Onyx: A 12nm Programmable Accelerator for Dense and Sparse Applications

Kalhan Koul[1*], Olivia Hsu[1*], Yuchen Mei[1], Sai Gautham Ravipati[1], Maxwell Strange[1], Jackson Melchert[1], Alex Carsello[1], Taeyoung Kong[1], Po-Han Chen[1], Huifeng Ke[1], Keyi Zhang[1], Qiaoyi Liu[1], Gedeon Nyengele[1], Zhouhua Xie[1], Akhilesh Balasingam[1], Jayashree Adivarahan[1], Ritvik Sharma[1], Christopher Torng[2], Joel S Emer[3], Fredrik Kjolstad[1], Mark Horowitz[1], Priyanka Raina[1]

***Abstract*—Onyx is a system-on-chip (SoC) with a coarse-grained reconfigurable array (CGRA) for accelerating sparse and dense tensor algebra and dense image processing and machine learning applications. To support multiple inputs, multiple dimensions, and fusion in sparse applications, Onyx utilizes composable memory primitives that operate on compressed storage and streams, and compute primitives that eliminate unnecessary calculations. Onyx also improves performance on dense applications with application-specialized processing elements, area-optimized memory tiles, and hybrid clock gating in the global buffer. Onyx achieves a peak energy efficiency of 756 INT16 GOPS/W, up to $565\times$ 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 machine learning, respectively, versus a state-of-the-art CGRA.**

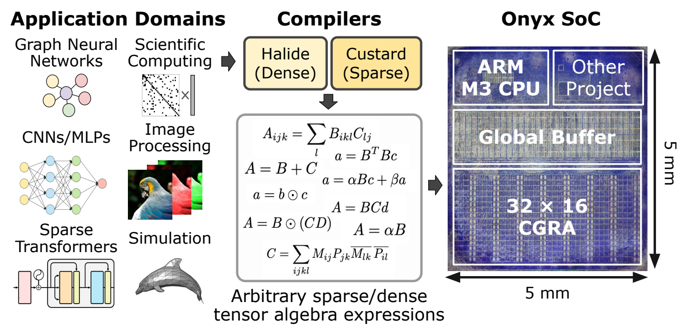

Fig. 1: End-to-end applications are broken down into dense and sparse kernels based on data sparsity. Then those kernels are mapped onto Onyx, which can accelerate both dense and sparse kernels.

I. Introduction

APPLICATIONS ranging from machine learning to scientific computing leverage hardware accelerators to improve performance and energy efficiency [1], [2]. Many of these accelerators focus on input data sparsity to skip ineffectual computation [3], [4], [5]. End-to-end applications, however, contain both dense and sparse portions [6]. Prior works typically combine specialized accelerators for different portions of an application on the same chip. For example, [7] contains separate accelerators for convolutional neural networks (CNNs) and graph convolutional networks (GCNs). These accelerators achieve high performance and efficiency for the applications they are designed for, but they are either inefficient or incapable of supporting other applications, and become unusable when a better algorithm is designed.

In contrast, programmable accelerators can accelerate applications as they evolve [8], [9]. These accelerators offer significant performance and energy efficiency improvements over general-purpose computing, but have primarily focused on dense data [10], [11]. Onyx overcomes the limitations of prior accelerators by supporting dense and sparse kernels on the same programmable accelerator as shown in Figure 1. End-to-end applications can be divided into sparse and dense portions, and then each of these portions can be accelerated on Onyx. Specifically, Onyx contains a coarse-grained reconfigurable array (CGRA) with composable tiles that allows it to support tensor algebra expressions with both sparse and dense tensors, multiple inputs, higher-order tensors, and fusion. In addition to sparse tensor algebra, Onyx also has dedicated support for image processing and machine learning applications. Onyx achieves this generality through the following contributions:

1) Composable memory primitives to store compressed representations of any-dimensional tensors.
2) Composable compute primitives that eliminate ineffectual computation and support all of sparse tensor algebra.
3) An automatic end-to-end sparse compiler that maps from high-level tensor index notation, also referred to as Einstein summation (Einsum) notation, to the hardware.
4) Optimizations to the sparse compiler, including software pipelining, kernel unrolling, and tensor tiling, for more performant mapping of sparse tensor expressions.
5) Application-driven processing element design and memory hierarchy optimizations to improve performance and efficiency of dense applications.

Onyx achieves up to 76% improvement in energy-delay product (EDP) over a state-of-the-art programmable accelerator on dense applications and is up to $565\times$ better in EDP versus CPUs with dedicated sparse libraries on sparse applications. Overall, this work demonstrates an approach to accelerate complete *domains of applications* on the same programmable hardware.

II. Onyx Architecture

Onyx is a system-on-chip (SoC) composed of a 32×16 CGRA, a 4 MB global buffer (GLB), and an ARM M3 processor [12], as shown in Figure 2. The CGRA has 384 processing element (PE) tiles and 128 memory (MEM) tiles. Each PE contains an arithmetic logic unit (ALU), a 64 byte register file (RF), and compute sparse primitives, described in Section III-C. Each MEM contains a 4 KB SRAM, a memory controller for dense applications [13], and a memory controller for sparse applications, described in Section III-B.

Authors[1] are with Stanford University, author[2] is at University of Southern California, and author[3] is at MIT. These authors* contributed equally.

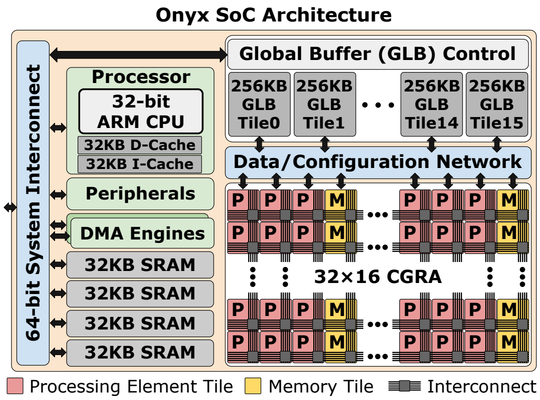

Fig. 2: Onyx SoC architecture. The accelerator contains a CGRA, a global buffer (GLB), and an ARM M3 control processor.

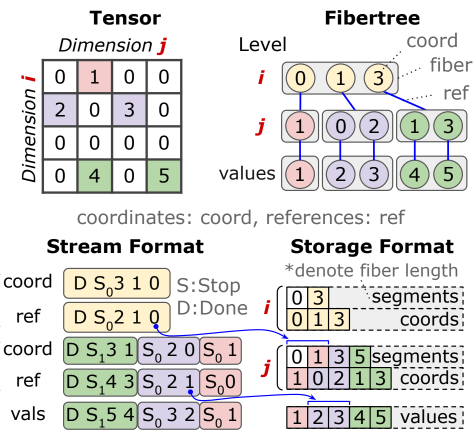

Fig. 3: Tensor in fibertree format, stream format, and storage format.

