Designing a Dataflow Hardware Accelerator with an Abstract Machine

Olivia Hsu
Stanford University, USA

Maxwell Strange
Stanford University, USA

Kunle Olukotun
Stanford University, USA

Mark Horowitz
Stanford University, USA

Fredrik Kjølstad
Stanford University, USA

Abstract

This paper motivates the use of domain-specific abstract machines for designing hardware accelerators. Specifically, we describe how we built a reconfigurable dataflow accelerator for sparse tensor algebra, a relatively complex domain, using the Sparse Abstract Machine (SAM). We show that leveraging an abstract dataflow representation (and its compiler) lead to a slew of benefits, including computational generality within the application domain, easier verification and debugging, boosted design productivity, and decoupled development of the toolchain from the hardware design and generation. These benefits allowed us to build the accelerator within a short time-frame with only two designers.

1 Introduction

We share our experiences of designing a reconfigurable dataflow accelerator (RDA) for sparse tensor algebra using a domain-specific language, compiler, and automatic tooling system with the hopes that it will encourage similar flows in the future. We build the design around a central abstraction—the Sparse Abstract Machine (or SAM) [9]—that serves as an idealized streaming dataflow representation for sparse tensor algebra algorithms. SAM is both an abstract machine model for sparse tensor algebra and an intermediate representation for its front-end compiler, Custard, which takes as input tensor index notation (or Einsum notation). By focusing accelerator design on the primitives in SAM, and their organization, it becomes easy to iterate on a design while maintaining end-to-end mapping support. Therefore, SAM is analogous to the purpose of an ISA (or LLVM) for von Neumann machines. We also found multiple benefits in design productivity and verification with this approach.

SAM identifies blocks that compose to describe the entire space of sparse tensor algebra expressions and dataflow orderings, allowing us to design an accelerator with these properties. Reconfigurable accelerators with this generality is unlike most sparse accelerator prior work, which only support fixed-function expressions [2, 5–7, 11–13, 16–18, 20].

2 System Overview

We successfully built, verified, and taped-out the reconfigurable sparse accelerator (more precisely a CGRA) over the course of about four months with only two designers. The designers implemented all of the on-chip RDA tiles (local memory tiles and sparse computation primitives) and verified the correctness of the entire suite of sparse applications. We tested our accelerator on eleven sparse tensor expressions [9]—which cover part of the the *long tail* of tensor expressions that are not addressed by fixed-function accelerators—across a regression of both randomly generated tensors and real-world SuiteSparse matrices [4]. An overview diagram of our system is shown in Figure 1. The rest of this section describes the general hardware flow and then the SAM simulator.

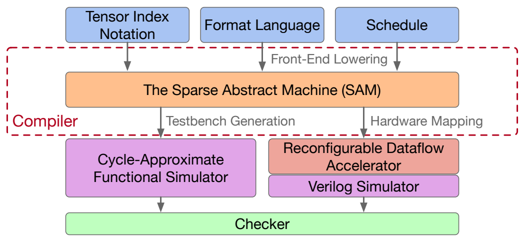

Figure 1. Overview of our sparse RDA toolchain [3, 9, 15].

2.1 Compilation to Hardware Description

The core of our accelerator is a hand-built set of ten modules that match the SAM primitive interfaces [9] and are written in Kratos [19]—a Python-embedded Verilog generation language. As we target a CGRA, processing and memory tiles are distributed throughout a reconfigurable interconnect, with each processing element capable of implementing a SAM primitive. The flexibility of this substrate allows tiles to be connected and reconfigured at run-time via a configuration bus without changing the Verilog collateral. First, the SAM compiler Custard compiles a set of expressions from index notation into SAM graphs. These SAM graphs are then automatically lowered to a hardware-aware SAM-like graph. These hardware-aware SAM-like graphs include transformation passes such as n-ary primitives to binary-primitive trees, individual primitives that are mapped to multiple hardware-modules, and bitwidth tagging for each connection. Hardware implementations could fuse many SAM primitives together (e.g. [14]). These designs would require custom primitive-merging passes when lowering to the hardware-aware SAM-like graph but is left as future work. We then leveraged a suite of open-source tools [10] to map the application onto the CGRA substrate. The output of this mapping contains a valid placement of all requested primitives/resources and a feasible routing for the interconnect. We note that individual hardware-aware SAM-like graphs could be hardened into a single-expression ASIC block rather than being targeted towards a reconfigurable substrate.

