Standalone Nested Loop Acceleration on CGRAs for Signal Processing Applications

Chilankamol Sunny, Satyajit Das, Kevin J M Martin, Philippe Coussy

Standalone Nested Loop Acceleration on CGRAs for Signal Processing Applications

Chilankamol Sunny[1][0000−0003−3826−3810], Satyajit Das[1][0000−0002−7550−2641], Kevin J. M. Martin[2][0000−0002−8122−1192], and Philippe Coussy[2][0000−0002−7222−5271]

[1] IIT Palakkad, Kerala, India
112004004@smail.iitpkd.ac.in, satyajitdas@iitpkd.ac.in
[2] Univ. Bretagne-Sud, UMR 6285, Lab-STICC, F-56100 Lorient, France
{kevin.martin,philippe.coussy}@univ-ubs.fr

Abstract. Coarse-Grained Reconfigurable Array (CGRA) architectures are becoming increasingly popular as low-power accelerators in compute and data intensive application domains such as security, multimedia, signal processing, and machine learning. The efficiency of a CGRA is determined by its architectural features and the compiler's ability to exploit the spatio-temporal configuration. Numerous design optimizations and mapping techniques have been introduced in this direction. However, the execution model has been overlooked, despite its critical role in ensuring the efficient acceleration of applications. Most of the existing CGRA implementations follow a hosted approach i.e., they execute the modulo scheduled innermost loop, entrusting outer loops to the host processor. This increases synchronization overhead with the host, mitigating the benefits of acceleration provided by the CGRA. In this paper, we propose a compilation flow that supports efficient standalone execution of nested loops. Experiments show that the standalone execution model leads to a maximum of 12.33× and an average of 6.75× performance improvement compared to the existing hosted execution model. In the proposed model, energy consumption is reduced up to 14.49× compared to that of the hosted one. We also compared our results with state-of-the-art standalone execution that uses loop flattening and achieved a maximum of 4.80× speed up with an average of 2.80×.

Keywords: Coarse grained reconfigurable array (CGRA) · Nested loop acceleration · Standalone execution model.

1 Introduction

Due to the architectural elasticity, Coarse-Grained Reconfigurable Array (CGRA) architectures offer high performance, energy efficiency, and flexibility [13]. A typical CGRA integrated system consists of an array of interconnected processing elements (PEs) tightly/loosely coupled with a host CPU, a context, and a data memory. Each PE is composed of a word-level configurable functional unit (FU), a regular register file, an instruction memory, and routers at the input and output. CGRAs have been proposed to cater to the needs of both High Performance Computing (HPC) and Low Power Computing (LPC) domains with various architectural and compilation novelties. Architectural improvements like dynamic voltage and frequency scaling (DVFS) [19], approximate arithmetic units [1] and efficient memory hierarchies [2] have been proposed to improve the performance and energy efficiency of CGRAs. To improve the performance of the compute-intensive applications, several loop optimization techniques have been adopted in the compilation flow, such as loop unrolling [6], modulo scheduling [8] and polyhedral loop optimizations [12]. However, most of the state-of-the-art CGRA compilation flow [8,9,18] uses modulo scheduling loop optimization due to its good performance for the innermost loop.

The majority of the existing CGRAs focus on the optimized innermost loop execution, entrusting outer loops to the host processor. This increases the synchronization overhead, diminishing the benefits of acceleration provided by the CGRA [4]. Hence, optimized mapping techniques and improved architectural designs are not sufficient to guarantee the best performance. The execution model, which defines how tasks or processes are partitioned, scheduled, and executed (includes configuration and execution time) on CGRAs, is equally important. This paper discusses and analyses the standalone and hosted execution models for CGRAs. In the standalone model, the entire nested loop structure is run on the CGRA with no host intervention. In the hosted model, CGRAs execute only the innermost loop letting the host processor execute the outer loops.

The major contributions of this work are: (a) An explorative study on the impact of execution models in determining the performance and energy efficiency of CGRAs. We demonstrate that a highly efficient mapping technique may not be sufficient to guarantee the best performance. The execution model is equally important, especially for kernels with deeply nested loops which is the case with most modern signal processing and AI applications [20]. (b) A compilation flow for CGRAs that supports the standalone execution of nested loops. The proposed approach modulo schedules the innermost loop using traditional modulo scheduling algorithms and executes loops at all levels of loop nesting on the CGRA, with no host intervention.

