Cascade: An Application Pipelining Toolkit for Coarse-Grained Reconfigurable Arrays

Jackson Melchert
Stanford University
USA

Yuchen Mei
Stanford University
USA

Kalhan Koul
Stanford University
USA

Qiaoyi Liu
Stanford University
USA

Mark Horowitz
Stanford University
USA

Priyanka Raina
Stanford University
USA

ABSTRACT

While coarse-grained reconfigurable arrays (CGRAs) have emerged as promising programmable accelerator architectures, they require automatic pipelining of applications during their compilation flow to achieve high performance. Current CGRA compilers either lack pipelining altogether resulting in low application performance, or perform exhaustive pipelining resulting in high power and resource consumption. We address these challenges by proposing Cascade, an end-to-end open-source application compiler for CGRAs that achieves both state-of-the-art performance and fast compilation times. The key contributions of this work are: (1) a novel post place-and-route (PnR) application pipelining technique for CGRAs, (2) a register resource usage optimization technique, and (3) an automated CGRA timing model generator. We integrate these into an end-to-end compilation flow that achieves 8 - 34× lower critical path delays and 7 - 190× lower energy-delay product (EDP) across a variety of dense image processing and machine learning workloads compared to a compiler without pipelining.

1 INTRODUCTION

In order to achieve commercial utility, CGRAs must demonstrate performance and energy-delay product (EDP) that are competitive with application-specific integrated circuits (ASICs). To do so, CGRAs need to execute applications at high clock frequencies, requiring carefully pipelined applications. The problem is that existing CGRA compilers fail at this task. They attempt to tackle mapping, scheduling, placement and routing (PnR), and pipelining of an application all within one optimization step [3–5, 8–10]. This coupling between the various pieces in the compilation flow makes the search space very large, making the compilers slow, produce poor results, and not scale well to large CGRAs.

CGRAs are typically composed of several processing element (PE) tiles and memory (MEM) tiles arranged in a grid, as shown in Fig. 1. The basic problem with pipelining CGRA applications is that the programmable wiring connecting the tiles together has large relative delay that the pipelining must take into account, but since data waves need to be balanced, adding pipeline registers along one path can require adding hardware to other paths. This additional hardware often changes the placement and routing of the CGRA, which causes this process to restart.

To address this issue, we took inspiration from FPGA compilers and decoupled mapping, scheduling, placement, routing, and pipelining into largely independent steps. We build upon the work presented in [6], which takes this approach but only does wire-independent pipelining, to create a compiler called Cascade. Cascade only needs minimal hardware support in the CGRA: configurable pipeline registers on the interconnect, an interconnect with single-cycle multi-hop connections, and the ability to adjust the schedules of the memory tiles at a cycle level. Using this hardware, we add post-PnR pipelining, register absorption, and incremental rescheduling to a staged FPGA-style compiler, resulting in state-of-the-art application performance and compilation times.

The contributions of this paper are:

(1) We adapt FPGA and ASIC-like pipelining and register retiming techniques [2, 7] to register-scarce CGRAs, and propose a technique for absorbing registers into memory tiles without affecting the mapping, placement, and routing.
(2) We propose a post-PnR pipelining technique that iteratively identifies the critical path in an application, breaks it by turning on interconnect registers, and performs rescheduling. Post-PnR pipelining accurately accounts for interconnect delays while avoiding cyclic rescheduling and PnR.
(3) An end-to-end open-source CGRA compiler [1], called Cascade, which has (a) an automatic CGRA timing model generator, (b) a static timing analysis (STA) tool that uses the timing model to determine the critical path of an application on a CGRA, and (c) a large set of existing and our proposed pipelining techniques integrated into an end-to-end flow.

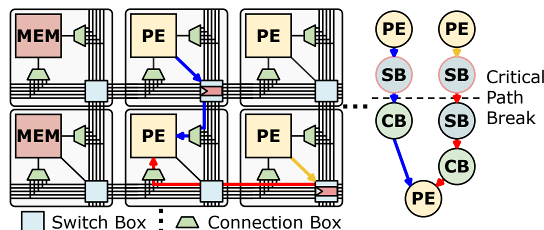

Figure 1: Post-PnR pipelining takes the place and route result represented as a dataflow graph and performs STA to identify the critical path. This path is broken by enabling registers in the switch box (SB), and the graph is branch delay matched.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
LATTE '24, April 28, 2024, San Diego, CA, United States
© 2024 Copyright held by the owner/author(s).

