ISOSceles: Accelerating Sparse CNNs through Inter-Layer Pipelining

Yifan Yang
MIT CSAIL
yifany@csail.mit.edu

Joel S. Emer
MIT CSAIL / NVIDIA
emer@csail.mit.edu

Daniel Sanchez
MIT CSAIL
sanchez@csail.mit.edu

***Abstract*—Sparse CNNs dramatically reduce computation and storage costs over dense ones. But sparsity also makes CNNs more data-intensive, as each value is reused fewer times. Thus, current sparse CNN accelerators, which process one layer at a time, are bottlenecked by memory traffic.**

We present *ISOSceles*, a new sparse CNN accelerator that dramatically reduces data movement through *inter-layer pipelining*: overlapping the execution of consecutive layers so that a layer's output activations are quickly consumed by the next layer without spilling them off-chip. Pipelining greatly increases reuse, but it is challenging to implement with existing approaches, which are limited to dense CNNs. ISOSceles relies on a novel *input-stationary output-stationary (IS-OS)* dataflow that consumes inputs and produces outputs in the same order, greatly reducing intermediate sizes over existing dataflows. ISOSceles implements IS-OS efficiently and leverages time-multiplexing and dynamic scheduling to pipeline multiple layers despite the large variations in work that sparsity induces.

On a wide range of sparse CNNs, ISOSceles outperforms a state-of-the-art accelerator by gmean 4.3× (up to 6.7×), and reduces traffic by 4.7× (up to 8.5×) while using less area.

I. Introduction

Convolutional Neural Networks (CNNs) achieve state-of-the-art performance on many machine learning tasks, but are computationally expensive. Exploiting *sparsity* is a promising way to reduce the compute and storage costs of CNNs. Sparse CNNs leverage the fact that a substantial fraction of values in weights and activations are zeros. Hardware accelerators exploit weight and activation sparsity by skipping multiplications by zero [7] and by not storing zero values [34].

Sparsity in weights and activations arises for different reasons, and to different degrees. First, weight pruning removes filter weights with near-zero values [19]. This process creates significant weight sparsity: prior work has shown that 80% to over 95% of weights can be pruned with negligible accuracy loss [26]. Second, activation sparsity arises because common non-linear activation functions like ReLU [29] convert negative activations into zeros. Prior accelerators have exploited sparsity in weights [44], activations [2], or both [17, 34] to reduce execution time, energy, and data movement over dense CNNs.

Sparse CNNs put more pressure on the memory system than dense ones, so existing accelerators are *dominated by data movement* rather than computation costs. This is because sparsity reduces computation more than memory footprint and traffic. For example, a convolutional layer with 90% sparse weights and activations reduces the footprint by 10×, but reduces multiplications by about $(1-0.9)\cdot(1-0.9)$, i.e., $100\times$. Reuse is also greatly decreased, as each weight and activation is used many fewer times. This reduces *arithmetic intensity*, the number of compute operations per byte of data fetched from memory. For example, sparsifying the ResNet-50 model reduces arithmetic intensity from 128 to 11 operations/byte.

Accelerating sparse CNNs requires techniques that reduce off-chip traffic. In this paper, we show that *pipelining* consecutive CNN layers is an effective approach. Most accelerators process one layer at a time, producing all output activations for a given layer before starting the next layer. Since input and output activations are large, they get spilled off-chip, causing substantial memory traffic. Inter-layer pipelining avoids this traffic by overlapping the execution of consecutive layers, so that each output activation produced by one layer is quickly consumed by the next layer. Thus, most activations are reused on-chip, and only the first layer's input activations and last layer's output activations incur off-chip traffic. Pipelining multiple layers requires maintaining their weights on-chip, *but sparsity makes this practical*. For example, with 90% weight sparsity, an accelerator can pipeline 10 layers with the same amount of on-chip storage that a dense accelerator uses to store weights for one layer. This improves activation reuse by about 10×.

Though inter-layer pipelining has major potential to accelerate sparse CNNs, it requires solving two key challenges: *(1)* finding a *dataflow* (i.e., a computation schedule) that minimizes the amount of intermediate activations between layers (so they can be consumed without spilling them off-chip) without adding other types of traffic (e.g., sacrificing reuse in weights) and that efficiently traverses the compressed data structures in sparse CNNs; and *(2)* building an accelerator that achieves *high utilization* despite the high dynamism introduced by sparsity: due to zero activations and weights, different layers have large and fast variations in work, so standard ways to pipeline them (e.g., running different layers on different parts of the chip) do not work well. We tackle these challenges with *ISOSceles*, the first accelerator that exploits inter-layer pipelining effectively to improve sparse CNN performance.

Prior accelerators use dataflows that cannot be pipelined effectively [2, 17, 34, 44] (Sec. II). For example, a dataflow may be *output-stationary*, producing outputs one element at a time but inducing poor reuse of inputs, or *input-stationary*, consuming inputs one element at a time but producing partial outputs out of order. These dataflows can be tiled, so that a single output tile is produced from an input tile. However, due to the nature of convolutions, output tiles are smaller than input tiles. This causes far-away reuse of *input halos* (the input elements used by multiple output tiles) and deep pipelines must use large tiles, which add substantial on-chip footprint. In fact, Fused-Layer [3] leverages tiling to pipeline *dense* CNNs, but achieves limited benefits (-6% to 27% speedups). This is mainly due to the higher arithmetic intensity of dense CNNs, but also to the limited degree of pipelining that tiling allows. Some dense accelerators, Brein [4] and Tangram [16], combine output- and input-stationary dataflows to pipeline two layers, but they are limited to pipelining only two layers. Finally, these pipelining approaches do not support sparsity, forgoing its performance and efficiency gains.

To tackle these challenges, we introduce a novel *input-stationary output-stationary (IS-OS)* dataflow (Sec. III). IS-OS consumes input activations and produces output activations in the same order, producing outputs in thin *wavefronts* instead of the 2D tiles of prior work. These wavefronts can be consumed immediately by the next layer, minimizing inter-layer storage. Moreover, IS-OS supports efficient indexing into compressed data structures (input and output activations, weights, and partial results), enabling all data to be stored in compressed form to reap the footprint benefits of sparsity. Finally, IS-OS does not add undue traffic to other structures (e.g., weights), and supports the same optimizations as other dataflows (e.g., tiling to accommodate layers whose weights do not fit on-chip).

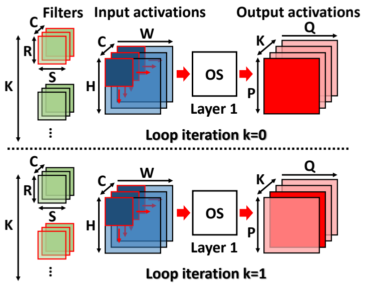
We then present the ISOSceles architecture (Sec. IV), which implements the IS-OS dataflow to pipeline sparse CNNs effectively. ISOSceles processes each layer with a unit that contains an input-stationary (IS) frontend and an output-stationary (OS) backend. The IS frontend accepts an input wavefront fetched from off-chip or delivered on-chip directly from the previous layer, and its MAC units convolve it with sparse weights. The OS backend features *mergers* that combine and reduce partial results to generate an output wavefront that can be delivered directly to another IS frontend or written off-chip. To achieve high utilization under sparsity, ISOSceles time-multiplexes and dynamically schedules PEs to avoid load imbalance and keep intermediate activations small. A programmable interconnect allows diverse CNN models to be mapped to ISOSceles.

We evaluate ISOSceles using cycle-level simulation and RTL synthesis on a variety of sparse CNNs (Sec. VI). Over the dense Fused-Layer [3] accelerator, ISOSceles is gmean 7.5× (up to 18.0×) faster, and reduces off-chip traffic by 3.6× by exploiting sparsity. ISOSceles outperforms SparTen [17], a state-of-the-art sparse CNN accelerator, by gmean 4.3× (up to 6.7×), and reduces traffic by 4.7× (up to 8.5×) with significantly less area.

II. Background and Motivation

In this section, we first introduce the key CNN computations and relevant dense CNN accelerators, then describe the advantages and challenges of sparse CNNs and accelerators.

A. Prior dense dataflows and accelerators



(a) Datatypes and accesses (highlighted) for first two iterations of outermost loop.

```
i = Tensor(shape=[C,H,W])
f = Tensor(shape=[K,C,R,S])
o = Tensor(shape=[K,P,Q])

for k = [0, K):
 for p = [0, P):
  for q = [0, Q):
   for c = [0, C):
    for r = [0, R):
     for s = [0, S):
      o[k,p,q] +=
          i[c,p+r,q+s]
          * f[k,c,r,s]
```

(b) Loop nest.

Fig. 1: Output-Stationary (OS) dataflow.

The key computation in a CNN layer is a 3D convolution of a set of C *input activation* planes of $H \times W$ elements (with one plane per *input channel*) with *filters* consisting of a $C \times R \times S$-element set of weights, to produce an *output activation* plane with $P \times Q$ elements (where $P = H - R + 1$ and $Q = W - S + 1$). A layer may have multiple *output channels* K, resulting from applying K filters to the input activations to produce K output activation planes. Fig. 1a shows the **filters**, **input activations**, and **output activations**, and their dimensions.

The computation of a layer can be expressed as a 6-level loop nest (omitting the optional loop for batch size, N). A specific ordering of the loop nest corresponds to a schedule for the computation, and is referred to as a *dataflow* [7, 33, 39]. *Tiling*, which changes the order of computation, can be represented with additional loop levels. Tiling reorders the computation to achieve reuse on smaller chunks, or tiles, of a tensor; tile sizes are chosen to fit in on-chip buffers.

Different dataflows induce different reuse for the input activations, output activations, and weights. For example, Fig. 1b shows the loop nest for the output-stationary (OS) dataflow. OS produces each output one element at a time, attaining maximum reuse of partial outputs. However, filter weights and input activations have worse reuse. Fig. 1a shows how OS proceeds over time, highlighting the inputs and filters used to produce the first and second output planes (corresponding to the two first iterations of the outermost loop in Fig. 1b). Similar to OS, input-stationary (IS) and weight-stationary (WS) dataflows prioritize reuse of inputs and weights [7].