The rest of the paper is organized as follows. Section 2 presents the background and related works. Section 3 introduces the proposed compilation flow. The experiments and results are discussed in Section 4. Section 5 concludes the paper.

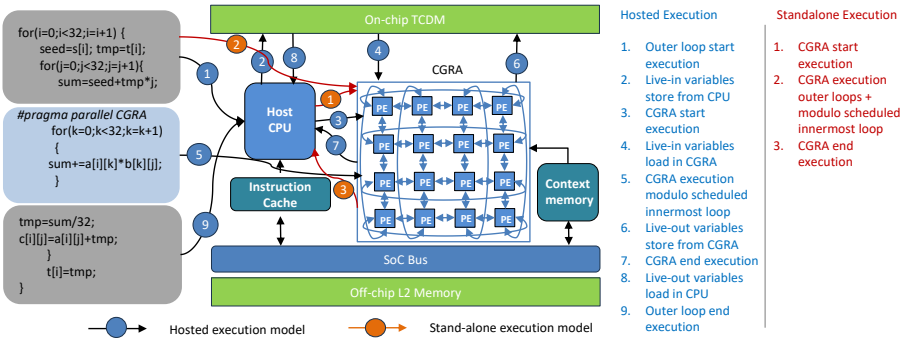

Fig. 1. Hosted and standalone execution model for CGRA loosely coupled with CPU

2 Background and Motivation

CGRAs, due to their architectural specialization, efficiently execute the pipelined innermost loop (single loop). Traditional CGRAs target to accelerate only the innermost loop for applications with nested loops, leaving outer loops for the host processor. This is referred to as *hosted* execution model (Fig. 1) in this paper. In this execution model, the variables needed for the CGRA to execute the innermost loop (*live-in variables*), and the variables processor needs from CGRA (*live-out variables*) are transferred through shared memory. The overhead due to added memory operations and communication for synchronization are shown in Fig. 1. To minimize the overhead, proposals like [12,10,21] perform several loop transformations (i.e. polyhedral transformation, loop flattening, loop fission). However, with the growing complexity of the loop nests in signal processing applications, in addition to the transformations, we need mechanisms to minimize the host intervention. Hence, the *standalone* execution of the entire loop nests is the ideal solution as presented in Fig. 1.

For the hosted execution of CGRAs, the works [5,8,14] perform *modulo scheduling* on the innermost loop. It is a software pipelining technique that facilitates overlapped execution of different iterations of a loop. The goal of modulo scheduling is to find a schedule of operations from different iterations of the innermost loop that can be repeated in a short interval called initiation interval (II), expressed in cycles. The data flow graph (DFG) formed by this repeating schedule is referred to as Modulo Data Flow Graph (MDFG). The set of operations that are executed once before and after the MDFG, form a couple of DFGs and are referred to as prologue and epilogue respectively [17]. As the hosted execution only executes the innermost loop, mapping of the MDFG onto the target CGRA is considered a DFG mapping problem [9,8]. The prologue and epilogue mappings are adapted from the mapped MDFG [16]. Fig. 2(d) shows the replication of MDFG mapping in Fig. 2(c). The innermost loop DFG in this example is presented in Fig. 2(a). Fig. 2(b) shows the *prologue epilogue*, and *MDFG*