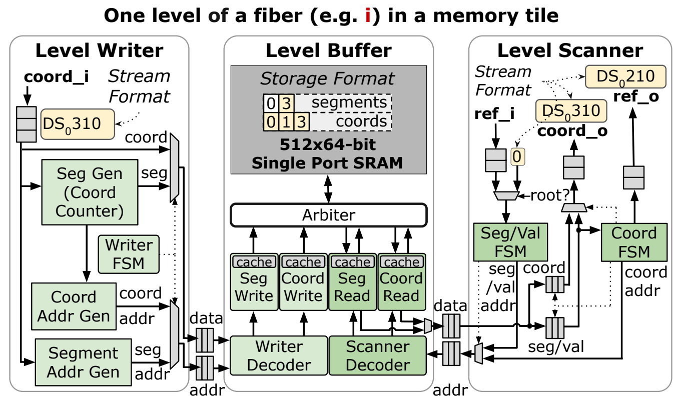

Fig. 4: Memory tile primitives: level writer, level buffer, and level scanner.

The CGRA has three PE columns per MEM column. The tiles are connected through a statically configured interconnect that supports static (for dense applications) and dynamic (for sparse applications) data movement [14]. The interconnect routes 17 bit data signals through switch boxes (SBs) and connection boxes (CBs) that are inside the PE and MEM tiles. Each PE and MEM has a switch box which selects the direction in which we send outgoing data, and connection boxes to bring incoming data. The GLB is composed of 16 GLB tiles. Each GLB tile has a 16-bit two-way connection to the CGRA, contains two 128 KB SRAM banks, load and store units, and a specialized configuration network [15].

III. Sparse Hardware

To accelerate arbitrary sparse tensor algebra, Onyx contains hardware implementations of the sparse acceleration primitives defined by the sparse abstract machine (SAM) [16]. We design (1) sparse memory primitives that convert between compressed storage and streams, (2) sparse compute primitives that eliminate ineffectual compute and support tensor algebra operations, and (3) a ready-valid interconnect.

A. Sparse Abstraction

We utilize the fibertree abstraction [17] and SAM's streaming tensor abstraction [16] to express tensors on Onyx. Figure 3 shows an example tensor, its corresponding fibertree format, its SAM stream representation, and its storage format as a doubly compressed sparse row (DCSR) data structure. Each level of a fibertree is a dimension of a tensor and is composed of fibers, which are lists of coordinates highlighted in gray and corresponding references to their payload (either another fiber or a value). In our example, we have the fiber [0, 1, 3] at level i, meaning that rows 0, 1, and 3 are nonzero. Each coordinate points to a child fiber at a lower level; this pointer is called a reference. The last level of a fibertree is composed of the actual values of the tensor. Fibertrees can be represented as streams or storage [16] in our hardware. There are three types of streams: coordinates, references, and values. Streams have hierarchical stop (S_n) tokens to denote boundaries between fibers, done tokens (D) to denote the end of a stream, and maybe tokens (M) to denote empty fibers.

B. Memory Tile Primitives

The memory tile is used to store a level of a fibertree. It contains three sparse memory controller primitives: level writer, level buffer, and level scanner. The level writer consumes a coordinate stream and writes it to the level buffer. It uses a coordinate address generator and a segment address generator to calculate coordinate and segment addresses and a coordinate counter to calculate the segment size. Specifically, the coordinate counter and coordinate address increment for each value token and the segment address increments with each stop token. A mux selects between writing a segment or a coordinate, and the address generators generate addresses to write to the corresponding array. For example, for the stream in Figure 4, the level writer first forwards the coordinates to write to the coordinate array. Simultaneously, the coordinate counter counts 3 tokens and generates (0, 3) to write to the segment array. To store the value level of the fibertree, we send the values instead of coordinates to the level buffer.

The level buffer receives writes and reads from the level writer and level scanner, respectively, and performs explicit decoupled data orchestration (EDDO [18]). It is either partitioned into segment and coordinate arrays or is a value array. It arbitrates between reads and writes with a cache line to avoid repetitive reads of the same 4-element word in consecutive accesses. Specifically, during the writing process, a write state machine writes each value to a cache line and does a group

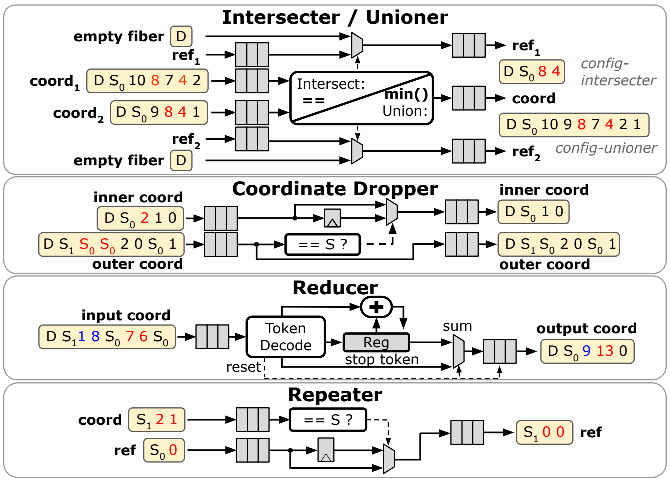

Fig. 5: PE tile primitives: intersecter/unioner, coordinate dropper, reducer, and repeater.

write when the line is full. If the cache line is not full, but the stream has ended, the state machine pushes the cache line with zeros filled in. When a read request comes in, the read state machine checks if the read request for that address was previously performed. If so, it uses the read cache line, otherwise it performs a new read request. With only a single memory port, all read and write accesses go through an arbiter that processes requests in order.

The level scanner takes in a reference stream pointing to a fiber(s) and produces a stream of coordinates and references to the next level. Each value in the incoming reference stream is the index of a fiber stored in the memory tile. That index points to the segment we want to read out of the memory tile. First, the level scanner receives an input reference stream. To produce next level coordinate and reference streams, the Seg/Val state machine reads the segment size from the level buffer, then the Coord state machine reads the coordinates. For each value token in the reference stream, the Seg/Val state machine issues two segment reads and reserves a stop token for the coordinate output stream in a reservation buffer. If the upcoming stop token is at level i, the reserved token is $i+1$. The two segment values are placed in the coordinate state machine. A counter iterates using these values to generate addresses to read from the coordinate array and places them in the coordinate FIFO. These coordinates are sent out of the tile as a coordinate stream, and an address generator generates the corresponding reference stream. In this example (for the highest tensor level), we use a root stream [0, D]. Otherwise, the memory tile receives a reference stream from upstream primitives. The level scanner first grabs the segment tuple (in this example (0, 3)) from the level buffer to determine the fiber length (3). Then the fiber length is used to grab the coordinates [0, 1, 3, S_0, D] from the level buffer and produce the coordinate stream. An address generator produces the corresponding reference for each coordinate to create the reference stream [0, 1, 2, S_0, D].

C. Processing Element Tile Primitives

The intersecter/unioner, reducer, repeater, and coordinate dropper primitives are added to the processing element tile. They perform computation on coordinates, references, and values to eliminate ineffectual compute. Figure 5 shows the implementation of each primitive, along with example input and output streams. The intersecter/unioner calculates the intersection/union of coordinate streams of two tensors to produce only non-zero output indices. The intersecter performs a two-way merge and only emits coordinates and references of both streams on equivalent coordinates, dequeuing the FIFO with the lower value. The unioner uses the same hardware but collects and outputs all coordinates (merging duplicate coordinates), also dequeuing the FIFO with the lower value. While complete, these primitives may produce empty fibers. The coordinate dropper removes the coordinates associated with empty fibers by registering, looking for, and dropping contiguous stop tokens in the inner coordinate stream and its corresponding outer coordinate as well. The reducer performs a sum over a stream, outputting a single value, and the repeater duplicates streams for tensor broadcasting.