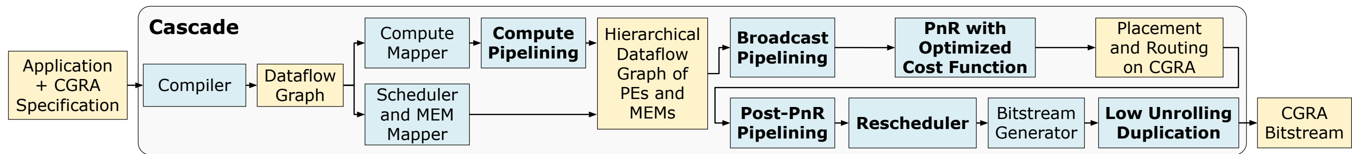

Figure 2: CGRA application compiler that takes an application and a CGRA specification and produces a bitstream. Compiler stages are blue. Intermediate representations are yellow. Bolded text indicates compiler stages added or modified in this work.

2 POST-PLACE-AND-ROUTE PIPELINING

Post-PnR pipelining, shown in Fig. 1, iteratively identifies the critical path and inserts pipelining registers to break it. After PnR is complete, we know exactly where each tile will be placed on the array and where the nets will be routed. Using the timing model and application STA tool we designed for CGRAs, we determine the critical path delay of the application. The CGRA timing model contains delays for both the PE operations and memories, as well as the delays of interconnect hops. Additionally, we can use STA with back-tracing to determine what the critical path is.

Adding registers on existing routes affects the execution of the application, so we must do branch delay matching in order to maintain application functionality. Branch delay matching matches the cycle arrival times of every piece of data arriving at every functional element in the application by adding registers on branches with shorter delay. When adding pipeline registers to a statically scheduled CGRA application, the schedule needs to be updated to reflect any changes to the compute latencies. After an application finishes post-PnR pipelining, we know all of the compute latencies. We send these latencies to the static scheduler to incrementally update the configuration of each memory tile used in the application.

3 OPTIMIZING REGISTER RESOURCE USAGE

The registers added by pipelining and branch delay matching need to be placed on the configurable interconnect. This added resource requirement increases execution energy, and for large applications, may cause placement or routability issues. Therefore, we introduce a technique for absorbing registers into memory tiles and register files that dramatically reduces the register resource usage while maintaining the benefits of pipelining. Note that the registers we want to remove from the configurable interconnect are *only those that are **not** on the critical path.* These are typically introduced on non-critical paths when we perform branch delay matching after pipelining the critical path. A register can be "absorbed" into a memory tile or register file in a PE tile, removing registers from the application. Removing a single register connected to the output of a memory tile can be achieved by scheduling the memory to start outputting data one cycle later.

4 CASCADE

Finally, we design an end-to-end open-source CGRA compiler called Cascade shown in Fig. 2, which like [6], has decoupled compiler stages, but achieves higher performance. Cascade has (a) an automatic CGRA timing model generator, (b) a static timing analysis tool that uses the timing model to determine the critical path of an application on a CGRA, and (c) a large set of existing and our proposed pipelining techniques integrated into an end-to-end flow that works both for dense and sparse applications.

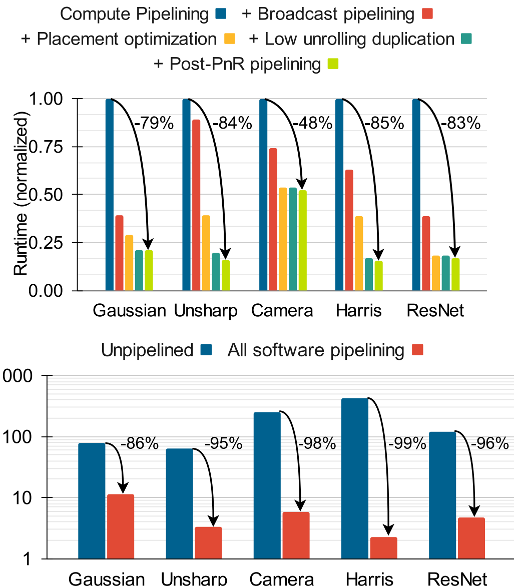

Figure 3: Incremental effect of each software pipelining technique on the runtime and EDP of dense applications.

5 RESULTS

The CGRA architecture that we use is a 32×16 array with 384 PE tiles and 128 MEM tiles. We perform physical design of the CGRA in GlobalFoundries 12 nm technology. We analyze the incremental impact of software pipelining techniques from Section 2 and Section 4 on the runtime of five dense applications from image processing and machine learning domains. This is shown in Fig. 3. The software pipelining techniques achieve an 84 - 97% decrease in runtime versus unpipelined implementations. Compute pipelining alone results in a 35 - 81% reduction in runtime compared to the un-pipelined applications, while the techniques applied during and after PnR result in an additional 48 - 85% reduction in runtime. The pipelining techniques result in an EDP decrease of 86 - 99%.

