This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

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

Kalhan Koul, *Member, IEEE*, Olivia Hsu, *Member, IEEE*, Yuchen Mei, Sai Gautham Ravipati, Maxwell Strange, Jackson Melchert, *Member, IEEE*, Alex Carsello, Taeyoung Kong, *Member, IEEE*, Po-Han Chen, *Graduate Student Member, IEEE*, Huifeng Ke, Keyi Zhang, Qiaoyi Liu, Gedeon Nyengele, *Member, IEEE*, Zhouhua Xie, Akhilesh Balasingam, Jayashree Adivarahan, Ritvik Sharma, Christopher Torng, Joel S. Emer, *Life Fellow, IEEE*, Fredrik Kjolstad, Mark Horowitz, *Life Fellow, IEEE*, and Priyanka Raina, *Life Fellow, IEEE*

***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 (ML) 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 (PEs), area-optimized memory tiles, and hybrid clock gating in the global buffer (GLB). Onyx achieves a peak energy efficiency of 756 INT16 GOPS/W, up to 565× better energy-delay product (EDP) for sparse kernels versus CPUs with sparse libraries, and up to 76% and 85% lower EDP for image processing and ML, respectively, versus the state-of-the-art CGRA.**

***Index Terms*—Coarse-grained reconfigurable array (CGRA), compilers, computer vision, image processing, machine learning (ML), reconfigurable accelerators, sparse matrices.**

I. Introduction

APPLICATIONS ranging from machine learning (ML) 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, Huang et al. [7] contain 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]. Additionally, many of the hardware accelerators mentioned above propose architectural advances that may or may not include hardware implementations [3], [4], [5] and none are fabricated [11].

Therefore, Onyx overcomes the limitations of prior accelerators by supporting dense and sparse kernels on the same fabricated, programmable accelerator, as shown in Fig. 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 has dedicated support for image processing and ML 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.

Received 10 February 2025; revised 30 June 2025; accepted 19 August 2025. This article was approved by Associate Editor Jie Gu. This work was supported in part by the Defense Advanced Research Projects Agency (DARPA) Domain-Specific System-on-Chip (DSSoC) Grant, in part by Stanford AHA Agile Hardware Center and Affiliates Program, in part by NSF Graduate Research Fellowship Program (GRFP), in part by Stanford SystemX Alliance, in part by the Semiconductor Research Corporation (SRC) Joint University Microelectronics Program (JUMP) 2.0 Center for Processing with Intelligent Storage and Memory (PRISM) Center, in part by the NSF CAREER Award under Grant 2238006, in part by the Intel HIP, and in part by the Samsung and Apple Stanford EE Ph.D. Fellowship. *(Kalhan Koul and Olivia Hsu contributed equally to this work.) (Corresponding author: Kalhan Koul.)*

Kalhan Koul, Olivia Hsu, Yuchen Mei, Sai Gautham Ravipati, Maxwell Strange, Jackson Melchert, Alex Carsello, Taeyoung Kong, Po-Han Chen, Huifeng Ke, Keyi Zhang, Qiaoyi Liu, Gedeon Nyengele, Zhouhua Xie, Akhilesh Balasingam, Jayashree Adivarahan, Ritvik Sharma, Fredrik Kjolstad, Mark Horowitz, and Priyanka Raina are with Stanford University, Stanford, CA 94305 USA (e-mail: kkoul@stanford.edu).

Christopher Torng is with the University of Southern California, Los Angeles, CA 90007 USA.

Joel S. Emer is with Massachusetts Institute of Technology, Cambridge, MA 02139 USA.

Color versions of one or more figures in this article are available at https://doi.org/10.1109/JSSC.2025.3604724.

Digital Object Identifier 10.1109/JSSC.2025.3604724

0018-9200 © 2025 IEEE. All rights reserved, including rights for text and data mining, and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: Stanford University Libraries. Downloaded on February 04,2026 at 15:54:59 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

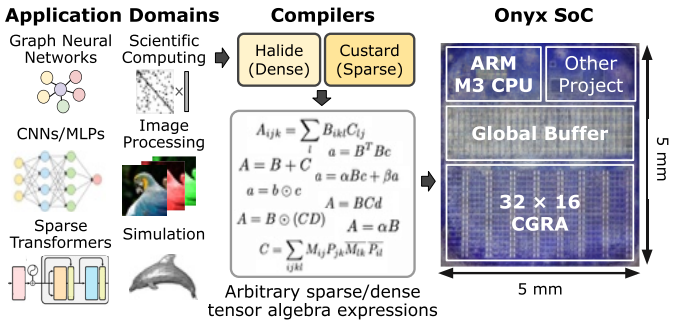


Fig. 1. End-to-end applications are broken down into dense and sparse kernels based on data sparsity. Then, those kernels are mapped onto Onyx, which can accelerate both dense and sparse kernels.

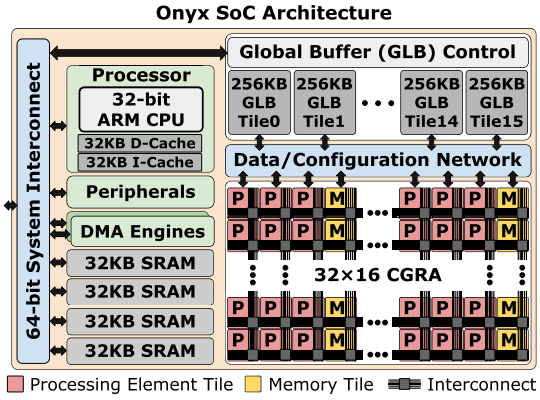


Fig. 2. Onyx SoC architecture. The accelerator contains a CGRA, a GLB, and an ARM M3 control processor.

5) Application-driven processing element (PE) 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× 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 Fig. 2. The CGRA has 384 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. 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 CBs 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].

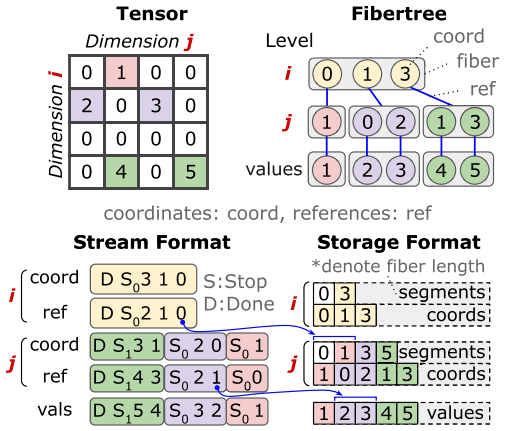


Fig. 3. Tensor in fibertree format, stream format, and storage format.

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. Fig. 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.