D. Ready-Valid Interconnect

The primitives described above produce data-dependent access patterns and have non-deterministic latency. We add ready-valid capability and FIFOs utilizing the hardware generation features described in [14] to the baseline architecture described in [10] to support these attributes. Naively, we could replace pipeline registers with two-element FIFOs. However, we avoid doubling the number of registers by designing a "SplitFIFO" composed of pipeline registers in adjacent tiles sharing control signals, which saves us 13% in interconnect area. Next, we add a bit to the 16-bit interconnect to designate the special tokens described above (e.g. stop, done). Finally, in our sparse application graphs, a producer node may broadcast to several consumer nodes. Mapped to our hardware, this is a producer FIFO sending data to an arbitrary number of consumer FIFOs on the interconnect. If any of the consumer FIFOs are not ready, we must prevent the producer FIFO from outputting valid data. To accomplish this, our CGRA performs an `and`, in SBs at broadcast points, on all incoming ready signals to generate an `enable` for the producer FIFO.

IV. Sparse Compiler

Our sparse application compiler leverages the Custard compiler from [16]. Custard takes as input three high-level domain-specific languages (DSLs) that describe the algorithm (or expression) [19], compression format [20], and schedule [21] of the sparse tensor algebra application and produces SAM dataflow graphs. The contribution of this work is creating an end-to-end compilation path from these high-level languages to hardware. We achieve this by designing a new back-end compilation path from SAM to the CGRA bitstream (Figure 6) through an Onyx-aware sparse dataflow graph intermediate representation (IR). Our compiler maps the sparse primitives from this hardware-aware dataflow IR onto Onyx's tiled architecture, places and routes (PnRs) the application [22] with a pipelining algorithm introduced in Section IV-E and generates the CGRA's configuration bitstream. Finally, it tiles the input data as described in Section IV-C, and pushes it through the configured CGRA to execute the application.

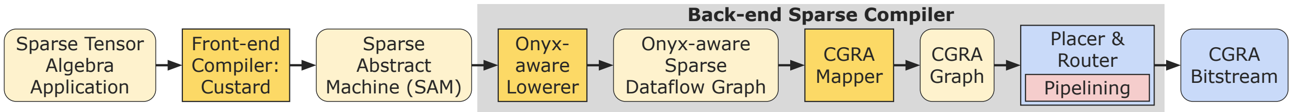

Fig. 6: Our end-to-end sparse compiler that starts with a sparse tensor algebra application (shown in Figure 7) and outputs a CGRA bitstream.

```
// Format language:
// comprised of level formats and mode orderings
Format dcsr({sparse, sparse}, {0, 1});
Format dcsc({sparse, sparse}, {1, 0});

// Declare input and output tensors
Tensor<int> X({I,J}, dcsr);
Tensor<int> B({I,K}, dcsr);
Tensor<int> C({K,J}, dcsc);

// Define the SpMSpM expression (algorithm)
IndexVar i, j, k;
X(i, j) = B(i, k) * C(k, j);

// Scheduling language: inner-product dataflow
IndexStmt stmt = A.getAssignment();
stmt = stmt.reorder({i, j, k})
```

Fig. 7: Input code to Onyx's sparse compiler for SpMSpM.

A. Running Example for a Sparse Application

We will use sparse-matrix sparse-matrix multiplication (SpMSpM) as a running example to demonstrate how the primitives in Onyx compose to implement arbitrary sparse tensor algebra expressions. SpMSpM is represented in Einsum notation as $\mathbf{X}_{ij} = \sum_k \mathbf{B}_{ik}\mathbf{C}_{kj}$, and in our example $\mathbf{X}$ and $\mathbf{B}$ are stored in doubly compressed sparse row (DCSR) format and $\mathbf{C}$ is stored in doubly compressed sparse column (DCSC) format for the inner-product dataflow ($i \rightarrow j \rightarrow k$ order). Custard compiles the input program for this example (shown in Figure 7) to a SAM dataflow graph.

Figure 8 shows how Onyx computes SpMSpM. Intermediate reference (ref) and coordinate (coord) streams are shown at each step of the SAM dataflow 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 output values to memory. Similarly, using our composable primitives, we can accelerate any sparse tensor algebra expression on our CGRA.

B. Onyx-aware Lowering

The lowerer performs dataflow graph rewrites and lowers SAM to an Onyx-aware IR (Figure 9). This IR is similar to SAM but defines type constraints on the neighbors of nodes and details wiring information between those neighbors.

First, we perform primitive remapping, performing automatic transformations for joiner primitives, memory tile primitives, and stream broadcasting. The lowerer transforms any N-ary SAM joiner into a tree of multiple binary joiners, supported by Onyx hardware. Next, since our sparse memory tile contains a level scanner, buffer, and writer, the lowerer transforms individual level scanners and level writers in the SAM graph to level scanner, buffer, writer primitive triplets. Finally, the lowerer also replaces all stream broadcasting nodes in SAM with multiple point-to-point primitive connections since the CGRA interconnect and router support broadcasting to multiple wires by default.

The Onyx-aware lowerer also performs wire elaboration. Each node in the Onyx-aware IR is similar to a SAM primitive but includes a list of valid neighboring nodes and the elaborated signals between those neighboring node types. As shown in Figure 9, the abstract connections in SAM are elaborated to multiple actual wires each with a specified bit-width (i.e. one-bit ready and valid, and multi-bit data).

C. Sparse Data Tiling

Our sparse compiler automatically generates tiled data that fits within the global buffer and sub-tiles that fit within the memory tile (Figure 10). For each sparse tensor expression written in Custard, the application compiler produces CPU application code to generate these tiles and sub-tiles for each dataset run. The sparse data tiling flow also handles the generation of tiles and sub-tiles for unrolled execution.

For performance and efficiency, we skip over empty tiles, and do not send them to Onyx. We accomplish this tiling by

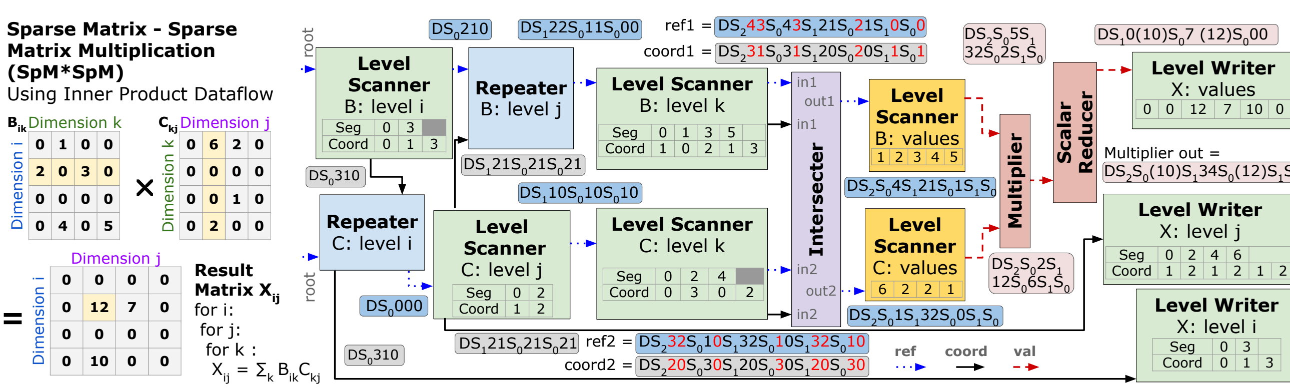

Fig. 8: SpMSpM ($X_{ij} = \sum_k B_{ik}C_{kj}$) SAM graph annotated with streams.

