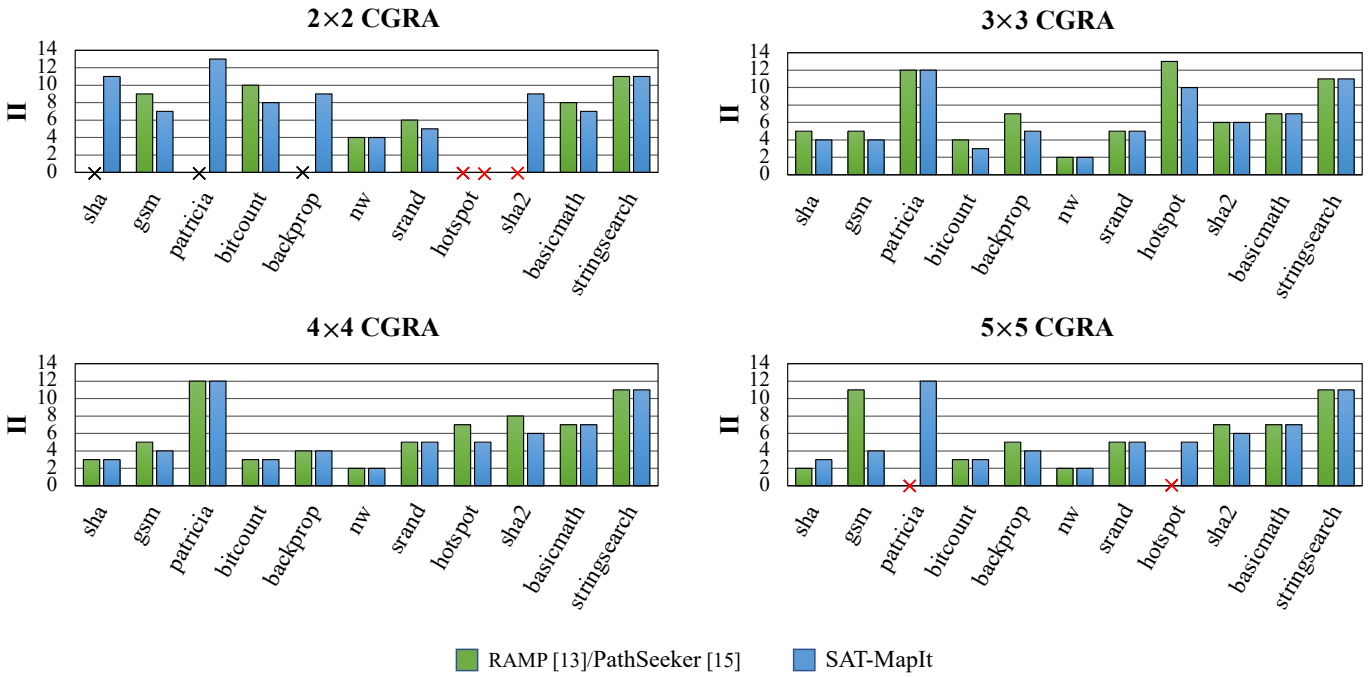

Fig. 6. Experimental results of the chosen benchmarks for different architecture sizes. We compare the II found by SAT-MapIt with respect to the best results obtained by RAMP and PathSeeker – lower is better. A red mark means that the process did not terminate before a timeout of 4000 seconds. A black mark means that the process was terminated when it reached a current II of 50 but still found no feasible solution.

