THEME ARTICLE: HOT CHIPS 2024

Designing Programmable Accelerators for Sparse Tensor Algebra

Kalhan Koul*, Zhouhua Xie*, Maxwell Strange, Sai Gautham Ravipati, Bo Wun Cheng, Olivia Hsu, Po-Han Chen, Mark Horowitz, Fredrik Kjolstad, Priyanka Raina, *Stanford University, Stanford, CA, 94305, USA*
**Equal Contribution*

Recent research has focused on leveraging sparsity in hardware accelerators to improve the efficiency of applications spanning scientific computing to machine learning. Most such prior accelerators are fixed-function, which is insufficient for two reasons. First, applications typically include both dense and sparse components, and second, the algorithms that comprise these applications are constantly evolving. To address these challenges, we designed a programmable accelerator called Onyx for both sparse tensor algebra and dense workloads. Onyx extends a coarse-grained reconfigurable array (CGRA) optimized for dense applications with composable hardware primitives to support arbitrary sparse tensor algebra kernels. In this paper, we show that we can further optimize Onyx by adding a small set of hardware features for parallelization that significantly increase both temporal and spatial utilization of the CGRA, reducing runtime by up to 6.2×.

In the last decade, there has been a rapid growth in the size of machine learning models. As a result, researchers have turned to sparsity to compress models and improve efficiency. Previous work has developed dedicated fixed-function hardware accelerators to exploit structured sparsity in end-to-end applications[1–3]. For example, Huang et al.[3] developed integrated dedicated accelerators for sparse convolutional neural networks and sparse graph convolutional networks on the same chip. There are also several works that exploit unstructured sparsity in specific kernels[4], where the kernel is typically a matrix multiplication. The key issue with these approaches is that end-to-end accelerators become obsolete after new algorithms are designed, and fixed-function blocks that only accelerate portions of end-to-end applications leave performance on the table.

Programmable accelerators offer a promising solution for applications with evolving algorithms. These accelerators achieve significant energy efficiency benefits compared to general-purpose computing[5] but have typically focused only on dense applications. One prior programmable accelerator, Capstan[6], accelerates sparse kernels, but does not fully leverage sparse iteration for unions and intersections. Modern end-to-end applications have both dense and sparse components[7] and can benefit significantly from the acceleration of both. Our work, Onyx[8,9], is the first programmable fabric that supports both dense and sparse acceleration. In dense mode, Onyx's coarse-grained reconfigurable array (CGRA) is optimized for high parallelism, performing hundreds of multiply-adds (or another operation) per cycle. In sparse mode, we instead configure our accelerator to eliminate unnecessary memory accesses and ineffectual computation. Onyx is the first programmable dataflow accelerator that provides a functional design with domain completeness. In this paper, we propose techniques to improve the spatial and temporal utilization of Onyx for sparse applications. Our contributions include: (1) hardware primitives for pipelining accelerator calls, thereby increasing temporal utilization; and (2) hardware primitives for broadcasting, filtering, and merging, enabling parallelism and high spatial utilization. Finally, we perform a detailed sweep across input matrix sparsities and (3) determine the optimal configuration for matrix multiplication performance.

THE ONYX CHIP

Accelerating arbitrary sparse tensor algebra expressions requires a general abstraction that can be

Digital Object Identifier 10.1109/XXX.0000.0000000

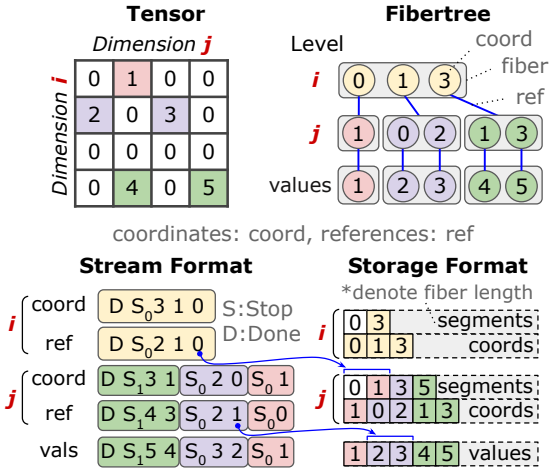


FIGURE 1. A 2D sparse tensor and its equivalent representations in the fibertree format, stream format, and storage format.