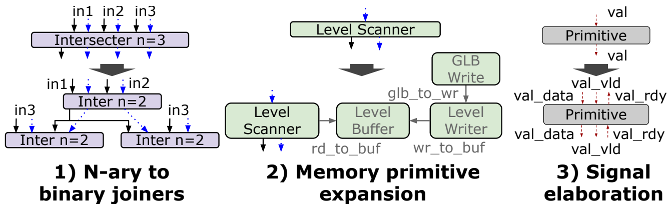

Fig. 9: Some of the transformations performed by the Onyx-aware lowering step in our back-end sparse compiler.

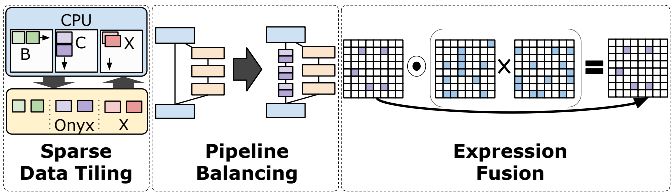

Fig. 10: Software optimizations in sparse application flow.

generating code to create $N \times N$ tiles and $M \times M$ sub-tiles, both tiled in coordinate space. The generated code produces tuples of sub-tiles within a tile, matching on shared dimensions. Sub-tile matching is different for each expression and dataset since the code compresses away empty sub-tiles and the shared dimensions in the input tensors depend upon Custard's input expression. For example, in matrix multiplication, sub-tiles for $\mathbf{B}$ and $\mathbf{C}$ are paired along the k dimension. During evaluation and testing, we sweep over N and M, ensuring that data does not overflow the global buffer or memory tile capacities.

D. Fusion

Onyx's generality supports higher-order tensors, multiple inputs, and, importantly, expression fusion, which eliminates the materialization of entire temporary tensors and ineffectual computation across all inputs. To illustrate fusion across multiple inputs, we extend our matrix multiplication example to a sampled matrix multiplication $X_{jl} = \sum_k B_{jl} C_{jk} D_{kl}$. Sparse tensor accelerators that only support unfused two input expressions (both fixed-function and reconfigurable) [23]–[27], would compute the sampled matrix multiplication as unfused expressions $T_{jl} = \sum_k C_{jk} D_{kl}$ and $X_{jl} = B_{jl} T_{jl}$. In the unfused case, the entire temporary tensor T_{jl} is materialized in memory. Additionally, this schedule may materialize nonzero elements that are not required in the final result tensor as shown in Figure 10. Our sparse application compiler can compose sparse memory and PE tiles to compute such multi-input expressions in a single accelerator call on the CGRA.

E. Pipelining

Our CGRA has the ability to register data on each hop of the interconnect to ensure short critical paths. To achieve maximum application frequencies, we exhaustively pipeline all routes on our CGRA with the SplitFIFOs described in Section III. However, exhaustively pipelining a fixed route does not take into account uneven paths in SAM graphs that can cause backpressure, which in turn causes bubbles in application execution. Therefore, we apply a balancing algorithm (Figure 10) before tile placement, which evens out these paths. Finally, the compiler places the balanced graph on the accelerator and performs exhaustive pipelining to ensure high application frequencies.

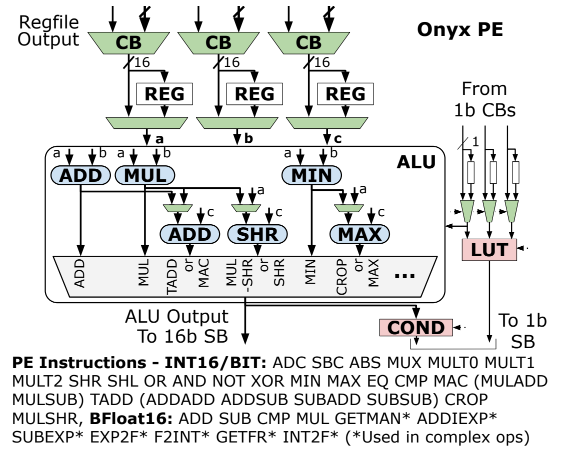

Fig. 11: Onyx PE with mul-add, add-add, mul-shift, and min-max.

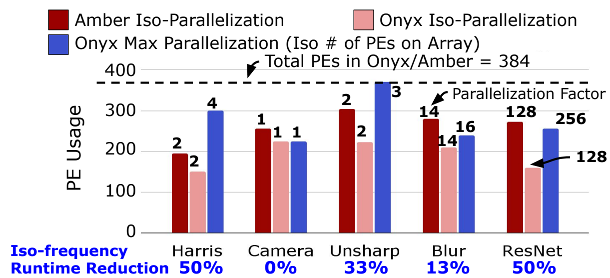

Fig. 12: Effect of PE specialization on array utilization.

V. Dense Optimizations

We use the CGRA from [10] as a baseline CGRA for dense image processing and machine learning applications. By increasing our compute density and improving our efficiency across the memory hierarchy we are more performant and energy efficient than the prior state-of-the-art CGRA.

A. PE Specialization

We increased the complexity of the computation done in the PE and specialized it to best fit the targeted application domain while still maintaining the flexibility to execute several applications. We used an automated application domain-driven PE specialization methodology [28] to optimize the PEs for a set of ML and image processing applications. We take dataflow graphs of our applications and perform subgraph mining to extract the most frequent computational subgraphs. We then perform an area-optimal merging of the most frequent subgraphs and add them to a baseline PE that contains a set of general purpose operations. The resulting Onyx PE is shown in Figure 11. The PE performs either single or multiple (e.g., multiply-add, add-add, multiply-shift, min-max) Int16 operations, or a single BFloat16 operation. With this specialization, we achieve a $12.5 - 41\%$ reduction in the number PEs used vs. Amber [10] for the same application parallelization. This allows us to further parallelize the applications, leading to up to 50% decrease in the number of execution cycles.

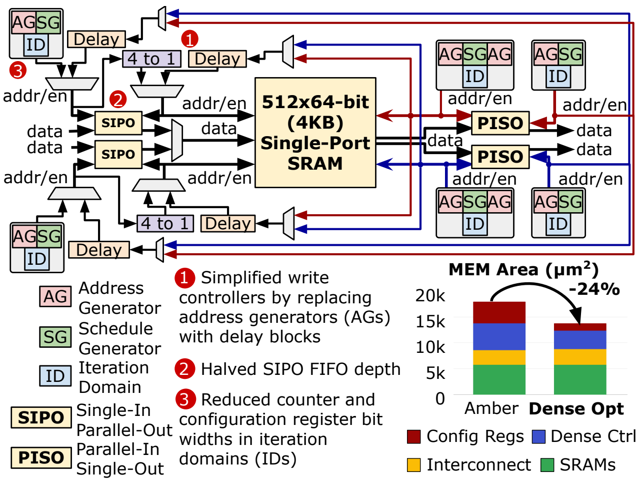

Fig. 13: Dense memory controller optimizations and area reduction.