Authorized licensed use limited to: Stanford University Libraries. Downloaded on February 04,2026 at 15:54:59 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

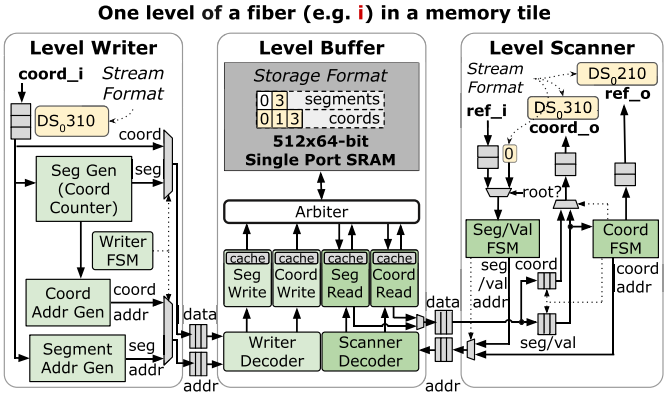


Fig. 4. Memory tile primitives: level writer, level buffer, and level scanner.

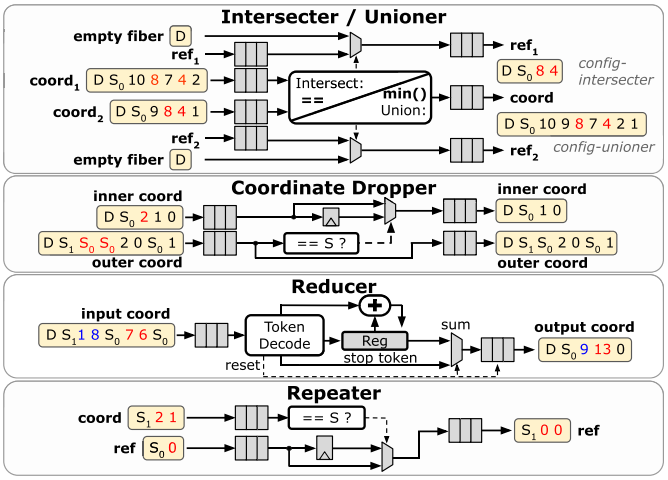


Fig. 5. PE tile primitives: intersecter/unioner, coordinate dropper, reducer, and repeater.

It uses a coordinate address generator (AG) and a segment AG 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 AGs generate addresses to write to the corresponding array. For example, for the stream in Fig. 4, the level writer first forwards the coordinates to write to the coordinate array. Simultaneously, the coordinate counter counts three 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 four-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 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 AG 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 AG produces the corresponding reference for each coordinate to create the reference stream [0, 1, 2, S_0, D].

C. PE Tile Primitives

The intersecter/unioner, reducer, repeater, and coordinate dropper primitives are added to the PE tile. They perform computation on coordinates, references, and values to eliminate ineffectual compute. Fig. 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

Authorized licensed use limited to: Stanford University Libraries. Downloaded on February 04,2026 at 15:54:59 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

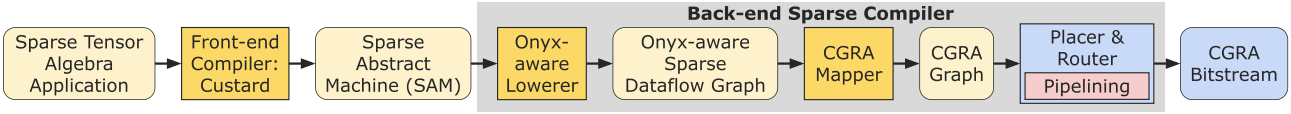


Fig. 6. Our end-to-end sparse compiler that starts with a sparse tensor algebra application (shown in Fig. 7) and outputs a CGRA bitstream.

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 an 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 (Fig. 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 [13], 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.

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 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 Fig. 7) to a SAM dataflow graph.

Fig. 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

```
1  // Format language:
2  // comprised of level formats and mode orderings
3  Format dcsr({sparse, sparse}, {0, 1});
4  Format dcsc({sparse, sparse}, {1, 0});
5
6  // Declare input and output tensors
7  Tensor<int> X({I,J}, dcsr);
8  Tensor<int> B({I,K}, dcsr);
9  Tensor<int> C({K,J}, dcsc);
10
11 // Define the SpMSpM expression (algorithm)
12 IndexVar i, j, k;
13 X(i, j) = B(i, k) * C(k, j);
14
15 // Scheduling language: inner-product dataflow
16 IndexStmt stmt = A.getAssignment();
17 stmt = stmt.reorder({i, j, k})
```

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

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—with some rewrites illustrated in Fig. 9—and lowers SAM to an Onyx-aware IR. This IR is similar to SAM but defines type constraints on the neighbors of nodes and details wiring information between those neighbors.

First, the lowerer performs primitive remapping, which automatically transforms certain primitives—like joiner, memory tile, and stream broadcasting primitives—to more closely resemble the Onyx hardware. For remapping joiner primitives, the lowerer transforms any N-ary SAM joiner into a tree of multiple binary joiners since the Onyx hardware only includes binary joiner implementations. We show an example of this remapping from one trinary intersect joiner to a tree of three binary intersect joiners in Fig. 9(a). Next, the lowerer performs memory primitive expansion to remap memory tile primitives, as shown in Fig. 9(b). In this transformation, the lowerer elaborates individual-level scanners and writers in the SAM graph to level scanner, buffer, and writer primitive triplets. This remapping transformation is necessary because each of the sparse memory tiles in Onyx must sequence reads from and writes to memory, which means that each tile contains a level scanner, buffer, and writer triplet. 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.

Authorized licensed use limited to: Stanford University Libraries. Downloaded on February 04,2026 at 15:54:59 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

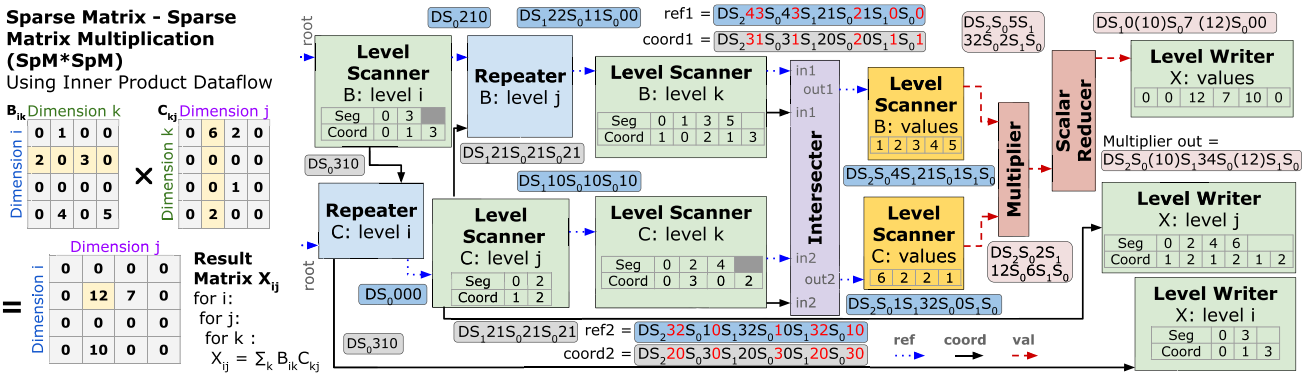


Fig. 8. SpMSpM ($X_{ij} = \sum_k B_{ik}C_{kj}$) SAM graph annotated with streams.