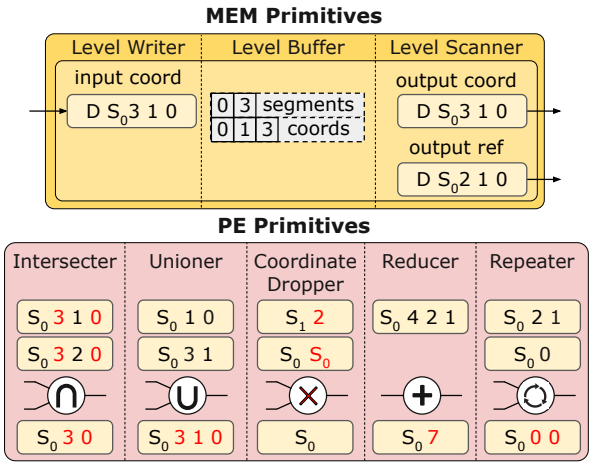


FIGURE 2. Memory tile sparse primitives showing conversion of streams to storage and storage to streams. PE tile sparse primitives showing intersecter, unioner, coordinate dropper, reducer, and repeater.

mapped onto a programmable accelerator. We utilize the fibertree abstraction[10] from the sparse abstract machine (SAM)[11] work. Figure 1 shows how a tensor can be represented as a fibertree. Each fibertree level is a dimension of the tensor. Each level is made up of fibers which are the groups of non-zero coordinates highlighted in gray. In this example, we have the fiber (0, 1, 3) at level *i*, indicating that rows 0, 1, and 3 have nonzero values. Each coordinate has a reference, or a pointer, to a child fiber at a lower level. The final level of the fibertree contains the actual values in the matrix.

Abstract Machine

SAM describes a set of primitives that can operate on fibertrees in stream and storage formats. We show these formats for the example tensor in Figure 1. The stop tokens (S_n) represent the end of the fibers, and the done tokens (D) represent the end of the streams. Reference streams are used as pointers to fibers at the next level. For example, the reference *0* in the *i* reference stream points to the first fiber in the *j* level storage, as shown by the blue arrow. Using reference streams, our design can perform the indirect data accesses necessary in sparse applications. The storage format consists of segment and coordinate arrays for each tensor level except the last level, which stores the value array. Segment arrays contain tuples that denote the start and end of each fiber. For example, in the *i* level storage, we have the tuple (0, 3) indicating that the fiber consists of the first three coordinates. Using the stream and storage formats shown here, our accelerator can perform operations, such as an intersection or a union, on compressed tensors.

SAM primitives can be composed to evaluate any sparse tensor algebra expression. There are three memory and five processing primitives. The memory primitives (Figure 2) include a level writer, which converts streams to the storage representation, a level buffer, which contains a memory wrapper to arbitrate writes and reads to the SRAMs, and a level scanner, which receives a reference and produces coordinate and reference streams for lower-level fibers. The processing primitives (Figure 2) include intersecters, coordinate droppers, unioners, repeaters, and reducers. Intersecters consume incoming coordinate streams and output the common coordinates, eliminating unnecessary memory accesses and computation. Intersecters may produce empty fibers, so coordinate droppers must remove existing coordinates with empty fibers and their corresponding stop tokens at lower levels. Unioners collect all incoming coordinates, for situations like element-wise addition. The repeater duplicates streams for broadcasting one tensor over another, and the reducer performs a sum over a stream. With these primitives, our design has the generality to evaluate arbitrary sparse tensor algebra expressions[11]. Figure 3 demonstrates how the primitives described above compose to compute matrix-vector multiplication.

This generality also allows us to accelerate expressions with higher-order tensors and multiple inputs. Our accelerator is particularly effective for complex multi-input kernels that would produce intermediate

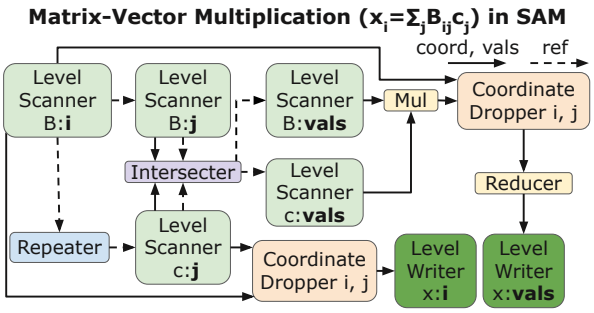
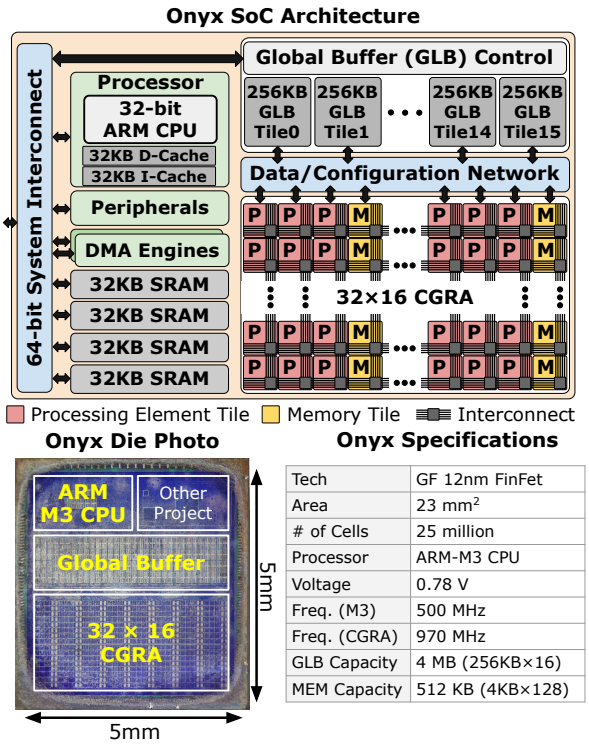


FIGURE 3. SAM graph for matrix-vector multiplication.