B. Memory Tile

We optimized our memory controllers for dense applications, as shown in Figure 13. In the baseline memory tile [10], each controller is composed of an iteration domain (ID), an address generator (AG), and a schedule generator (SG), which generate an affine access pattern at each memory port [29]. Our optimizations consist of (1) simplifying the write port controller, (2) reducing the depth of the serial-in, parallel-out (SIPO) buffer and (3) reducing the bit widths of counters and configuration registers. We could simplify the write controller because complex write address generation is only needed in update (read-modify-write) operations, in which case we can reuse the addresses generated by the read controller by adding a delay block. To justify reducing the SIPO buffer and bit widths of counters and configuration registers, we analyzed the maximum SIPO depth and number of bits needed in our dense applications and removed unnecessary hardware. These optimizations reduce the area of the memory tile by 24%.

C. Global Buffer

In a CGRA, it is critical that unused elements dissipate as little power as possible since the number of active elements varies widely between applications. Given the large idle energy of memories, memories should be clocked only while being used. However, clock gating the GLB requires special consideration because some blocks may be gated statically while others' status are only known at runtime. Sending data from the CPU or off-chip to the GLB requires dynamic gating, whereas for a given application, load, store, and dynamic partial reconfiguration (DPR) usage is known ahead of time and can be statically gated. We implement a hybrid static/dynamic gating scheme for the different GLB blocks as shown in Figure 14. The GLB tile is composed of two banks, a bank mux, store (ST), load (LD), and DPR DMAs, and ring, AXI, and configuration switches. On the workload shown that double buffers data into and out of the CGRA, Tile 1 receives a processor write from the subsystem and loads data into the CGRA. The LD DMA can be statically clock gated because the schedule is known, whereas the banks and AXI switch are dynamically clock gated. Tile 2 receives

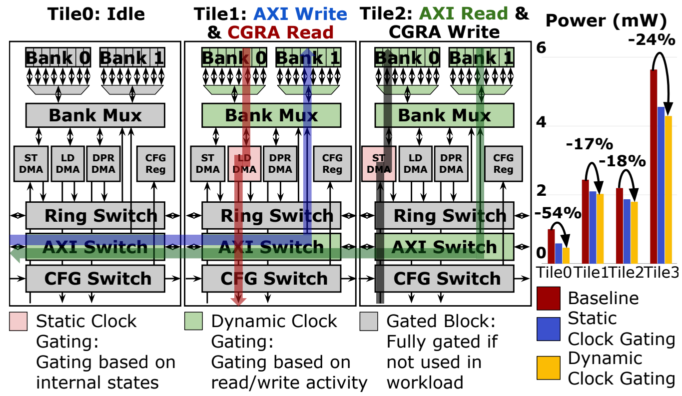

Fig. 14: Static/dynamic clock gating in the global buffer.

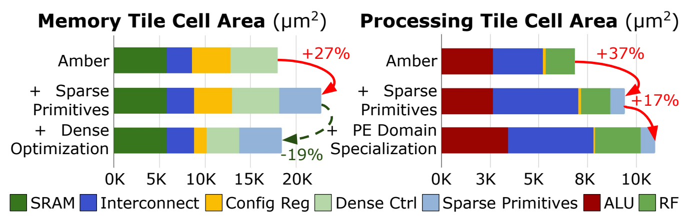

Fig. 15: Memory and PE tile synthesis areas showing the changes after optimizations and added features.

a processor read request and data from the CGRA. In Tile 2, the ST DMA can be statically clock gated but the banks and AXI switch are dynamically gated. In this example, our gating scheme achieves a 24% reduction in GLB power.

VI. Results

Onyx is fabricated in GlobalFoundries 12 nm technology. Figure 1 shows an annotated die photo. We use an I/O voltage of 1.8 V and a core voltage of 0.78 V unless otherwise noted. Onyx reaches a maximum frequency of 500 MHz for the processor subsystem and 970 MHz for the CGRA, and achieves a peak performance of 571 INT16 GOPS and peak energy efficiency of 756 GOPS/W.

A. Area Breakdown

The layout area of Onyx is 23.00 mm^2. The global buffer occupies 4.45 mm^2, with each GLB tile occupying 0.249 mm^2. The CGRA layout area is 8.28 mm^2, where each PE tile is 0.011 mm^2 and each memory tile is 0.023 mm^2 in area. Figure 15 shows the cell areas of the PE and memory tiles in Amber [10] and Onyx. To understand the changes in the area vs. Amber, we look at unflattened synthesis area results. Adding sparse primitives to the memory tile increases the memory tile size by 27%. The memory tile optimizations described in Section V-B decrease the area by 19%. For the PE tile, adding sparse primitives increases the area by 37% with most of the area increase coming from the interconnect. Adding application specialization further increases the PE tile area by 17%, but allows us to achieve higher application unrolling as discussed in Section V-A.

B. Dense Application Results

1) Image Processing and Convolutional Layers: We use the Halide compiler [30] [13] to map image processing (Gaussian

TABLE I: Runtime, energy, and accuracy results for CNN models on 1,000 ImageNet samples.

Model	Formats	Runtime (ms/frame)	Energy @ 0.74V (mJ/frame)	Accuracy (Drop)
MCUNet	BFloat16 (activations, weights, accumulation)	92.79	8.52	68.4% (-0% vs. [35])
MobileNetV2	BFloat16 (activations, weights, accumulation)	72.80	10.28	72.2% (+0.3% vs. [36])
ResNet-18	INT8 (activations & weights), INT16 (accumulation)	88.67	20.82	68.9% (-0.86% vs. [37])

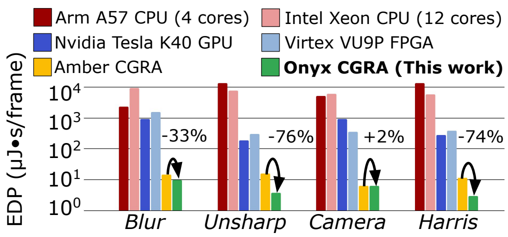

Fig. 16: EDP compared with CPU, GPU, FPGA, and a state-of-the-art CGRA for image processing and computer vision applications.

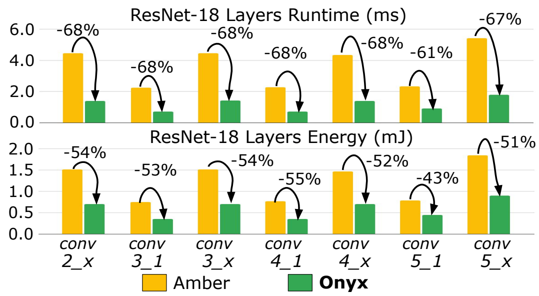

Fig. 17: Runtime and energy compared with state-of-the-art CGRA for ResNet convolutional layers.