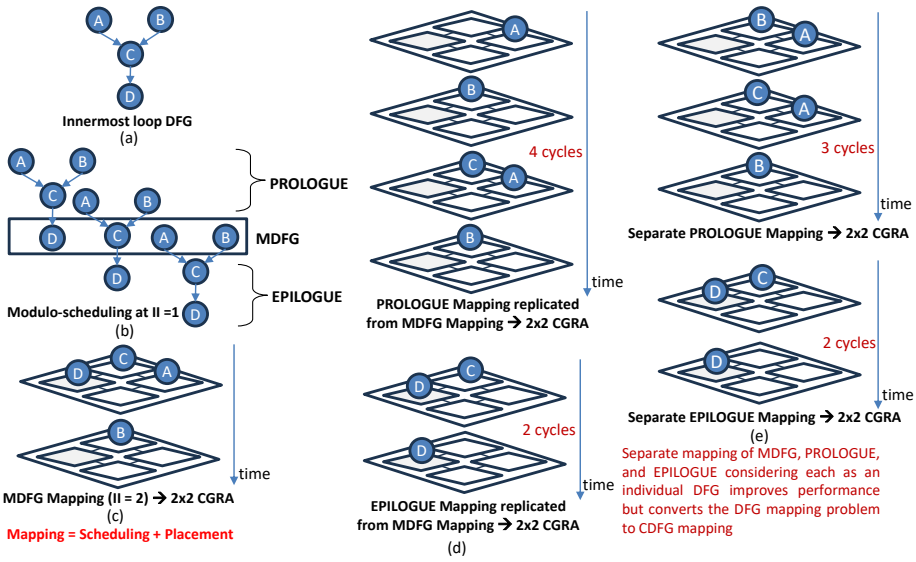

Fig. 2. Modulo scheduling example with MDFG, prologue, epilogue mapping

after modulo scheduling. In this example, the two-cycle-long MDFG mapping (Fig. 2(a)) is replicated twice to prepare the prologue mapping, resulting in a schedule length of 4 (Fig. 2(d)). Similarly, the epilogue mapping is also prepared from the MDFG mapping with a schedule length of 2. However, replicating the MDFG mapping does not always guarantee the optimum solution. As shown in Fig. 2(e), a mapping solution of schedule length 3 is obtained by mapping the prologue DFG directly onto the CGRA rather than adapting from the MDFG mapping which resulted in a higher schedule length. The larger schedule length may seem very less in a single-nested loop. However, for loop nests if the iteration count increases, the cumulative effect of the larger schedule results in degraded performance. Thankfully, prologue and epilogue DFGs always contain a smaller number of operations per cycle compared to the MDFG due to the inherent construct of MDFG. As the number of nodes is less, mapping the prologue and epilogue as individual DFGs results in a lower schedule length. This is a CDFG mapping problem where, the prologue, epilogue, and MDFG are considered as individual DFGs, and while mapping, control flow between the DFGs needs to be satisfied. In this paper, the standalone execution is achieved by the mapping of the entire application CDFG along with the modulo scheduled innermost loop kernel.

The state-of-the-art solution, Cheng et al [18] proposes support for the standalone execution where the loop nests are flattened into a single-nested loop (Fig. 3) to facilitate the DFG mapping. The resultant DFG is modulo scheduled and executed on the CGRA. The solution suffers from inflated DFG when the

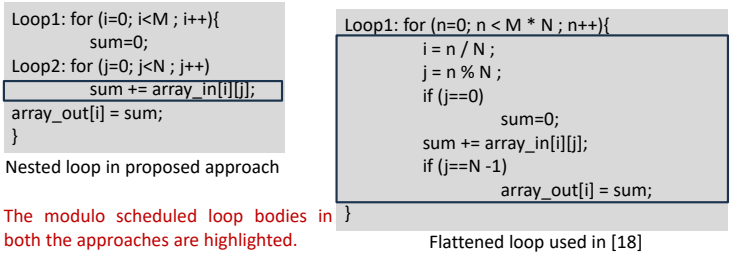

Fig. 3. Example of where modulo scheduling is applied in the proposed and loop transformation based standalone execution models

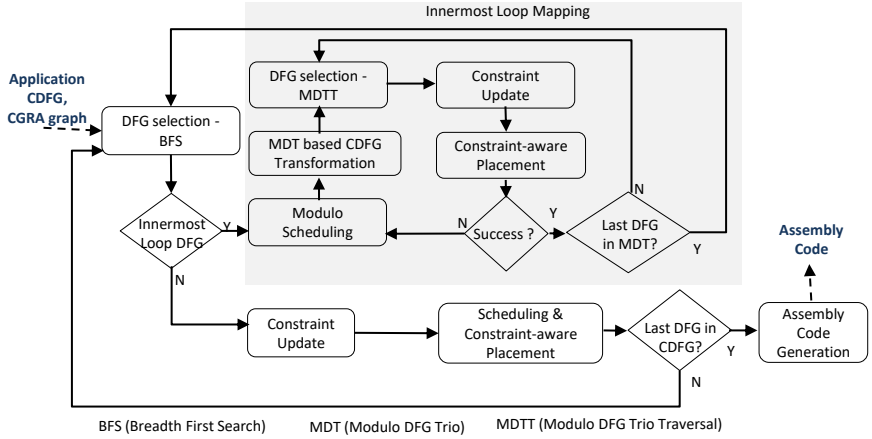

Fig. 4. Proposed compilation flow supporting standalone execution of nested loops and modulo scheduling of innermost loop