Most dense CNN accelerators follow a fixed dataflow and operate on one layer at a time, incurring a significant cost for holding input and output activations off-chip [6, 7, 9, 14, 24]. We focus on the few that support inter-layer pipelining.

Dense accelerators with pipelining: The Fused-Layer CNN accelerator [3] pipelines the execution of multiple layers. Fused-Layer uses a tiled OS dataflow, producing output activations tile by tile and reusing them as the input to the next layer. Fig. 2 illustrates this dataflow across four layers, for the case where each layer has a single input and output channel (i.e., $C = K = 1$). Fig. 2 shows the input tiles required at previous layers to produce an output tile at the last layer. Due to the nature of convolutions, output tiles are smaller than input tiles, causing far-away reuse of *input halos*, the elements of an input tile shared by multiple output tiles. To reduce the impact of halos, tiles need to be relatively large, requiring substantial on-chip storage to hold tiles and halos. Furthermore, input halos

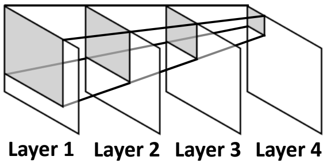


Fig. 2: Fused-Layer CNN accelerator dataflow (using layers with a single input and output channel).

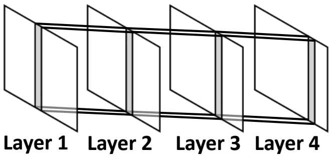


Fig. 3: Input-stationary output-stationary (IS-OS) dataflow (using layers with a single input and output channel).

grow with the number of pipelined layers (Fig. 2), increasing their overheads. While Fused-Layer reduces traffic substantially for some CNNs, it yields modest speedups (-6% to 27%), largely because dense networks are less bottlenecked by data movement than sparse ones.

Fig. 3 shows how our novel IS-OS dataflow differs from prior tile-based pipelining in the dense setting. Like Fig. 2, it shows multiple layers. The key difference is that IS-OS produces thin and tall *wavefronts* instead of 2D tiles. Activations are produced and consumed in the same order, and wavefronts in earlier layers are ahead in the W dimension (by S positions per layer, i.e., by the width of the filters). This reduces the sizes of intermediates and avoids or greatly reduces halos, enabling deeper pipelines. Moreover, IS-OS supports sparsity efficiently by allowing the efficient (concordant) traversal of compressed data structures in sparse CNNs, unlike Fused-Layer (Sec. II-B). Finally, IS-OS spatially parallelizes loop dimensions like H (activation height) to maximize vertical reuse and handle sparsity with simple hardware. Fused-Layer parallelizes differently, on channel dimensions C and K, to simplify inter-PE communication. Sec. III describes IS-OS in detail.

Prior work beyond Fused-Layer proposed *dense* dataflows that allow some inter-layer pipelining. Brein [4] and Tangram [16] pipeline two layers by chaining an output-stationary layer with an input-stationary layer. While simpler than Fused-Layer, they can only pipeline two layers. By contrast, our IS-OS dataflow allows pipelining an arbitrary number of layers.

B. Leveraging sparsity in CNNs

Much prior work has leveraged sparsity to reduce the cost of CNN inference. *Weight pruning* zeroes out weights based on some criteria. In *unstructured pruning* [19], all weights below a threshold are pruned away. Then the resulting sparse CNN is retrained to retain accuracy. Prior work has shown that 50–95% of weights can be trimmed with no or negligible accuracy loss [18, 19, 26]. *Structured pruning* techniques zero out a larger groups of weights, e.g., an entire channel [21]. Structured pruning allows using existing hardware efficiently, like GPUs, but quickly degrades accuracy by imposing strict constraints on weights. In this work, we focus on unstructured pruning, which allows greater pruning but requires new hardware to handle the resulting unstructured sparsity.

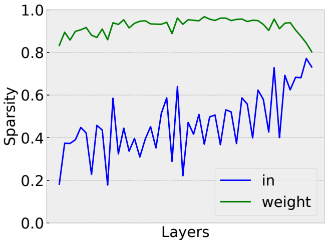


Fig. 4: Input activation and weight sparsity in ResNet-50.

Another source of sparsity, *activation sparsity*, arises from non-linear activation functions like ReLU [29], which turn negative activations into zeros. Activation sparsity is input-dependent and unstructured, as negative values may appear at any location.

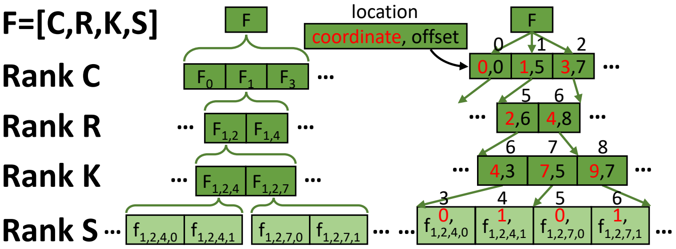


Fig. 5: A sparse weight tensor illustrated in the fibertree abstraction [39] (left) and its Compressed Sparse Fiber (CSF) format (right).

Leveraging both weight and activation sparsity yields the largest reductions in execution time, energy, and data movement. Fig. 4 shows the weight and activation sparsity of a pruned ResNet-50 CNN [26]. Each data point represents one layer and higher means sparser (100% denotes all zero values). As Fig. 4 shows, weights are typically sparser than activations, with 90% sparsity across layers in this case. Activation sparsity varies more, ranging from 20% to 80%.

Compressed data structures: An advantage of sparse data is that it can be compressed, e.g., storing only nonzero values and their coordinates, saving memory capacity and traffic. Sparse tensors can be described abstractly (i.e., without the details of their specific format) through the fibertree representation [39]. Fig. 5 (left) shows the fibertree of a 4D sparse **filter** tensor. The fibertree shows only the nonzero values in the tensor, and elements at each level, or *rank* [39], represent a sub-tensor that includes all the children below that element. For example, the root node F represents the entire sparse filter. F_1 at the C rank points to the set of nonzero weights with the same input channel $c = 1$ ($F[1,:,:,:]$). Note that F_2 is not present, meaning that input channel $c = 2$ is empty. $F_{1,2,4}$ at the K rank represents all filter weights at input channel $c = 1$, row id $r = 2$, and output channel $k = 4$ ($F[1,2,4,:]$). The leaf nodes are the filter weights; their subscripts denote their **coordinates** at every rank.

Actual designs must store tensors using one of the myriad concrete formats [10, 39] rather than the abstract representation above. Compressed Sparse Fiber (CSF) [25, 37], shown in Fig. 5 (right), is a commonly used format for higher-dimensional sparse tensors (CSF generalizes the CSR/CSC formats for 2D matrices). CSF stores a sparse tensor using one array per rank. Each array element is a tuple of the **coordinate** at the current rank and an offset to the next rank's array, where all the subsequent elements share the same coordinate. For example, tuple (**2**,6) at rank R denotes that elements at rank K's array at indices [6,8) all share the same R coordinate of **2** (the upper bound 8 is indicated by the next tuple, (**4**,8)). The array at the bottom rank (S) consists of tuples of coordinates (at rank S) and values.

Unlike uncompressed data structures, compressed formats can be traversed efficiently only in some orders. For example, in CSF, traversing all ranks sequentially is efficient. These are called *concordant traversals* [39]. By contrast, other traversals, i.e., *discordant traversals*, are inefficient. For example, accessing elements at random in CSF would require a bisection search.

To be efficient, sparse CNN dataflows should maximize the use of concordant traversals. This requires a careful codesign of the dataflow and data structures, and makes some operations harder. For example, tiling and halos are more complex with a representation like CSF. Specifically, Fused-Layer's tiled dataflow [3] would require discordant traversals of halos even if activations were produced in a tiled CSF format, causing needless fetches. Instead, IS-OS traverses activations in *wavefronts* (Fig. 3), achieving concordant traversals of all compressed data structures, making IS-OS more efficient on sparse CNNs.

Sparse accelerators: SCNN, SparTen, and GoSPA are representative sparse single-layer CNN accelerators (Sec. VII discusses other accelerators). SCNN [34] exploits both weight and activation sparsity. Its input-stationary dataflow allows efficient concordant traversal of input activation and weights. SparTen [17] improves on SCNN by introducing a novel intersection-based PE design and a load balancing scheme. Sparse input activations and weights are represented as bit masks, enabling efficient dot products in the PEs. GoSPA [12] improves on SparTen through a novel implicit intersection mechanism that uses statically known weights to avoid fetching input activations that will not produce output activations. These accelerators execute sparse CNNs layer by layer. Their input/output stationary dataflows create long output/input activation reuse. Thus, pipelining multiple layers would require a large amount of on-chip storage.

III. IS-OS Dataflow

We now present the input-stationary-output-stationary (IS-OS) dataflow for efficient pipelining in sparse CNNs. For ease of understanding, Sec. III-A first presents IS-OS in a *dense* setting. Sec. III-B then explains the *sparse* IS-OS dataflow.

A. Dense IS-OS dataflow

Input-output behavior: A key characteristic of IS-OS is that it consumes input activations and produces output activations in the same order, resulting in thin *wavefronts* between pipeline stages instead of wide tiles. Fig. 3 from Sec. II showed this basic idea for the case with a single input and output channel. Fig. 6 shows this input-output behavior in more detail for the general case, with multiple input and output channels. Fig. 6 highlights the input and output wavefronts for one layer at two different points in time. Each wavefront is a single column, i.e., a vertical slice of a single activation plane at a specific channel. Over time, the IS-OS layer consumes the input wavefront by wavefront, advancing first through the input channels, C and then in the horizontal input dimension, W. The output wavefront is always S positions behind the input wavefront in the horizontal output dimension, Q, i.e., it lags by the horizontal size of the filters. This is because, in a convolution, the output wavefront at horizontal dimension $P = i - S$ is dependent on input wavefronts with horizontal dimension W in $(i - S, i]$. Thus, output wavefronts are produced when their dependencies are cleared.