Tech	GF 12nm FinFet
Area	23 mm^2
# of Cells	25 million
Processor	ARM-M3 CPU
Voltage	0.78 V
Freq. (M3)	500 MHz
Freq. (CGRA)	970 MHz
GLB Capacity	4 MB (256KB×16)
MEM Capacity	512 KB (4KB×128)

FIGURE 4. Onyx SoC block diagram showing the CGRA, the global buffer (GLB), and the ARM M3 processor subsystem. Onyx die photo and specifications.

tensors, such as matricized tensor times Khatri-Rao product (MTTKRP $X_{ij} = \sum_{kl} B_{ikl} C_{jk} D_{jl}$). Our work supports expression fusion over multiple inputs, which allows the accelerator to avoid storing intermediate tensors and eliminate any unnecessary computation across all inputs. A fused implementation of MTTKRP performs up to 11.8× faster than its unfused implementation on the same programmable fabric.

Architecture

The Onyx (Figure 4) system on chip (SoC) contains a 32×16 CGRA of processing element and memory tiles, a 4 MB global buffer (GLB), and an ARM M3 processor subsystem. There are 384 processing element tiles (PEs), each of which has an arithmetic logic unit (ALU), a 64-byte register file, and the sparse processing primitives described above. Additionally, there are 128 memory tiles (MEMs), each of which has 4 KB of SRAM, address generation logic for dense applications, and the sparse memory primitives described above. The CGRA has three columns of PE tiles for each column of MEM tiles. The tiles are connected to each other over a 17-bit ready-valid interconnect that allows for routing in all directions. 16 bits are used for data, and 1 bit is used to designate the control tokens in the sparse streaming abstraction described above. The GLB is composed of 16 tiles. Each tile has two 128 KB SRAMs, a bidirectional connection to the top row of the CGRA, and a specialized network for application configuration. The ARM M3 orchestrates application acceleration, including data movement and accelerator configuration, and performs any operations not supported by the accelerator.

UTILIZATION OPTIMIZATIONS

The composable memory and processing primitives implemented in Onyx allow us to map any tensor algebra expression onto our accelerator, but by themselves, they do not achieve high temporal or spatial utilization of the accelerator. CPU overhead between accelerator calls results in low temporal utilization, leading to low performance. To improve temporal utilization, we implement **tile pipelining**, which allows us to overlap the loading of the next tile of data with the processing of the current tile. Low spatial utilization of the array of PEs and MEMs also leads to low performance on Onyx. To improve spatial utilization, we add three hardware primitives to the array that allow us to maximally **unroll** applications on our accelerator: a stream filter, an arbiter, and an output address generator. We describe these optimizations in detail in the rest of this section.

Tile Pipelining

One crucial consideration when accelerating tensor algebra is efficiently mapping large tensors onto constrained on-chip memories. For example, in the case of a large matrix multiplication, an input matrix may exceed the size of available memory. Therefore, like other push-memory accelerators, we *tile* input matrices into smaller chunks and accelerate those chunks individually to produce partial results. Onyx accelerated

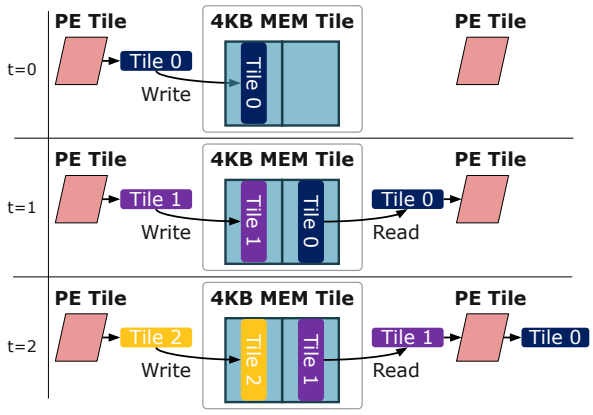


FIGURE 5. Overlapping execution by pipelining tensor tiles onto our accelerator.

each individual tile pair with one kernel call. This meant that for very sparse tensors, a significant portion of the application runtime was spent on CPU cycles for accelerator management and not on actual computation. To improve runtime, we implemented *tile pipelining*, which pipelines the tiled execution of a large tensor operation onto and off of the accelerator.