number of loops gets increased causing increased II and high energy consumption. IPA [3] approach proposes to support the standalone execution of nested loops using *register allocation-based direct-mapping* of CDFG onto CGRAs achieving good performance and energy results. However, the compilation flow proposed in that work uses partial loop unrolling instead of modulo scheduling the innermost loop. In this paper, we extend the solution proposed in the IPA [3] to support the standalone execution of the entire loop nests with the innermost loop modulo scheduled (Fig. 3).

3 Proposed Approach

3.1 Overview

We introduce a novel compilation flow for CGRAs that supports standalone execution of the entire application containing loop nests. The innermost loop uses modulo scheduling for the pipelined implementation.

As the problem is defined as a CDFG mapping problem, we adapt the register allocation-based approach proposed in Das et al [3] where the basic blocks (BB) (individual DFGs) are mapped onto CGRA with a simultaneous scheduling and placement algorithm. The variables that are used in multiple BBs (*symbol variables*) are mapped in registers using *target location*, and *reserved location* constraints in placement. In this paper, we propose a mapping of modulo scheduled innermost loop DFG along with the other BBs in the CDFG. The primary challenge in this strategy is to integrate the MDFG, prologue, and epilogue generated by modulo scheduling in the original kernel CDFG and traverse efficiently to find valid mappings for the entire CDFG. To meet this challenge, we have designed a CDFG transformation and traversal technique which are explained in the following section.

The proposed compilation flow comprises two tracks as depicted in Fig. 4, one for the innermost loop mapping and the other for mapping the rest of the CDFG. The first step in the compilation flow is to choose the BB for mapping. The selection is done by the breadth-first search (BFS) traversal of the CDFG, the technique proven to generate the least number of constraints in CDFG mapping [3]. If the selected BB (DFG) corresponds to the innermost loop, it is first modulo scheduled and then placed onto the CGRA, respecting the register allocation constraints imposed by the mappings of already mapped BBs. Every other BB is mapped by following a simultaneous scheduling and placement approach [3]. The data integrity over different BB mappings is maintained by the constraint-aware placement technique. The constraint update step in the compilation flow sets the register allocation constraints that guide the choice of registers in the placement process. These constraints ensure reserved usage of registers for variables that are used in multiple BBs. Once all BBs are mapped, the compiler generates the assembly code for the entire CDFG mapping.

Simultaneous Scheduling and Placement A priority-based list scheduling algorithm is used to schedule the DFG nodes and an incremental version of Levi's algorithm [11] is used for placement and routing (binding). Failing to bind a node, the compilation flow transforms the DFG dynamically and continues with the mapping process. If a transformation that improves the mapping possibilities cannot be identified, backtracking is performed, and mapping restarts with a new mapping context. This is done by choosing the next BB from the set of previously mapped BBs. If the mapping is successful, a stochastic pruning is applied on the partial mapping set to prevent it from growing exponentially.

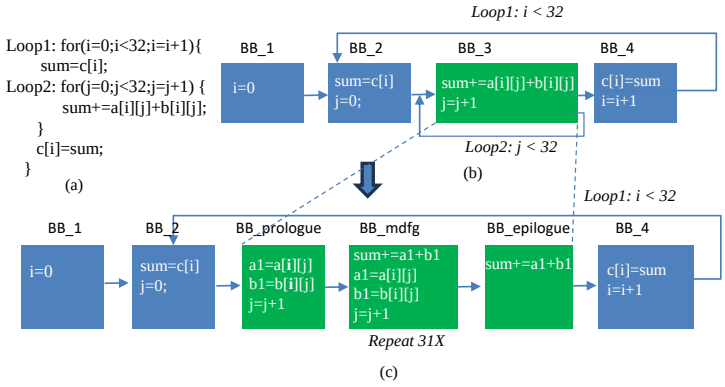

Fig. 5. MDT based CDFG transformation: (a) Sample for loop; (b) corresponding CDFG; (c) CDFG after MDT based transformation

3.2 Innermost Loop Mapping

If the DFG selected for mapping corresponds to the innermost loop, it is modulo scheduled [16] and placed onto the CGRA, following the innermost loop track in the compilation flow.

