Dataflow Blocks: Modular Time-Multiplexing for CGRAs

Xuesi Chen, Nishanth Subramanian, Karthik Ramanathan, Nathan Beckmann, Brandon Lucia
Carnegie Mellon University
{xuesic, nsubram2, kramana2, blucia}@andrew.cmu.edu beckmann@cs.cmu.edu

***Abstract*—Recent works on energy-minimal dataflow architecture executed on coarse-grained reconfigurable array (CGRA) sacrifices area and performance for better energy efficiency, which open up opportunities for better hardware utilization and greater performance. We propose dividing dataflow graphs into Dataflow Blocks as the basic module for scheduling instructions to be time-multiplexed. Our preliminary results show a 2.5x increase in performance per area compared to a state-of-the-art CGRA, RipTide, based on simulation results. Therefore, we believe our full-stack work that spans from C code compilation down to hardware modifications has great potential to increase the performance per area for arbitrary code execution on energy-minimal CGRAs.**

I. Introduction and Related Works

Extreme edge device deployment on chip-scale satellites, wildlife monitoring and medical devices are expected to operate on ultra-low power (<1mW) for years [3, 11, 15]. Given the low energy budget and diverse applications extreme-edge processing needs to accommodate, the computer platform should be energy efficient, programmable, and as performant as possible [7].

The CGRAs, consisted of a grid of processing elements (PEs) connected by on-chip networks (NoC), which is a good execution platform for energy-minimal dataflow architecture. Prior CGRA-based energy-minimal dataflow architectures [4, 6] avoid control and data movement overheads intrinsic to von Neumann architecture, allowing operands to fire in-order upon data arrivals. Unlike application-specific integrated circuits (ASICs), the programmability of CGRAs allow computations in diverse applications (e.g. machine learning, signal processing) [5] with multiple generations of algorithmic updates [8].

Previous CGRAs [1, 9, 13, 16] use time-multiplexing to improve performance, but at the cost of programmability or energy efficiency, making them unsuitable to extreme edge computing. HyCUBE [9], REVAMP [1], ULP-SRP [10], IPA [2] execute statically scheduled instructions, allowing both spatial and temporal instruction mapping to PEs to increase performance. However, their applications are limited to inner loop kernel accelerations.

While Revel [16] and Fifer [13] did address nested loops and other irregular control, they either employs energy draining queues for inter-stage data transfers [13] or dataflow PEs with costly instruction buffers and register files [16]. Thus, they are not suitable for energy-minimal edge computing.

The state-of-art energy-minimal order-dataflow CGRA architecture, RipTide, has the ability to execute arbitrary code written in C. RipTide adopts many techniques that trade area and performance for energy efficiencies. As a result, RipTide reduces energy so much that it is unable to use all available energy in certain environments [3] (e.g. from a solar panel under direct sunlight). Hence, it makes sense to ask: **how can we improve performance with a small cost in energy?**

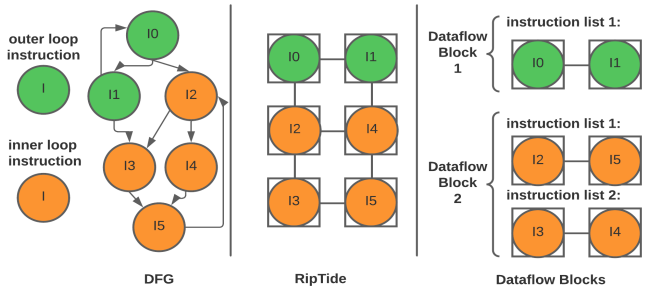

Fig. 1. Dataflow Block partition on a toy nested loop example. With the introduction of modular time-multiplexing, the number of PEs required on the CGRA shrink from 6 to 2.

One key energy saving technique RipTide uses is mapping only one operand to each PEs to avoid reconfiguration. Scheduling instructions across space but not time introduces spatial inefficiency that ultimately hurts area and performance. In nested loop examples shown in Fig 1, the instructions corresponding to the outer loop execution are left idle while waiting for the inner loop execution to finish. Thus, we believe introducing time-multiplexing to RipTide will render performant CGRA edge processor that has the programmability to execute arbitrary code written in C. Time-sharing PEs can also decrease the number of PEs needed for execution, fitting complex programs that previously need larger fabrics or allowing multi-tenancy.