We introduce new features to both the memory and processing primitives to enable tile pipelining. For the memory primitives, we design the level buffer as a depth-two FIFO of levels (Figure 5), where the reads and writes are handled by two separate state machines that never operate on the same level simultaneously. When reading and writing a level, the two state machines each maintain their own base and bound registers for their respective memory segments. When a write finishes, the write state machine pushes the base and bound to a level status FIFO and updates the write base register with the starting address of the next available line. When a read finishes and the level status FIFO is not empty, the read state machine pops the FIFO to get the base and bound for the next level to read. This mechanism allows us to overlap the processing of one pair of tiles with the next. Additionally, all processing primitives return to an idle state after receiving a done token, which ensures proper sequencing of unrelated streams.

Unrolling

Unrolling a kernel on Onyx is constrained by the number of input and output links from the GLB to the CGRA, which can be a maximum of 16 each way in the Onyx design. To determine the number of links used by a sparse tensor algebra expression, we can

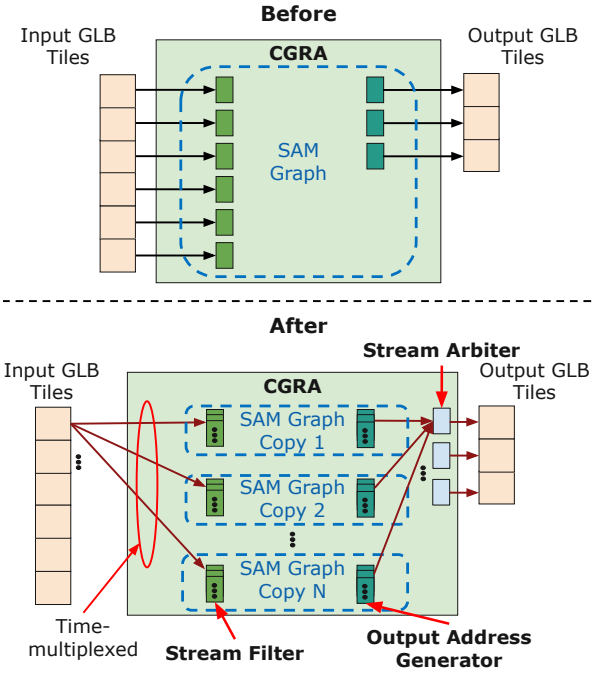


FIGURE 6. CGRA showing the three key primitives needed to enable application unrolling: stream filter, arbiter, and output address generator.

analyze the input and output tensor dimensions for the expression. For example, matrix multiplication has two input matrices and one output matrix, where each matrix's fibertree has three levels. This expression would use six GLB inputs and three GLB outputs. This means we can only accelerate two matrix multiplies in parallel, resulting in very low spatial utilization of the accelerator. To fix this limitation, we time-multiplex our GLB outputs (accelerator inputs) and arbitrate our GLB inputs (accelerator outputs). To accomplish this, we add stream filter, arbiter, and output address generator primitives to our design as shown in Figure 6. These primitives allow us to place several copies of the same kernel on our accelerator simultaneously, greatly improving array utilization and performance.

The stream filter (Figure 7) is placed in the level writer. It sends data from one GLB output to several memory tiles. First, we add a configuration register to each memory tile that contains an identifier (ID). Then, we prepend our GLB output streams with that ID to signify which memory tiles should capture the incoming stream. This allows us to time-multiplex our GLB outputs in a single accelerator call.

SAM graphs mapped onto the accelerator may complete execution in any order since their runtime is

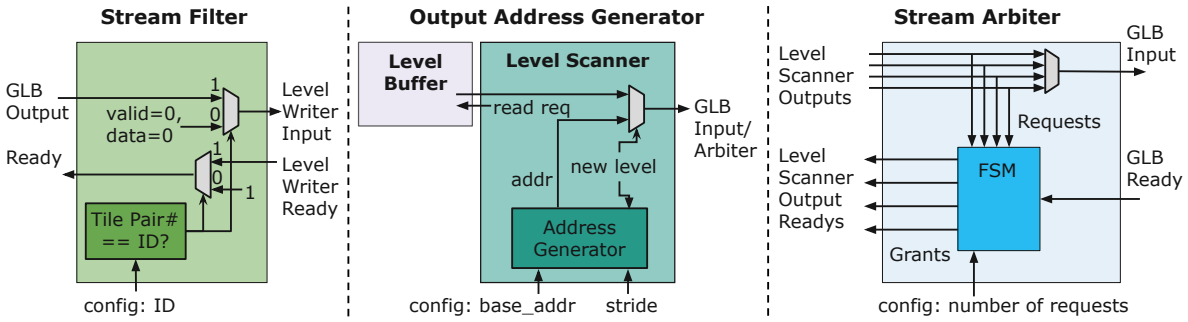


FIGURE 7. Diagram of added primitives for unrolling. The stream filter shows interaction with GLB and level writer. The output address generator shows interaction with level buffer and GLB. The arbiter shows the interaction with level scanner and GLB.