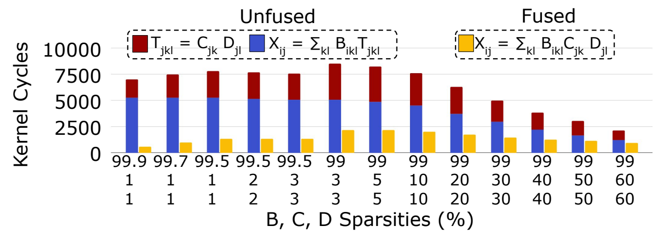

Fig. 18: MTTKRP fusion runtime results for different input sparsities.

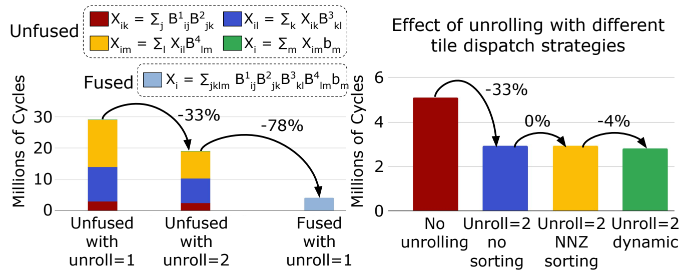

Fig. 19: Left: Iterative matrix-matrix/-vector multiplication fusion results on SuiteSparse matrices (*qiulp, west2021, watt_2, tols2000, tols2000-vectorized*) truncated to a shared dimension of 2100. Right: SpMSpM runtime with three unrolling strategies: no unrolling, NNZ sorting, and dynamic dispatching.

blur, unsharp masking, Harris corner detection, and camera pipeline) and machine learning applications to our CGRA. We compare our image processing results against an ARM Cortex A57 CPU, an Intel Xeon CPU, an NVIDIA Tesla K40 GPU, a Xilinx Virtex Ultrascale FPGA and the Amber CGRA [10]. With the improvements described in Section V, we achieve up to a 76% energy-delay product (EDP) reduction vs. Amber on image processing applications (Figure 16). For convolutional layers, we achieve 61–68% lower runtime and 43–55% lower energy vs. Amber (Figure 17).

2) Performance of CNN Models for Image Classification: We implement MCUNet [31], MobileNetV2 [32] and ResNet-18 [33] for image classification tasks on the ImageNet [34] dataset. For MCUNet, we perform end-to-end inference, storing all intermediate activations in the global buffer. For MobileNetV2 and ResNet-18, we perform layer-by-layer inference due to on-chip memory constraints. To enable faster evaluation, we analytically select a subset of 1,000 samples from the 50,000 validation images, ensuring that the baseline accuracy on the GPU matches the reference. For ResNet-18, we applied post-training quantization to convert weights and activations from FP32 to INT8. Scaling factors were constrained to powers of two, enabling efficient quantization and dequantization via bit shifting. Table I presents the data formats, runtime, energy, and accuracy for each model on the Onyx chip. The accuracy drop compared to the reference is shown in parentheses and is negligible for all three models.

C. Sparse Application Results

We use the sparse application compiler described in Section IV to map sparse expressions onto our accelerator. We evaluate 2D sparse expressions on matrices from the SuiteSparse [38] dataset (99.0-99.95% sparsity, 1813×1813-2021×2021 dimensions) or generated uniform random sparse matrices. For expressions with 3D tensors, we generate uniform random tensors (67% sparsity, 8×37×10-28×35×54 dimensions), unless otherwise noted. This section first details the performance impact of the compiler optimizations introduced in Section IV and then shows our final sparse application results across several kernels.

1) Compiler Optimizations: First, we start with expression fusion. As described in Section IV-D, we utilize fusion to eliminate temporary tensors and ineffectual compute across multi-input expressions. Figure 18 shows the runtime of unfused graphs versus a fused graph for matricized-tensor times Khatri-Rao product (MTTKRP) with different input sparsities (on 10×10×10 and 10×10 tensors). As B gets sparser and C and D get denser, the speedup increases up to 11.8×. Figure 19 (left) shows the runtime for a four-iteration iterative sparse matrix-vector multiplication. Since unfused kernels have fewer inputs, we can benefit from unrolling. Even with this optimization, the fused kernel performs 4.6× faster than the unfused kernels.

Application pipelining is performed as described in Section IV-E. Figure 21 shows matrix-multiplication runtime results with no pipelining, when only exhaustive pipelining is applied, and when balancing is applied before exhaustive pipelining. Balancing ensures that there are no imbalanced paths in the mapped application graph and exhaustive pipelining ensures high application frequency.

Since our CGRA is homogeneous, we can *unroll* or par-

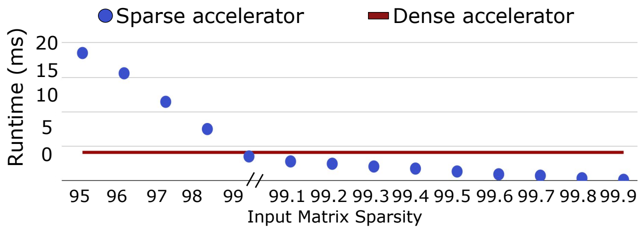

Fig. 20: Sparse accelerator vs. dense accelerator runtime results on a 512×512 matrix multiplication of varying sparsity.

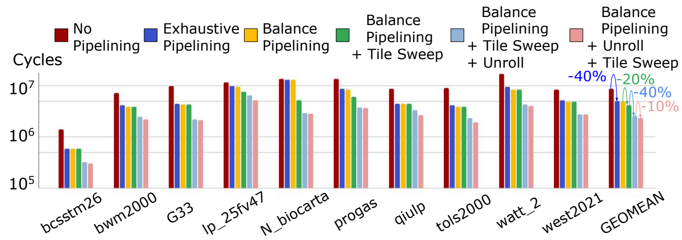

Fig. 21: SpMSpM performance improvement results after applying exhaustive pipelining, balance pipelining, a tile sweep, and unrolling.

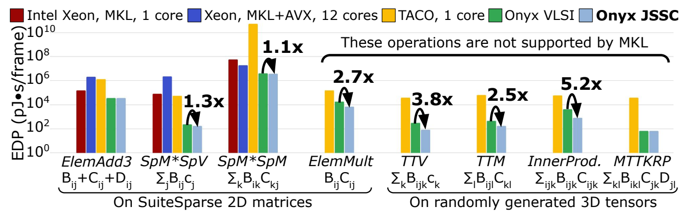

Fig. 22: EDP compared with CPUs with sparse acceleration libraries for a variety of sparse 2D and 3D kernels.

allelize a bitstream over the array to improve performance. Unfortunately, feeding tiles into two parallel matrix multiplications in order can lead to load imbalance and result in suboptimal performance. In Figure 19 (right), we compare the results for three tile dispatch strategies. The first approach is in order and is the baseline. The second strategy sorts the tile pairs based on the sum of the number of non-zeros in the two inputs. Finally, the third approach dynamically dispatches data whenever a region of the accelerator is finished. This requires duplicating input data in our global buffer, but achieves the highest performance, $1.76\times$ faster than no unrolling. Figure 21 shows a $1.6\times$ improvement in performance for sparse matrix multiplication with unrolling.