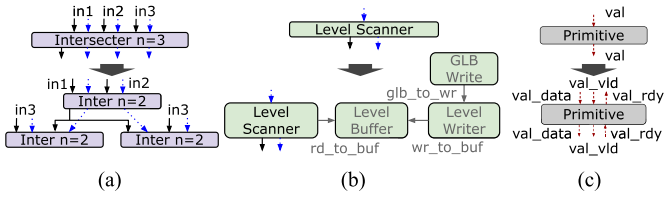


Fig. 9. Some of the transformations performed by the Onyx-aware lowering step in our back-end sparse compiler. (a) N-ary to binary joiners. (b) Memory primitive expansion. (c) Signal elaboration.

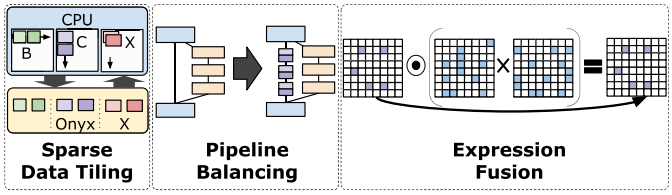


Fig. 10. Software optimizations in sparse application flow.

After all of the primitives are remapped, the Onyx-aware lowerer also performs signal 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 Fig. 9(c), the abstract signal 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). The example in Fig. 9(c) demonstrates how the lowerer would elaborate the single "values" input and "values" output connections of a generic SAM primitive into actual input and output "values" connections that implement ready-valid signaling.

C. Sparse Data Tiling

Our sparse compiler automatically generates tiled data that fits within the GLB and sub-tiles that fit within the memory tile (Fig. 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 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 **B** and **C** are paired along the k dimension. During evaluation and testing, we sweep over N and M, ensuring that data do not overflow the GLB 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], [24], [25], [26], [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 Fig. 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 before tile placement, which evens out these paths. The balancing algorithm (Fig. 10) finds reconverging paths in the dataflow graph, which occur at joiner nodes (the second blue box), and inserts an equivalent amount of SplitFIFOs (purple boxes) on the path with fewer primitives. Finally, the compiler places the balanced graph on the accelerator

Authorized licensed use limited to: Stanford University Libraries. Downloaded on February 04,2026 at 15:54:59 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

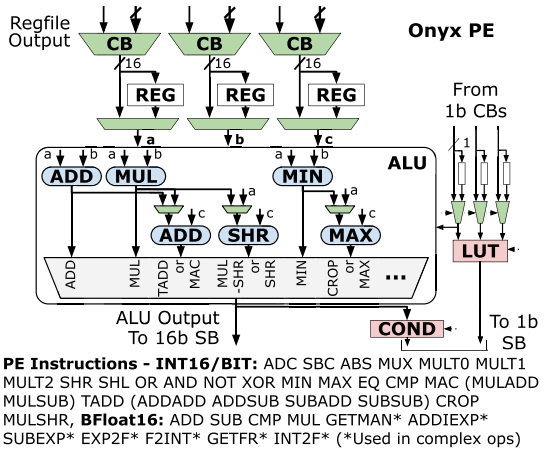


Fig. 11. Onyx PE with mul–add, add–add, mul–shift, and min–max.

and performs exhaustive pipelining to ensure high application frequencies. We leverage prior work [13], [22] for the final steps of our sparse compiler, which include mapping the lowered dataflow graph of the sparse application onto an abstract graph of CGRA nodes, placement and routing onto the CGRA, and bitstream generation.

V. Dense Optimizations

We use the CGRA from [10] as a baseline CGRA for dense image processing and ML 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 Fig. 11. The PE performs either single or multiple (e.g., multiply–add, add–add, multiply–shift, min–max) Int8/Int16 operations or a single BFloat16 operation. With this specialization, we achieve a 12.5%–41% reduction in the number PEs used versus Amber [10] for the same application parallelization. This allows us to further parallelize the applications, leading to up to a 50% decrease in the number of execution cycles (as shown in Fig. 12). Finally, since our PEs are composable, we can support larger integer formats, such as Int32.

B. Memory Tile

We optimized our memory controllers for dense applications, as shown in Fig. 13. In the baseline memory tile [10], each controller is composed of an iteration domain (ID), an

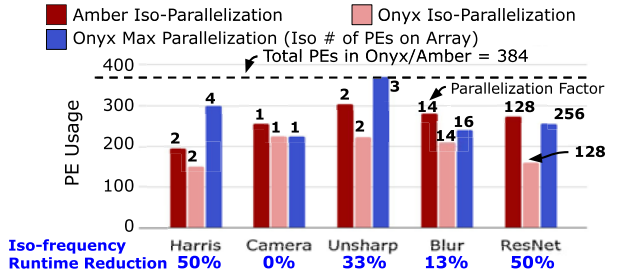


Fig. 12. Effect of PE specialization on array utilization.

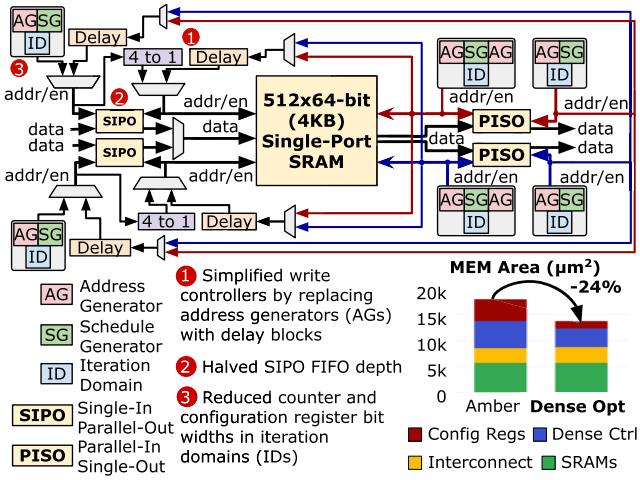


Fig. 13. Dense memory controller optimizations and area reduction.

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 is 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 Fig. 14. The GLB tile is composed of two banks, a bank mux, store (ST),

Authorized licensed use limited to: Stanford University Libraries. Downloaded on February 04,2026 at 15:54:59 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

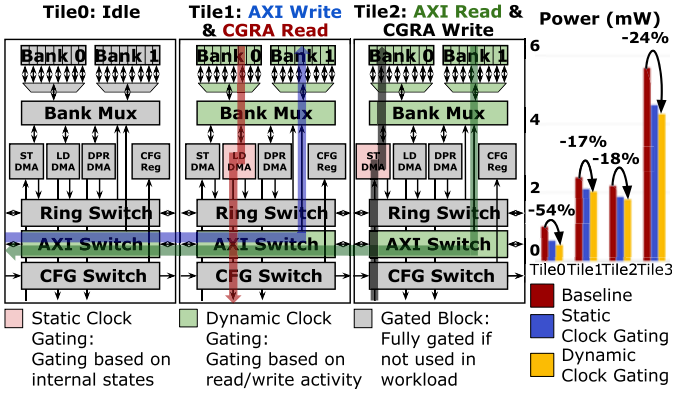


Fig. 14. Static/dynamic clock gating in the GLB.