data-dependent. Therefore, we implement an inverse of the stream filter in the level scanner to write the output tiles coming out of these graphs in the correct order in the GLB. Specifically, we add two configurations to each memory tile: a maximum output size and stride. The maximum output size determines the spacing between outputs, and the stride is the number of simultaneously accelerated kernels. An output address generator (Figure 7) uses these configurations to reserve fixed-size blocks in the GLB. The level scanner prepends the generated address to the output stream for the GLB, signaling the output write location.

The arbiter (Figure 7) allows us to collect outputs from different level scanners and route them to the GLB. We implement round-robin arbitration to ensure fairness and prevent starvation. The stream arbiter reuses the same FIFOs and four input ports as the other PE processing primitives to save area. For scenarios with an unrolling factor greater than 4, arbiters are hierarchically assembled by the compiler. With the stream filter and output address generator primitives described above, we can now maximally unroll our sparse applications. Unrolling significantly improves accelerator utilization as shown in Figure 8. For the applications shown, the utilization increases by up to 7.5×.

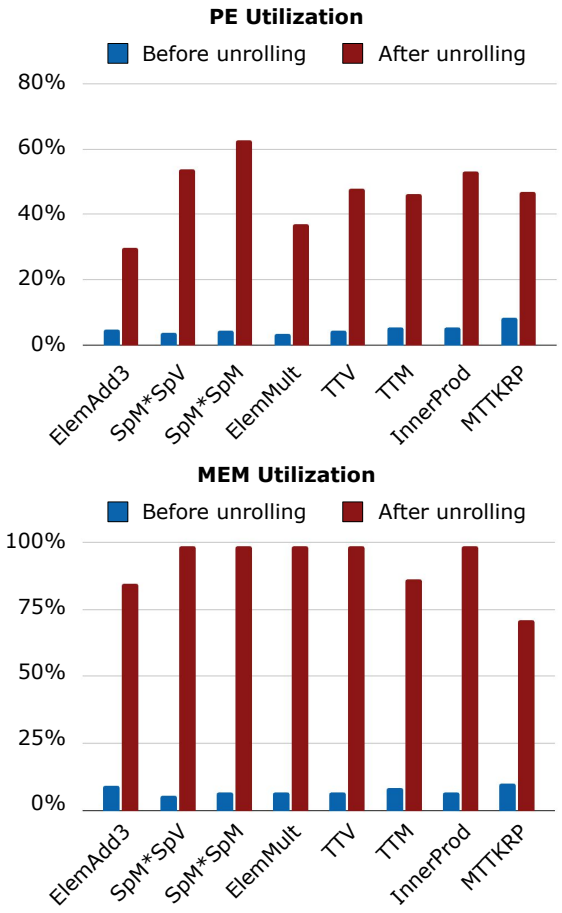


FIGURE 8. PE and MEM tile utilization before and after enabling maximal unrolling. Further unrolling is prevented by a lack of memory tiles or failures in application place and route.

EVALUATION

Hardware Overhead

We synthesize the design (in Intel 16 technology) both without and with the new hardware features. Our design contributions increase the PE tile area by 0.86% and the memory tile area by 0.56%. The significant runtime improvements shown in the next section justify the marginal area added by the additional primitives.

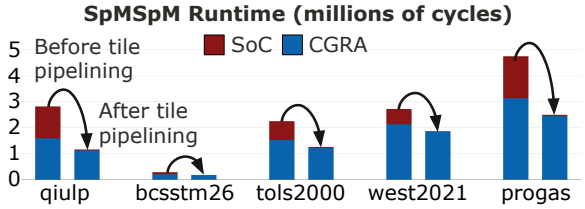


FIGURE 9. Improvements in both SoC and accelerator runtimes using tile pipelining.

Runtime Results

We use an end-to-end sparse compiler[9] to map sparse kernels onto our CGRA. For 2D input tensors, we evaluate sparse matrix-sparse matrix multiplication (SpMSpM) on 5 SuiteSparse[12] matrices (~2000x2000 with 99.0 - 99.5% sparsity). For 3D input tensors, we evaluate MTTKRP, an expression from scientific computation, on 5 randomly generated sparse tensors (98% uniform random sparsity, ~100x200x200). We implemented our design in Verilog, and the runtime results are derived from RTL simulations that incorporate estimated processor overheads.