Fig. 7 shows the input-output behavior at a finer granularity, showing how wavefronts are produced and consumed channel by channel. Fig. 7 shows a single intermediate layer. It's important to note that an IS-OS layer will consume input wavefronts and produce output wavefronts independently within the channels.

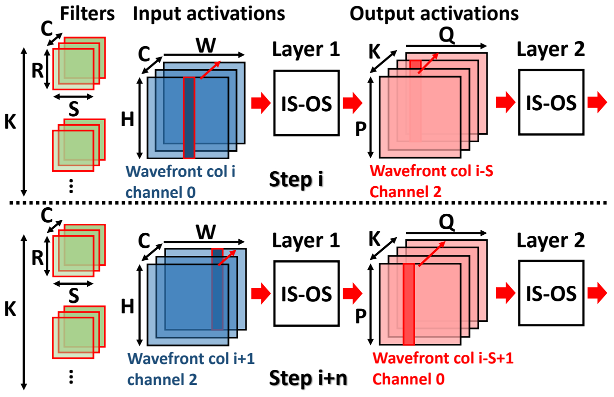


Fig. 6: Input-stationary-output-stationary (IS-OS) dataflow.

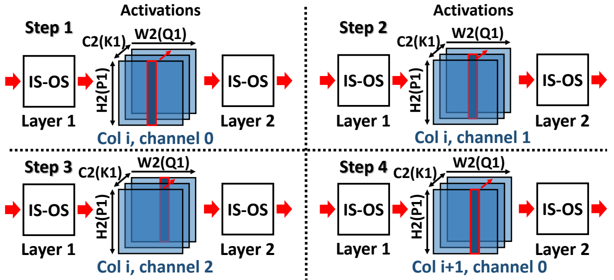


Fig. 7: Step by step example of IS-OS dataflow.

```
i = Tensor(shape=[H,W,C]) # compressed
f = Tensor(shape=[C,R,K,S]) # compressed
o = Tensor(shape=[P,Q,K]) # compressed

# IS frontend
tmp1 = Tensor(shape=[H,R,K,Q]) # uncompressed on Q
for h = [0,H): # spatial
 for w = [0,W): # pipeline
  for c = [0,C):
   for r in [0,R):
    for k in [0,K):
     for s in [0,S): # spatial
      tmp1[h,r,k,w-s] += i[h,w,c] * f[c,r,k,s]
tmp1_t = tmp1.transpose() # [H,R,K,Q] -> [K,Q,H,R]
# OS backend
tmp2 = Tensor(shape=[P,K,Q]) # compressed
for p = [0,P): # spatial
 for k = [0,K):
  for q = [0,Q): # pipeline
   for r in [0,R):
    h = p + r
    tmp2[p,k,q] += tmp1_t[k,q,h,r]
o = tmp2.transpose() # [P,K,Q] -> [P,Q,K]
```

Fig. 8: Loop nest for the IS-OS dataflow. Black code describes the dense IS-OS dataflow. Red code denotes additions for the sparse IS-OS dataflow.

In other words, the input and output channels are not the same at a given point in time, as the two diagrams in Fig. 6 show. But they are always S columns apart.

IS-OS loop nest: Fig. 8 shows the IS-OS loop nest, i.e., the implementation of each IS-OS block in Fig. 6.

Unlike prior dataflows, we write IS-OS *not* using a single loop nest, but two: an *input-stationary (IS) frontend* and an *output-stationary (OS) backend*. Importantly, frontend and backend are *pipelined*, using a small amount of intermediate storage to communicate. This intra-layer pipelining of frontend and backend enables the input-output behavior described above, which in turn enables inter-layer pipelining.

While for the dense case, the code in Fig. 8 could be written as a single loop nest, this would require costly discordant traversals for the sparse case. Instead, Fig. 8 is written using two loop nests that access all data structures in a single order, plus two transposition operations. The transpositions enable efficient concordant traversal of the data structures, and also have a straightforward sparse implementation (mergers).

The IS frontend (top loop in Fig. 8) consumes input activations wavefront by wavefront (proceeding through channels, C and then input activation horizontal dimension, W). Each input value at channel C is multiplied with $K \times R \times S$ filter values corresponding to the K output channels. This produces $K \times R \times S$ values per input element. These values are buffered and accumulated along the S (horizontal filter) dimension to produce a stream of partial results, which are written to the `tmp1` array.

The OS backend (bottom loop in Fig. 8) produces output activations wavefront by wavefront (proceeding along output channels, K, and output activation horizontal dimension, P). To do this, the backend consumes partial results, which the frontend placed in `tmp1`, and accumulates them along the R (vertical filter) dimension. However, `tmp1` has dimension order $H \times R \times K \times Q$, so reducing across R would require non-contiguous accesses. To avoid these, `tmp1` is transposed (i.e., its dimensions are permuted) between the frontend and backend, creating a `tmp1_t` array with dimension order $K \times Q \times H \times R$, so that the backend can accumulate across the innermost dimension. Similarly, the OS backend performs one final transpose to leave the output channel, K, as the innermost dimension, so that output wavefronts are produced channel by channel, then column by column (consistent with how inputs are consumed).

It is important to realize that the intermediate arrays (`tmp1`, `tmp1_t`, and `tmp2`) need not be produced completely before being consumed (which would occupy a lot of storage): pipelining the frontend and backend, and pipelining the backend with the frontend of the next layer, means that all intermediates are pipelined across the Q dimension (output activation column). Pipelining still needs some intermediate storage to hold partial results (essentially the size of $K \times R \times S$ output activation wavefronts), but this is reasonably small in practice.

B. Sparse IS-OS dataflow

The previous section has introduced the dense IS-OS dataflow in a way that lends itself well to sparsity. We describe the high-level effects of sparsity here, and present the full ISOSceles implementation of this sparse dataflow in the next section.

Assume that input activations, output activations, and filters are in a compressed format like CSF (Sec. II-B). This brings several benefits. First, this reduces the sizes of all structures, as only nonzeros are stored. Second, because the code in Fig. 8 traverses all data structures sequentially, resulting in efficient concordant traversals on CSF structures; and transposes of sparse data structures, as we will see later, can be efficiently implemented with high-radix mergers. Third, sparse IS-OS avoids ineffectual work: specifically, only nonzero inputs and weights are multiplied to produce partial results.

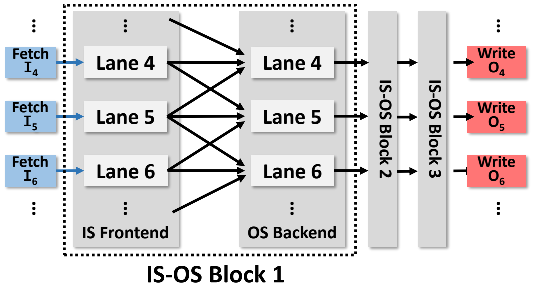


Fig. 9: ISOSceles spatial components.

Despite these benefits, sparsity also brings challenges. Specifically, it introduces large and fast variations of work across layers, as the number of effectual multiplications in a layer is unknown in advance. This requires techniques to load balance hardware resources across layers to avoid low utilization.

Finally, since input and output activations are sparse, input and output wavefronts need not be simple columns as Fig. 6 showed: because zeros are not represented, each wavefront will be a *wavy line* consisting of the earliest unprocessed nonzero along the input channel dimension at the input (and output channel dimension at the output). This lets different parts of the layer run at slightly different positions, with synchronization dictated only by data dependences. Though this is difficult to visualize, the hardware is simple, since it just streams in nonzero values and their coordinates.

IV. ISOSceles Architecture

We now present the ISOSceles architecture, which efficiently implements the IS-OS dataflow and pipelines multiple layers. For ease of understanding, Sec. IV-A first presents ISOSceles's structure. This shows the key implementation techniques without concerns for limited resources or low utilization. Sec. IV-B shows how time-multiplexing helps ISOSceles achieve high hardware utilization. Finally, Sec. IV-C discusses how ISOSceles can handle both large and small layers through tiling and spatial/temporal mapping.

A. ISOSceles spatial structure

Overview: Fig. 9 shows a version of ISOSceles that is *fully spatial*, i.e., without time-multiplexing. Each ISOSceles IS-OS block processes one neural network layer. To pipelines multiple layers, this implementation handles each layer with a separate hardware IS-OS block, and blocks communicate intermediate activations through FIFO queues. This spatial design is a stepping stone to ISOSceles's full implementation, which we will extend with time-multiplexing in Sec. IV-B to handle multiple layers with a single IS-OS block.

Because each layer uses a different set of filters, in this spatial design each IS-OS block uses a separate filter buffer that stores all the layer's filter weights on-chip. (We will later share a single filter buffer across layers, and tile large layers whose weights do not fit on-chip.) ISOSceles uses the Compressed Sparse Fiber (CSF) [25, 37] format for all data structures.

IS-OS block overview: Each ISOSceles IS-OS block consists of an IS frontend and an OS backend that implement the two loop nests described in Fig. 8. The frontend and backend are

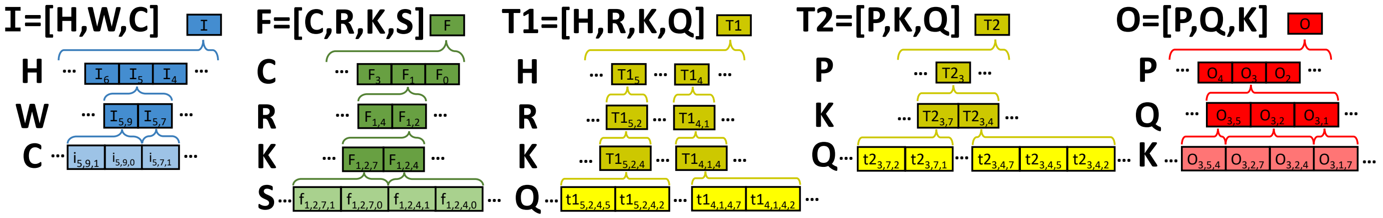


Fig. 10: Data structures used in ISOSceles. I: input activation, F: filter weight, T_1, T_2: partial results, O: output activation.

