SparTen: A Sparse Tensor Accelerator for Convolutional Neural Networks

Ashish Gondimalla*
agondima@purdue.edu
*School of Electrical and Computer Engineering
Purdue University

Noah Chesnut
noachesnut11@gmail.com

Mithuna Thottethodi*
mithuna@purdue.edu

T. N. Vijaykumar*
vijay@ecn.purdue.edu

ABSTRACT

Convolutional neural networks (CNNs) are emerging as powerful tools for image processing. Recent machine learning work has reduced CNNs' compute and data volumes by exploiting the naturally-occurring and actively-transformed zeros in the feature maps and filters. While previous *semi-sparse* architectures exploit one-sided sparsity either in the feature maps or the filters, but not both, a recent *fully-sparse* architecture, called Sparse CNN (SCNN), exploits two-sided sparsity to improve performance and energy over dense architectures. However, sparse vector-vector dot product, a key primitive in sparse CNNs, would be inefficient using the representation adopted by SCNN. The dot product requires finding and accessing non-zero elements in matching positions in the two sparse vectors – an *inner join* using the position as the *key* with a single *value field*. SCNN avoids the inner join by performing a Cartesian product capturing the relevant multiplications. However, SCNN's approach incurs several considerable overheads and is not applicable to non-unit-stride convolutions. Further, exploiting reuse in sparse CNNs fundamentally causes systematic load imbalance not addressed by SCNN. We propose *SparTen* which achieves efficient inner join by providing support for native two-sided sparse execution and memory storage. To tackle load imbalance, *SparTen* employs a software scheme, called *greedy balancing*, which groups filters by density via two variants, a software-only one which uses whole-filter density and a software-hardware hybrid which uses finer-grain density. Our simulations show that, on average, *SparTen* performs 4.7x, 1.8x, and 3x better than a dense architecture, one-sided sparse architecture, and SCNN, respectively. An FPGA implementation shows that *SparTen* performs 4.3x and 1.9x better than a dense architecture and a one-sided sparse architecture, respectively.

CCS CONCEPTS

• **Computer systems organization** → **Neural networks**; **Special purpose systems**.

KEYWORDS

Convolutional neural networks, Sparse tensors, Accelerators

ACM Reference Format:
Ashish Gondimalla, Noah Chesnut, Mithuna Thottethodi, and T. N. Vijaykumar. 2019. SparTen: A Sparse Tensor Accelerator for Convolutional Neural Networks . In *The 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52), October 12–16, 2019, Columbus, OH, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3352460.3358291

1 INTRODUCTION

Advances in convolutional neural networks (CNNs) have resulted in highly-accurate recognition of image data [22, 25, 27, 34]. CNNs comprise many layers (e.g., 20-100) each of which employ numerous *filters* (e.g., 128-1024) to identify features resulting in heavy compute and large intermediate data which raise both compute and memory bandwidth concerns for performance and energy.

To reduce the compute and data volume, previous work [2, 42] exploits the naturally-occurring zeros in the *feature maps* – the output of a CNN layer which is input to the next layer – due to the Rectifier Linear Unit (ReLU), which converts negative values to zeros. These schemes exploit *one-sided sparsity* – zeros only in the feature maps while leaving the filter values unchanged. Targeting *two-sided sparsity*, recent work [20, 21] actively prunes filter values below a threshold to be zeros. Such pruning is followed by retraining to ensure that accuracy is maintained. This sparsity leads to 2-3x memory size reduction, and 4-9x compute reduction (the zeros in the tensors are independent giving quadratic compute reductions). However, current architectures for dense CNNs would be inefficient for sparse tensor computations (e.g., GPGPU's SIMT and systolic array [18, 23], respectively, would have divergence and irregularity problems).

While current sparse microarchitectures have made some advances, they fall short of some of the following design goals:

[G1] *avoid transfer of zeros in both feature maps and filters,*
[G2] *avoid computing with zeros in both feature maps and filters,*
[G3] *maintain accuracy, and*
[G4] *achieve efficient sparse computation.*

Current proposals that handle only *one-sided sparsity* (e.g.,in feature maps [2, 42] achieve only partial compute and data reduction. Though EIE [19] targets sparsity in both filters and feature maps only in fully-connected layers, EIE is performance-equivalent to the one-sided schemes in that it discards zeros in the filter but incurs

Table 1: Design Goals (N/a = not applicable)

Design Goals	Cambricon-X	Cnvlutin	Cambricon-S	SCNN	SparTen
Avoid transfer of all zeros	No	No	No	Yes	Yes
Avoid computing with all zeros	No	No	No	Yes	Yes
Maintain accuracy	Yes	Yes	No	Yes	Yes
Efficient fully- sparse computation	N/a	N/a	N/a	No	Yes

compute idling due to the discard. Cambricon-S [43] employs coarse-grain pruning to force regularity in filter sparsity which does not avoid all zeros and affects accuracy (elaborated in Section 6), and stores and retrieves zeros in the feature maps but discards zeros in the computation. Bit-serial schemes [1, 11, 36] that skip zero bits in the compute by leveraging Booth encoding (1) transfer zero values incurring on-chip SRAM area and energy, and memory bandwidth and energy, (2) incur load imbalance (as does any sparse scheme), and (3) face other issues (Section 6). The *semi-sparse* value schemes are summarized in Table 1. In contrast, Sparse CNN (SCNN) [33], a *fully-sparse* scheme, targets *two-sided sparsity* in both the filters and feature maps. However, there are two shortcomings.