Tile pipelining allows us to significantly improve runtime by both reducing SoC overheads and overlapping the execution of consecutive tiles. Figure 9 shows runtime improvements on the five SuiteSparse matrices. SoC overhead drops by up to 31.7× as we no longer manage acceleration at the memory tile size, but instead at the global buffer tile size. Additionally, the accelerator runtime reduces by up to 1.41× as well because we can overlap the execution of different tiles.

Unrolling improves runtime significantly by accelerating several kernels in parallel. To understand the benefits and limitations of unrolling, we show SpMSpM and MTTKRP runtimes on a single tensor each in Figure 10. For SpMSpM on the matrix *qiulp*, with an unroll factor of 7, we see a 4.6× runtime improvement. For MTTKRP on a randomly generated tensor with 98% sparsity, with an unroll factor of 7, we see a 3.2× runtime improvement.

There are two reasons for not achieving an ideal speedup: load imbalance and limited global buffer bandwidth. First, since the runtime of individual kernels is data-dependent, we run into load balancing issues (as shown in Figure 10 for SpMSpM). There are several possible strategies for load balancing, which we leave as future work. Another reason for not achieving an ideal speedup when unrolling is limited GLB bandwidth (as shown in Figure 10 for MTTKRP). When data transfer to the accelerator takes longer than the execution, more unrolling does not improve runtime and

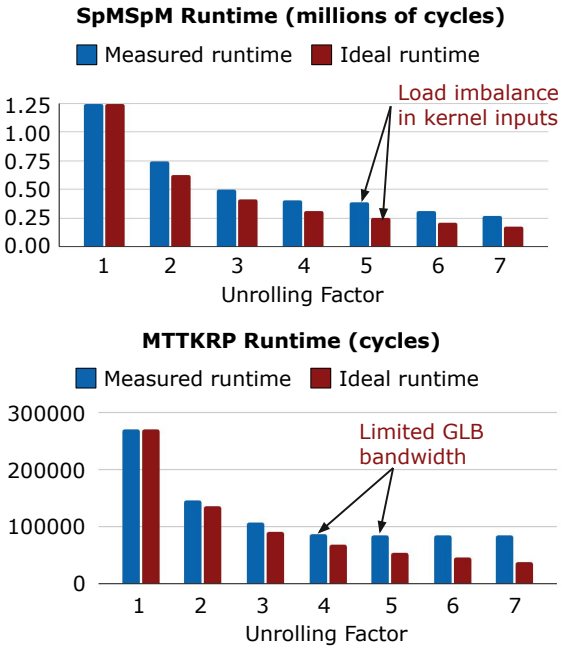


FIGURE 10. SpMSpM and MTTKRP runtime results with different unrolling factors.

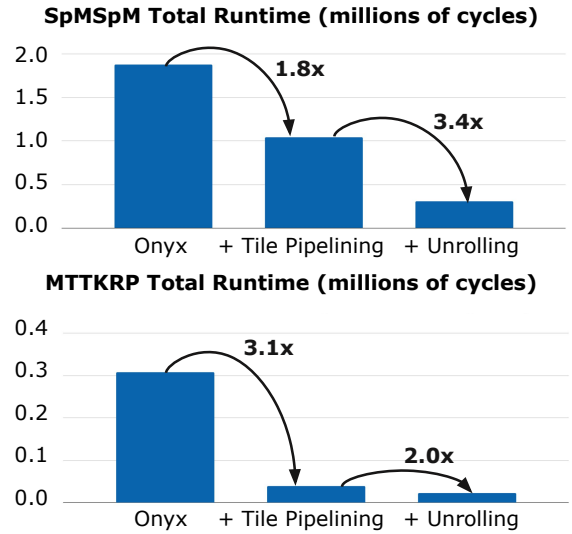


FIGURE 11. SpMSpM and MTTKRP runtime results after tile pipelining and unrolling.

wastes accelerator resources. Future work can explore identifying the bottlenecks for a given expression and utilizing free global buffer links to reduce runtime.

Figure 11 shows runtime improvements on SpMSpM and MTTKRP, respectively, after both optimizations are performed. We perform 6.1× better on matrix

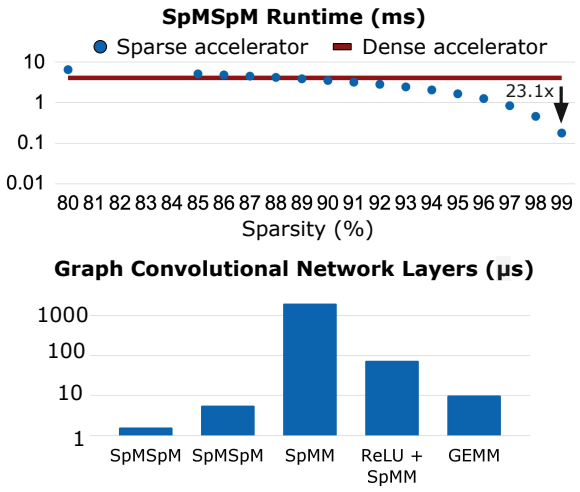


FIGURE 12. Runtime comparison of sparse and dense accelerator configurations for matrix multiplication and runtime of GCN layers on a 100-node graph.

multiplication and 6.2× better on MTTKRP than the baseline Onyx.