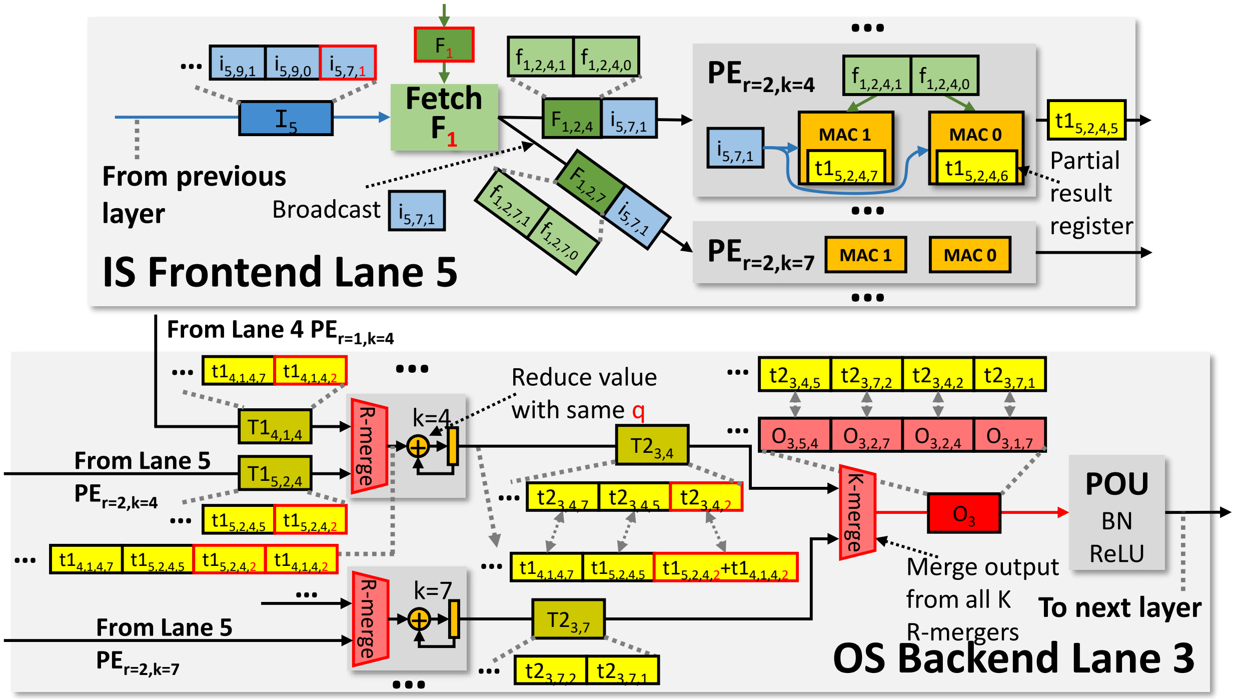


Fig. 11: ISOSceles IS frontend and OS backend.

pipelined to reduce intermediate storage. Internally, frontend and backend both consist of a number of *lanes*. The vertical dimension of activations (H for the frontend and P for the backend) is spatially mapped across lanes, so each lane handles a single row of input or output activations. (For now, assume we have enough lanes to accommodate the whole layer; Sec. IV-C describes how to handle larger layers.)

Overview of lanes and cross-lane communication: Each lane of the IS frontend reads across the horizontal input activation dimension (W) and performs a partial convolution, accumulating across the horizontal filter dimension (S). Each lane of the OS backend collects partial results from the frontend and produces results along the horizontal output activation dimension (Q). To do this, each OS backend lane accumulates partial results along the vertical filter dimension (R). These partial results are generated by different lanes of the IS frontend, so each backend lane consumes values from R nearby frontend lanes. For example, Fig. 9 shows the cross-lane communication pattern with $R = 3$ filters: each backend lane consumes partial results from the $R = 3$ surrounding frontend lanes, e.g., backend lane 5 consumes partial results from frontend lanes 4, 5, and 6. Modern CNNs typically use small filter kernels (e.g. $R = 1, 3, 5$), so the NoC overhead of this cross-lane communication is limited. Each backend lane connects with the five closest frontend lanes directly. Larger kernels ($R > 5$) can be executed through multi-hop communication between backend lanes. Furthermore, lanes are decoupled through queues to tolerate load imbalance and on-chip memory access latency.

Main memory accesses: ISOSceles interfaces with off-chip memory through input activation fetchers and output activation writers, shown in Fig. 9. Each per-lane fetcher reads one input activation row (dimension H) of the first layer from main memory. No other off-chip reads are needed in the entire pipeline because all subsequent IS-OS blocks consume output activations from the previous IS-OS block. Similarly, each per-lane writer streams one output activation row (dimension P) of the last layer to main memory.

The IS-OS dataflow enables efficient concordant traversals of input and output activations. Therefore, fetchers and writers are implemented as simple FSMs that traverse compressed tensors (like prior work [22, 42, 43]). To hide memory latency, they are decoupled from the main execution pipeline using queues.

We now detail the implementation of frontend and backend lanes using a concrete example, shown in Fig. 11. Fig. 10 shows the format of the sparse tensors used in the example.

IS frontend lane: Each frontend lane computes a partial convolution of one input activation row and its associated weights. The lane *(1)* consumes the input activation row element by element, *(2)* fetches the filter weights based on the input activation's channel (C), and *(3)* sends the activation value and filter weights to a set of PEs, which perform the multiply-accumulates.

Fig. 11 illustrates this process for frontend lane 5. The frontend consumes input activation sub-tensor I_5 (shown as a stream of nonzeros $I_{5,7,1}$, $I_{5,9,0}$...), one element per cycle. Since the current input activation is from input channel 1, the filter subtensor F_1 is fetched. The filter fetcher fetches the corresponding weights ($F_{1,2,4}$, $F_{1,2,7}$...) from the shared filter buffer. These weights are associated with different rows, r, and output channels, k, and are sent to different PEs.

The filter buffer is shared across lanes, so it needs to support the highest throughput in our design (serving up to 4096 elements/cycle in our implementation). We achieve this cheaply with three techniques. First, since each input fetches many weights, the filter buffer uses wide words to supply many weights in parallel. Second, the filter buffer is heavily banked, especially along input channels. Third, when multiple lanes request weights for the same input channel, we *coalesce* their requests and serve them with one access, avoiding stalls.

Each frontend lane has $R \times K$ PEs. $\mathrm{PE}_{r,k}$ is responsible for handling a filter sub-tensor with a specific row, r, and output channel, k. In our example, filter $F_{1,2,4}$ (along with input activation $I_{5,7,1}$) is sent to $\mathrm{PE}_{r=2,k=4}$. The PE conducts a vector (weight) scalar (input activation) product and accumulates results in partial result registers $t1_{5,2,4,6}$ and $t1_{5,2,4,7}$. Upon receiving input activation $I_{5,7,1}$, the PE outputs partial result $t1_{5,2,4,5}$ if it is not zero. This is because input ($I_{5,7,1}$) at column 7 can only contribute to output activation at column 6 and 7 (assuming a length $S = 2$ filter). The PE then knows that the partial convolution on column 5 is completed and thus pushes partial result at column 5 ($t1_{5,2,4,5}$) to its output queue, to be consumed by the backend. Each PE only holds S partial results, $t1_{5,2,4,6}$ and $t1_{5,2,4,7}$. By generating partial results $T1$ (`tmp1` in Fig. 8) in a pipelined fashion, ISOSceles never needs to materialize $T1$ entirely and stores it compressed on-chip. As described in Sec. III-A, the uncompressed size needed to hold partial results is a reasonably small $R \times K \times S$ per lane.

OS backend lane: Each OS backend lane produces one output activation row (along dimension Q) by accumulating partial results $t1$ produced by the frontend. Accumulation happens along the R (vertical filter) dimension. Each backend lane sources partial results from R (adjacent) frontend lanes. To do this, the lane *(1)* consumes partial result tensors from appropriate frontend queues, *(2)* reorders them using a set of *mergers* to leave H and R as the innermost dimensions, *(3)* accumulates partial results over the R dimension to complete the convolution, and *(4)* serializes all accumulated values using a final merger. The lane also implements batch normalization and non-linear activation functions to produce the final output activations.

Fig. 11 shows backend lane 3, which is responsible for producing output row 3. We first focus on output channel $k = 4$: *(1)* The lane consumes partial result sub-tensors $T1_{4,1,4}$ (from frontend lane 4 $\mathrm{PE}_{r=1,k=4}$) and $T1_{5,2,4}$ (from frontend lane 5 $\mathrm{PE}_{r=2,k=4}$) (we omit partial results from other lanes for simplicity). *(2)* The R-merger at $k = 4$ reorders $T1_{4,1,4}$ ($h = 4, r = 1$) and $T1_{5,2,4}$ ($h = 5, r = 2$) so that H and R appears at the innermost dimension in the merged stream. As a consequence Q appears at the outermost dimension ($t1_{4,1,4,2}$, $t1_{5,2,4,2}$, $t1_{5,2,4,5}$, $t1_{4,1,4,7}$...), which reduces memory footprint for the upcoming reduction process. *(3)* The reducer accumulates partial results (with the same column index, e.g., $q = 2$) over the R (vertical filter) dimension using a simple adder. This completes the convolution. $t1_{4,1,4,2}$ and $t1_{5,2,4,2}$ are added together (and so are subsequent elements with same column index q). The resulting $T2_{3,4}$ is output activation element of row 3 at channel $k = 4$.