(1) **Inefficient sparse microarchitecture:** Sparse vector-vector dot product, a key primitive in sparse CNNs, requires finding non-zero elements in matching positions in two sparse vectors and accessing those elements – an *inner join* with the position as the key and a single value field. While implementing the one-sided *inner join* is straightforward, the two-sided *inner join* is challenging. SCNN avoids the *inner join* by observing that as a filter 'slides' in the 'X' (height) and 'Y' (width) dimensions over the feature map during the convolution, each (non-zero) filter cell in a given channel (in the "Z" dimension) is multiplied by all the (non-zero) feature map cells in the channel. Because an *inner join* would result in these same multiplications, SCNN performs this *Cartesian product* of multiplications in parallel without any explicit *inner join*. However, these *concurrent* multiplications produce unrelated products destined for *different* output cells while spreading over *different* multipliers the products destined for the *same* output cell. Consequently, the Cartesian-product approach incurs many overheads: (a) numerous implicit barriers (i.e., a 512-channel map may incur 64 barriers to produce one output cell), (b) a high-bandwidth cross bar to route products to partial sums (e.g., 16x32 8-bit crossbar [33]) and many address calculators (e.g., 32 adders for 16 multipliers), and (c) either compute underutilization at the input maps' X-Y borders and for smaller filters (e.g., 1x1 filters), or more buffering for larger filters. Further, this approach is not applicable to non-unit-stride convolutions in CNNs (e.g., *ResNets*) or non-convolutional deep neural networks.

(2) **Load Imbalance:** SCNN employs Eyeriss's *input stationary* approach [8] where the input feature maps are held in the processing elements (PEs) to capture input reuse across the filters which are broadcast to the units to capture filter reuse across inputs (two-way reuse). However, because different input maps inevitably have different sparsities and because all the maps are multiplied by the same filter, the PEs with denser maps would lag behind those with sparser maps by the next filter broadcast. Buffering the later filters would not address this *systematic* load imbalance. The load imbalance can be alleviated if each PE independently fetches the filters but at the complete loss of filter reuse via broadcast. The alternative of holding the filters and broadcasting the input maps (i.e., *filter stationary*) also incurs the same problem (e.g., EIE's input map buffering absorbs temporary imbalance across the filters but does not address this systematic imbalance). Thus, this fundamental reuse-imbalance tension is not addressed by any previous scheme.

Sparse BLAS libraries, including cuSPARSE for GPUs, support sparse linear algebra in High Performance Computing (HPC). However, current HPC hardware and software optimizations for sparse linear algebra (e.g., [5, 44]) do not address these shortcomings.

To that end, we propose *SparTen*, a fully-sparse tensor accelerator architecture, which makes the following contributions:

First, *SparTen*'s microarchitecture achieves efficient *inner join* for full sparsity instead of avoiding the *inner join* like SCNN. *SparTen* provides support for native two-sided sparse execution and memory storage. In contrast, previous architectures except SCNN, being semi-sparse, do not provide this support, whereas SCNN is inefficient. *SparTen* confines the products for one output cell to one multiplier and distributes those for different output cells on different multipliers for parallelism. Thus,*SparTen* avoids SCNN's Cartesian-product overheads. Further, *SparTen* employs clustered, asynchronous *compute units* to handle sparsity. As such, *SparTen* is a general sparse linear algebra accelerator applicable to sparse CNNs using any convolutional stride (and non-convolutional deep neural networks (DNNs) and sparse HPC, though not the focus of this paper), unlike SCNN.

Second, to address load imbalance across the computer units of a cluster, we observe that while input-stationary and filter-stationary approaches may seem equivalent in capturing reuse, *SparTen* employs the latter because the filters do not change during recognition. Based on this observation, *SparTen* employs an offline software scheme, called *greedy balancing (GB)*, which has two variants: a software-only one (GB-S) and a software-hardware hybrid (GB-H). GB-S groups the filters by density at the granularity of whole filters whereas GB-H's grouping is at a finer granularity (e.g., 128 elements) for better balance than GB-S. Both variants' grouping occurs in software and both "shuffle" the filters' output positions within a cluster. GB-S's whole-filter granularity implies that static, offline "unshuffling" by rearranging the next layer's weights suffices. GB-S's "unshuffling" is similar to column combining (CC) [26] but the shuffling criteria are completely different: GB *groups* filters by *density* whereas to improve systolic utilization CC *merges* whole sparse filters by *jigsaw-fitting* filters so that only a few filters have non-zero values in the same tensor positions (Section 6). Further, unlike CC, GB-H's finer granularity requires a *multi-stage permutation network within each cluster* to dynamically unshuffle the finer-grain partial sums to the appropriate output sum within the cluster. Unlike SCNN's high-bandwidth cross bar, this low-bandwidth network routes a result only once per partial sum (e.g., 32 values after 1287 $\approx$ 18 multiply-adds assuming 32 compute units/cluster, 128-element partial sum, and 7x compute sparsity). Thus, *SparTen* efficiently achieves both filter reuse and load balance.

Our simulations show that, on average, *SparTen* performs 4.7x, 1.8x, and 3x better than a dense architecture, one-sided sparse architecture, and SCNN, respectively. *SparTen* also achieves 1.5x and 1.3x lower compute and memory energy than the one-sided architecture. An FPGA implementation shows that *SparTen* performs 4.3x

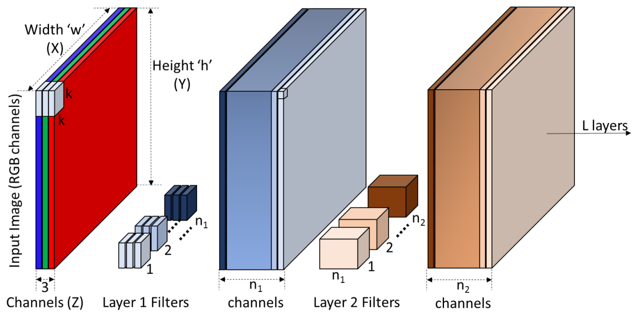


Figure 1: Deep Neural Network

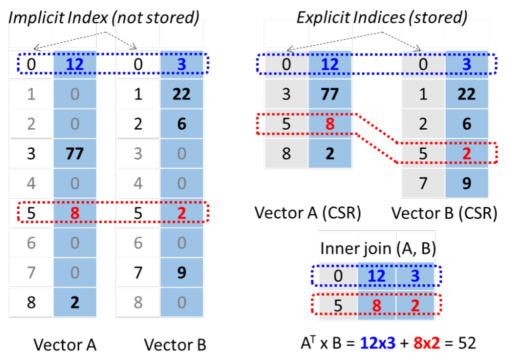

Implicit Index (not stored)
Explicit Indices (stored)
Vector A
Vector B
Vector A (CSR)
Vector B (CSR)
Inner join (A, B)
0 12 3
5 8 2
A^T x B = 12x3 + 8x2 = 52

Figure 2: Sparse vector-vector multiplication using CSR

and 1.9x better than a dense architecture and a one-sided sparse architecture, respectively.