Benchmarks	[13]/[15]	SAT-MapIt	Δ
sha	41.01	3.22	-37.79
gsm	2.81	1.25	-1.56
patricia	1351.15	5.39	-1345.76
bitcount	2.63	1.68	-0.95
backprop	1262.69	3.39	-1259.3
nw	0.01	0.56	0.55
srand	0.32	1.15	0.83
hotspot	4000	4000	0
sha2	4000	2.21	-3997.79
basicmath	0.01	0.62	0.61
stringsearch	0.19	1.02	0.83

TABLE I
MAPPING TIME (SECONDS) ON A 2 × 2 CGRA

Benchmarks	[13]/[15]	SAT-MapIt	Δ
sha	0.23	2.86	2.63
gsm	4.14	4.35	0.21
patricia	48.98	16.31	-32.67
bitcount	5.86	7.84	1.98
backprop	44.62	12.27	-32.35
nw	0.03	1.56	1.53
srand	0.09	3.9	3.81
hotspot	13.19	28.43	15.24
sha2	61.7	3.13	-58.57
basicmath	0.07	1.9	1.83
stringsearch	3.27	3.55	0.28

TABLE II
MAPPING TIME (SECONDS) ON A 3 × 3 CGRA

Benchmarks	[13]/[15]	SAT-MapIt	Δ
sha	32.04	7.23	-24.81
gsm	32.04	10.46	-21.58
patricia	421.05	39.28	-381.77
bitcount	0.44	21.55	21.11
backprop	57.17	25.1	-32.07
nw	0.08	3.63	3.55
srand	0.25	8.79	8.54
hotspot	3556.62	3734.77	178.15
sha2	696.6	7.19	-689.41
basicmath	0.22	4.11	3.89
stringsearch	0.02	7.62	7.6

TABLE III
MAPPING TIME (SECONDS) ON A 4 × 4 CGRA

Benchmarks	[13]/[15]	SAT-MapIt	Δ
sha	6.25	28.93	22.68
gsm	0.29	21.4	21.11
patricia	4000	75.16	-3924.84
bitcount	0.01	47.62	47.61
backprop	981.73	52.43	-929.3
nw	0.02	7.75	7.73
srand	0.02	21.64	21.62
hotspot	4000	108.02	-3891.98
sha2	675.12	16.88	-658.24
basicmath	0.5	8.7	8.2
stringsearch	0.02	15.3	15.28

TABLE IV
MAPPING TIME (SECONDS) ON A 5 × 5 CGRA

1, and each PE contains four local registers. We vary the size of the mesh from 2×2 up to 5×5. The Z3 solver is used to solve our SAT formulation. All experiments are performed on a machine with 2.6 GHz 6-Core Intel Core i7. For PathSeeker, each experiment was repeated 10 times given its randomized nature.

SAT-MapIt achieves better IIs. The performance of a mapping is first and foremost measured by the II achieved, because this, in turn, is a measure of the level of parallelism obtained. In our experiments we compare the II of SAT-MapIt with those of SoA, for each benchmark explored. This is depicted in Figure 6 which shows the performance obtained by all techniques for different CGRA configurations. For SoA, we report the best result among the two algorithms. Our tool is able to systematically find the best mapping solution.

SAT-MapIt uses tight resources better. By focusing on the 2×2 size CGRA, we can see that SAT-MapIt always finds the best solution, and twice (`patricia` and `backprop`) it is even able to find a valid mapping where the SoA could not. This showcases the effectiveness of our methodology particularly when the mapping problem becomes more challenging.

SAT-MapIt is faster when runtimes are high. Given that the II found are better than SoA, we now analyse the time needed to find such solutions, and report it in Tables I, II, III and IV. It can be noticed that our tool running time is longer than SoA in 26 out of 44 experiments, and that in these 26 cases the average time difference is only 15.28 seconds, with a standard deviation of 34.97. On the other hand, in the 18 cases in which our tool is faster, the average time difference is 962.24 seconds, with a standard deviation of 1438.78. This shows that SAT-MapIt is significantly faster when it matters, i.e. when computation times are high.

Limitations of SAT-MapIt Currently, our tool does not apply any routing strategy. This limitation manifests in the `sha` kernel of a 5×5 CGRA, where we achieve an II of 3, while SoA can find an II of 2 by adding a routing node. This is the only case, out of the 44 experiments shown, where the effect of this limitation can be noticed.