TABLE I
ONYX SPECIFICATIONS

Technology	GlobalFoundries 12nm FinFet
Project Area	23 mm^2
Processor	ARM-M3 CPU
Voltage	0.78 V
Frequency (Processor)	500 MHz
Frequency (CGRA)	970 MHz
Peak Performance	571 INT16 GOPS
Peak Energy Efficiency	756 GOPS/W

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 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. Fig. 1 shows an annotated die photograph. As shown in Table I, 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. Our results are compared against reported numbers, and we do not normalize for technology differences.

A. Area Breakdown

The layout area breakdown of Onyx is shown in Table II. To understand how these areas compare with prior work, Fig. 15 shows the cell areas of the PE and memory tiles in Amber [10] and Onyx. To understand the changes in the area versus 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

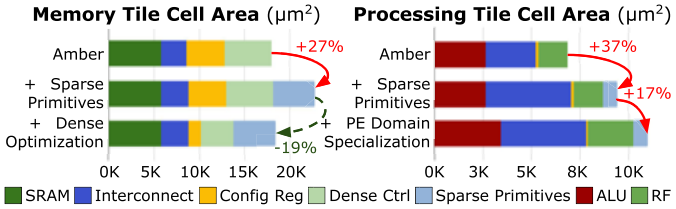


Fig. 15. Memory and PE tile synthesis areas showing the changes after optimizations and added features.

TABLE II
ONYX LAYOUT AREA

Onyx	23.00 mm^2
Global Buffer	4.45 mm^2
GLB Tile (each)	0.249 mm^2
CGRA	8.28 mm^2
PE Tile (each)	0.011 mm^2
MEM Tile (each)	0.023 mm^2

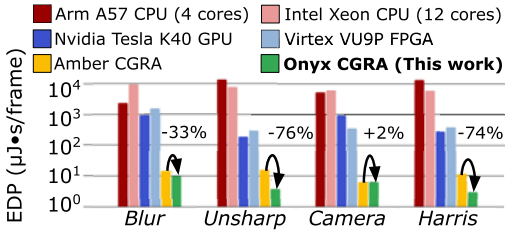


Fig. 16. EDP compared with CPU, GPU, FPGA, and a state-of-the-art CGRA for image processing and computer vision applications.

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 [13], [30] to map image processing (Gaussian blur, unsharp masking, Harris corner detection, and camera pipeline) and ML applications to our CGRA. All images, weights, and outputs are stored on chip. 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% EDP reduction versus Amber on image processing applications (Fig. 16). For convolutional layers, we achieve 61%–68% lower runtime and 43%–55% lower energy versus Amber (Fig. 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 each model, weights are streamed on chip for each individual layer. For MCUNet, we perform end-to-end inference, storing all intermediate activations in the GLB. For MobileNetV2 and ResNet-18, we perform layer-by-layer inference due to on-chip memory constraints. More

Authorized licensed use limited to: Stanford University Libraries. Downloaded on February 04,2026 at 15:54:59 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

TABLE III
RUNTIME, ENERGY, AND ACCURACY RESULTS FOR CNN MODELS ON 1000 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])

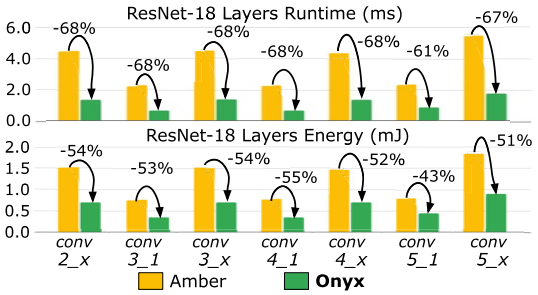


Fig. 17. Runtime and energy compared with the state-of-the-art CGRA for ResNet convolutional layers.

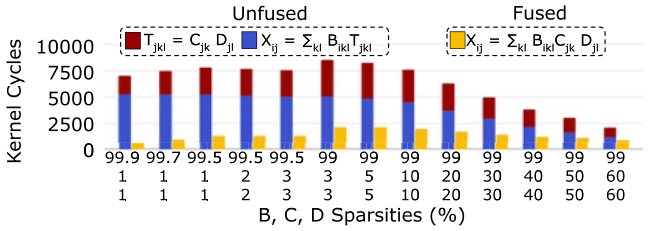


Fig. 18. MTTKRP fusion runtime results for different input sparsities.

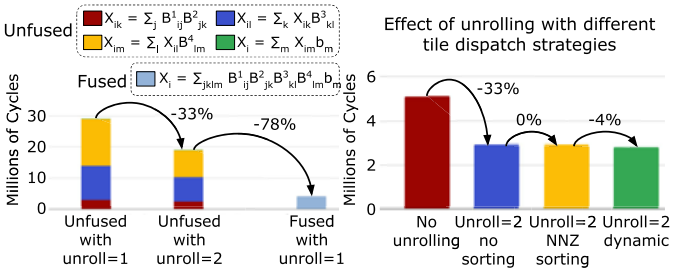


Fig. 19. Left: iterative matrix–matrix/-vector multiplication fusion results on SuiteSparse matrices (*qiulp, west2021, watt_2, tols2000, tols2000-vectorized*) truncated to a shared dimension of 2100. Right: SpMSpM runtime with three unrolling strategies: no unrolling, NNZ sorting, and dynamic dispatching.

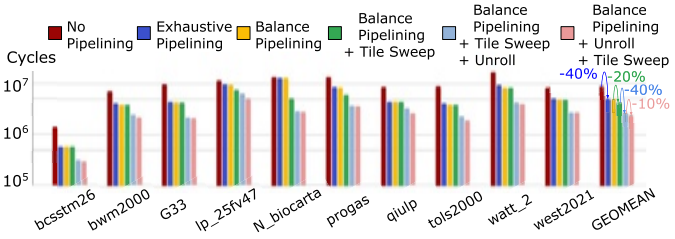


Fig. 20. SpMSpM performance improvement results after applying exhaustive pipelining, balance pipelining, a tile sweep, and unrolling.

specifically, for MobileNetV2 and ResNet-18, we stream output activations off chip, reorganize the data, and then stream them back onto the chip, which allows us to optimally map each layer on the CGRA. Future work will explore accelerating this data reorganization. To enable faster evaluation, we analytically select a subset of 1000 samples from the 50000 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 III 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 2-D 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 3-D tensors, we generate uniform random tensors (67% sparsity, 8×37 ×10–28×35×54 dimensions), unless otherwise noted. We stream tiles of tensors onto the chip for operands that are larger than the GLB. All sparse application intermediates used to compute tiled results are stored on chip. 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. Fig. 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×. Fig. 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. Fig. 20 shows the 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 parallelize a bitstream over the array to improve performance.