Modulo Scheduling Modulo scheduling starts with computing the minimum possible II (MII), determined by the resource and recurrence constraints of the input DFG and the target CGRA. The nodes in the DFG are then modulo scheduled [16] with an II equal to MII. Failing to find a schedule or placement solution for this II, the process is restarted with an incremented II and repeated until a valid mapping is found. Modulo Scheduling a DFG splits it into three DFGs, the prologue, MDFG, and epilogue, together called Modulo-DFG-Trio (MDT) in the discussion hereafter. This calls for a local graph transformation in the CDFG.

MDT based CDFG Transformation Fig 5 illustrates the MDT based CDFG transformation we introduce to apply modulo scheduling in conjunction with direct CDFG mapping. The figure presents a sample *for* loop and the corresponding CDFG. Fig 5 (c) gives the transformed CDFG in which the innermost loop DFG is replaced with the MDT. Edges connecting these DFG nodes are set such that the control flows from the outer loop BB to prologue, from prologue to MDFG, and from MDFG to epilogue. The epilogue DFG is connected to the immediate successor of the innermost loop DFG in the original CDFG. MDFG is executed multiple times, determined by the number of times the innermost loop DFG is unrolled to prepare the modulo schedule.

DFG Selection by Modulo DFG Trio Traversal (MDTT) Unlike the conventional approach of preparing the prologue and epilogue mappings from the

Table 1. Modulo DFG Trio (MDT) Traversal explained

Let P, M and E be the set of all variables in prologue, MDFG and epilogue respectively; S be the set of all symbol variables (variables used in multiple BBs) in the MDT.	
M = P ∪ E S = (P ∩ M) ∪ (M ∩ E) ∪ (P ∩ E) P ∩ M = P ; M ∩ E = E; P ∩ E ⊆ M Therefore S = P ∪ E ∪ (P ∩ E) = M	#symbol variables in prologue = n(S ∩ P) = n(M ∩ P) = n(P) in MDFG = n(S ∩ M) = n(M ∩ M) = n(M) in epilogue = n(S ∩ E) = n(M ∩ E) = n(E) n(M) = n(P) + n(E) – n(P ∩ E) => n(M) >n(P) & n(M) >n(E)
Modulo DFG Trio Traversal (MDTT) MDFG –>Prologue –>Epilogue; if n(P) >n(E) MDFG –>Epilogue –>Prologue; otherwise	

MDFG mapping, we propose to map the three DFGs separately (respecting the modulo schedule generated in the previous step). As shown in the motivating example (Fig. 2(e)), mapping the DFGs separately helps in achieving shorter schedule lengths for the prologue and/or epilogue, leading to improved performance and energy efficiency. This is now a nested CDFG mapping problem, solved by employing the register allocation approach. However, in the case of the CDFG formed by MDT, all variables in all three DFGs are symbol variables (variables used in multiple BBs), by the inherent construct of modulo schedule. One location constraint is generated each time a symbol variable is placed for the first time. Hence, the ordering of DFGs for mapping is crucial (especially in the case of MDT) to ensure the flexibility of mapping. We propose MDT traversal (MDTT) technique, explained in Table 1, for DFG selection from the MDT. Mapping BBs with a greater number of symbol variables earlier helps to reduce the number of location constraints [3]. Hence, MDTT chooses the MDFG first, the DFG with the highest number of symbol variables in the MDT. Next, it selects the prologue, if the number of variables in the prologue is more than that of the epilogue and epilogue otherwise.

Constraint Update and Placement While mapping the MDFG, the compiler fetches the register allocation constraints generated by the previously mapped BBs and finds a placement solution that meets these constraints. The next DFG in the MDT is placed considering the constraints generated by the MDFG as well as the previous BBs. Similarly, the mapping of the next BB in the outer loop will be bound by the constraints set by the MDT as well. This technique of constraint-aware placement maintains the data integrity between the separately mapped BBs of the CDFG.