2 SPARSE CNNS

As mentioned in Section 1, there are many layers in a CNN each of which uses many filters to extract features [22, 25, 27, 34]. A layer's output feature map is the next layer's input map. Each layer's input map is a tensor with height h, width w, and depth d which is the number of channels equal to the number of filters in the previous layer (Figure 1). Each filter is also a tensor with typically equal height and width, k, and the same channel count as that of the input map (Figure 1). Each layer's output map dimensions are h x w x n, assuming n filters in the layer.

Each output map cell – a *cell* is a scalar – is a *tensor product* of the input map and a filter (Figure 1). This product can be expressed as a vector-vector multiplication. The next output map cell results from the dot product with the filter "slid over" by a stride along the height and width dimensions, one at a time, in a two-dimensional convolution. The filer does not slide along the channel dimension.

Each layer in a dense CNN results in $h * w * k^2 * d * n$ multiply-adds assuming d input- and n output-channels, ignoring boundary effects. In a dense CNN, each filter cell in a channel is reused by every cell in the input map's corresponding channel ($h * w$ times). Similarly, each input map cell is reused for every filter ($k^2 * n$ times in all).

Sparse CNNs dramatically cut the amount of compute (e.g., 4-9x) and data (e.g., 2-3x) by avoiding zeros. In doing so, the reuse patterns remain the same though the reuse counts decrease. However, implementing sparse vector-vector multiplication, the key primitive in sparse CNNs, is challenging.

2.1 SCNN

As discussed in Section 1, Cnvlutin [2] and others [42] exploit one-sided sparsity of either the feature maps or the filters but not both. SCNN [33] extends the idea to full, two-sided sparsity. SCNN employs a cluster (e.g., 8x8) of asynchronous processing elements (PEs) comprising several multipliers (e.g., 4x4) and accumulators (e.g., 32). Each PE holds a tensor of the input map as per the *input stationary* approach. The filters are broadcast to the PEs to be multiplied by relevant tensors of the input. Thus, SCNN captures both each filter's reuse across the input maps and each input map's reuse across the filters. To ensure good utilization of the multipliers and accumulators, each PE operates on a few filters at a time (e.g., 8).

SCNN employs HPC's Compressed Sparse Row (CSR) representation for sparse tensors using pointers to non-zero values. Multiplying two CSR tensors requires finding matching non-zero positions in the tensors and accessing the corresponding values – an *inner join* using the position as the *key* with a single *value field*. CSR requires incrementally searching a tensor's pointers for a match against the next pointer in the other tensor (Figure 2). Upon finding a matching position, accessing the corresponding value requires tracking the number of previous non-zero positions from the tensor start to the matching position in each of the input tensors. Implementing the one-sided *inner join* is straightforward, but the two-sided *inner join* using CSR or CSC is inefficient.

SCNN avoids implementing the *inner join* via the observation that as a filter "slides" in the "X" (height) and "Y" (width) dimensions over the feature map during the convolution, each (non-zero) filter cell in a given channel (in the "Z" dimension) is multiplied by all the (non-zero) feature map cells in the channel (Figure 1). Because an *inner join* would result in these same multiplications, SCNN performs this *Cartesian product* of multiplications in parallel without any explicit *inner join*. This strategy guarantees that no unnecessary multiplications with zeros occur while avoiding the *inner join*. The PE adds each product to the appropriate output map cell held in an accumulator buffer. The output tensor is computed first as a dense tensor (i.e., includes zeros). After applying *Rectified linear unit (ReLU)* which converts negative values into zeros, the output tensor is converted to the sparse CSR format.

The input map is tiled along the X-Y dimensions (Figure 1). Each PE is assigned a tile whose size is limited by the number of accumulator buffers to hold the tile's output. There is near-neighbor inter-PE communication to handle more than one neighboring input tile contributing to the same output cell at the tile boundary. SCNN authors argue that instead of tiling the input, tiling the output would incur other problems such as replication of the input cells contributing to more than one output tile and the Cartesian product may result in multiplying unnecessary input-filter products. However, extracting such an X-Y tile, where each dimension has variable number of elements due to sparsity, is hard.

2.1.1 SCNN's overheads: In SCNN's Cartesian-product strategy, however, the concurrent multiplications produce unrelated products destined for *different* output cells while spreading over different multipliers the products destined for the same output cell. Consequently, this approach incurs many considerable overheads.

First, because the products are unrelated, each product has to be routed to its sum held in an accumulator buffer. Because every product needs to be routed and because the products destined for a given partial sum may come from any multiplier, the routing requires a high-bandwidth cross bar between the multipliers and adders (SCNN uses a 16x32 crossbar). Further, each product needs to compute the address of its partial sum. While the calculation is simple (amounts to computing the differences between coordinates of the input map cell and the filter cell), each product's computation is different requiring many adders (e.g., at least 32 adders for the 16 multipliers per PE). While the latency of these overheads can be hidden, the complexity and energy costs are high.

Second, the Cartesian-product strategy assumes that a filter cell is multiplied by every input map cell, which is true only for unit-stride convolutions. For k-stride convolutions, a filter cell is multiplied by every k^{th} input map cell. Further, a filter cell is multiplied by only one input map cell for the fully-connected layers of RNNs, LSTMs, and MLPs. As such, the Cartesian-product strategy restricts SCNN's applicability only to CNNs with unit-stride convolutions.

