Onyx: A Programmable Accelerator for Sparse Tensor Algebra

Kalhan Koul[1], Maxwell Strange[1], Jackson Melchert[1], Alex Carsello[1], Yuchen Mei[1], Olivia Hsu[1], Taeyoung Kong[1], Po-Han Chen[1], Huifeng Ke[1], Keyi Zhang[1], Qiaoyi Liu[1], Gedeon Nyengele[1], Akhilesh Balasingam[1], Jayashree Adivarahan[1], Ritvik Sharma[1], Zhouhua Xie[1], Christopher Torng[2], Joel S Emer[3], Fredrik Kjolstad[1], Mark Horowitz[1], Priyanka Raina[1]
[1]Stanford University, CA, USA; [2]University of Southern California, CA, USA; [3]MIT, MA, USA

I. Introduction

SPARSE tensor algebra is used to accelerate a wide variety of applications such as graph analytics, scientific computing, and machine learning (ML) by eliminating unnecessary computation. Specialized accelerators have been proposed for specific sparse tensor expressions, such as matrix multiplication, but they fail to map other sparse kernels (e.g. element-wise or multi-input expressions) that are needed in these applications. To support a variety of sparse expressions we need a programmable accelerator. Coarse-grained reconfigurable arrays (CGRAs) are widely used programmable accelerators, but prior CGRAs have primarily focused on dense applications. We present Onyx, the first programmable accelerator for both dense and sparse applications. Onyx is fabricated in GF 12nm. We design programmable hardware primitives that can be composed to accelerate arbitrary sparse tensor algebra expressions, including those with higher-dimensional tensors, multiple inputs, and fusion. Onyx achieves up to 565x better energy-delay product (EDP) for sparse kernels vs. CPUs with sparse libraries, and up to 76% and 85% lower EDP for image processing and ML, respectively, vs. prior CGRAs.

II. Architecture Overview

The Onyx SoC (Figure 1) has a CGRA with 384 processing elements (PE) and 128 memory tiles (MEM) in an interconnect that supports both static data movement for dense applications and dynamic data movement for sparse applications. The PE contains a 64B register file and an ALU supporting INT16 and BFLOAT16 operations. Additionally, the PE supports more complex INT16 multiply-add, multiply-shift, max-min, and add-add operations, which are common operations we mined from image processing and ML applications. The MEM contains 4KB of SRAM and memory controllers that support affine access patterns for dense applications. The CGRA connects to a 4 MB global buffer (GLB) that has 16 tiles, each with load and store engines. An Arm M3 processor orchestrates kernels on the CGRA and GLB. The interconnect is composed of switch boxes (SBs) which route data to and from tiles and connection boxes which route data into the core of the tile. We add sparse primitives to these tiles to eliminate unnecessary compute in our sparse kernels.

III. Sparse Acceleration Hardware

Prior sparse accelerators have focused on sparsity in specific kernels, mostly matrix multiplication, leaving out support for other kernels, higher-order tensors, multiple inputs, and fusion. Onyx addresses this by building on a streaming representation called the Sparse Abstract Machine (SAM) [1] that uses fibertrees [2]–[4] (Figure 2) to store and process any-order tensors and to eliminate unnecessary memory accesses. We convert stored tensors into coordinate, reference (pointer to

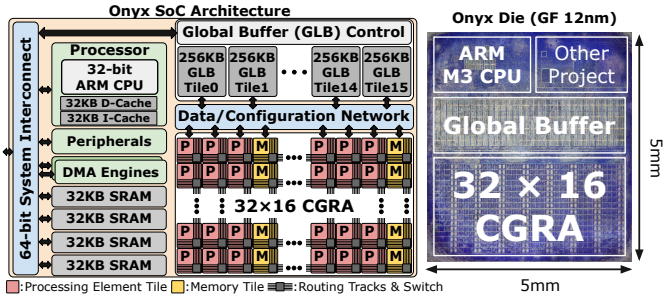

Fig. 1. Onyx SoC architecture with a CGRA containing 384 PEs and 128 MEMs and die photo of Onyx.

fiber), and value streams that move between CGRA tiles using level writer, level buffer, and level scanner primitives, which we implement in the memory tile (Figure 2). We partition a single-port 512x64b SRAM in the level buffer into segment and coordinate arrays. The level writer consumes a coordinate stream and generates segment and coordinate addresses to write to the partitioned memory. The level scanner takes a reference pointing to a segment pair, requests that pair from the level buffer to determine the coordinate array length, and then generates addresses to read the coordinate stream from the level buffer. It also produces the corresponding reference stream for downstream tiles. We add cache lines in the level buffer to avoid redundant reads of the same 4-element word in consecutive accesses.