Authorized licensed use limited to: Stanford University Libraries. Downloaded on February 04,2026 at 15:54:59 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

TABLE IV
COMPARISON TABLE

	This Work	Amber JSSC 22 [10]	Zhang VLSI 22 [42]	Huang VLSI 22 [7]
Architecture	SoC w/CGRA	SoC w/CGRA	CNN/Image Processing PE Array	Sparsity-Aware CNN-GCN
Programmability	✓	✓	✓	×
Sparse/Dense	Sparse/Dense	Dense Only	Dense Only	Sparse/Dense
Technology/Area	12 nm/23 mm2	16 nm/20.1 mm2	22 nm/8.8 mm2	28 nm/8.3 mm2
Formats Supported	BFloat16, INT16	BFloat16, INT16	INT16	INT8
# of Cores	384 PEs, 1 M3	384 PEs, 1 M3	576 PEs, RISC Core, Custom M33	1024 8-bit MACs
Total SRAM	4.5 MB	4.5 MB	1428 KB, 2MB MRAM	292 KB
Voltage & Frequency	0.78 V @970 MHz	0.84 V @580 MHz 1.29 V @955 MHz	0.5 - 1.0 V 56 KHz - 190 MHz	10 - 200 MHz
Peak Performance (Normalized to MAC=2OPs for all)	571 INT16 GOPS @(0.78 V, 850 MHz)	367 INT16 GOPS @(1.29 V, 955 MHz)	414 INT16 GOPS @(1.0 V, 180 MHz)	Dense: 410 GOPS Sparse: 3.3 TOPS
Peak Energy Efficiency (Normalized to MAC=2OPs for all)	756 INT16 GOPS/W @(0.66 V, 500 MHz)	538 INT16 GOPS/W @(0.84 V, 580 MHz)	7.0 INT16 TOPS/W @(0.5 V, 16 MHz)	Dense: 3.1 TOPS/W Sparse: 25.1 TOPS/W

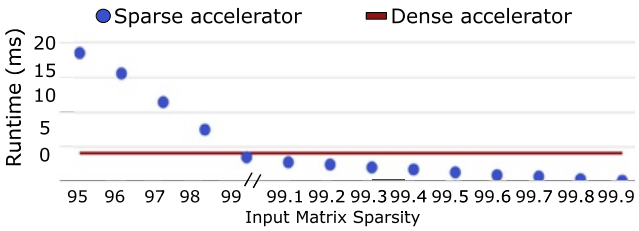


Fig. 21. Sparse accelerator versus dense accelerator runtime results on a 512 × 512 matrix multiplication of varying sparsity.

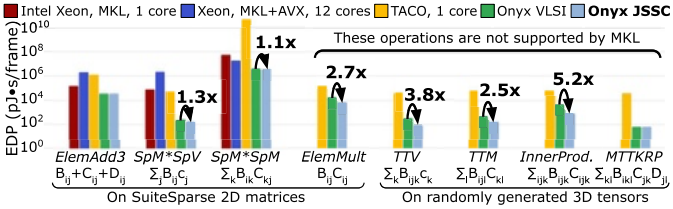


Fig. 22. EDP compared with CPUs with sparse acceleration libraries for a variety of sparse 2-D and 3-D kernels.

Unfortunately, feeding tiles into two parallel matrix multiplications in order can lead to load imbalance and result in suboptimal performance. In Fig. 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 GLB but achieves the highest performance, 1.76× faster than no unrolling. Fig. 20 shows a 1.6× 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 GLB. Fig. 20 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 Versus Dense Matrix Multiplication: For a 512 × 512 matrix multiplication with varying input sparsity, we compare our dense accelerator configuration versus its sparse counterpart in Fig. 21. After 99% matrix sparsity, we should configure our accelerator as a sparse accelerator. At 99.9% input sparsity, we perform 30× better than the dense accelerator configuration.

Given the data above in Fig. 21, 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 ten iterations.

Utilizing the scheduling optimizations described above, we perform up to 5.2× better in EDP on sparse tensor algebra expressions versus our VLSI submission [41] and 4.4 – 565× in EDP better versus CPUs with sparse libraries, as shown in Fig. 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. [43] design several fixed-function accelerators for 3-D 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, Yang et al. [5], Dadu et al. [44], Zhong et al.

Authorized licensed use limited to: Stanford University Libraries. Downloaded on February 04,2026 at 15:54:59 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

[45], and Lu et al. [46] optimize matrix multiplications over different levels of input sparsity, while Hegde et al. [3] and Srivastava et al. [47] support operations on high-order tensors, such as MTTKRP. Additionally, Rucker et al. [4] support tensor addition but do not take full advantage of sparse iteration for intersects/unions. Symphony [48] 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 [10] and [42] (shown in Table IV). However, Feng et al. [10] and Zhang et al. [42] are limited to dense applications only and achieve lower peak performance. There are several other works that explore CGRA architectures [8], [9], [11], [49], [50], 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 ML 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 et al., "A 0.32–128 TOPS, scalable multi-chip-module-based deep neural network inference accelerator with ground-referenced signaling in 16 nm," *IEEE J. Solid-State Circuits*, vol. 55, no. 4, pp. 920–932, Apr. 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 J. Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, Jan. 2017.

[3] K. Hegde et al., "ExTensor: An accelerator for sparse tensor algebra," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 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 *Proc. 54th Annu. IEEE/ACM Int. Symp. Microarchitecture*. New York, NY, USA: Association for Computing Machinery, Oct. 2021, pp. 1022–1035, doi: 10.1145/3466752.3480047.

[5] Y. Yang, J. S. Emer, and D. Sanchez, "Trapezoid: A versatile accelerator for dense and sparse matrix multiplications," in *Proc. ACM/IEEE 51st Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 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," *Proc. IEEE*, vol. 109, no. 10, pp. 1706–1752, Oct. 2021.

[7] W.-C. Huang et al., "A 28-nm 25.1 TOPS/W sparsity-aware CNN-GCN deep learning SoC for mobile augmented reality," in *Proc. IEEE Symp. VLSI Technol. Circuits (VLSI Technol. Circuits)*, Honolulu, HI, USA, Jun. 2022, pp. 42–43.

[8] V. Govindaraju et al., "DySER: Unifying functionality and parallelism specialization for energy-efficient computing," *IEEE Micro*, vol. 32, no. 5, pp. 38–51, Sep. 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 *Proc. Int. Conf. Field Program. Log. Appl.*, Aug. 2005, pp. 106–111.

[10] K. Feng et al., "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. 59, no. 3, pp. 947–959, Mar. 2023.

[11] R. Prabhakar et al., "Plasticine: A reconfigurable architecture for parallel paterns," *ACM SIGARCH Comput. Archit. News*, vol. 45, no. 2, pp. 389–402, Jun. 2017, doi: 10.1145/3140659.3080256.

[12] *ARM Cortex-M3*. Accessed: Jun. 2023. [Online]. Available: https://developer.arm.com/Processors/Cortex-M3

[13] K. Koul et al., "AHA: An agile approach to the design of coarse-grained reconfigurable accelerators and compilers," *ACM Trans. Embedded Comput. Syst.*, vol. 22, no. 2, pp. 1–34, Jan. 2023, doi: 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 Comput. Archit. Lett.*, vol. 22, no. 1, pp. 45–48, Jan. 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, *arXiv:2301.00861*.

[16] O. Hsu et al., "The sparse abstract machine," in *Proc. 28th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.* New York, NY, USA: Association for Computing Machinery, Mar. 2023, pp. 710–726, doi: 10.1145/3582016.3582051.