4 Experiments and Results

In this section, we evaluate the performance of the standalone execution vs the hosted execution. In the hosted execution, the innermost loop is modulo scheduled using *EpiMap* [8] and executed onto CGRA. The outer loops of the applications are run and controlled by the host CPU. The proposed standalone

Table 2. Listed kernels and their loop characteristics

Kernel	Nest depth	Max # Iterations	Loop nest structure
Matrix Multiplication	3	32X32X32 = 32 768	Imperfect
Histogram Equalization	2	80X60 = 4 800	Perfect
2D Non-Separable Filter	4	58X78X3X3 = 40 716	Imperfect
FIR Filter	2	190X10 = 1 900	Imperfect
DCT	3	8X8X8 = 512	Imperfect
Bicg	2	32X32 = 1 024	Imperfect
2D Convolution	4	58X38X3X3 = 19 836	Imperfect
Sobel	4	62X62X3X3 = 34 596	Imperfect

execution runs the entire application onto CGRA without any interruption from the CPU. To be fair with the comparison, the modulo scheduled innermost loop is mapped using *EpiMap* like approach. Here, the prologue and epilogue DFGs are mapped separately as individual DFGs instead of adapting from the MDFG mapping. We also present a performance comparison study with a state-of-the-art standalone solution by using loop transformation modeled in Cheng et al [18], which reduces the communication and memory overhead by flattening loop nests into a single-nested loop.

4.1 Experimental Setup

The proposed compilation flow for the standalone execution is implemented by using Java and Eclipse Modeling Framework (EMF). The target CGRA for all our experiments is a 4×4 PE array configuration of state-of-the-art Integrated Programmable Array (IPA) architecture [4], loosely coupled with a host CPU as shown in Fig. 1. The CPU is a RISCV [7] core based on a four pipeline stages micro-architecture optimized for energy- efficient operations in digital signal processing (DSP). A multi-banked tightly coupled data memory (TCDM) facilitates data communication between IPA and the CPU. Energy results are computed using the switching activity obtained by simulating the placement-and-routed netlist design. The CGRA design is synthesized with Cadence Genus Synthesis Solution using 90nm CMOS technology libraries. Placement and Routing are performed using Cadence Innovus and power analysis is done with Cadence Voltus at the supply of 0.9 V, in typical process conditions. A set of loop-intensive signal processing kernels including those from PolyBench [15] benchmark suite is chosen for our experiments. Table 2 features these kernels with the number of levels of loop nesting (depth of nesting), maximum number of iterations, and loop nest structures present. Nesting of loops is perfect if all the assignment statements are in the innermost loop otherwise it is imperfect nesting. Due to the additional assignments in the outer loops, the imperfect nested loops usually have more live-in and live-out variables.

4.2 Results and Discussion

Performance Comparison of Different Execution Models Table 3 presents execution latency in cycles on hosted and standalone execution models. The

Table 3. Performance and Energy comparison between hosted and standalone execution of applications with innermost loop modulo scheduled

Kernel	Execution Model	Latency (cycles)	Throughput (Mbps)	Energy (μJoule)	Speed-up	Throughput gain	Energy gain
Matrix Multiplication	Hosted	464 159	1.24	287.64	4.10x	4.10x	4.88x
	Standalone	113 310	5.07	58.97			
Histogram Equalization	Hosted	25 225	106.83	15.31	1.63x	1.63x	1.90x
	Standalone	15 484	174.03	8.06			
2D Non-Separable Filter	Hosted	2 783 615	0.97	1702.74	**12.33x**	**12.33x**	**14.49x**
	Standalone	225 768	11.94	117.49			
FIR Filter	Hosted	43 365	2.59	26.37	6.87x	6.87x	8.03x
	Standalone	6,308	17.80	3.28			
DCT	Hosted	14 450	2.49	8.79	5.14x	5.14x	6.01x
	Standalone	2 813	12.77	1.46			
Bicg	Hosted	12 452	2.89	7.56	1.93x	1.93x	2.25x
	Standalone	6 451	5.57	3.36			
2D Convolution	Hosted	1 352 406	1.00	845.47	10.70x	10.70x	12.85x
	Standalone	126 446	10.66	65.80			
Sobel	Hosted	2 534 676	0.91	1583.88	11.32x	11.32x	13.60x
	Standalone	2 23 844	10.27	116.49			

standalone execution achieves an average speed-up of 6.75× with a maximum of 12.33× over the hosted model. As discussed in the previous sections, hosted execution incurs a communication, and memory exchange overhead with the host CPU, resulting in increased latency. The overhead is directly related to the total number of control transfers and memory operations performed which in turn depends on the kernel structure like the outer loop count and the number of live-in and live-out variables. This is evident from the speed-up figures that standalone execution reports. For instance, the highest speed-ups are achieved on 2D Non-Separable Filter and 2D convolution kernels that feature the highest number of outer loop iterations and live-in/live-out variables among the kernels we considered. The average compilation times for the hosted and standalone executions are 11.73 and 18.61 seconds respectively.