Third, SCNN reports intra-PE underutilization due to fewer non-zero values in the input tile or filter than needed by the 4x4 multiplier array in each PE. The lower count may be caused by natural sparsity or small filters (e.g., 1x1 filters) despite operating on multiple filters as mentioned above. More filters would alleviate this problem but would also need more accumulator buffers.

Fourth, each filter tensor is broadcast to all the PEs to capture filter reuse and unrelated outputs need buffering. Consequently, limited filter buffering at the PEs implies numerous global, inter-PE barriers (e.g., a 512-channel input feature map may incur 64 barriers to produce one output cell). These barriers expose inter-PE load imbalance due to (1) varying feature map sparsity, (2) truncated input tiles at the input maps' edges due to the X-Y input tiling, and (3) input tiles leaving a laterally-inverted L-shaped remainder in the input map. Using smaller tiles to decrease this imbalance would reduce intra-PE utilization or reuse.

While the above second and third reasons are specific to SCNN's input tiling, the first reason is not. To capture filter reuse across the input maps, SCNN holds the input maps in the PEs while the filters are broadcast to the PEs. However, the varying sparsity among the input maps induces *systematic* load imbalance among the PEs. The PE holding a denser map would repeatedly take longer with *most* filters than a PE holding a sparser map. No amount of buffering would address this imbalance. Holding the filters instead of the input maps has the same effect. Solving this imbalance by not broadcasting the filters and having each PE independently fetch the filters to balance the PEs' work across the filters would result in the loss of filter reuse. SCNN does not address this *fundamental* tension between reuse and load imbalance due to which SCNN reports significant PE idling at the barriers.

3 SPARTEN

Recall from Section 1 that *SparTen* makes two contributions. First, *SparTen* achieves efficient *inner join* by providing support for native two-sided sparse execution and memory storage, instead of avoiding the *inner join* like SCNN. To avoid SCNN's execution overheads due to its Cartesian-product approach, *SparTen* confines one output cell's products to one multiplier and distributes different output cells

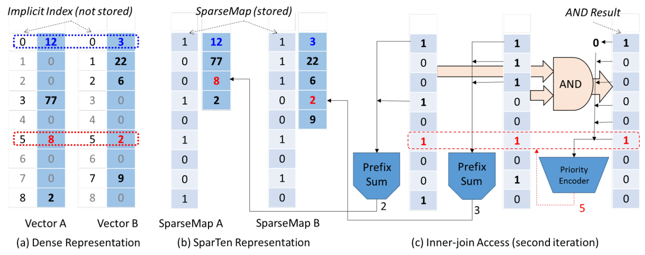


Figure 3: Efficient sparse vector-vector multiplication

to multipliers for parallelism. Second, to address load imbalance while achieving full reuse, *SparTen* employs an offline software scheme called *greedy balancing* which groups filters by density via two variants, a software-only one which uses whole-filter density and a software-hardware hybrid which uses finer-grain density.

3.1 Efficient inner join

Instead of HPC's CSR or CSC representations, *SparTen* uses a *bit-mask* representation where a sparse tensor is a two tuple of a bit mask, called *SparseMap*, and a set of non-zero values. The SparseMap has 1's for positions with non-zero values and 0's otherwise (Figure 3). For ease of implementation, tensors are broken up into *chunks* of n positions to give n-bit SparseMaps and the corresponding (variable number of) non-zero data values (e.g., $n = 128$).

The vector-vector dot product is a basic unit of computation for tensor processing. Implementing *inner join*, and therefore sparse vector-vector dot product, with the bit-mask representation is efficient. The key steps in an *inner-join* are: (1) finding the matching non-zero bit positions in the two tensors and (2) accessing the corresponding values. The first step is achieved by ANDing the tensors' SparseMaps to find the matches which are the set bits in the *AND-result* (Figure 3). Using the matching positions, the compute unit multiplies the corresponding pairs of values in the tensors, one at a time, as follows (from top to bottom in Figure 3). To find the next matching pair's position, we clear the current matching pair's bit in the *AND-result* when the pair is done (shown by a zero next to the topmost match position in the *AND-result* in Figure 3). To identify the next matching pair, we need the next topmost set bit in the *AND-result*. This bit is identified by a priority encoder (priority decreases from top to bottom). In the second step, a prefix sum circuit counts the number of 1's in each of the input tensors' SparseMaps above the newly-identified bit (the second-from-top matching position Figure 3). For each tensor, the count provides the number of non-zero values before the current value (Figure 3 shows the counts for the second-from-top match position). These counts provide the address offsets needed for accessing the current values. Upon multiplying the two values, the compute unit accumulates the product to the partial sum held locally in the unit. Fortunately, prefix sum [29] and priority encoder have well-studied, efficient implementations with carry lookahead-like logarithmic delays in the SparseMap bit width instead of ripple carry-like linear delays. Recall from Section 2.1 that *inner join* using CSR or CSC involves inefficient searching for matching non-zero positions and counting the non-zero values from the start until the matching positions to obtain the matching values' address offsets.

Surprisingly, the bit-mask representation is also space- efficient in addition to being compute-efficient. The space efficiency of the representation depends on the degree of sparsity. In HPC, sparse data is extremely sparse where the pointer representation is efficient (e.g., only 0.1% of values are non-zero). In contrast, the sparsity in machine learning models is much less (e.g., 33%-50% of values are non-zero). Consequently, a bit-mask representation is smaller. Here is a simple analysis: If we assume that a fraction f is non-zero in a set of n l-bit values, then a pointer representation needs $f*n*log_2 n+f*n*l$ bits whereas the bit-mask representation needs $n+f*n*l$ bits. Note that this count does not include any additional pointers needed for managing the variable number of elements in sparse data which are common to both representations. For the pointer scheme to be smaller $f*n*log_2 n<n$ which implies that $f<1log_2 n$ which means f has to be small for large n. In CNNs, the number of filter values is of the order of several millions requiring f to be smaller than 120 for the pointer scheme to be better whereas the observed f is around 13 to 12. Some CSC or CSR formats use zero-string run-length encoding to compress the pointers (e.g., [19]). However, shorter run lengths achieve higher compression (e.g., run length of 4 requires 2 bits) but incur (1) redundant pointers for strings of zeroes longer than the run length, making representation efficiency depend on the sparsity and zero run-length distribution and (2) redundant zero compute for such redundant pointers. Simple bit-mask representation avoids these overheads.