Figure 12 (top) shows runtime results for a 512×512 matrix multiplication for different input data sparsities for our CGRA configured as a dense accelerator and as a sparse accelerator. After 89% sparsity, configuring the CGRA to be a sparse accelerator is advantageous. At 99% sparsity, the sparse accelerator performs 23.1× better than the dense accelerator, and at 99.9% sparsity it performs 165× better. Figure 12 (bottom) shows runtime results for a graph convolutional network (GCN) comprising several sparse and dense layers demonstrating how our CGRA can accelerate end-to-end applications. Future work includes exploring different matrix multiplication schedules (i.e. inner product vs. Gustavson's algorithm), different tiling schemes, and further optimizing GLB–accelerator bandwidth.

CONCLUSION

Our work explores improving state-of-the-art sparse kernel acceleration on programmable fabrics. Compared against the same kernels and schedules in prior work, we achieve up to 15× better spatial utilization and up to 6.2× runtime improvement. Our work makes significant progress in improving performance for sparse inputs and enables mapping end-to-end applications that can leverage both sparse and dense acceleration on the same accelerator fabric.

ACKNOWLEDGMENTS

This work was supported by the DARPA DSSoC grant, the Stanford AHA Agile Hardware Center and Affiliates Program, Intel's Science and Technology Center (ISTC), Stanford SystemX Alliance, SRC JUMP 2.0 PRISM Center, NSF CAREER Award (2238006), Hellman Faculty Scholar Program, Apple Stanford EE PhD Fellowship, and NSF GRFP.

REFERENCES

1. A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "SCNN: An accelerator for compressed-sparse convolutional neural networks," in *Proc. 2017 ACM/IEEE 44th Ann. Int. Symp. Comput. Archit. (ISCA)*, 2017, pp. 27–40, doi:10.1145/3079856.3080254.
2. S. Song, D. Han, S. Kim, S. Kim, G. Park, and H.-J. Yoo, "GPPU: A 330.4-uJ/task neural path planning processor with hybrid GNN acceleration for autonomous 3D navigation," in *Proc. 2023 IEEE Symp. VLSI Technol. Circuits*, 2023, pp. 1–2, doi: 10.23919/VLSITechnologyandCir57934.2023.10185367.
3. W.-C. Huang, I.-T. Lin, W.-C. Chen, L.-Y. Lin, N.-S. Chang, C.-P. Lin, C.-S. Chen, and C.-H. Yang, "A 28-nm 25.1 TOPS/W sparsity-aware CNN-GCN deep learning SoC for mobile augmented reality," in *Proc. 2019 Symp. VLSI Circuits*, 2022, pp. 42–43, doi:10.1109/VLSITechnologyandCir46769.2022.9830261.
4. Y. Yang, J. S. Emer, and D. Sanchez, "Trapezoid: A versatile accelerator for dense and sparse matrix multiplications," in *Proc. 2024 ACM/IEEE 51st Ann. Int. Symp. Comput. Archit. (ISCA)*, 2024, pp. 931–945, doi:10.1109/ISCA59077.2024.00072.
5. K. Feng, T. Kong, K. Koul, J. Melchert, A. Carsello, Q. Liu, G. Nyengele, M. Strange, K. Zhang, A. Nayak, J. Setter, J. Thomas, K. Sreedhar, P.-H. Chen, N. Bhagdikar, Z. A. Myers, B. D'Agostino, P. Joshi, S. Richardson, C. Torng, M. Horowitz, and P. Raina, "Amber: A 16-nm system-on-chip with a coarse-grained reconfigurable array for flexible acceleration of dense linear algebra," *IEEE J. Solid-State Circuits*, vol. 58, no. 4, pp. 1–13, 2023, doi:10.1109/JSSC.2023.3313116.
6. A. Rucker, M. Vilim, T. Zhao, Y. Zhang, R. Prabhakar, and K. Olukotun, "Capstan: A vector RDA for sparsity," *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 1022–1035, doi:10.1145/3466752.3480047.
7. S. Dave, R. Baghdadi, T. Nowatzki, S. Avancha, A. Shrivastava, and B. Li, "Hardware acceleration of sparse and irregular tensor computa-

tions of ML models: A survey and insights," *Proc. IEEE*, vol. 109, no. 10, pp. 1706–1752, 2021, doi:10.1109/JPROC.2021.3098483.