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

[18] M. Pellauer et al., "Buffets: An efficient and composable storage idiom for explicit decoupled data orchestration," in *Proc. Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*. New York, NY, USA: Association for Computing Machinery, 2019, pp. 137–151, doi: 10.1145/3297858.3304025.

[19] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 1–29, Oct. 2017.

[20] S. Chou, F. Kjolstad, and S. Amarasinghe, "Format abstraction for sparse tensor algebra compilers," *Proc. ACM Program. Lang.*, vol. 2, pp. 1–30, Oct. 2018.

[21] R. Senanayake et al., "A sparse iteration space transformation framework for sparse tensor algebra," in *Proc. ACM Program. Lang.*, vol. 4, Nov. 2020, pp. 1–30. [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 Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 43, no. 10, pp. 3055–3067, Oct. 2024.

[23] G. Zhang, N. Attaluri, J. S. Emer, and D. Sanchez, "Gamma: Leveraging Gustavson's algorithm to accelerate sparse matrix multiplication," in *Proc. 26th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Apr. 2021, pp. 687–701.

[24] S. Pal et al., "OuterSPACE: An outer product based sparse matrix multiplication accelerator," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2018, pp. 724–736.

[25] Z. Zhang, H. Wang, S. Han, and W. J. Dally, "SpArch: Efficient architecture for sparse matrix multiplication," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2020, pp. 261–274.

[26] Z. Li et al., "Spada: Accelerating sparse matrix multiplication with adaptive dataflow," in *Proc. 28th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, vol. 2. New York, NY, USA: Association for Computing Machinery, Jan. 2023, pp. 747–761, doi: 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 *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays (FPGA)*. New York, NY, USA: Association for Computing Machinery, 2022, pp. 65–77, doi: 10.1145/3490422.3502357.

[28] J. Melchert et al., "APEX: A framework for automated processing element design space exploration using frequent subgraph analysis," in *Proc. 28th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.* New York, NY, USA: Association for Computing Machinery, Mar. 2023, pp. 33–45, doi: 10.1145/3582016.3582070.

[29] Q. Liu et al., "Unified buffer: Compiling image processing and machine learning applications to push-memory accelerators," *ACM Trans. Archit. Code Optim.*, vol. 20, no. 2, pp. 1–26, Mar. 2023, doi: 10.1145/3572908.

Authorized licensed use limited to: Stanford University Libraries. Downloaded on February 04,2026 at 15:54:59 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

[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 *Proc. 34th ACM SIGPLAN Conf. Program. Lang. Design Implement.* New York, NY, USA: Association for Computing Machinery, Jun. 2013, pp. 519–530, doi: 10.1145/2491956.2462176.

[31] J. Lin, W.-M. Chen, Y. Lin, J. Cohn, C. Gan, and S. Han, "MCUNet: Tiny deep learning on IoT devices," in *Proc. Adv. Neural Inf. Process. Syst.*, 2020, pp. 11711–11722.

[32] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted residuals and linear bottlenecks," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 4510–4520.

[33] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 770–778.

[34] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2009, pp. 248–255.

[35] H Lab,. *MCUNet: Tiny Deep Learning on Microcontrollers*. GitHub Repository. Accessed: Nov. 15, 2024. [Online]. Available: https://github.com/mit-han-lab/mcunet

[36] PyTorch Team. *PyTorch Hub: MobileNet V2*. Accessed: Nov. 15, 2024. [Online]. Available: https://pytorch.org/hub/pytorchvisionmobilenetv2/

[37] TorchVision Contributors. *ResNet-18 Pretrained Model on ImageNet*. Accessed: Nov. 15, 2024. [Online]. Available: https://pytorch.org/vision/main/models/generated/torchvision.models.resnet18.html

[38] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1–25, Nov. 2011.

[39] F. G. Gustavson, "Two fast algorithms for sparse matrices: Multiplication and permuted transposition," *ACM Trans. Math. Softw.*, vol. 4, no. 3, pp. 250–269, Sep. 1978.

[40] *Intel Math Kernel Library*, Intel Corp., Santa Clara, CA, USA, 2009.

[41] K. Koul et al., "Onyx: A 12 nm 756 GOPS/W coarse-grained reconfigurable array for accelerating dense and sparse applications," in *Proc. IEEE Symp. VLSI Technol. Circuits (VLSI Technol. Circuits)*, Jun. 2024, pp. 1–2.

[42] Q. Zhang et al., "A 22 nm 3.5TOPS/W flexible micro-robotic vision SoC with 2MB eMRAM for fully-on-chip intelligence," in *Proc. IEEE Symp. VLSI Technol. Circuits (VLSI Technol. Circuits)*, Jun. 2022, pp. 72–73.

[43] 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 *Proc. IEEE Symp. VLSI Technol. Circuits (VLSI Technol. Circuits)*, Jun. 2023, pp. 1–2.

[44] V. Dadu, J. Weng, S. Liu, and T. Nowatzki, "Towards general purpose acceleration by exploiting common data-dependence forms," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2019, pp. 924–939.

[45] K. Zhong et al., "FEASTA: A flexible and efficient accelerator for sparse tensor algebra in machine learning," in *Proc. 29th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, vol. 3. New York, NY, USA: Association for Computing Machinery, Apr. 2024, pp. 349–366, doi: 10.1145/3620666.3651336.

[46] L. Lu et al., "Sanger: A co-design framework for enabling sparse attention using reconfigurable architecture," in *Proc. 54th Annu. IEEE/ACM Int. Symp. Microarchitecture*. New York, NY, USA: Association for Computing Machinery, Oct. 2021, pp. 977–991, doi: 10.1145/3466752.3480125.

[47] N. Srivastava, H. Jin, S. Smith, H. Rong, D. Albonesi, and Z. Zhang, "Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2020, pp. 689–702.

[48] M. Pellauer et al., "Symphony: Orchestrating sparse and dense tensors with hierarchical heterogeneous processing," *ACM Trans. Comput. Syst.*, vol. 41, nos. 1–4, pp. 1–30, Dec. 2023.

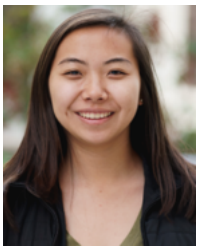
[49] C. Torng, P. Pan, Y. Ou, C. Tan, and C. Batten, "Ultra-elastic CGRAs for irregular loop specialization," in *Proc. IEEE Int. Symp. High-Performance Comput. Archit. (HPCA)*, Feb. 2021, pp. 412–425.

[50] S. A. Chin et al., "CGRA-ME: A unified framework for CGRA modelling and exploration," in *Proc. IEEE 28th Int. Conf. Appl.-Specific Syst., Architectures Processors (ASAP)*, Jul. 2017, pp. 184–189.



Kalhan Koul (Member, IEEE) received the B.S. degree (Hons.) in electrical engineering and the B.A. degree in liberal arts (Plan II Hons.) from The University of Texas, Austin, TX, USA, in 2018, and the M.S. and Ph.D. degrees in electrical engineering from Stanford University, Stanford, CA, USA, in 2021 and 2025, respectively.