The data is held in two parts: the first part comprises an array of two tuples each of which is a chunk's SparseMap followed by a pointer to the chunk's non-zero data values. The array is as long as there are chunks in a filter or feature map. There are three such arrays for each layer, one each for (a) all the filters, (b) the input map, and (c) the output map. The second part holds the non-zero values, which are variable in number, making their layout a little more involved, especially for the non-read-only feature maps. Further, as we see below, different parts of a layer's feature map is output concurrently by different clusters of compute units in *SparTen*. As such, a contiguous layout of all the values of the output feature map, which vary in number from one chunk to the next, or of even the SparseMaps and the values together, may serialize the clusters' writes of the values to memory. At the other extreme, if a chunk's values were discontiguous from those of another chunk then there may be fragmentation and fine-grained memory management issues. Note that this issue arises for all one- or two-sided sparse architectures that exploit sparsity in the feature maps. *SparTen*'s simple compromise is to lay out contiguously each cluster's output values while keeping apart different clusters' outputs in different memory regions ensuring that the clusters need not be serialized for writing the values to memory. At the same time, because there are only a few tens of clusters, fragmentation and management issues are alleviated. Each cluster's region holds the values of a contiguous sub-tensor of the output map by slicing the X or Y dimensions while leaving the Z (channel) axis intact. Because of the data pointer with each SparseMap, only each of a chunk's bit mask and value vector has to be contiguous, a cluster's region need not be. So it is adequate to allocate for the average case with some padding (e.g., 10%), while using a watermark-based fallback for additional allocations in the background as the cluster continues to work.

All the data is stored in the axes order of Z, X and Y. The Z-first format ensures that the SparseMaps for an input map tensor or filter are contiguous for a compute unit access. We pad the SparseMaps

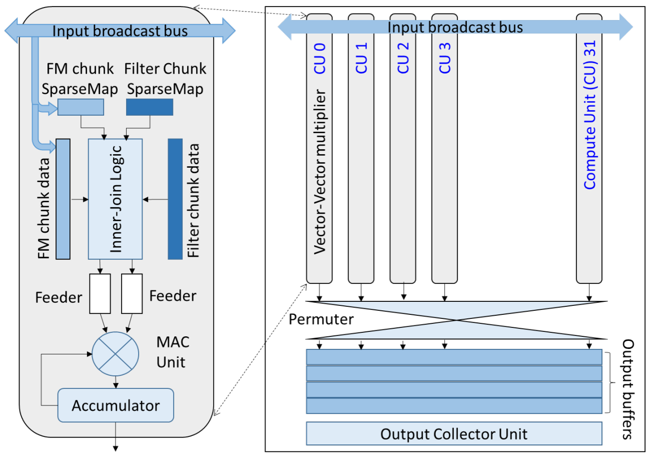


Figure 4: *SparTen* microarchitecture

with 0's when the channel count is a non-multiple of 128 (chunk size). An extreme special case is the initial 3-channel input image which is dense (i.e., zeroes are present). The input can be formatted into *SparTen*'s representation by simply creating bit masks with three 1's padded by 125 0's and a pointer to the dense data (values are not padded).

Finally, SCNN, Cnvlutin, and Cambricon-X use CSR and EIE uses a variant of HPC's compressed sparse column representation (CSC). Cambricon-S [43] is the only other architecture to use a bit-mask representation. Cambricon-S uses a *common* bit-mask for a set of coarsely-pruned filters (constructed offline in software) but does not support sparse representation for the feature maps (i.e., stores and retrieves zeros in the feature maps). Instead, Cambricon-S skips zeros in the feature maps during computation. In contrast, *SparTen* supports native two-sided sparse execution which retrieves, consumes, produces, and stores only non-zeros.

3.2 SparTen microarchitecture

We envision *SparTen* as a sparse tensor accelerator attached to the CPU-memory bus. The accelerator exposes BLAS-like interfaces for matrix-vector ($C \leftarrow Ax+y$) and matrix-matrix multiplications ($C \leftarrow A \times B$ with some simplifications. The interface allows for incremental construction of vectors to handle non-contiguous layout of tensors. Effectively all tensors are linearized on-the-fly into vectors for the matrix-vector or matrix-matrix operations.

Because sparse computation is not efficient in a SIMD, SIMT, or systolic organization, *SparTen* employs asynchronous *compute units* (left in Figure 4). Each compute unit, comprising a multiplier, an accumulator, the *inner join* circuitry (Section 3.1), and buffers for inputs and output, performs a sparse vector-vector dot product. A cluster of the units together (e.g., 32) perform a sparse matrix-vector multiplication to produce a sparse output vector in our representation (right in Figure 4). A *SparTen* implementation typically has many clusters for high compute bandwidth (e.g., 32 which gives 1K multiply-accumulate units in all). A key feature of *SparTen* is that the clusters produce different outputs and work independently. For energy efficiency, the compute units employ simple state machine control instead of program control.

To capture both filter reuse across input maps and input map reuse across filters, each compute unit has a local buffer to hold a filter chunk which is reused across multiple input map chunks. While computation proceeds chunk-by-chunk, memory transfers may occur in a different granularity (e.g., 128 B). At the start of a CNN layer, the CPU instructs each compute unit of a cluster to fetch and hold a chunk of a filter (SparseMaps and non-zero data values). The CPU then issues a fetch of an input map chunk from on-chip or off-chip memory which is broadcast to the cluster's compute units which then computes the filter-map dot product using the *inner join* circuitry. Each compute unit locally accumulates an output cell's partial sum as more input channels may remain to be processed (Figure 4). The CPU then issues the rest of the input chunks to each compute unit to complete its output cell, including any ReLU operation which may results in a zero. Consecutive compute units produce consecutive output channels. An *output collector* collects each compute unit's output, discards zero values, and on-the-fly produces a sparse output tensor (SparseMap and non-zero data values) broken up into chunks (Figure 4). If the channel count is not a multiple of 128 then (1) the output collector pads the SparseMap with zero bits (as many as there are compute units) and (2) the CPU rounds off the padding to the nearest multiple of 128. The permute network in Figure 4 is explained later in Section 3.3.