2.2 SAM Simulator Description

Our accelerator toolchain includes a cycle-approximate functional Python simulator along with agile software testing that models and verifies the SAM abstraction. In the simulator, each SAM primitive is represented as a class with methods for input connections, output connections, and an `update` state-machine. We hand implemented the behavior of all primitives such that blocks communicate with each other via streams as prescribed by SAM. Each primitive models a basic hardware implementation, meaning that useful cycle counts are extracted from their execution. Simulator blocks are also verified individually against a battery of handwritten unit tests. The SAM compiler also automatically generates testbenches for full SAM graph tests—SAM graphs composed of individual primitives that compute an entire sparse tensor algebra algorithm. Full testbenches contain: object initialization for each primitive in the SAM graph, a cycle (iteration) loop, IO connections for every graph edge at the start of each cycle, and block updates at the end of each cycle. Running full tests allows us to verify the correctness of the abstraction and its composition. Once we know the SAM behavior is correct, the simulator can also verify hardware implementations (as a gold checker) and guide architectural design (as a design-space exploration/productivity tool).

3 Lessons Learned

Our experience building this accelerator demonstrated a slew of benefits in terms of design productivity and verification, stemming from implementing an abstract dataflow machine. Although these benefits are well-known for von Neumann machines, we highlight them for RDA design as well.

Automation. In our tooling methodology, we automatically generate or compile as much of the code and hardware collateral as possible. To add a new primitive (operation), we only need to implement its behavior in the simulator and Kratos and then register the primitive with the hardware-aware lowering step. The rest of the flow is automatic. Specifically, the automatic portions were: the front-end compiler Custard, the generation of simulator testbenches from SAM graphs, the compiler transformations from SAM graphs to hardware-aware graphs, mapping placement and routing on to the RDA, and CGRA configuration bitstream generation. Since SAM has only a few primitives, accelerators supporting these primitives (either directly or indirectly) make for easy compiler targets. Additionally, as new applications were pushed through the compilation toolchain, they were immediately ready for execution on the hardware. Designers are now free to consider additional applications without worrying about mapping these applications by hand.

Generality. Similar to automatic compilation, our hardware design also inherits the benefit of generality from the abstract machine. The hardware design reuses the same set of ten blocks across different application mappings. Since these blocks form the basis of sparse tensor algebra [9], the hardware design is general and can compute any expression across many schedules. This generality is important as many scheduled algorithms can perform asymptotically bad under certain data [1, 8, 9], making point-solution accelerators that were previously proposed unsatisfactory in general. In order to find an optimal schedule, we must search through high-level schedules using simulator performance (either cycle-approximate or post-synthesis). However, the purposeful design of our flow—specifically the hardware generality, separation of algorithm and schedule in the input API, and cycle-approximate simulator—will allow others to leverage autoscheduling techniques. We differentiate generality from automation since it is possible to have one without the other. For example, a system with a compiler toolchain that can only handle a limited set of fixed expressions without general block reuse would be automatic but not general.

Ease of Debugging. As both the hardware design and simulator implement the interfaces defined in SAM, we can debug the hardware at the interface of each module. Using the functional simulator written in software, we were able to pinpoint bugs to specific SAM blocks—and therefore their corresponding hardware module—which minimizes the tedious task of tracing and analyzing Verilog simulation waveforms. Since the SAM abstraction defines the interfaces for each block, similar to an ISA, we are able to compare all intermediate streams with the hardware behavior. Intermediate result checking then allows us to quickly debug.

Decoupled Development. SAM stands in between its compiler and any hardware design/implementation. This well-defined interface enables designers on each side to work in a decoupled manner. We were able to build the compiler and accelerator independently of each other, and they integrated smoothly at the SAM abstraction once each side was functional. Our flow can better leverage software engineers with minimal hardware knowledge to develop the tooling infrastructure—consisting of the compiler, simulator, and additional code collateral.

4 Conclusion

We hope that this experience demonstrates how the Sparse Abstract Machine can enable the rapid development of future accelerators. We also hope that compiler designs and automatic tooling like ours, targeting an abstract machine for portability, will improve the programmability and usability of dataflow accelerators.

Acknowledgments

Olivia Hsu was supported by an NSF GRFP Fellowship and Maxwell Strange was supported by the Apple Stanford Electrical Engineering PhD Fellowship in Integrated Systems. This work was supported in part by DARPA under the Domain-Specific System on Chip (DSSoC) Program. This research was also supported in part by the Stanford Agile Hardware (AHA) Center, Google Research Scholar program, and Stanford Data Analytics for What's Next (DAWN) Affiliate Program[1].