There are K (output channel dimension) R-mergers, each working to produce one row of a specific output channel k. In our example of lane 3, other than $T2_{3,4}$ generated by R-merger ($k = 4$), output row 3 at channel 7 ($T2_{3,7}$) is generated by R-merger ($k = 7$). T_2 in lane 3 is organized as a compressed tensor of shape $K \times Q$. However, the IS frontend of the next layer consumes activations channel by channel then column by column (K is the innermost loop). *(4)* A final K-merger serializes all K accumulated streams ($T2_{3,4}$, $T2_{3,7}$...) so that K is the innermost loop at the output activation O_3. This emits output in the same order they're consumed by the next layer. Finally, O_3 is passed to the Point-wise Operation Unit (POU), including batch normalization (BN) and ReLU (increasing activation sparsity). The final output activation row is then sent to the next layer.

The backend uses cheap *scalar* mergers, which suffice because each frontend lane consumes a single element per cycle. R-mergers have a low radix and are implemented using a combinational comparator tree [43]. K-mergers need a higher radix ($K = 256$), so we implement them with a pipelined min-heap [5].

B. Time-multiplexing in ISOSceles

The spatial design described so far (Fig. 9) uses one IS-OS block per layer and one lane per activation row. Fig. 12 shows the ISOSceles design with time-multiplexing support, which instead has *a single IS-OS block* with a *fixed number of lanes*. Multiple lanes are pipelined *temporally* instead of spatially, by *time-multiplexing* them over the single IS-OS block.

In this section, we discuss the rationale and changes needed to temporally pipeline multiple layers. In Sec. IV-C, we discuss how to use a fixed number of lanes efficiently.

The key motivation for temporal pipelining is that the spatial approach, where each IS-OS block handles a single layer, suffers from severe underutilization. First, sparsity introduces underutilization within each layer. For example, the MAC units in the PE arrays often spend cycles idle because filter weights are often zero. Second, work imbalance across layers causes underutilization too: each layer requires a different amount of work (e.g., because layers use filters with different sizes and sparsities), and this amount of work changes over time (e.g., due to zero input activations). When consecutive layers are pipelined spatially, using multiple IS-OS blocks, one of them becomes the bottleneck and leaves others even more underutilized. By time-multiplexing layers over a single IS-OS block, we can avoid both of these issues.

Effective time-multiplexing requires two extensions: the addition of per-layer *contexts* that hold the intermediate values

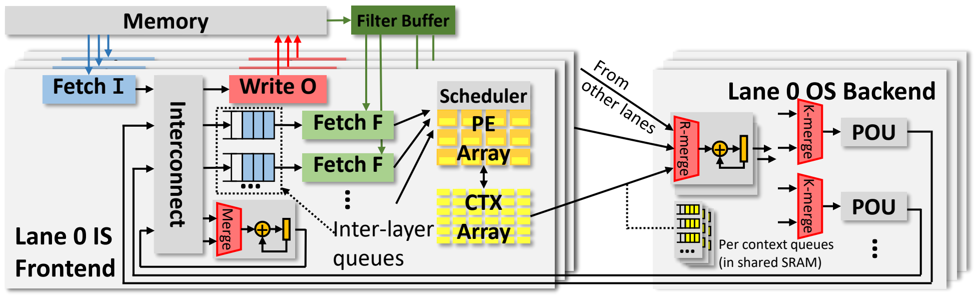
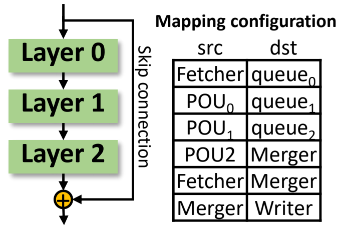


Fig. 12: ISOSceles with time-multiplexing.



Mapping configuration

src	dst
Fetcher	queue_0
POU_0	queue_1
POU_1	queue_2
POU2	Merger
Fetcher	Merger
Merger	Writer

Fig. 13: Mapping ResNet block to ISOSceles.

for each layer, and *dynamic scheduling* to decide which layer (context) to process among the ready ones at different points in the pipeline. These modifications are the same as those needed to make a processor pipeline multi-threaded.

Contexts: Fig. 12 shows the addition of per-layer contexts at different points in the frontend and backend lanes. These are SRAM memories that hold the distinct intermediate state of each layer. The main points where this happens are at the PE arrays in the frontend (where contexts hold partial results that are then consumed by backend lanes), and the R-mergers (where contexts hold the $T1$ tensors being transposed).

Our implementation supports pipelining 2–16 layers, so our contexts increase state by a factor of 16. Different contexts share the same physical SRAM memories, so when fewer contexts are used, more space can be allocated to each context (e.g., each PE array context can hold more partial results, increasing decoupling between frontend and backend lanes).

Replicated fetchers, K-mergers, and POUs: Since time-multiplexing increases utilization, the very first and last stages of each lane can bottleneck throughput. Therefore, as Fig. 12 shows, we replicate the filter fetchers in the frontend, and both the final K-mergers and POUs in the backend, instead of time-multiplexing them (using 16 replicas in our implementation). This allows each frontend/backend lane to consume/produce multiple elements per cycle, and is a small cost (Sec. V).

PE array: Because convolution layers have different filter sizes ($S = 1,3,5$), using a fixed number of MAC units per PE (as in the spatial design) causes underutilization. For example, if the PE is designed to handle $S = 5$, when a layer with $S = 1$ is mapped to the PE, 80% of the MAC units are idle, creating significant internal fragmentation. ISOSceles addresses this by having coarse-grain PEs, where each PE contains more than S MAC units (e.g. 8 in our implementation). The filter fetcher now sends a long vector of compressed weights (which may span multiple r and k) along with the input activation value to the PE each cycle. The PE performs the same scalar vector products and accumulates them to the wider partial result registers. This coarse-grain PE design also benefits the context array's design: it adopts the same wide-word design as the filter buffer, because partial results are read/written contiguously to the context array from the PE. The PE double-buffers partial results to hide the access latency of the context array SRAM.

Dynamic scheduling: ISOSceles performs dynamic scheduling of the PE arrays to control the overall throughput of stages. Because the amount of work per stage changes over time, a static schedule that apportions a fixed number of PEs to each layer is insufficient and would cause load imbalance. However, doing cycle-by-cycle dynamic scheduling would be too expensive, as there are many PEs and multiple contexts. Instead, dynamic scheduling is performed periodically: every 100 cycles, the scheduler reallocates the PEs among layers based on the demand of each layer in the prior 100-cycle interval. The scheduler monitors the number of MACs requested by each layer in an 100-cycle interval and allocates PEs to layers proportionally to their demand. In practice, we find this dynamic scheduling algorithm to effectively smooth out load imbalance.

Programmable interconnect: Diverse CNN models can be mapped to ISOSceles by configuring the interconnect shown in Fig. 12. Fig. 13 shows an example of how we map one ResNet block to ISOSceles. Each ResNet block (left) consists of a three-layer main branch and a skip connection. At the end of the block, activations from the main branch and the skip branch are added together. Each inter-layer connection is directly translated into a hardware unit connection. For layer0, the fetcher pushes input activation from off-chip to queue0. Queue0 connects with the pipeline (from intersect0 to POU0) responsible for the inference of layer 0. The output of layer 0 is pushed from POU0 to queue1 (to be consumed by layer 1). The same is true for other layers.

Configuring ISOSceles: Beyond mapping layer connections onto the interconnect, ISOSceles hardware is configured with the layer information. This includes configuring the activation fetchers and writers, filter fetcher, context array, and POUs. We currently write this manually, but this can be automated with simple extensions to deep learning frameworks like PyTorch.

C. Handling different layer sizes and types

ISOSceles is flexible enough to process convolutional layers of widely varying sizes. Because ISOSceles streams through input and output activation wavefronts horizontally (i.e., over the W and Q dimensions), there is no limit to how large these can be. But the vertical dimensions (H and P) are spatially mapped across lanes and ISOSceles uses a fixed number of frontend and backend lanes (64 in our implementation). In addition, ISOSceles handles other layer types like depth-wise convolution and fully-connected layers with minimal hardware changes.

Handling large layers: In ISOSceles, each frontend lane consumes one input activation row, and each backend lane produces one output activation row. When the number of rows (P) in the activation exceeds the number of hardware lanes, ISOSceles

TABLE I
Configuration of the ISOSceles system.

Lane Parameter	Value	ISOSceles Parameter	Value
Multiplier width	8b	# Lanes	64
Accumulator width	16b	Filter buffer	1MB
# MAC units	64	DRAM bandwidth	128GB/s
Context array	8KB	Summary	
Queues	8KB	Total # MAC units	4096
# Merger	16	Total memory size	2MB
Merger radix	256	Frequency	1GHz

TABLE II
Area breakdown of ISOSceles.

ISOSceles	Area (mm^2)	Lane	Area (mm^2)
64 Lanes	18.4	64 MAC Units	0.069
Filter buffer	7.5	Mergers	0.060
		Buffers	0.121
		Fetcher	0.010
		Crossbar	0.021
		Others	0.007
Total	26.0	Total	0.288

TABLE III
Configuration of the SparTen system.

Cluster Parameter	Value	SparTen Parameter	Value
Multiplier width	8b	# Clusters	64
Accumulator width	16b	Filter buffer	1MB
# MAC units	64	DRAM bandwidth	128GB/s
Buffers	64KB	Summary	
Frequency	1GHz	Total # MAC units	4096
		Total memory size	5MB

tiles the output row dimension P to accommodate it. Tiling P is also conducted when the context array memory requirement (proportional to P) exceeds its capacity. Without considering striding and padding, due to the nature of convolution, input activations ($H \times W$) are slightly larger than output activations ($P \times Q$). For each output tile, the corresponding input activation tile is slightly larger. Therefore, for two disjoint output tiles, their corresponding input tiles will have some overlapping rows at the boundary, also referred as input *halos* [15, 34]. When processing the output tiles, ISOSceles needs to read the input halos from off-chip memory. Traversing these halos in the compressed input are concordant since each row of input belongs to a separate sub-tensor. With a large number of lanes, halos are a small portion of the input activation, so their effect on off-chip memory traffic and performance is limited.