He is currently a GPU Architect at Nvidia, Santa Clara, CA, USA, where he works on power modeling and optimizations for machine learning workloads. His research focused on agile hardware design flows, coarse-grained reconfigurable array (CGRA) architectures, and accelerators for sparse applications.



Olivia Hsu (Member, IEEE) received the B.S. degree in electrical engineering and computer science from the University of California at Berkeley, Berkeley, CA, USA, in 2019, and the M.S. degree in computer science from Stanford University, Stanford, CA, USA, in 2025, where she is currently pursuing the Ph.D. degree in computer science.

She is an incoming Assistant Professor of electrical and computer engineering at Carnegie Mellon University, Pittsburgh, PA, USA. Her current research includes accelerator architectures, programming systems, heterogeneous systems, and domain-specific compilers.



Yuchen Mei received the B.S. degree in electronic information science and technology from Nanjing University, Nanjing, China, in 2021, and the M.S. degree in electrical engineering from Stanford University, Stanford, CA, USA, in 2023, where he is currently pursuing the Ph.D. degree in electrical engineering, under the supervision of Prof. Priyanka Raina.

His research interests include application mapping and optimization for domain-specific accelerators, with a focus on heterogeneous reconfigurable computing fabrics for machine learning.



Sai Gautham Ravipati received the M.S. degree from Stanford University, Stanford, CA, USA, in 2025, and the B.Tech. degree in electrical engineering from IIT Madras, Chennai, India, in 2023. He is currently pursuing the Ph.D. degree in electrical engineering with Stanford University.

His research interests include programming systems, domain-specific compilers, and hardware-software co-design.



Maxwell Strange received the B.S. degree in computer engineering and computer science from the University of Wisconsin–Madison, Madison, WI, USA, in 2017, and the M.S. degree in electrical engineering from Stanford University, Stanford, CA, USA, in 2020, where he is currently pursuing the Ph.D. degree in electrical engineering under the supervision of Prof. Mark Horowitz.

His research focuses on developing infrastructure and tools to facilitate agile hardware development as part of the efforts by the AHA! Agile Hardware Center, Stanford University. His research interests include domain-specific hardware architectures, hardware/software codesign, and embedded systems design.

Authorized licensed use limited to: Stanford University Libraries. Downloaded on February 04,2026 at 15:54:59 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.



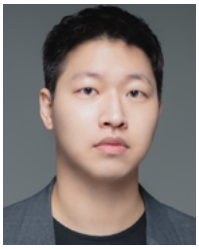
Jackson Melchert (Member, IEEE) received the B.S. degree (Hons.) in electrical engineering from the University of Wisconsin–Madison, Madison, WI, USA, in 2018, and the Ph.D. degree in electrical engineering from Stanford University, Stanford, CA, USA, in 2024.

He is currently a Compiler Engineer at Efficient, Pittsburgh, PA, USA, where he works on mapping applications to lower power devices. His research focused on optimizing configurable hardware to approach the performance and efficiency of application-specific accelerators.



Alex Carsello received the B.S. degree in electrical engineering and the B.S. degree in computer engineering from Washington University in St. Louis, Saint Louis, MO, USA, in 2017, and the M.S. and Ph.D. degrees in electrical engineering from Stanford University, Stanford, CA, USA, in 2020 and 2025, respectively.

He is interested in agile physical design tools and methodologies, reconfigurable computing, and domain-specific architectures for image processing and machine learning.



Taeyoung Kong (Member, IEEE) received the B.S. degree in electrical and computer engineering from Seoul National University, Seoul, South Korea, in 2017, and the M.S. and Ph.D. degrees in electrical engineering from Stanford University, Stanford, CA, USA, in 2020 and 2024, respectively.

His doctoral work focused on architecture and runtime software support for CGRAs. He is currently a Machine Learning Architect at Google, Sunnyvale, CA, USA, where he works on the TPU architecture and HW-SW codesign for LLM workloads.



Po-Han Chen (Graduate Student Member, IEEE) received the B.S. degree in electrical engineering and computer science and the M.S. degree in electrical engineering from National Tsing Hua University, Hsinchu, Taiwan, in 2016 and 2018, respectively. He is currently pursuing the Ph.D. degree in electrical engineering with Stanford University, Stanford, CA, USA, under the supervision of Prof. Priyanka Raina.

Before joining Stanford, he was a Digital Circuit Designer at MediaTek, Hsinchu, where he worked on developing hardware architectures for image processing pipelines. Most of his previous works were related to computational photography algorithms, such as digital refocusing. He is interested in designing hardware accelerators. He is focusing on analyzing and designing the architecture of CGRAs to create high-performance, energy-efficient, and reconfigurable computing platforms.



Huifeng Ke received the B.A.Sc. degree in electrical engineering from the University of Toronto, Toronto, ON, Canada, in 2020, and the M.S. degree in electrical engineering from Stanford University, Stanford, CA, USA, in 2023. He is currently pursuing the Ph.D. degree in electrical engineering under the supervision of Prof. Tony Nowatzki and Prof. Jason Cong.

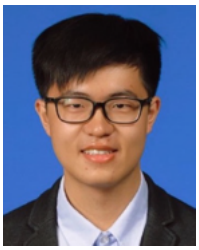
His research interests include reconfigurable hardware architectures and accelerators for Boolean satisfiability (SAT).



Keyi Zhang received the B.S. degree in computer science and engineering from Bucknell University, Lewisburg, PA, USA, in 2017, and the M.S. and Ph.D. degrees in computer science from Stanford University, Stanford, CA, USA, in 2021 and 2022, respectively.

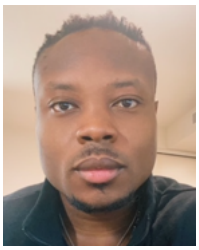
During his Ph.D. degree, he was advised by Prof. Mark Horowitz. He is currently a Lead Systems Engineer at EfficientAI, Santa Clara, CA, USA, a start-up focusing on designing highly efficient and ultralow-power reconfigurable fabrics. His research interests include software and hardware co-design, compiler optimization, hardware generator frameworks, and debugging generated hardware systems.

Dr. Zhang was a recipient of the Apple Ph.D. Fellowship in Integrated Systems.

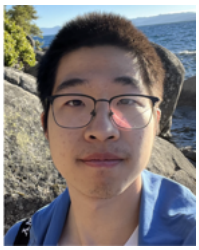


Qiaoyi Liu received the B.S. degree in electronic engineering from Tsinghua University, Beijing, China, in 2017, and the M.S. and Ph.D. degrees in computer science from Stanford University, Stanford, CA, USA, in 2020 and 2024, respectively.

His research focuses on computer architecture and compilation for domain-specific accelerators, with a particular interest in optimizing schedules and memory mappings for compute-intensive tensor applications, such as image processing and deep learning.



Gedeon Nyengele (Member, IEEE) received the B.S. degree in electrical engineering from Georgia Institute of Technology, Atlanta, GA, USA, in 2017. He is currently pursuing the Ph.D. degree in electrical engineering with Stanford University, Stanford, CA, USA, with a focus on novel methodologies for system-on-chip design, under the supervision of Prof. Mark Horowitz,.