References

[1] Peter Ahrens, Fredrik Kjolstad, and Saman Amarasinghe. 2022. Autoscheduling for Sparse Tensor Algebra with an Asymptotic Cost Model. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 269–285. https://doi.org/10.1145/3519939.3523442

[2] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. 2019. Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9, 2 (2019), 292–308. https://doi.org/10.1109/JETCAS.2019.2910232

[3] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2018. Format Abstraction for Sparse Tensor Algebra Compilers. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 123 (October 2018), 30 pages.

[4] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1–25.

[5] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. 2016. EIE: Efficient inference engine on compressed deep neural network. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 243–254.

[6] Xin He, Subhankar Pal, Aporva Amarnath, Siying Feng, Dong-Hyeon Park, Austin Rovinski, Haojie Ye, Yuhan Chen, Ronald Dreslinski, and Trevor Mudge. 2020. *Sparse-TPU: Adapting Systolic Arrays for Sparse Matrices.* Association for Computing Machinery, New York, NY, USA, Chapter 19, 1–12. https://doi.org/10.1145/3392717.3392751

[7] Kartik Hegde, Jiyong Yu, Rohit Agrawal, Mengjia Yan, Michael Pellauer, and Christopher W. Fletcher. 2018. UCNN: Exploiting Computational Reuse in Deep Neural Networks via Weight Repetition. In *Proceedings of the 45th Annual International Symposium on Computer Architecture* (Los Angeles, California) *(ISCA '18)*. IEEE Press, 674–687. https://doi.org/10.1109/ISCA.2018.00062

[8] Rawn Henry, Olivia Hsu, Rohan Yadav, Stephen Chou, Kunle Olukotun, Saman Amarasinghe, and Fredrik Kjolstad. 2021. Compilation of Sparse Array Programming Models. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 128 (October 2021), 29 pages. https://doi.org/10.1145/3485505

[9] Olivia Hsu, Maxwell Strange, Jaeyeon Won, Ritvik Sharma, Kunle Olukotun, Joel Emer, Mark Horowitz, and Fredrik Kjolstad. 2022. The Sparse Abstract Machine. https://doi.org/10.48550/ARXIV.2208.14610

[10] 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

[11] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. 2018. OuterSPACE: An outer product based sparse matrix multiplication accelerator. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 724–736.

[12] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. 2017. SCNN: An accelerator for compressed-sparse convolutional neural networks. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 27–40. https://doi.org/10.1145/3079856.3080254

[13] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. 2020. SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 58–70. https://doi.org/10.1109/HPCA47549.2020.00015

[14] Alexander Rucker, Matthew Vilim, Tian Zhao, Yaqi Zhang, Raghu Prabhakar, and Kunle Olukotun. 2021. Capstan: A Vector RDA for Sparsity. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 1022–1035. https://doi.org/10.1145/3466752.3480047

[15] Ryan Senanayake, Changwan Hong, Ziheng Wang, Amalee Wilson, Stephen Chou, Shoaib Kamil, Saman Amarasinghe, and Fredrik Kjolstad. 2020. A Sparse Iteration Space Transformation Framework for Sparse Tensor Algebra. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 158 (Nov. 2020), 30 pages. https://doi.org/10.1145/3428226

[16] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. 2020. MatRaptor: A sparse-sparse matrix multiplication accelerator based on row-wise product. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 766–780.

[17] Xinfeng Xie, Zheng Liang, Peng Gu, Abanti Basak, Lei Deng, Ling Liang, Xing Hu, and Yuan Xie. 2021. SpaceA: Sparse Matrix Vector Multiplication on Processing-in-Memory Accelerator. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 570–583. https://doi.org/10.1109/HPCA51647.2021.00055

[18] Guowei Zhang, Nithya Attaluri, Joel S Emer, and Daniel Sanchez. 2021. GAMMA: leveraging Gustavson's algorithm to accelerate sparse matrix multiplication. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 687–701.

[19] Keyi Zhang. 2022. Kratos: Debuggable Hardware Generator. https://github.com/Kuree/kratos

[20] Zhekai Zhang, Hanrui Wang, Song Han, and William J Dally. 2020. SpArch: Efficient architecture for sparse matrix multiplication. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 261–274.

[1]Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the aforementioned funding agencies