When the sizes of filters in large layers exceed filter buffer capacity, ISOSceles runs them layer by layer and, if necessary, tiles the output channel K dimension so that each tile fits on-chip. We show later (Sec. VI-C) that, even in this single-layer execution mode, ISOSceles is still incurs less traffic than prior accelerators thanks to the IS-OS dataflow.

Handling small layers: When P is significantly smaller than the number of lanes (e.g., 16 with 64 lanes), mapping a whole activation row to one lane is inefficient (e.g., leaving 75% of lanes idle in our example). ISOSceles solves this by mapping one row to multiple adjacent processing lanes. In the frontend, at every step, one input activation element is consumed and the corresponding weight sub-tensor ($K \times R \times S$) is fetched. Instead of sending the input activation and weight sub-tensor to one frontend lane, ISOSceles splits the weight sub-tensor according to its output channel dimension K and dispatches each weight partition to one frontend lane. Each frontend and backend lane only handles a subset of K and all backend lanes in aggregate handle all output activations. In the loop nest view (Fig. 8), ISOSceles makes the K dimension partially spatial. Reflected in Fig. 7, when P=16 and we have 64 lanes, we consume 4 columns in parallel to fill all lanes.

Handling other layers: The POU handles point-wise layers like batch normalization and ReLU. Depth-wise convolution is supported by simply disabling input channel C accumulation and fetching filters from only one output channel K per input activation. Grouped convolution can be supported with a similar mechanism. ISOSceles executes fully-connected layers in an input-stationary way, reusing most of the frontend structure. For batch 1 inference, it performs an SpMV between the sparse weight matrix and input activation vector. Weights are streamed from DRAM (as they exhibit no reuse). All frontend lanes share the same input activation; each lane processes a weight sub-column to generate a partial result sub-column. The operation is completed once all inputs are consumed. Fully connected layers bypass the backend logic and directly send output from the context array. Global average pooling is supported by treating it as a convolution where kernel size matches input size.

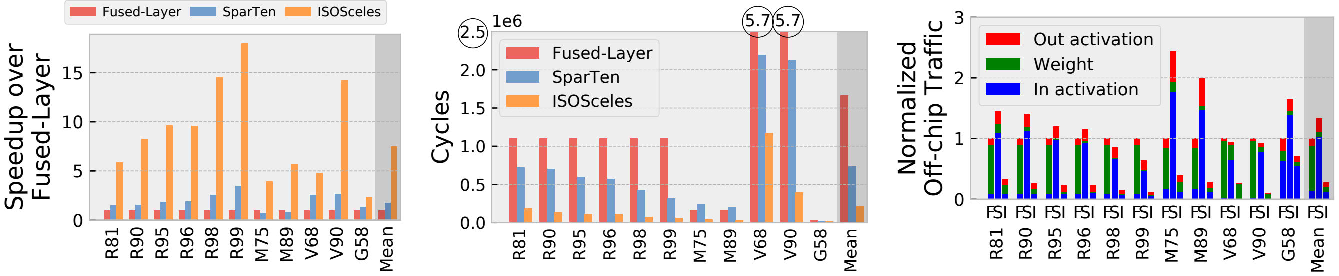
V. Experimental Methodology

System: We build a cycle-level simulator to evaluate ISOSceles. Table I shows its configuration. Our design consists of 64 lanes (frontend and backend), an 1 MB global filter buffer, and an 128 GB/s High-Bandwidth Memory (HBM) interface. Each lane contains 64 8-bit multipliers, 8 KB of context arrays to hold partial results, 8 KB of queues to support inter-layer pipelining, and 16 radix-256 throughput-1 mergers. The full system has 4096 MAC units with a total of 2 MB on-chip storage.

We have implemented ISOSceles's components in RTL and synthesized them in 45 nm using the FreePDK library [30], with a 1 GHz target frequency. Table II shows ISOSceles's area breakdown by component. Overall, this is a small accelerator even at 45 nm, and ISOSceles would take just 4.7 mm^2 when scaled to 16 nm [34]. By comparison, a single HBM2e interface is about 15 mm^2 [11, 31]. ISOSceles is able to saturate this interface while using a small amount of area.

Baselines: We compare with Fused-Layer [3] and SparTen [17]. Fused-Layer is a *dense* CNN accelerator that pipelines multiple layers. We implement its 2D tile-based dataflow and configure it to have similar area as ISOSceles, using a 2.5MB filter buffer and the same memory bandwidth and number of MAC units as ISOSceles. SparTen is a state-of-the-art accelerator for *sparse* CNNs. We model its output-stationary dataflow, memory traffic, and layer-by-layer execution strategy for sparse CNNs. We enhance our SparTen baseline with GoSPA's activation filtering optimization [12] to reduce further traffic (Sec. II). Table III lists its configuration. SparTen is sized to match ISOSceles's bandwidth and number of MAC units. However, SparTen requires 5 MB (over 2× more) on-chip storage.

Workloads: We use ResNet-50 [20], MobileNetV1 [23], VGG-16 [36], and GoogLeNet [40] as representative sparse CNNs on ImageNet [13]. ResNet-50 and MobileNetV1 are pruned using STR [26] to achieve 6 different levels of weight sparsity for ResNet-50 (81%, 90%, 95%, 96%, 98%, and 99%) and 2 levels for MobileNetV1 (75% and 89%). We prune VGG-16 and GoogLeNet with magnitude-based pruning so that their weight sparsities (68% and 58%) match prior work [17, 34]. We further emulate an aggressive version of VGG-16, where 90% of the weights are pruned to show that increasing weight sparsity has



(a) Speedup over Fused-Layer.

(b) Cycles comparison.

(c) Off-chip traffic of ISOSceles (I) and SparTen (S) normalized to Fused-Layer (F).

Fig. 14: ISOSceles speedups, and comparison of cycles and off-chip traffic across CNNs. R81-R99: ResNet-50 w/ 81%-99% weight sparsity. V68, V90: VGG-16 w/ 68% and 90% weight sparsity. G58: GoogLeNet w/ 58% weight sparsity. M75, M89: MobileNetV1 w/ 75% and 89% weight sparsity.

TABLE IV
PIPELINEABLE WORKLOADS IN RESNET-50 WITH 96% WEIGHT SPARSITY (R96), *L*: NUMBER OF LAYERS.

Workload name	*L*	layers	Workload name	*L*	layers
l1.0.conv1	4	layer1.0.conv{1,2,3}, layer1.0.downsample	l1.1.conv1	6	layer1.1.conv{1,2,3}, layer1.2.conv{1,2,3}
l2.0.conv1	4	layer2.0.conv{1,2,3}, layer2.0.downsample	l2.1.conv1	3	layer2.1.conv{1,2,3}
l2.2.conv1	6	layer2.2.conv{1,2,3}, layer2.3.conv{1,2,3}	l3.0.conv1	4	layer3.0.conv{1,2,3}, layer3.0.downsample
l3.1.conv1	3	layer3.1.conv{1,2,3}	l3.2.conv1	6	layer3.2.conv{1,2,3}, layer3.3.conv{1,2,3}
l3.4.conv1	6	layer3.4.conv{1,2,3}, layer3.5.conv{1,2,3}	l4.0.conv1	4	layer4.0.conv{1,2,3}, layer4.0.downsample
l4.1.conv1	6	layer4.1.conv{1,2,3}, layer4.2.conv{1,2,3}			

a compounding factor on compute intensity reduction. Overall, we evaluate 11 sparse CNNs (6 ResNet-50: R81, R90, R95, R96, R98, R99; 2 MobileNetV1: M75, M89; 2 VGG-16: V68, V90; 1 GoogLeNet:G58). Fused-Layer runs the dense CNNs on uncompressed data.

Benchmarks: We execute ResNet-50, MobileNetV1 and VGG-16 end-to-end, including all layers (FC, batch normalization, etc.); for GoogLeNet, we evaluate a subset of representative layers. We pipeline layers greedily from the start of network to the end until the storage requirements (filter buffer, context array, or queues) exceed on-chip resource availability. Since pooling layers are not amenable to pipelining, we run them unpipelined. If a single layer is too large to fit on-chip, it is tiled (following Sec. IV-C).

For ResNet-50, ISOSceles pipelines at ResNet block granularity. For denser ResNet, it usually pipelines 1-2 ResNet blocks (4-6 convolutional layers) including the skip connection. In later layers (e.g. l4.0.conv1), weights are large, so ISOSceles can only execute them layer by layer to not overflow the filter buffer. Table IV shows all the pipelineable layers in R96 (only the first convolution layer and FC layer are not pipelined). Sparser ResNet-50 (R98 and R99) can pipeline more layers, typically 9-15 layers. On average, ISOSceles pipelines 3-5 layers per run for ResNet-50 with different sparsity levels. To model the extra traffic incurred by the skip connection in SparTen, we fuse the skip connection with the last convolutional layer in a ResNet block. For MobileNetV1, we can pipeline 3-7 layers until the context array reaches its capacity and pooling layer serves as pipeline boundary. In VGG-16, ISOSceles tiles the first 4 layers on *P* because their activation heights exceeds the number of lanes. Then we can pipeline 2-3 convolutional layers until the pooling layer. For the sparser VGG (V90), we further tile on *P* for convolutional layers in the middle (even though their activation heights are smaller than number of lanes) to reduce the context array capacity requirement for each layer. In this way, ISOSceles can turn 6 non-pipelineable convolutional layers (`features.24-40`) into two pipelines (each containing a group of three convolutional layers). Finally, we evaluate the Inception 3a block in GoogLeNet. Branches 2 and 3 (each containing 2 layers) are pipelined, and the single-layer branches 1 and 4 are executed separately. We evaluate inference without batching, since it will provide no benefit for these designs.