8. K. Koul, M. Strange, J. Melchert, A. Carsello, Y. Mei, O. Hsu, T. Kong, P.-H. Chen, H. Ke, K. Zhang, Q. Liu, G. Nyengele, A. Balasingam, J. Adivarahan, R. Sharma, Z. Xie, C. Torng, J. Emer, F. Kjolstad, M. Horowitz, and P. Raina, "Onyx: A 12nm 756 GOPS/W coarse-grained reconfigurable array for accelerating dense and sparse applications," in *Proc. 2024 IEEE Symp. VLSI Technol. Circuits*, 2024, pp. 1–2, doi:10.1109/VLSITechnologyandCir46783.2024.10631383.
9. K. Koul, M. Strange, J. Melchert, A. Carsello, Y. Mei, O. Hsu, T. Kong, P.-H. Chen, H. Ke, K. Zhang, Q. Liu, G. Nyengele, A. Balasingam, J. Adivarahan, R. Sharma, Z. Xie, C. Torng, J. Emer, F. Kjolstad, M. Horowitz, and P. Raina, "Onyx: A programmable accelerator for sparse tensor algebra," in *2024 IEEE Hot Chips 36 Symposium (HCS), Stanford, CA, USA*, 2024, pp. 1-91, doi: 10.1109/HCS61935.2024.10665150.
10. V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, *Efficient Processing of Deep Neural Networks*. Morgan & Claypool Publishers, 2020.
11. O. Hsu, M. Strange, R. Sharma, J. Won, K. Olukotun, J. S. Emer, M. A. Horowitz, and F. Kjolstad, "The sparse abstract machine," in *Proc. 28th ACM Int. Conf. Archit. Support for Program. Lang. Oper. Syst. (ASPLOS)*, 2023, pp. 710–726, doi:10.1145/3582016.3582051.
12. T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, pp. 1–25, 2011.

Kalhan Koul is an Electrical Engineering Ph.D. student at Stanford University. He received an M.S. in Electrical Engineering from Stanford University in 2021. His research interests are in accelerators and agile hardware design. Contact him at kkoul@stanford.edu.

Zhouhua Xie is an undergraduate student at Stanford University. His research interests include domain-specific hardware architectures and hardware-software co-design. Contact him at xzh015@stanford.edu.

Maxwell Strange is an Electrical Engineering Ph.D. student at Stanford University. He received an M.S. in Electrical Engineering from Stanford University in 2020. His research interests include domain-specific hardware architectures, hardware-software co-design, and embedded systems design. Contact him at mstrange@stanford.edu.

Sai Gautham Ravipati is an Electrical Engineering M.S. student at Stanford University. He received a B.Tech. degree in Electrical Engineering from the IIT Madras in 2023. His research interests include programming systems, domain-specific compilers, and hardware-software co-design. Contact him at sgautham@stanford.edu.

Bo Wun Cheng is an Electrical Engineering Ph.D. student at Stanford University. He received an M.S. in Computer Science from National Tsing Hua University (Taiwan) in 2023. His research interests are in reconfigurable accelerators and accelerator memory systems. Contact him at bwcheng@stanford.edu.

Olivia Hsu is a Computer Science Ph.D. student at Stanford University. She received a B.S. in Electrical Engineering and Computer Science from the University of California, Berkeley in 2019. Her research interests include accelerator architectures, programming systems, and domain-specific compilers. Contact her at owhsu@stanford.edu.

Po-Han Chen is an Electrical Engineering Ph.D. student at Stanford University. He received his M.S. in Electrical Engineering from National Tsing Hua University (Taiwan) in 2018. His research interests include coarse-grained reconfigurable arrays (CGRAs) and high-performance and energy-efficient computing platforms. Contact him at pohan@stanford.edu.

Mark Horowitz is the Yahoo! Founders Professor at Stanford University and chair of the electrical engineering department. He is an IEEE fellow, an ACM fellow, and a member of the National Academy of Engineering and the American Academy of Arts and Science. He received a Ph.D. from Stanford University in 1984. His current research includes updating both analog and digital design methods, agile hardware design, and applying engineering to biology. Contact him at horowitz@ee.stanford.edu.

Fredrik Kjolstad is an Assistant Professor of Computer Science at Stanford University. He received a Ph.D. degree in Computer Science from the Massachusetts Institute of Technology in 2020. His research interests include sparse computing, compilers, and programming models and languages. Contact him at kjolstad@stanford.edu.

Priyanka Raina is an Assistant Professor of Electrical Engineering at Stanford University. She received a Ph.D. in Electrical Engineering and Computer Sci-

ence from MIT in 2018. Her research interests are in domain-specific hardware architectures and agile hardware–software codesign methodology. Contact her at praina@stanford.edu.