Figure 3 shows the intersecter/unioner, reducer, repeater, and coordinate dropper primitives in the PE tile that perform computation on the fibertree indexing data. The intersecter/unioner calculates the intersection/union of coordinate

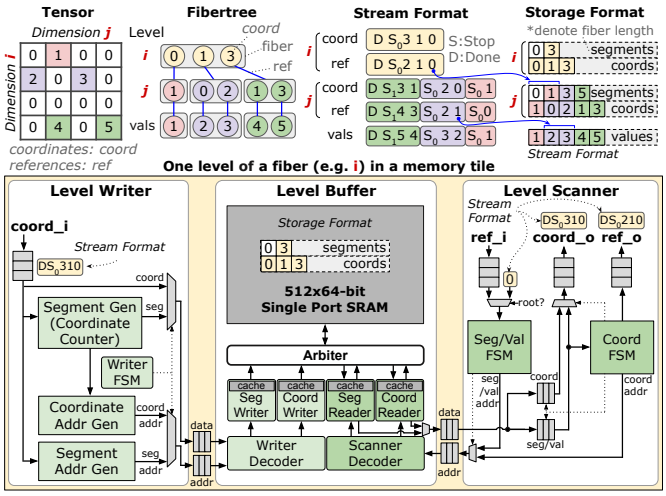

Fig. 2. Sparse MEM primitives that read and write fibertrees.

streams of two tensors. The intersecter only emits coordinates and references of both streams when the coordinates are equal. The unioner uses the same hardware but outputs all coordinates (merging duplicate coordinates and using empty tokens for unknown references). These primitives, while complete, can produce empty fibers. The coordinate dropper removes these tokens by looking for contiguous stop tokens in the inner coordinate stream and dropping the corresponding outer coordinate. Finally, the reducer primitive performs a reduction over a stream, outputting a single value, and the repeater primitive duplicates streams for tensor broadcasting.

To support non-deterministic latency, we add ready-valid capability to the interconnect by replacing pipeline registers with FIFOs. We avoid doubling the number of registers by designing a "SplitFIFO" with pipeline registers in adjacent tiles sharing control signals, saving 13% in interconnect area.

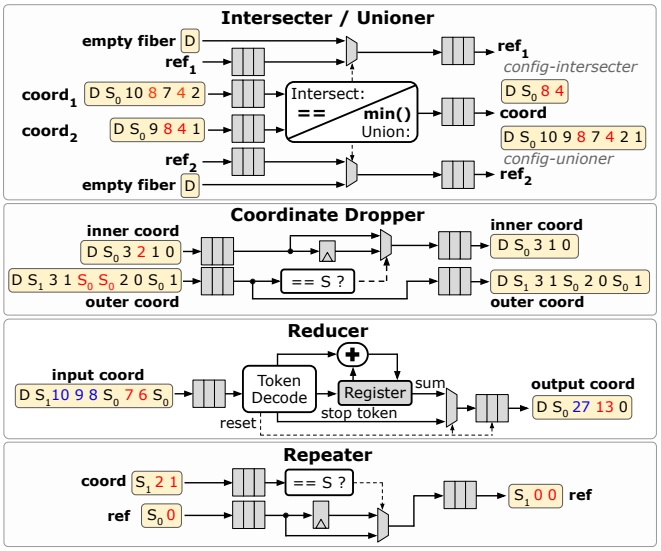

Fig. 3. Sparse PE primitives that compute on tensor indices and addresses.

IV. Mapping Sparse Applications

Our sparse applications are written in Custard [1]. The Custard compiler transforms the application into a dataflow graph of SAM primitives. Those primitives are then mapped onto PE and MEM tiles, place and routed on our CGRA, and a bitstream is generated.

To demonstrate how the sparse primitives implement arbitrary sparse expressions, we show sparse inner-product matrix multiplication ($X_{ij} = \sum_k B_{ik}C_{kj}$) in Figure 4. Intermediate reference and coordinate streams are shown at each step of the graph. First, we have two pairs of level scanners (B: level i and C: level j) and repeaters for the outer dimension of each matrix. The level scanner/repeater pairs produce replicated references indicating nonzero rows/columns (in matrix multiplication, B is replicated for each j and C for each i). Next, those replicated references are fed into level scanners for the shared dimension (k) of each matrix. These level scanners produce reference-coordinate pairs that are intersected to determine the values that need to be multiplied. The resulting intersected pairs are then used to read values, multiply, and reduce them, before, finally, a level writer writes the values to memory. Using this composable set of primitives, we can accelerate any sparse tensor algebra expression on our CGRA. Our generality supports higher-order tensors, multiple inputs, and, importantly, fusion for expressions such as matricized tensor times Khatri-Rao product (MTTKRP: $X_{ij} = \sum_{kl} B_{ikl}C_{jk}D_{jl}$). MTTKRP's fused schedule eliminates ineffectual compute across all inputs and is up to 6.6x faster than two unfused kernels.