Figure 5 illustrates the hardware support for efficient on-the-fly conversion. A simple per-value zero-detection (EXNOR gate) generates the SparseMap. The output values must be compacted by eliminating any interleaving zeros. Such compacting is easily achieved using an *inverted* prefix sum that counts zeros in the SparseMap to shift the non-zero elements accordingly. Figure 5 illustrates the operation for the sixth value (shown in red) from the left. Because there are two zeros to the left of it, the value is shifted to the left by two positions. (The figure assumes that the values in the third and fourth positions have already been shifted left by one position each.) Unlike issue queue compaction in out-of-order-issue processors [15] which needs to be fast, this output compaction need not be fast (e.g., 32 output cells produced in parallel after $3*3*2567 \approx 329$ multiply-adds assuming 32 compute units/cluster, 3x3x256 filters, and 7x compute sparsity).

The cluster's output is written to the output bit-mask and data value arrays in memory (pointers provided by the CPU). The cluster returns the count of the non-zero output values to the CPU to increment the output map value array pointer for the next round of output. To ensure that each cluster produces a contiguous sub-tensor of the output map in the cluster's memory region (Section 3.1), the CPU issues the corresponding input-map sub-tensors and all the filters to the same cluster (capturing input map reuse and filter reuse). *SparTen* assigns one output cell per compute unit which, unlike SCNN, requires no near-neighbor communication.

The CPU places many requests to keep the compute units busy (i.e., there is request buffering). There is output buffering to achieve filter reuse across multiple input maps. Because the filters are held only one chunk at a time which may not be the entire filter, reusing each filter chunk across multiple inputs implies buffering the corresponding incomplete outputs until completion (by processing the rest of each filter and input map). Fortunately, an entire input map and filter, including all the chunks, produce just one output cell, so the buffer size is modest (e.g., 32 output cells per compute unit). However, this buffering cannot address the systematic load imbalance across the filters which we address in Section 3.3. To hide memory

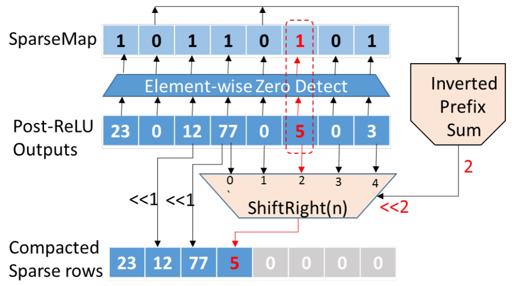


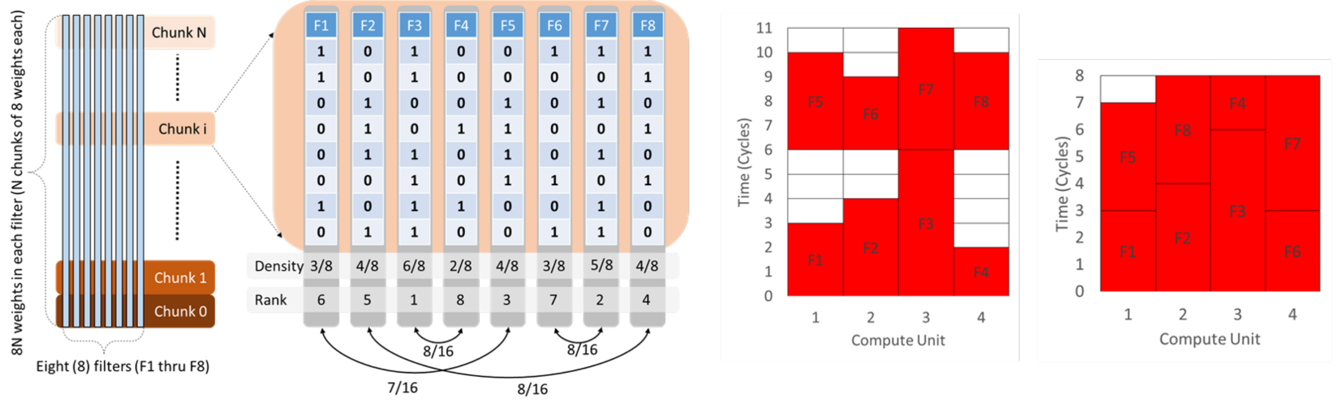
Figure 5: On-the-fly conversion to sparse representation

latency, the input map, filter and output map are double-buffered so that later input map chunks are fetched and broadcast, and the previous output map data is written while processing the current input chunks to produce the current output data. For example, the input map and filter each has a 128-bit mask for the SparseMap and a 128-byte block for data values, and the output data is 32 1-byte cells. Thus, in a 32-unit cluster, the total buffering is [128 bytes + 128 bits (input) + 128 bytes + 128 bits (filter) + 32 bytes (output)] * 32 (units) * 2 (double-buffering) = 20 KB (i.e., 640 B per multiplier). In comparison, SCNN needs 26 KB per PE including accumulator double buffering (i.e., 1.625 KB per multiplier) [33]. *SparTen*'s buffering increases due to its load balancing scheme (Section 3.3), pushing *SparTen* closer to SCNN in buffer capacity.