Energy consumption comparison of different execution models Table 3 lists the energy results for the hosted and standalone execution models. The results demonstrate that the memory operations performed in the live-in and live-out phases of the hosted execution significantly increase energy consumption. The standalone model achieves an average of 8.00× reduction in energy consumption over hosted execution by eliminating the communication overhead with the host CPU. A maximum reduction of 14.49× is reported for the 2D non-separable filter as it performs the highest number of memory operations to transfer the live-in and live-out variables between the host and the CGRA.

Performance comparison with loop transformed standalone execution Fig. 6(a) compares the execution latencies between two standalone approaches. Both approaches use modulo scheduled innermost loop. However, Cheng et al [18]

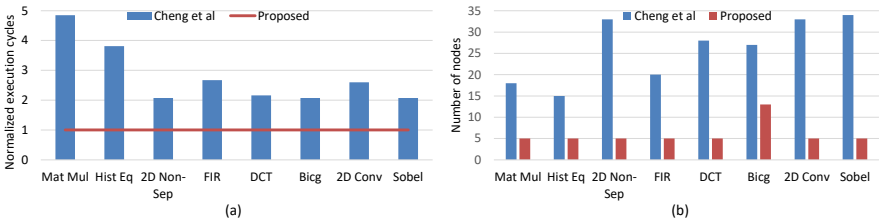

Fig. 6. Comparison between two standalone approaches (loop flattening-based vs proposed) (a) Performance comparison; (b) Comparison between the Number of nodes in the DFG to be modulo scheduled

transforms the loop nests into a single-nested loop by flattening whereas the proposed approach maps the CDFG directly using constraint aware placement. The results show that the proposed approach achieves an average of 2.80× (with a maximum of 4.80×) speed up over the loop transformation approach. The innermost loop DFG sizes of different kernels that are modulo scheduled in the two approaches are presented in Fig. 6(b). Due to inflated DFGs, transformation-based approaches deal with larger DFGs to be mapped, hence the performance deteriorates.

5 Conclusion

This paper presented a standalone execution model for the acceleration of nested loops on CGRAs. The main contribution is the compilation flow that: i) modulo-schedules the innermost loop and achieves reduced execution latencies by separately mapping prologue, MDFG, and epilogue DFGs. ii) combines modulo-scheduling with direct CDFG mapping for efficient standalone execution of nested loops. The results show that the standalone model leads to a maximum of 12.33× and an average of 6.75× performance improvement compared to the hosted model. The proposed approach reports also a maximum of 4.80× and an average of 2.80× speed-up when compared to the standalone solution.

References

1. Akbari, O., Kamal, M., Afzali-Kusha, A., Pedram, M., Shafique, M.: X-cgra: An energy-efficient approximate coarse-grained reconfigurable architecture. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **39**(10) (2019). https://doi.org/10.1109/TCAD.2019.2937738
2. Dai, L., Wang, Y., Liu, C., Li, F., Li, H., Li, X.: Reexamining cgra memory sub-system for higher memory utilization and performance. In: 2022 IEEE 40th International Conference on Computer Design (ICCD). IEEE (2022). https://doi.org/10.1109/ICCD56317.2022.00017
3. Das, S., Martin, K.J., Coussy, P., Rossi, D., Benini, L.: Efficient mapping of cdfg onto coarse-grained reconfigurable array architectures. In: 2017 22nd Asia