As described in Section IV-C, tiling can have a significant effect on performance. Smaller tiles perform less computation on the array but are faster and reduce the ratio of kernel acceleration to the processor overhead of reconfiguring the global buffer. Figure 21 shows our most performant results when we run an exhaustive tiling sweep before and after unrolling the application. Performing the tiling sweep after unrolling is slightly better since the ratio of kernel time to processor time changes with unrolling.

2) Sparse vs. Dense Matrix Multiplication: For a 512×512 matrix multiplication with varying input sparsity, we compare our dense accelerator configuration versus its sparse counterpart in Figure 20. After 99% matrix sparsity, we should configure our accelerator as a sparse accelerator. At 99.9% input sparsity, we perform $30\times$ better than the dense accelerator configuration. Future work includes improving primitive performance, supporting other dataflows (for example, Gustavson's algorithm [39]), and adding new primitives to increase accelerator utilization. With these improvements, the sparse accelerator will perform better on less sparse matrices.

3) Final Sparse Results: We evaluate our sparse tensor algebra kernels on EDP versus a CPU (12-core Intel Xeon) using sparse libraries (Math Kernel Library (MKL) [40] and TACO [19]). For single-core MKL baselines, we ran the library with no optimizations enabled. For 12-core MKL+AVX, MKL was configured and run with AVX512 and OMP enabled. For the TACO baseline, we used default schedules for each expression. In all baseline configurations, we take the median runtime across 10 iterations. Utilizing the scheduling optimizations described above, we perform up to $5.2\times$ better in EDP on sparse tensor algebra expressions versus our VLSI submission [41] and $4.4-565\times$ in EDP better versus CPUs with sparse libraries as shown in Figure 22. Onyx excels in expressions with higher-order tensors and multiple inputs, where we can leverage fusion to outperform the baselines.

D. Related Work

Typically, fixed-function accelerators are used to accelerate sparse workloads. For example, Huang et al. [7] target augmented reality with dedicated sparse-CNN and sparse-GCN engines but only support accelerating these two operations. Similarly, Song et al. [42] design several fixed-function accelerators for 3D navigation, including a sparse matrix multiplication engine, but do not accelerate full application domains or exploit sparsity in other kernels.

Several other works focus on accelerating sparse kernels. For example, [5], [43]–[45] optimize matrix-multiplications over different levels of input sparsity, while [3], [46] support operations on high-order tensors such as MTTKRP. Additionally, [4] supports tensor addition, but does not take full advantage of sparse iteration for intersects/unions. Symphony [47] is a sparse architecture that supports all of sparse tensor algebra but has no hardware implementation. As opposed to Onyx, none of these works have silicon results.

On the other hand, there are programmable accelerators that support application domains, such as Feng et al. [10] and Zhang et al. [48]. However, [10] and [48] are limited to dense applications only and achieve lower peak performance. There are several other works that explore CGRA architectures ([11], [49], [50], [8], [9]), but they focus solely on dense applications and only have simulation or FPGA-based implementations.

VII. Conclusion

Onyx is a programmable SoC designed to accelerate both dense and sparse applications. By increasing compute density and optimizing across the memory hierarchy, we achieve efficient performance and efficiency in both image processing and machine learning workloads. With composable memory and compute hardware primitives and an end-to-end sparse compiler with scheduling optimizations, we accelerate any sparse tensor algebra expression. As the first programmable accelerator to support arbitrary dense or sparse applications, Onyx demonstrates an approach to accelerating several *application domains* on a unified hardware fabric.

References

[1] B. Zimmer, R. Venkatesan, Y. S. Shao, J. Clemons, M. Fojtik, N. Jiang, B. Keller, A. Klinefelter, N. Pinckney, P. Raina, S. G. Tell, Y. Zhang, W. J. Dally, J. S. Emer, C. T. Gray, S. W. Keckler, and B. Khailany, "A 0.32–128 TOPS, scalable multi-chip-module-based deep neural network inference accelerator with ground-referenced signaling in 16 nm," *IEEE Journal of Solid-State Circuits*, vol. 55, no. 4, pp. 920–932, 2020.

[2] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2016.

[3] K. Hegde, H. Asghari-Moghaddam, M. Pellauer, N. Crago, A. Jaleel, E. Solomonik, J. Emer, and C. W. Fletcher, "ExTensor: An accelerator for sparse tensor algebra," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 319–333.

[4] A. Rucker, M. Vilim, T. Zhao, Y. Zhang, R. Prabhakar, and K. Olukotun, "Capstan: A vector RDA for sparsity," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1022–1035. [Online]. Available: https://doi.org/10.1145/3466752.3480047

[5] Y. Yang, J. S. Emer, and D. Sanchez, "Trapezoid: A versatile accelerator for dense and sparse matrix multiplications," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, 2024, pp. 931–945.

[6] S. Dave, R. Baghdadi, T. Nowatzki, S. Avancha, A. Shrivastava, and B. Li, "Hardware acceleration of sparse and irregular tensor computations of ML models: A survey and insights," *Proceedings of the IEEE*, vol. 109, no. 10, pp. 1706–1752, 2021.

[7] 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 *2019 Symposium on VLSI Circuits*, 2022, pp. 42–43.

[8] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim, "DySER: Unifying functionality and parallelism specialization for energy-efficient computing," *IEEE Micro*, vol. 32, no. 5, pp. 38–51, 2012.

[9] F.-J. Veredas, M. Scheppler, W. Moffat, and B. Mei, "Custom implementation of the coarse-grained reconfigurable ADRES architecture for multimedia purposes," in *International Conference on Field Programmable Logic and Applications*, 2005, pp. 106–111.

[10] 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 Journal of Solid-State Circuits*, pp. 1–13, 2023.

[11] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, "Plasticine: A reconfigurable architecture for parallel paterns," *SIGARCH Comput. Archit. News*, vol. 45, no. 2, p. 389–402, Jun. 2017. [Online]. Available: https://doi.org/10.1145/3140659.3080256

[12] "ARM Cortex-M3," https://developer.arm.com/Processors/Cortex-M3.

[13] K. Koul, J. Melchert, K. Sreedhar, L. Truong, G. Nyengele, K. Zhang, Q. Liu, J. Setter, P.-H. Chen, Y. Mei, M. Strange, R. Daly, C. Donovick, A. Carsello, T. Kong, K. Feng, D. Huff, A. Nayak, R. Setaluri, J. Thomas, N. Bhagdikar, D. Durst, Z. Myers, N. Tsiskaridze, S. Richardson, R. Bahr, K. Fatahalian, P. Hanrahan, C. Barrett, M. Horowitz, C. Torng, F. Kjolstad, and P. Raina, "AHA: An agile approach to the design of coarse-grained reconfigurable accelerators and compilers," *ACM Trans. Embed. Comput. Syst.*, vol. 22, no. 2, Jan 2023. [Online]. Available: https://doi.org/10.1145/3534933

[14] J. Melchert, K. Zhang, Y. Mei, M. Horowitz, C. Torng, and P. Raina, "Canal: A flexible interconnect generator for coarse-grained reconfigurable arrays," *IEEE Computer Architecture Letters*, vol. 22, no. 1, pp. 45–48, 2023.