Recall from Section 2.1.1 that in SCNN's Cartesian-product strategy, the concurrent multiplications produce unrelated products destined for *different* output cells while spreading over different multipliers the products destined for the same output cell. This approach incurs numerous barriers, a high-bandwidth crossbar and many address calculators per PE, cannot handle non-unit-stride convolutions (or non-convolutional DNNs), and either underutilizes PEs at the input maps' X-Y borders and for smaller filters (e.g., 1x1), or requires more buffering for larger filters. In contrast, *SparTen* leverages its efficient *inner join* to confine the products for one output cell to one multiplier and distribute those for different output cells on different multipliers for parallelism. Consequently, *SparTen* avoids SCNN's problems. Because *SparTen*'s *inner join* produces different output cells in separate compute units, *SparTen* applies to convolutions of any stride (and also to non-convolutional DNNs) and avoids SCNN's cross bars. Each of *SparTen*'s compute units produces only one output cell at a time (1) requiring just one address calculation for *all* the products in a chunk and (2) allowing more room for input buffering and hence fewer implicit barriers at input broadcast. Producing a single output cell also means that a chunk can capture many channels for smaller filters to achieve high compute-unit utilization without increasing output buffering as all the channels contribute to the same output cell. Because *SparTen* assigns one output cell per compute unit there is no underutilization due to input tiling.

3.3 Greedy balancing

Because different filters held in a cluster's compute units inevitably have different sparsity and because all the filters are multiplied by the same input map, the units with denser filters would lag behind those with sparser filters by the next input map broadcast. The broadcast would impose an implicit barrier across the compute units exposing the load imbalance, as seen in SCNN Section 2.1.1. For example, across a collection of 20 layers' filters from *ResNet-152*, utilization



(a) GB-H Operation Example (8 filters and chunk size = 8)

(b) Utilization without GB-H (on 4 Compute Units)

(c) Utilization with GB-H (on 4 Compute Units)

Figure 6: Greedy Balancing (GB-H)

would vary from 52% to 65% at best. Buffering the later input maps would not address this *systematic* load imbalance which can be alleviated if each compute unit independently fetches the maps but at the complete loss of input map reuse. Thus, this reuse-imbalance tension is fundamental.

To achieve both reuse and load balance, we propose an offline load-balancing approach, called *greedy balancing (GB)*. While the true data-dependent estimate of work requires us to count the work where both the feature map *and* the filter are non-zero, we found that load-balancing based solely on the density of filters is an effective proxy. Because filters do not change during execution, one option may be to sort offline a layer's filters by density so that the filters within a cluster are similar in density. Because input maps are computed online, the input-stationary approach is not amenable to such offline processing. For this reason, *SparTen* employs the filter-stationary approach. Further, the offline filter processing cost is amortized over numerous input images.

GB has two variants: a software-only one (*GB-S*) and a software-hardware hybrid (*GB-H*). GB-S simply sorts all the filters of a layer by whole-filter density so that filters assigned to a cluster are similar in density. However, this sorting "shuffles" the filters' output positions in the channel axis. To match the new positions with the filters' weights in the next layer, GB-S statically "unshuffles" the next layer's weights in software (once for all image inputs). Thus, the offline processing proceeds layer by layer, unshuffling each layer's weights to match the previous layer and then sorting the layer's filters for load balance.

Because the filter density varies across chunks, GB-S incurs per-chunk load imbalance during execution (recall that chunks impose implicit barriers). GB-H addresses this issue by offline sorting the filters on a per-chunk basis. However, per-chunk sorting implies that the partial sum outputs of the chunks of a filter are "shuffled" to different positions. This shuffling cannot be fixed statically. Accordingly, GB-H employs a *multi-stage permutation network* to "unshuffle" the partial sum of each chunk to the appropriate output sum. However, because the filters of a layer are distributed across *SparTen*'s clusters for parallelism, (per-chunk) sorting together all the filters of a layer would require a global network connecting all the clusters. Restricting the sorting only to the filters within a cluster would need only a local, per-cluster network but may result in filters with varying density within each cluster, which is the original problem. To solve this dilemma, GB-H assigns twice as many filters to a cluster as its compute units and collocates on the first compute unit the per-chunk densest and sparsest filters within the cluster, the second densest and second sparsest filters on the second unit, and so on. Thus, the load imbalance across the pairs is much less than that across the individual filters. In Figure 6(a), we show eight filters, *F1* through *F8*, with a chunk size of eight weights, the densities for chunk *i* (density is the number of 1's in the SparseMap), and the density-sorted rank order across the filters. The figure shows the filter pairings for GB-H's colocation (e.g., F1 and F5, and F2 and F8). Figure 6(b) and (c), respectively, show the utilization on four compute units without GB-H and with GB-H (the shaded, useful time cycles in each unit correspond to the 1's in the SparseMap whereas the unshaded cycles are wasted due to load imbalance). In the rare case of filters being too few, assigning two filters per compute unit in GB-H may underutilize the cluster more than improve load balance. Fortunately, this condition can be detected statically and GB-S (or GB-H) can be turned off.

Further, because such collocation can alleviate any residual imbalance even in GB-S, we employ whole-filter collocation, pairing dense and sparse filters as described above. This whole-filter collocation can be unshuffled offline in software (as described above) and hence does not require any network, unlike GB-H's per-chunk collocation. Note that instead of GB, dynamically dispatching filters to idle compute units (1) would result in more filter movement (i.e., loss of filter reuse) and (2) is unlikely to perform as well as GB which statically collocates appropriate filter pairs.