and South Pacific Design Automation Conference (ASP-DAC). IEEE (2017). https://doi.org/10.1109/ASPDAC.2017.7858308
4. Das, S., Martin, K.J., Rossi, D., Coussy, P., Benini, L.: An energy-efficient integrated programmable array accelerator and compilation flow for near-sensor ultralow power processing. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **38**(6) (2018). https://doi.org/10.1109/TCAD.2018.2834397
5. Dave, S., Balasubramanian, M., Shrivastava, A.: Ramp: Resource-aware mapping for cgras. In: Proceedings of the 55th Annual Design Automation Conference (2018). https://doi.org/10.1145/3195970.3196101
6. Dragomir, O.S., Stefanov, T., Bertels, K.: Loop unrolling and shifting for reconfigurable architectures. In: 2008 International Conference on Field Programmable Logic and Applications. IEEE (2008). https://doi.org/10.1109/FPL.2008.4629926
7. Gautschi, M., Schiavone, P.D., Traber, A., Loi, I., Pullini, A., Rossi, D., Flamand, E., Gürkaynak, F.K., Benini, L.: Near-threshold risc-v core with dsp extensions for scalable iot endpoint devices. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **25**(10) (2017). https://doi.org/10.1109/TVLSI.2017.2654506
8. Hamzeh, M., Shrivastava, A., Vrudhula, S.: Epimap: Using epimorphism to map applications on cgras. In: Proceedings of the 49th Annual Design Automation Conference (2012). https://doi.org/10.1145/2228360.2228600
9. Hamzeh, M., Shrivastava, A., Vrudhula, S.: Regimap: Register-aware application mapping on coarse-grained reconfigurable architectures (cgras). In: Proceedings of the 50th Annual Design Automation Conference (2013). https://doi.org/10.1145/2463209.2488756
10. Lee, J., Seo, S., Lee, H., Sim, H.U.: Flattening-based mapping of imperfect loop nests for cgras. In: Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis (2014). https://doi.org/10.1145/2656075.2656085
11. Levi, G.: A note on the derivation of maximal common subgraphs of two directed or undirected graphs. Calcolo **9**(4) (1973). https://doi.org/10.1007/BF02575586
12. Liu, D., Yin, S., Liu, L., Wei, S.: Polyhedral model based mapping optimization of loop nests for cgras. In: Proceedings of the 50th Annual Design Automation Conference (2013). https://doi.org/10.1145/2463209.2488757
13. Liu, L., Zhu, J., Li, Z., Lu, Y., Deng, Y., Han, J., Yin, S., Wei, S.: A survey of coarse-grained reconfigurable architecture and design: Taxonomy, challenges, and applications. ACM Comput. Surv. **52**(6) (Oct 2019). https://doi.org/10.1145/3357375
14. Park, H., Fan, K., Mahlke, S.A., Oh, T., Kim, H., Kim, H.s.: Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In: Proceedings of the 17th international conference on Parallel architectures and compilation techniques (2008). https://doi.org/10.1145/1454115.1454140
15. Pouchet, L.N., Grauer-Gray, S.: Polybench: The polyhedral benchmark suite, 2012 (2012), `http://www-roc.inria.fr/~pouchet/software/polybench`
16. Rau, B.R.: Iterative modulo scheduling: An algorithm for software pipelining loops. In: Proceedings of the 27th annual international symposium on Microarchitecture (1994). https://doi.org/10.1145/192724.192731
17. Rau, B.R., Schlansker, M.S., Tirumalai, P.P.: Code generation schema for modulo scheduled loops. SIGMICRO Newsl. **23**(1–2), 158–169 (dec 1992). https://doi.org/10.1145/144965.145795
18. Tan, C., Xie, C., Li, A., Barker, K.J., Tumeo, A.: Opencgra: An open-source unified framework for modeling, testing, and evaluating cgras. In: 2020 IEEE

38th International Conference on Computer Design (ICCD). IEEE (2020). https://doi.org/10.1109/ICCD50377.2020.00070
19. Torng, C., Pan, P., Ou, Y., Tan, C., Batten, C.: Ultra-elastic cgras for irregular loop specialization. In: 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA). IEEE (2021). https://doi.org/10.1109/HPCA51647.2021.00042
20. Wijerathne, D., Li, Z., Mitra, T.: Accelerating edge ai with morpher: An integrated design, compilation and simulation framework for cgras (2023). https://doi.org/10.48550/arXiv.2309.06127
21. Wijerathne, D., Li, Z., Pathania, A., Mitra, T., Thiele, L.: Himap: Fast and scalable high-quality mapping on cgra via hierarchical abstraction. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **41**(10) (2021). https://doi.org/10.1109/TCAD.2021.3132551