VI. EVALUATION

A. *Performance and memory traffic*

Fig. 14a reports the speedups of SparTen and ISOSceles over Fused-Layer on all 11 sparse CNNs. Fused-Layer and ISOSceles pipeline multiple layers (as described in Sec. V), but SparTen does not. SparTen and ISOSceles leverage weight and activation sparsity, but Fused-Layer does not.

By leveraging sparsity, ISOSceles outperforms Fused-Layer by gmean 7.5× (and up to 18.0× on R99). Speedups grow with sparsity, as sparser networks have less compute and memory traffic. Furthermore, sparser networks have smaller filter weights, which allows ISOSceles to pipeline even more layers (from up to 6 layers in R81, to up to 15 layers in R99). For example, speedups on ResNet increase from 5.9× to 18.0× when weight sparsity grows from 81% to 99%.

By leveraging inter-layer pipelining, ISOSceles outperforms SparTen by gmean 4.3× (and up to 6.7×, on MobileNetV1 with 89% weight sparsity). Speedups on ResNet with various weight sparsity level range from 3.9× to 5.6×. ISOSceles is 5.6× and 6.7× faster than SparTen on MobileNetV1 with two sparsity levels. SparTen performs worse than Fused-Layer on MobileNetV1 due to frequent depth-wise convolutions, which have low compute intensity. Inter-layer pipelining reduces traffic more than sparsity for these layers, resulting in better performance. For VGG-16, ISOSceles is 1.9× and 5.3× faster on the variants with 68% and 90% weight sparsity, respectively. Finally, ISOSceles outperforms SparTen by 1.7× on GoogLeNet.

Fig. 14b reports the number of cycles (lower is better) that Fused-Layer, SparTen, and ISOSceles take to run each sparse CNN, and Fig. 14c shows the total off-chip memory traffic

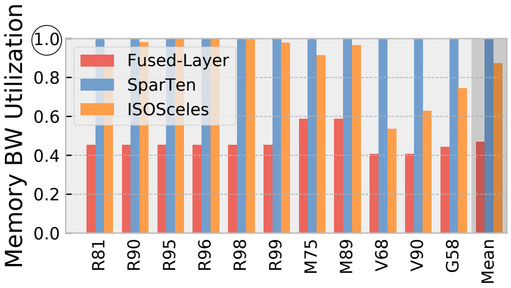


Fig. 15: Memory bandwidth utilization.

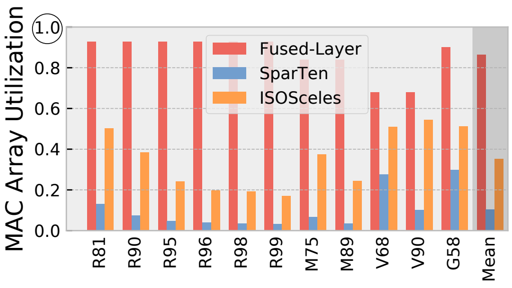


Fig. 16: MAC array utilization.

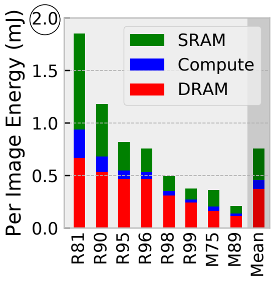


Fig. 17: Energy breakdown.

(lower is better). Each bar is normalized to Fused-Layer's traffic, and is broken down by weight and activation traffic.

Fig. 14c shows that ISOSceles enjoys dramatically lower memory traffic than both Fused-Layer and SparTen, highlighting the synergy between inter-layer pipelining and sparsity. By contrast, there is no clear winner between Fused-Layer and SparTen. Thanks to pipelining, Fused-Layer features little activation traffic. But Fused-Layer is dominated by weight traffic, since weights are dense. SparTen has much lower weight traffic owing to its use of sparsity. But SparTen is dominated by activation traffic since it does not pipeline layers. Except in the sparsest ResNet variants, this limitation makes SparTen incur more traffic than Fused-Layer, sometimes by large amounts (up to 2.4× more in M75), and 1.3× more on average.

By contrast, Fig. 14c shows that ISOSceles achieves both low weight traffic (thanks to sparsity) and activation traffic (thanks to pipelining). ISOSceles enjoys 3.6× less traffic than Fused-Layer, and 4.7× less traffic than SparTen.

B. Architectural analysis

Fig. 15 compares the off-chip memory bandwidth utilization for the baselines and ISOSceles across CNNs. The height of each bar shows the average fraction of time memory is being used; 1.0 means bandwidth is saturated all the time. Fused-Layer only utilizes 47% of the off-chip bandwidth. This is because dense CNN inference has high compute intensity and is compute-bound most of the time. SparTen, on the other hand, always saturates memory bandwidth, and thus is memory-bound on sparse CNNs. This is caused by its single-layer execution strategy, which generates large off-chip transfers of activations. By contrast, ISOSceles reduces memory bandwidth pressure through inter-layer pipelining. As a result, 3 of the 11 networks no longer need the full 128GB/s bandwidth, and are mainly compute-bound. This is why its traffic reductions over SparTen are generally larger than its performance gains. For the other 8 networks, ISOSceles is also bandwidth-bound, so its higher compute intensity translates directly into speedups.

Fig. 16 shows the MAC array utilization rate across all networks. ISOSceles achieves an average 35% MAC array utilization, 3.4× larger than SparTen. This is because the improved compute intensity due to inter-layer pipelining increases utilization. Fused-Layer achieves nearly 100% MAC utilization, again showing that it is compute-bound. Because sparse CNNs have 15× fewer MACs than dense ones, ISOSceles widely outperforms Fused-Layer despite having lower MAC array utilization.

Note that, while Fused-Layer is more compute-bound than ISOSceles, it also incurs much more memory traffic. Therefore, if we increase the number of MACs in Fused-Layer to make it memory-bound (which would cost significant area), ISOSceles would still be about 3× faster than Fused-Layer.

ISOSceles's MAC array utilization decreases when ResNet becomes sparser. This is because higher sparsity reduces the number of effectual operations and makes ResNet more memory-bound. VGG-16, on the other hand, achieves a relatively high MAC utilization, over 50%. We note that 100% average utilization is not achieved for several reasons. First, each network contains a mix of compute- and memory-bound phases. For example, in VGG-16, convolutional layers are heavily pipelined and expose decent reuse so that they are compute-bound. The final three FC layers are essentially SpMV operations and the large weight matrix exhibits no reuse. Therefore, ISOSceles is memory-bound in these layers. The accelerator spends about 60% time in compute-bound phases, and about 40% in memory-bound phases. Second, the dynamic scheduler reallocates MAC units across layers every 100 cycles. The fragmentation caused by load imbalance causes some underutilization.

Fig. 17 shows the energy per end-to-end inference (obtained using a 14/12nm process with commercial tools) on the ResNet-50 and MobileNetV1 CNNs, broken down by component. Energy consumption per image ranges from 0.2mJ to 1.9mJ across CNNs. SRAM energy is mainly associated with filter buffer reads and context array accesses for MAC operations. As CNNs become sparser, operational intensity falls, reducing the contribution of SRAM and compute energy. Overall, DRAM energy dominates, especially for sparser networks. This shows that reducing DRAM traffic (through inter-layer pipelining) is the correct way to improve energy efficiency on sparse CNN inference: due to their much higher traffic, the other accelerators will be even more severely dominated by DRAM energy, even if their on-chip structures are simpler. VGG-16 end-to-end consumes 10.1mJ (V68) and 3.7mJ (V90) per image and shows similar breakdowns. The large model size and number of MACs contribute to the higher energy consumption.

C. Effect of pipelining

We now break down ISOSceles's benefits over SparTen by focusing on one CNN, ResNet-50 with 96% weight sparsity (R96). Beyond ISOSceles and SparTen, we also evaluate ISOSceles-single, which uses the IS-OS dataflow but executes the inference layer by layer instead of pipelining. Fig. 18 reports the cycles

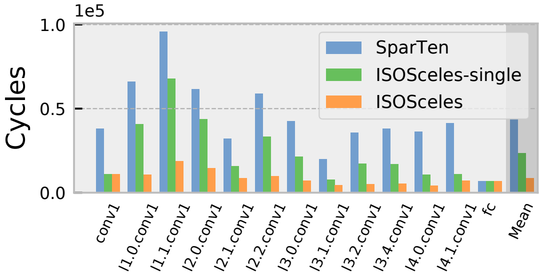


Fig. 18: Execution time (cycles) of different layer groups/pipelines on R96.

taken by each of the different layer pipelines that make up R96 (each group of bars shows an ISOSceles pipeline, or their equivalent group of layers for the other accelerators). Because the number of activation rows in the first layer exceeds the number of lanes, ISOSceles tiles the first layer on P (Sec. IV-C) and executes it tile by tile. Therefore ISOSceles has the same performance as ISOSceles-single. For other parts of the CNN, ISOSceles pipelines 4-6 convolutional layers (1-2 ResNet blocks) based on storage requirements. ISOSceles runs FC layer by layer.

ISOSceles-single reduces execution time by 1.9× over SparTen. Its off-chip traffic reductions match its speedups for all layers, because R96 is memory-bound (as we saw in Fig. 15). This shows that the IS-OS dataflow has merits over prior dataflows by itself. This is because the IS-OS dataflow reads/writes input/output activations once per layer, incurring only compulsory traffic. SparTen's OS dataflow has poor reuse of input activations and may read them multiple times. Tiling in SparTen helps input reuse but it still causes more traffic than compulsory. In this ResNet execution, layers that are not pipelined (first convolution and FC layer), running in ISOSceles-single mode, only account for 16% of execution time.

Comparing ISOSceles and ISOSceles-single shows the benefits of inter-layer pipelining. From Fig. 18, ISOSceles reduces execution time by 2.6× over ISOSceles-single. Since ResNet is memory-bound, traffic reductions are again similar: ISOSceles incurs 2.7× less traffic than ISOSceles-single.