We propose a compiler and architecture co-design that divides a DFG of any arbitrary program into Dataflow Blocks in the compiler. Within each Dataflow Block, we further group instructions into stages that specify what instructions will be time-multiplexed to the CGRA upon reconfiguration. During execution, the architecture rotates between stages within a block until it runs out of work to do. Static-routed NoC is chosen for the execution model for energy efficiency.

While the TRIPS processor [12, 14], a VLIW-dataflow hybrid also partitions programs to large hyperblocks for execution, hyperblocks statically schedules instructions. Dataflow Blocks, on the other hand, simply rotates instructions within a block, allowing data and control dependencies to guide the execution. On top of that, software pipelining in Dataflow

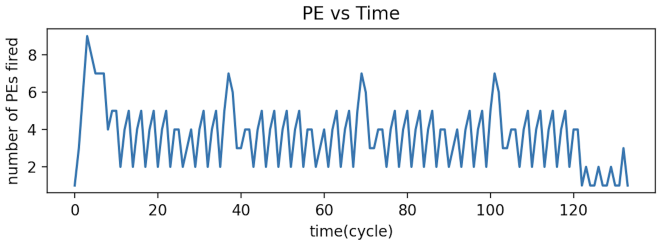

Fig. 2. Limit study shows dmv execution with stages of periodicity.

Blocks is much simpler given that operation scheduling is handled in a distributed fashion due to dataflow firing rule.

II. Preliminary Studies and Simulation Results

Our preliminary study is done on a Python simulator, dataflow-sim, which is a high-level abstraction of RipTide. It removes the complexity of implementing routing and NoC and estimates PE utilization and execution time.

To start with, we run a limit study to answer two questions: 1. What is the number of PEs needed to run workloads if routing and PE placements are not considered? 2. Given enough PE resources, what does the ideal execution pattern look like?

The result of the limits study shows two insights: 1. Program execution time has diminishing returns as the number of available PEs increase. 2. At different stages of execution, predominately marked by control branching (e.g. loop boundaries), the instructions firing trend displays a repeated pattern with certain initiation interval (II), as shown in Fig 2.

Therefore, we propose a compiler and architecture co-design that divides an DFG of any arbitrary program into ”stages” using a compiler. We group the instruction in the ”stages” as Dataflow Blocks. Instruction scheduling is performed inside of the Dataflow Blocks to emulate the repeated instruction firing pattern suggested by limit study.

We manually divide DFG into Dataflow Blocks and programmed a simple instruction scheduler. The runtime scheduler in the simulator starts by mapping a Dataflow Block onto the fabric and executes by rotating stages in a Round-Robin fashion until the whole Dataflow Block runs out of work to do. There are two options for deciding which Dataflow Block should execute next. Current simulation results show that for simple nested for loops, the Round-Robin execution order is sufficient. However, as we run into more complicated workloads with complex control, it might make sense to choose the Dataflow Blocks with most work to do. The amount of work a Dataflow Block needs to do is based on the number of instructions in the Dataflow Block that are ready to be fired.

Simulation results show that a reasonable Dataflow Blocks partition and instruction scheduling gives us close to ideal performance per area, exceeding the baseline RipTide implementation by 2.5x. Fig 3 shows the performance comparison for dmv with different scheduling and hardware support generated by the simulator. The ideal performance will require PE-level time-multiplexing, which is too energy-consuming for edge computing. However, Dataflow Blocks allow the performance for static-routed coarse-grain time-multiplexing to be almost as good.

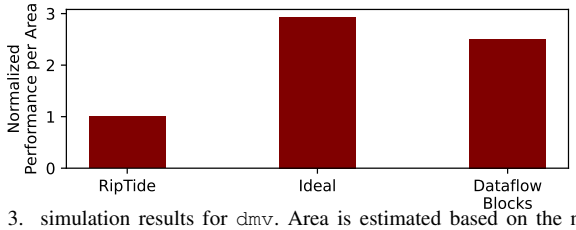

Fig. 3. simulation results for dmv. Area is estimated based on the number of PEs needed for execution.

III. Future Design and Implementations

We plan on continuing the exploration of Dataflow Block partition and scheduling algorithms using the simulator. Once we narrow down the selection of compiler and architecture design, RTL synthesis will be performed. We leverage RipTide as our baseline, which has a compiler that generates dataflow graphs and a 6x6 CGRA fabric synthesized in Intel 22FFL for execution. In the below subsections, we discuss detailed compiler and architecture changes we will be working on in the near future.

A. Compiler Design

At the end of DFG generation, the compiler is expected to estimate the maximum number of PEs needed to execute a workload by identifying the critical path and maximum dependency width.