VI. Conclusion

In this paper we present a tool, called SAT-MapIt, for modulo-scheduling loops onto CGRAs. We find that previous techniques, mainly based on classic graph algorithms such as Max-Clique enumeration, do not always explore the scheduling space effectively. We propose a new SAT formulation of the modulo scheduling problem on CGRA that fully explores the scheduling space and finds the lowest II possible for a given DFG. To define the mapping problem through a SAT formulation, we introduce a new custom schedule called Kernel Mobility Schedule, which is used with the data-flow graph of the loop to be mapped, and with the architectural information of the CGRA, to generate all the constraints that the SAT solver needs to obey. Overall, SAT-MapIt finds better solutions than the SoA alternatives [13] [15], achieving better results in 47.72% of the cases, and even identifying valid mappings where other tools could not find a valid solution.

References

[1] Z. Li, D. Wijerathne, X. Chen, A. Pathania, and T. Mitra, "ChordMap: Automated Mapping of Streaming Applications onto CGRA," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 306–319, 2021.

[2] M. Karunaratne, A. K. Mohite, T. Mitra, and L.-S. Peh, "HyCUBE: A CGRA with reconfigurable single-cycle multi-hop interconnect," in *Proceedings of the 54th Design Automation Conference*, 2017, pp. 1–6.

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

[4] T. Oh, B. Egger, H. Park, and S. Mahlke, "Recurrence cycle aware modulo scheduling for coarse-grained reconfigurable architectures," in *Proceedings of the 2009 Conference on Languages, Compilers, and Tools for Embedded Systems*, 2009, pp. 21–30.

[5] H. Lee, D. Nguyen, and J. Lee, "Optimizing stream program performance on CGRA-based systems," in *Proceedings of the 52th Design Automation Conference*, 2015, pp. 1–6.

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

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

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

[9] H. Park, K. Fan, S. A. Mahlke, T. Oh, H. Kim, and H.-s. Kim, "Edge-centric modulo scheduling for coarse-grained reconfigurable architectures," in *Proceedings of the 17th International Conference on Parallel Architecture and Compilation Techniques*, 2008, pp. 166–176.

[10] M. Hamzeh, A. Shrivastava, and S. Vrudhula, "EPIMap: Using Epimorphism to map applications on CGRAs," in *Proceedings of the 49th Design Automation Conference*, 2012, pp. 1284–1291.

[11] L. Chen and T. Mitra, "Graph minor approach for application mapping on CGRAs," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 7, no. 3, pp. 1–25, 2014.

[12] M. Hamzeh, A. Shrivastava, and S. Vrudhula, "REGIMap: Register-aware application mapping on coarse-grained reconfigurable architectures (CGRAs)," in *Proceedings of the 50th Design Automation Conference*, 2013, pp. 1–10.

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

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

[15] ——, "PathSeeker: A Fast Mapping Algorithm for CGRAs," *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pp. 268–273, 2022.

[16] S. A. Chin and J. H. Anderson, "An Architecture-Agnostic Integer Linear Programming Approach to CGRA Mapping," in *Proceedings of the 55th Design Automation Conference*, 2018, pp. 1–6.

[17] Y. Miyasaka, M. Fujita, A. Mishchenko, and J. Wawrzynek, "SAT-Based Mapping of Data-Flow Graphs onto Coarse-Grained Reconfigurable Arrays," in *International Conference on Very Large Scale Integration-System on a Chip*. Springer, 2020, pp. 113–131.

[18] G. Zacharopoulos, L. Ferretti, E. Giaquinta, G. Ansaloni, and L. Pozzi, "RegionSeeker: Automatically identifying and selecting accelerators from application source code," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 4, pp. 741–754, 2018.

[19] S. Hack, D. Grund, and G. Goos, "Register allocation for programs in SSA-form," in *International Conference on Compiler Construction*. Springer, 2006, pp. 247–262.