VII. Additional Related Work

Prior work has proposed accelerators that leverage sparsity beyond SCNN, SparTen, and GoSPA (Sec. II-B). Cambricon-X [44] exploits weight sparsity by storing weights compressed on-chip. It uses sparse weights to fetch the corresponding input activations in an irregular way. Eyeriss [7] gates MAC units when an input is 0 to save energy, but this incurs idle cycles. Cnvlutin [2] skips ineffectual multiplications caused by activation sparsity, reducing cycles over Eyeriss. Eyeriss v2 [8] proposes a flexible NoC to handle sparse layers with varying reuse. ESCALATE [27] decomposes convolution using SVD into lower-rank operations: a small set of kernels and a large coefficient matrix that is more amenable to pruning. Its architecture maximizes reuse of these structures. Dual-side sparse tensor core [41] leverages bitmap-based im2col and outer-product based SpGEMM to support weight and activation sparsity in GPU tensor cores.

The work mentioned above exploits value sparsity in CNNs, i.e., eliminating ineffectual work cause by values that are zero. Another line of work focuses on exploiting bit-level sparsity by performing bit-serial computation and skipping zero bits. Bit-pragmatic [1] skips zero bits in input activations and weights. A new encoding scheme and compute schedule reduce its area and energy. Bitlet [28] introduces bit interleaving to condense multiple values (with zero bits) into fewer values (with all 1 bits) to improve the utilization of bit-serial PEs. Bit Fusion [35] leverages the fact that different CNN layers require different bit precision. It proposes a flexible multiplier array that handles variable-precision operations. Bit sparsity is orthogonal to ISOSceles. ISOSceles could be extended to leverage it.

Prior accelerators on sparse matrix-sparse matrix multiplication can also be used to accelerate sparse convolution by leveraging the Toeplitz transformation [39], though at an efficiency cost. ExTensor [22] leverages hierarchical intersection to eliminate ineffectual work. OuterSPACE [32] and SpArch [45] use an outer-product based dataflow. And Gamma [43] and MatRaptor [38] implement Gustavson's dataflow to reduce memory traffic and accelerator area. SpArch and Gamma use high-radix mergers for efficiency, which ISOSceles takes inspiration from. But these accelerators are not designed for CNNs: they cannot pipeline operations, run a single matrix multiplication at a time, and are tuned for very sparse matrices, so they have far less compute throughput (32–64 MACs) than ISOSceles.

While ISOSceles and these prior accelerators target different problems, ISOSceles's techniques are general and should apply to sparse matrix and tensor acceleration. For instance, small changes to ISOSceles would allow it to support Gustavson's dataflow (by using the fetcher, PE array, and K-merger, and bypassing other modules), which pipelines naturally. Supporting sparse matrix-sparse matrix multiplication would allow leveraging ISOSceles's sparse pipelining in other applications, like transformers and sparse tensor algebra.

VIII. Conclusion

Sparse CNNs are dramatically more efficient than dense ones, but also more memory-bound due to their limited reuse. We have shown that inter-layer pipelining is an effective way to increase reuse for sparse CNNs and substantially improve performance. We have presented a novel dataflow, IS-OS, that allows pipelining many CNN layers with a minimal amount of intermediate state; and ISOSceles, an accelerator that implements this dataflow and leverages time-multiplexing to pipeline multiple layers at high utilization, avoiding the load imbalance issues caused by sparsity. ISOSceles outperforms a state-of-the-art accelerator by 4.3× gmean, chiefly by dramatically reducing off-chip traffic by 4.7×.

Acknowledgments

We sincerely thank Nithya Attaluri, Robert Durfee, Fares Elsabbagh, Axel Feldmann, Kendall Garner, Aleksandar Krastev, Hyun Ryong (Ryan) Lee, Quan Nguyen, Nikola Samardzic, Shabnam Sheikhha, Victor Ying, and the anonymous reviewers for their helpful feedback. This work was supported in part by the Semiconductor Research Corporation under contract 2020-AH-2985, and by the National Science Foundation under grant CCF-2217099.

References

[1] J. Albericio, A. Delmás, P. Judd, S. Sharify, G. O'Leary, R. Genov, and A. Moshovos, "Bit-pragmatic deep neural network computing," in *Proc. MICRO-50*, 2017.

[2] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *Proc. ISCA-43*, 2016.

[3] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer CNN accelerators," in *Proc. MICRO-49*, 2016.

[4] K. Ando, K. Ueyoshi, K. Orimo, H. Yonekawa, S. Sato, H. Nakahara, S. Takamaeda-Yamazaki, M. Ikebe, T. Asai, T. Kuroda *et al.*, "Brein memory: A single-chip binary/ternary reconfigurable in-memory deep neural network accelerator achieving 1.4 TOPS at 0.6 W," *IEEE Journal of Solid-State Circuits*, 2017.

[5] R. Bhagwan and B. Lin, "Fast and scalable priority queue architecture for high-speed network switches," in *Proc. of the IEEE Infocom*, 2000.

[6] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proc. ASPLOS-XIX*, 2014.

[7] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: a spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Proc. ISCA-43*, 2016.

[8] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 2019.

[9] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun *et al.*, "DaDianNao: A machine-learning supercomputer," in *Proc. MICRO-47*, 2014.

[10] S. Chou, F. Kjolstad, and S. Amarasinghe, "Format abstraction for sparse tensor algebra compilers," in *Proc. OOPSLA*, 2018.

[11] S. Dasgupta, T. Singh, A. Jain, S. Naffziger, D. John, C. Bisht, and P. Jayaraman, "Radeon RX 5700 Series: The AMD 7nm Energy-Efficient High-Performance GPUs," in *Proc. ISSCC*, 2020.

[12] C. Deng, Y. Sui, S. Liao, X. Qian, and B. Yuan, "GoSPA: an energy-efficient high-performance globally optimized sparse convolutional neural network accelerator," in *Proc. ISCA-48*, 2021.

[13] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *Proc. CVPR*, 2009.

[14] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "ShiDianNao: Shifting vision processing closer to the sensor," in *Proc. ISCA-42*, 2015.

[15] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "Tetris: Scalable and efficient neural network acceleration with 3d memory," in *Proc. ASPLOS-XXII*, 2017.

[16] M. Gao, X. Yang, J. Pu, M. Horowitz, and C. Kozyrakis, "Tangram: Optimized coarse-grained dataflow for scalable nn accelerators," in *Proc. ASPLOS-XXIV*, 2019.

[17] A. Gondimalla, N. Chesnut, M. Thottethodi, and T. Vijaykumar, "SparTen: A sparse tensor accelerator for convolutional neural networks," in *Proc. MICRO-52*, 2019.

[18] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding," in *Proc. ICLR*, 2015.

[19] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Proc. NeurIPS*, 2015.

[20] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. CVPR*, 2016.

[21] Y. He, X. Zhang, and J. Sun, "Channel pruning for accelerating very deep neural networks," in *Proc. ICCV*, 2017.

[22] 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 *Proc. MICRO-52*, 2019.

[23] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[24] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proc. ISCA-44*, 2017.

[25] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," in *Proc. OOPSLA*, 2017.

[26] A. Kusupati, V. Ramanujan, R. Somani, M. Wortsman, P. Jain, S. Kakade, and A. Farhadi, "Soft threshold weight reparameterization for learnable sparsity," in *Proc. ICML*, 2020.

[27] S. Li, E. Hanson, X. Qian, H. H. Li, and Y. Chen, "ESCALATE: Boosting the efficiency of sparse cnn accelerator with kernel decomposition," in *Proc. MICRO-54*, 2021.

[28] H. Lu, L. Chang, C. Li, Z. Zhu, S. Lu, Y. Liu, and M. Zhang, "Distilling bit-level sparsity parallelism for general purpose deep learning acceleration," in *Proc. MICRO-54*, 2021.

[29] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proc. ICML*, 2010.

[30] Nangate Inc., "The NanGate 45nm Open Cell Library," http://www.nangate.com/?page_id=2325, 2008.

[31] NVIDIA, "NVIDIA DGX station A100 system architecture," https://images.nvidia.com/aem-dam/Solutions/Data-Center/nvidia-dgx-station-a100-system-architecture-white-paper.pdf, 2021.

[32] 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 *Proc. HPCA-24*, 2018.

[33] A. Parashar, P. Raina, Y. S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer, "Timeloop: A systematic approach to dnn accelerator evaluation," in *Proc. ISPASS*, 2019.

[34] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "SCNN: An accelerator for compressed-sparse convolutional neural networks," in *Proc. ISCA-44*, 2017.

[35] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, V. Chandra, and H. Esmaeilzadeh, "Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network," in *Proc. ISCA-45*, 2018.

[36] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proc. ICLR*, 2015.

[37] S. Smith and G. Karypis, "Tensor-matrix products with a compressed sparse tensor," in *Proc. of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, 2015.

[38] N. Srivastava, H. Jin, J. Liu, D. Albonesi, and Z. Zhang, "MatRaptor: A sparse-sparse matrix multiplication accelerator based on row-wise product," in *Proc. MICRO-53*, 2020.

[39] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks," *Synthesis Lectures on Comp. Arch.*, 2020.

[40] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proc. CVPR*, 2015.

[41] Y. Wang, C. Zhang, Z. Xie, C. Guo, Y. Liu, and J. Leng, "Dual-side sparse tensor core," in *Proc. ISCA-48*, 2021.

[42] Y. Yang, J. S. Emer, and D. Sanchez, "SpZip: Architectural support for effective data compression in irregular applications," in *Proc. ISCA-48*, 2021.

[43] G. Zhang, N. Attaluri, J. S. Emer, and D. Sanchez, "Gamma: Leveraging Gustavson's algorithm to accelerate sparse matrix multiplication," in *Proc. ASPLOS-XXVI*, 2021.

[44] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-X: An accelerator for sparse neural networks," in *Proc. MICRO-49*, 2016.

[45] Z. Zhang, H. Wang, S. Han, and W. J. Dally, "SpArch: Efficient architecture for sparse matrix multiplication," in *Proc. HPCA-26*, 2020.