Meanwhile, the Dataflow Blocks should be indicated. Back edges, control operators on the graph are keys elements for us to identify Dataflow Blocks. This is because they signal the boundaries of loops, which can be software pipelined. We are also considering profile-guided compilation, especially for aiding us through identifying Dataflow Blocks on workloads with complicated controls.

Within a dataflow block, list scheduling and software pipelining techniques are used to assign instructions to PEs based on the maximum number of PEs needed. This will result in stages, which are groups of instructions mapped to PEs. The length of each stage does not exceed the number of PEs allocated but should be maximized.

B. Hardware Modification

Homogenizing PE functionalities is required for time-multiplexing support. Banks should also be added to PEs for storing the instructions lists in the current Dataflow Block. The NoC needs to have store control flow filters and routing connections in its banks. We will add selectors to point to the right stage when reconfiguration to the whole fabric happens.

C. Evaluation

We will use Cadence Xcelium for correctness verification and performance measurement. Cadence Joules will be used for power estimation.

We plan on comparing our performance and energy data with the baseline RipTide, especially with workloads that have irregular loops, nested loops and control heavy workloads (e.g. dmv, dmm, fft, bfs, SpMV, etc.). On top of that, a quantitative comparison with HyCUBE, REVAMP, Fifer,

Revel in terms of MOPS/mW and applications supported is intended to be the evaluation scope.

REFERENCES

[1] T. K. Bandara, D. Wijerathne, T. Mitra, and L.-S. Peh, "Revamp: A systematic framework for heterogeneous cgra realization," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 918–932.

[2] S. Das, D. Rossi, K. J. Martin, P. Coussy, and L. Benini, "A 142mops/mw integrated programmable array accelerator for smart visual processing," in *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2017, pp. 1–4.

[3] B. Denby and B. Lucia, "Orbital edge computing: Nanosatellite constellations as a new class of computer system," in *ASPLOS 25*, 2020.

[4] G. Gobieski, A. O. Atli, K. Mai, B. Lucia, and N. Beckmann, "Snafu: an ultra-low-power, energy-minimal cgra-generation framework and architecture," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 1027–1040.

[5] G. Gobieski, N. Beckmann, and B. Lucia, "Intermittent deep neural network inference," in *SysML Conference*, 2018, pp. 1–3.

[6] G. Gobieski, S. Ghosh, M. Heule, T. Mowry, T. Nowatzki, N. Beckmann, and B. Lucia, "A programmable, energy-minimal dataflow compiler and architecture," pp. 546–564, 2022.

[7] G. Gobieski, B. Lucia, and N. Beckmann, "Intelligence beyond the edge: Inference on intermittent embedded systems," in *ASPLOS*, 2019.

[8] N. P. Jouppi, D. H. Yoon, M. Ashcraft, M. Gottscho, T. B. Jablin, G. Kurian, J. Laudon, S. Li, P. Ma, X. Ma *et al.*, "Ten lessons from three generations shaped google's tpuv4i: Industrial product," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 1–14.

[9] 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 Annual Design Automation Conference 2017*, 2017, pp. 1–6.

[10] C. Kim, M. Chung, Y. Cho, M. Konijnenburg, S. Ryu, and J. Kim, "Ulp-srp: Ultra low-power samsung reconfigurable processor for biomedical applications," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 7, no. 3, pp. 1–15, 2014.

[11] B. Lucia, V. Balaji, A. Colin, K. Maeng, and E. Ruppel, "Intermittent computing: Challenges and opportunities," *2nd Summit on Advances in Programming Languages (SNAPL 2017)*, 2017.

[12] R. Nagarajan, K. Sankaralingam, D. Burger, and S. W. Keckler, "A design space evaluation of grid processor architectures," in *Proceedings. 34th ACM/IEEE International Symposium on Microarchitecture. MICRO-34*. IEEE, 2001, pp. 40–51.

[13] Q. M. Nguyen and D. Sanchez, "Fifer: Practical acceleration of irregular applications on reconfigurable architectures," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 1064–1077.

[14] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore, "Exploiting ilp, tlp, and dlp with the polymorphous trips architecture," in *Proceedings of the 30th annual international symposium on Computer architecture*, 2003, pp. 422–433.

[15] F. Tavares, "Kicksat 2," May 2019. [Online]. Available: https://www.nasa.gov/ames/kicksat

[16] J. Weng, S. Liu, Z. Wang, V. Dadu, and T. Nowatzki, "A hybrid systolic-dataflow architecture for inductive matrix algorithms," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 703–716.