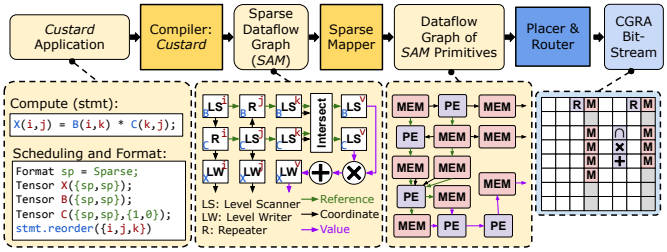

Fig. 4. Our compiler, with a sparse matrix multiplication mapping example.

V. Measured Results

Onyx is fabricated in GF 12 nm technology (Figure 1). We use the Halide compiler [5] to map dense applications. We evaluate the sparse expressions on real-world 2D matrices from the SuiteSparse dataset. For expressions with 3D tensors, we generate our own synthetic data. Figure 5 shows the measured results. On image processing, we achieve a 33%, 76% and 74% EDP reduction vs. Amber [6] on blur, unsharp, and Harris, respectively. For ResNet layers, we achieve 61–68% lower runtime and 43–55% lower energy vs. Amber. On sparse tensor algebra kernels, we achieve a 4.4x–565x geomean improvement in EDP versus a CPU using sparse libraries (MKL and TACO). Onyx runs at a maximum frequency of 970MHz at 0.78V, and achieves a peak energy efficiency of 756 INT16 GOPS/W, 41% better than the prior CGRA [6].

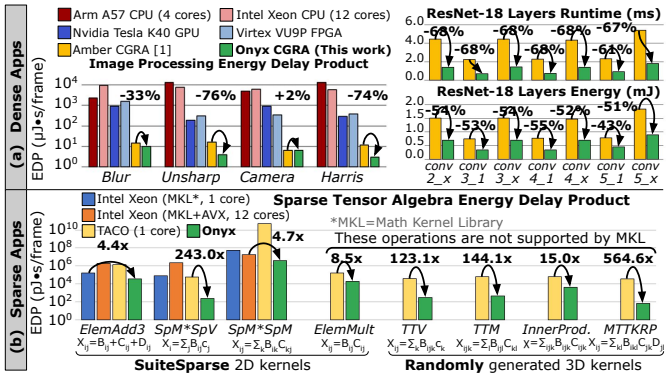

Fig. 5. Measured results on (a) dense applications: image processing EDP vs. other platforms, and machine learning (ResNet-18) runtime and energy vs. Amber (b) sparse applications: EDP vs. CPU with MKL/TACO sparse library.

VI. Conclusion

Unlike most systems, which put together specialized accelerators for different applications on the same chip, Onyx is a unified programmable accelerator that supports arbitrary dense and sparse tensor algebra applications and demonstrates a systematic methodology for extending CGRAs to new domains.

References

[1] Sze et al., Efficient processing of deep neural networks, Morgan & Claypool Publishers, 2020.

[2] Hsu et al., The sparse abstract machine, ASPLOS 2023.

[3] Ragan-Kelley et al., Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines, PLDI 2013.

[4] Carsello et al., Amber: A 367 GOPS, 538 GOPS/W 16nm SoC with a coarse-grained reconfigurable array for flexible acceleration of dense linear algebra, VLSI 2022.

REFERENCES

[1] O. Hsu, M. Strange, R. Sharma, J. Won, K. Olukotun, J. S. Emer, M. A. Horowitz, and F. Kjølstad, "The sparse abstract machine," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 710–726. [Online]. Available: https://doi.org/10.1145/3582016.3582051

[2] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, *Efficient Processing of Deep Neural Networks*. Morgan & Claypool Publishers, 2020.

[3] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–29, 2017.

[4] S. Chou, F. Kjolstad, and S. Amarasinghe, "Format abstraction for sparse tensor algebra compilers," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 123:1–123:30, October 2018.

[5] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 519–530. [Online]. Available: https://doi.org/10.1145/2491956.2462176

[6] A. Carsello, K. Feng, T. Kong, K. Koul, Q. Liu, J. Melchert, G. Nyengele, M. Strange, K. Zhang, A. Nayak, J. Setter, J. Thomas, K. Sreedhar, P.-H. Chen, N. Bhagdikar, Z. Myers, B. D'Agostino, P. Joshi, S. Richardson, R. Bahr, C. Torng, M. Horowitz, and P. Raina, "Amber: A 367 GOPS, 538 GOPS/W 16nm SoC with a coarse-grained reconfigurable array for flexible acceleration of dense linear algebra," in *2022 IEEE Symposium on VLSI Technology and Circuits*, 2022, pp. 70–71.