Zhouhua Xie received the B.S. degree in electrical engineering from Stanford University, Stanford, CA, USA, in 2025, where he is currently pursuing the Ph.D. degree in electrical engineering under the supervision of Prof. Priyanka Raina.



Akhilesh Balasingam received the B.S. degree in electrical engineering from Stanford University, Stanford, CA, USA, in 2025, where he is currently pursuing the M.S. degree in computer science.

He was an Intern at Apple, Cupertino, CA, USA. His research interests include computer architecture, compilers, and the design of hardware–software systems for high-performance computing and machine learning.

Authorized licensed use limited to: Stanford University Libraries. Downloaded on February 04,2026 at 15:54:59 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.



Jayashree Adivarahan received the B.S. degree in electrical engineering and the B.S. degree in computer systems engineering from Arizona State University, Tempe, AZ, USA, in 2025. She is currently pursuing the Ph.D. degree in electrical and computer engineering with the University of Southern California, Los Angeles, CA, USA, under the supervision of Prof. Viktor Prasanna, with a focus in HW/SW co-design, computer architecture, and hardware accelerator design.

She has interned with Northrop Grumman, Baltimore, MD, USA, and with Intel, Chandler, AZ, USA. She also participated in a research internship at Stanford University, Stanford, CA, USA.

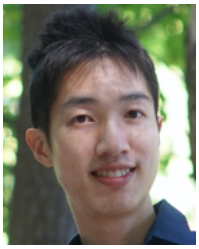
Ms. Adivarahan was a recipient of NSF GRFP and the USC Annenberg Fellowship.



Ritvik Sharma received the B.Tech. degree in electrical engineering from IIT Delhi, New Delhi, India, in 2021, and the M.S. degree in electrical engineering from Stanford University, Stanford, CA, USA, in 2023, where he is currently pursuing the Ph.D. degree in electrical engineering under the supervision of Prof. Sara Achour.

He was an Intern with IonQ, College-Park, MD, USA, and with Google, Sunnyvale, CA, USA. His research interests include compiler tools for optimization of structured and rank sparse data and quantum simulations.

Mr. Sharma received Stanford's Stranford Graduate Fellowship from 2021 to 2024.



Christopher Torng received the B.S., M.S., and Ph.D. degrees in electrical and computer engineering from Cornell University, Ithaca, NY, USA, in 2012, 2016, and 2019, respectively.

From 2019 to 2022, he was a Post-Doctoral Researcher at Stanford University, Stanford, CA, USA, operating in the leadership of Stanford AHA Agile Hardware Project. Since 2023, he has been with the University of Southern California, Los Angeles, CA, USA, where he is currently an Assistant Professor of electrical and computer engineering. His research interests include domain-specific hardware architectures and agile hardware design methodologies.



Joel S. Emer (Life Fellow, IEEE) received the B.S. (Hons.) and M.S. degrees in electrical engineering from Purdue University, West Lafayette, IN, USA, in 1974 and 1975, respectively, and the Ph.D. degree in electrical engineering from the University of Illinois at Urbana–Champaign, Champaign, IL, USA, in 1979.

He is currently a Senior Distinguished Research Scientist with Nvidia's Architecture Research Group, Santa Clara, CA, USA, and a Professor of practice at Massachusetts Institute of Technology, Cambridge, MA, USA. Previously, he was with Intel, Chandler, AZ, USA, where he was an Intel Fellow and the Director of Microarchitecture Research. At Intel, he led the VSSAD Group, which he had previously been a member of at Compaq, Palo Alto, CA, USA, and Digital Equipment Corporation, Maynard, MA, USA. He has made architectural contributions to a number of VAX, Alpha, and X86 processors and has contributed to the quantitative approach to processor performance evaluation, simultaneous multithreading technology, processor reliability analysis, cache and pipeline organization, and spatial architectures for deep learning.

Dr. Emer is a fellow of the Association for Computing Machinery (ACM) and a member of the National Academy of Engineering (NAE). He received both the Eckert-Mauchly and Rau Awards and ECE Alumni Awards from Purdue University and the University of Illinois. He has multiple best paper awards and seven papers selected as the IEEE Micro Top Picks in Computer Architecture.



Fredrik Kjolstad is currently an Assistant Professor of computer science at Stanford University, Stanford, CA, USA. He works on topics in compilers, programming models, and systems, with an emphasis on compiler techniques that make high-level languages portable.

Dr. Kjolstad has received the NSF CAREER Award, the MIT Sprowls Ph.D. Thesis Award in Computer Science, the Tau Beta Phi Teaching Honor Roll, the Google ML and Systems Junior Faculty Awards, and several distinguished paper awards.



Mark Horowitz (Life Fellow, IEEE) received the B.S. and M.S. degrees in electrical engineering from Massachusetts Institute of Technology, Cambridge, MA, USA, in 1978, and the Ph.D. degree from Stanford University, Stanford, CA, USA, in 1984.

He has worked on many processor designs, from early RISC chips to distributed shared memory multiprocessors, and in 1990, he took leave from Stanford University to help start Rambus Inc., a company designing high-bandwidth memory interface technology. His work at both Rambus and Stanford University drove high-speed link designs for many decades. In the 2000s, he started a collaboration with Marc Levoy in computational photography, which led to light-field photography and microscopy. He is currently the Yahoo! Founders Professor at Stanford University, where he is the Chair of the Electrical Engineering Department. He remains interested in learning new things, and building interdisciplinary teams. His current research includes updating both analog and digital design methods, agile hardware design, and applying engineering to biology.

Dr. Horowitz is a fellow ACM and a member of the National Academy of Engineering and American Academy of Arts and Science.



Priyanka Raina (Life Fellow, IEEE) received the B.Tech. degree in electrical engineering from IIT Delhi, New Delhi, India, in 2011, and the M.S. and Ph.D. degrees in electrical engineering and computer science from Massachusetts Institute of Technology, Cambridge, MA, USA, in 2013 and 2018, respectively.

She was a Visiting Research Scientist with NVIDIA Corporation, Santa Clara, CA, USA, in 2018. She is currently an Assistant Professor of electrical engineering with Stanford University, Stanford, CA, USA, where she works on domain-specific hardware architectures and agile hardware–software co-design methodology.

Dr. Raina is a 2018 Terman Faculty Fellow. She was a co-recipient of the Best Demo Paper Award at VLSI 2022, the Best Student Paper Award at VLSI 2021, the JSSC Best Paper Award 2020, the Best Paper Award at MICRO 2019, and the Best Young Scientist Paper Award at ESSCIRC 2016. She received the DARPA Young Faculty Award and the Sloan Research Fellowship in 2024, the NSF CAREER Award in 2023, Intel Rising Star Faculty Award in 2021, and Hellman Faculty Scholar Award in 2019. She was the Program Chair for IEEE Hot Chips 2020. She currently serves as an Associate Editor for IEEE JOURNAL OF SOLID-STATE CIRCUITS and IEEE SOLID-STATE CIRCUITS LETTERS.

Authorized licensed use limited to: Stanford University Libraries. Downloaded on February 04,2026 at 15:54:59 UTC from IEEE Xplore. Restrictions apply.