6 CONCLUSION

Cascade is an open-source end-to-end CGRA compiler that achieves high performance and fast compilation through the use of a decoupled FPGA-style compilation flow and new pipelining techniques. Cascade achieves 8 - 34× lower critical path delays and 7 - 190× lower EDP across a variety of dense image processing and machine learning workloads compared to a compiler without pipelining. While Cascade is a standalone compiler, the pipelining techniques can be integrated into other CGRA compilers, enabling the creation of high performance CGRA compilation infrastructure and encouraging research into CGRAs as promising acceleration platforms.

7 ACKNOWLEDGEMENT

This work was supported by funding from SRC/DARPA JUMP 2.0 PRISM Center, NSF CAREER (award number: 2238006), DARPA DSSoC, Stanford Agile Hardware (AHA) Center, Stanford SystemX Alliance and Apple Stanford EE PhD Fellowship in Integrated Systems.

REFERENCES

[1] [n. d.]. Cascade Compiler. https://github.com/StanfordAHA/aha. Accessed: 2024-01-30.

[2] Pierre-yves Calland, Anne Mignotte, Olivier Peyran, Yves Robert, and Frederic Vivien. 1998. Retiming DAGs [direct acyclic graph]. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 17, 12 (1998), 1319–1325. https://doi.org/10.1109/43.736571

[3] Mahdi Hamzeh, Aviral Shrivastava, and Sarma Vrudhula. 2012. EPIMap: Using Epimorphism to Map Applications on CGRAs. In *Proceedings of the 49th Annual Design Automation Conference* (San Francisco, California) *(DAC '12)*. Association for Computing Machinery, New York, NY, USA, 1284–1291. https://doi.org/10.1145/2228360.2228600

[4] Manupa Karunaratne, Aditi Kulkarni Mohite, Tulika Mitra, and Li-Shiuan Peh. 2017. HyCUBE: A CGRA with Reconfigurable Single-Cycle Multi-Hop Interconnect. In *Proceedings of the 54th Annual Design Automation Conference 2017* (Austin, TX, USA) *(DAC '17)*. Association for Computing Machinery, New York, NY, USA, Article 45, 6 pages. https://doi.org/10.1145/3061639.3062262

[5] Xiangyu Kong, Yi Huang, Jianfeng Zhu, Xingchen Man, Yang Liu, Chunyang Feng, Pengfei Gou, Minggui Tang, Shaojun Wei, and Leibo Liu. 2023. MapZero: Mapping for Coarse-Grained Reconfigurable Architectures with Reinforcement Learning and Monte-Carlo Tree Search. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 46, 14 pages. https://doi.org/10.1145/3579371.3589081

[6] Kalhan Koul, Jackson Melchert, Kavya Sreedhar, Leonard Truong, Gedeon Nyengele, Keyi Zhang, Qiaoyi Liu, Jeff Setter, Po-Han Chen, Yuchen Mei, Maxwell Strange, Ross Daly, Caleb Donovick, Alex Carsello, Taeyoung Kong, Kathleen Feng, Dillon Huff, Ankita Nayak, Rajsekhar Setaluri, James Thomas, Nikhil Bhagdikar, David Durst, Zachary Myers, Nestan Tsiskaridze, Stephen Richardson, Rick Bahr, Kayvon Fatahalian, Pat Hanrahan, Clark Barrett, Mark Horowitz, Christopher Torng, Fredrik Kjolstad, and Priyanka Raina. 2023. AHA: An Agile Approach to the Design of Coarse-Grained Reconfigurable Accelerators and Compilers. *ACM Trans. Embed. Comput. Syst.* 22, 2, Article 35 (Jan 2023), 34 pages. https://doi.org/10.1145/3534933

[7] Charles E. Leiserson, Flavio M. Rose, and James B. Saxe. 1983. Optimizing Synchronous Circuitry by Retiming (Preliminary Version). In *Third Caltech Conference on Very Large Scale Integration*, Randal Bryant (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 87–116. https://doi.org/10.1007/978-3-642-95432-0_7

[8] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. 2002. DRESC: A Retargetable Compiler for Coarse-Grained Reconfigurable Architectures. In *2002 IEEE International Conference on Field-Programmable Technology, 2002. (FPT). Proceedings.* 166–173. https://doi.org/10.1109/FPT.2002.1188678

[9] Tony Nowatzki, Michael Sartin-Tarm, Lorenzo De Carli, Karthikeyan Sankaralingam, Cristian Estan, and Behnam Robatmili. 2013. A General Constraint-Centric Scheduling Framework for Spatial Architectures. *SIGPLAN Not.* 48, 6 (Jun 2013), 495–506. https://doi.org/10.1145/2499370.2462163

[10] Zhongyuan Zhao, Weiguang Sheng, Qin Wang, Wenzhi Yin, Pengfei Ye, Jinchao Li, and Zhigang Mao. 2020. Towards Higher Performance and Robust Compilation for CGRA Modulo Scheduling. *IEEE Transactions on Parallel and Distributed Systems* 31, 9 (2020), 2201–2219. https://doi.org/10.1109/TPDS.2020.2989149