[15] T. Kong, K. Koul, P. Raina, M. Horowitz, and C. Torng, "Hardware abstractions and hardware mechanisms to support multi-task execution on coarse-grained reconfigurable arrays," 2023. [Online]. Available: https://arxiv.org/abs/2301.00861

[16] 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

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

[18] M. Pellauer, Y. S. Shao, J. Clemons, N. Crago, K. Hegde, R. Venkatesan, S. W. Keckler, C. W. Fletcher, and J. Emer, "Buffets: An efficient and composable storage idiom for explicit decoupled data orchestration," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 137–151. [Online]. Available: https://doi.org/10.1145/3297858.3304025

[19] 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.

[20] 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.

[21] R. Senanayake, C. Hong, Z. Wang, A. Wilson, S. Chou, S. Kamil, S. Amarasinghe, and F. Kjolstad, "A sparse iteration space transformation framework for sparse tensor algebra," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Nov. 2020. [Online]. Available: https://doi-org.stanford.idm.oclc.org/10.1145/3428226

[22] J. Melchert, Y. Mei, K. Koul, Q. Liu, M. Horowitz, and P. Raina, "Cascade: An application pipelining toolkit for coarse-grained reconfigurable arrays," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2024.

[23] G. Zhang, N. Attaluri, J. S. Emer, and D. Sanchez, "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*, 2021, pp. 687–701.

[24] S. Pal, J. Beaumont, D.-H. Park, A. Amarnath, S. Feng, C. Chakrabarti, H.-S. Kim, D. Blaauw, T. Mudge, and R. Dreslinski, "OuterSPACE: An outer product based sparse matrix multiplication accelerator," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 724–736.

[25] Z. Zhang, H. Wang, S. Han, and W. J. Dally, "SpArch: Efficient architecture for sparse matrix multiplication," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 261–274.

[26] Z. Li, J. Li, T. Chen, D. Niu, H. Zheng, Y. Xie, and M. Gao, "Spada: Accelerating sparse matrix multiplication with adaptive dataflow," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 747–761. [Online]. Available: https://doi.org/10.1145/3575693.3575706

[27] L. Song, Y. Chi, A. Sohrabizadeh, Y.-k. Choi, J. Lau, and J. Cong, "Sextans: A streaming accelerator for general-purpose sparse-matrix dense-matrix multiplication," in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 65–77. [Online]. Available: https://doi.org/10.1145/3490422.3502357

[28] J. Melchert, K. Feng, C. Donovick, R. Daly, R. Sharma, C. Barrett, M. A. Horowitz, P. Hanrahan, and P. Raina, "APEX: A framework for automated processing element design space exploration using frequent subgraph analysis," 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. 33–45. [Online]. Available: https://doi.org/10.1145/3582016.3582070

[29] Q. Liu, J. Setter, D. Huff, M. Strange, K. Feng, M. Horowitz, P. Raina, and F. Kjolstad, "Unified buffer: Compiling image processing and machine learning applications to push-memory accelerators," *ACM Trans. Archit. Code Optim.*, vol. 20, no. 2, Mar 2023. [Online]. Available: https://doi.org/10.1145/3572908

[30] 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

[31] J. Lin, W.-M. Chen, Y. Lin, C. Gan, and S. Han, "MCUNet: Tiny deep

learning on IoT devices," *Advances in Neural Information Processing Systems*, vol. 33, 2020.

[32] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 4510–4520.

[33] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[34] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255.

[35] H. Lab, "MCUNet: Tiny deep learning on microcontrollers," GitHub repository. Available: https://github.com/mit-han-lab/mcunet, accessed: November 15, 2024.

[36] PyTorch Team, "PyTorch hub: MobileNet V2," https://pytorch.org/hub/pytorch_vision_mobilenet_v2/, accessed: November 15, 2024.

[37] TorchVision Contributors, "ResNet-18 pretrained model on ImageNet," https://pytorch.org/vision/main/models/generated/torchvision.models.resnet18.html, accessed: November 15, 2024.

[38] 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.

[39] F. G. Gustavson, "Two fast algorithms for sparse matrices: Multiplication and permuted transposition," *ACM Trans. Math. Softw.*, vol. 4, no. 3, 1978.

[40] I. Corporation, *Intel Math Kernel Library. Reference Manual*. Santa Clara: Intel Corporation, 2009.

[41] 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 *2024 IEEE Symposium on VLSI Technology and Circuits (VLSI Technology and Circuits)*, 2024, pp. 1–2.

[42] S. Song, D. Han, S. Kim, S. Kim, G. Park, and H.-J. Yoo, "GPPU: A 330.4-μJ/task neural path planning processor with hybrid GNN acceleration for autonomous 3D navigation," in *2023 IEEE Symposium on VLSI Technology and Circuits*, 2023, pp. 1–2.

[43] V. Dadu, J. Weng, S. Liu, and T. Nowatzki, "Towards general purpose acceleration by exploiting common data-dependence forms," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 924–939.

[44] K. Zhong, Z. Zhu, G. Dai, H. Wang, X. Yang, H. Zhang, J. Si, Q. Mao, S. Zeng, K. Hong, G. Zhang, H. Yang, and Y. Wang, "FEASTA: A flexible and efficient accelerator for sparse tensor algebra in machine learning," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 349–366. [Online]. Available: https://doi.org/10.1145/3620666.3651336

[45] L. Lu, Y. Jin, H. Bi, Z. Luo, P. Li, T. Wang, and Y. Liang, "Sanger: A co-design framework for enabling sparse attention using reconfigurable architecture," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 977–991. [Online]. Available: https://doi.org/10.1145/3466752.3480125

[46] N. Srivastava, H. Jin, S. Smith, H. Rong, D. Albonesi, and Z. Zhang, "Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 689–702.

[47] M. Pellauer, J. Clemons, V. Balaji, N. Crago, A. Jaleel, D. Lee, M. O'Connor, A. Parashar, S. Treichler, P.-A. Tsai, S. W. Keckler, and J. S. Emer, "Symphony: Orchestrating sparse and dense tensors with hierarchical heterogeneous processing," *ACM Trans. Comput. Syst.*, vol. 41, no. 1–4, Dec. 2023. [Online]. Available: https://doi.org/10.1145/3630007

[48] Q. Zhang, H. An, Z. Fan, Z. Wang, Z. Li, G. Wang, H.-S. Kim, D. Blaauw, and D. Sylvester, "A 22nm 3.5TOPS/W flexible micro-robotic vision SoC with 2MB eMRAM for fully-on-chip intelligence," in *2022 IEEE Symposium on VLSI Technology and Circuits*, 2022, pp. 72–73.

[49] C. Torng, P. Pan, Y. Ou, C. Tan, and C. Batten, "Ultra-elastic CGRAs for irregular loop specialization," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 412–425.

[50] S. A. Chin, N. Sakamoto, A. Rui, J. Zhao, J. H. Kim, Y. Hara-Azumi, and J. Anderson, "CGRA-ME: A unified framework for CGRA modelling and exploration," in *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2017, pp. 184–189.