The CPU simply instructs each compute unit to fetch and hold a collocated pair of filter chunks. Each compute unit sequentially multiplies the input map chunk with the two filters' chunks producing two partial sums which are routed by the permutation network (Figure 4). Because the output collector (Section 3.2) comes *after* the permutation network, the latter does not complicate the former. Unlike SCNN's high-bandwidth cross bar, this low-bandwidth network routes a result only once per chunk of multiply-adds (e.g.,

Table 2: Hardware parameters (* multiply-accumulate units)

Architecture	MACs*/cluster		# clusters		buffer/MAC
	large	small	large	small	
Dense	32	16	32	16	8 B
SCNN	16	16	64	16	1.63 KB
SparTen	32	16	32	16	0.97 KB

32 values after 1287 ≈ 18 multiply-adds assuming 32 compute units/cluster, 128-element chunk and 7x compute sparsity). Further, the permutation latency can be hidden under the next chunk's computation. As such, unlike high-bandwidth permutation networks (*e.g.* Benes network [6], Clos network [10], our low-bandwidth network needs significantly fewer resources. However, collocation doubles the buffering for the filters and outputs (each collocated filter produces a different output cell), so that *SparTen*'s total buffering increases to [128 bytes + 128 bits (input) + (128 bytes + 128 bits) * 2 (collocated filters) + 32 bytes * 2 (collocated output)] * 32 (units) * 2 (double-buffering) = 31 KB (i.e., 992 B per multiplier). This buffering is smaller than SCNN's (Section 3.2).

Due to the low-bandwidth demand, *SparTen*'s network significantly "thins" all the links and switches of traditional permutation networks. For example, a fully-provisioned network must accommodate the possibility that *all* values traverse the bisection concurrently. Instead, we limit bisection bandwidth to just four values at a time (i.e., 8 4-value batches routed in 18 cycles in the above example). If more values must traverse the bisection, they are scheduled in later cycles left vacant by the low bandwidth demand. Our analysis reveals that using modest bandwidth (1/8th of full provisioning) is more than adequate to handle GB's demand. Similarly, though GB results in two collocated outputs being produced per cluster, the low-bandwidth demand implies that a single output collector can sequentially produce the two output SparseMaps.

4 METHODOLOGY

We evaluate *SparTen* using a cycle-level simulator and a working System Verilog implementation realized on an FPGA.

Simulator: We build a cycle-level performance simulator for each of a dense accelerator, SCNN, and *SparTen*. The *SparTen* simulator can be configured for one-sided or two-sided sparsity. For the dense accelerator, the simulator captures the zero computations, which provide opportunity for the sparse architectures, *without* imposing sparse computation overheads (i.e., inner-join, permutation network, and output compaction). For SCNN, the simulator faithfully captures the key sources of performance loss (e.g., intra-PE idling and inter-PE barriers). We set SCNN input tile size to be 6x6 which performs the best in a search of the tile size space based on 1K accumulator buffers and output group size of 8 [33]). For *SparTen*, the simulator captures any residual load imbalance even after greedy balancing and any other idling. Additionally, we also configure *SparTen* for one-sided sparsity to act as a proxy for previous one-sided schemes (e.g., Cnvlutin, Cambricon-X, and EIE's zero-idling). We use detailed energy estimates based on our Verilog implementation.

Simulated systems: We simulate an aggressive configuration for AlexNet and VGGNet, and a scaled-down configuration for the smaller GoogLeNet models. The hardware parameters are listed in Table 2. *In our comparisons, we ensure that all the architectures have closely similar resources, such as compute units, on-chip buffering, and memory bandwidth, so that the performance differences stem only from the architectural differences and not resource disparities.* A TPU-like dense accelerator needs only 8 bytes of buffering per multiply-accumulate unit (MAC) to achieve the highest performance [40]. Adding more buffers will hurt the accelerator's energy without improving performance. In dense accelerators, every element in a tensor is guaranteed to be multiplied so that each MAC in a systolic pipeline needs to (double-) buffer only the input element pair and output. In sparse architectures including *SparTen*, however, not knowing statically which element would be multiplied forces more conservative buffering per MAC.

FPGA implementation: Using our working RTL, we implement one *SparTen* cluster with 32 compute units in System Verilog realized on a Terasic DE2-150 FPGA development board which has an Intel/Altera Cyclone IV FPGA interfaced to an external SDRAM (2.8Gbps). Running at 50 MHz, the FPGA has over 300 DSP multiply-accumulate units, 150K logic elements and 820 KB of RAM blocks. We use Intel's Quartus Prime (v15.0) with Qsys system builder for synthesis. We integrate our accelerator with a soft core (Nios II). We check the numerical correctness of our implementation.

Table 3: Benchmarks

Bench-mark	input	input density	filter	# filters	filter density
AlexNet					
Layer0	224x224x3	100%	11x11x3	64	84%
Layer1	55x55x64	38%	5x5x64	192	38%
Layer2	27x27x192	24%	3x3x192	384	35%
Layer3	13x13x384	20%	3x3x384	256	37%
Layer4	13x13x256	24%	3x3x256	256	37%
GoogLeNet					
Inc._3a_1x1	28x28x192	58%	1x1x192	64	38%
Inc._3a_3x3red	28x28x192	58%	1x1x192	96	41%
Inc._3a_3x3	28x28x96	68%	3x3x96	128	43%
Inc._3a_5x5red	28x28x192	58%	1x1x192	16	35%
Inc._3a_5x5	28x28x16	85%	5x5x16	32	33%
Inc._3a_poolprj	28x28x192	58%	1x1x192	32	47%
Inc._5a_1x1	7x7x832	31%	1x1x832	384	37%
Inc._5a_3x3red	7x7x832	31%	1x1x832	192	38%
Inc._5a_3x3	7x7x192	42%	3x3x192	384	39%
Inc._5a_5x5red	7x7x832	31%	1x1x832	48	35%
Inc._5a_5x5	7x7x48	69%	5x5x48	128	38%
Inc._5a_poolprj	7x7x832	31%	1x1x832	128	36%
VGGNet					
Layer0	224x224x3	100%	3x3x3	64	58%
Layer1	224x224x64	57%	3x3x64	64	21%
Layer2	224x224x64	49%	3x3x64	128	34%
Layer3	112x112x128	52%	3x3x128	128	36%
Layer4	112x112x128	36%	3x3x128	256	53%
Layer5	56x56x256	39%	3x3x256	256	24%
Layer6	56x56x256	49%	3x3x256	256	42%
Layer7	56x56x256	16%	3x3x256	512	32%
Layer8	28x28x512	27%	3x3x512	512	27%
Layer9	28x28x512	30%	3x3x512	512	34%
Layer10	28x28x512	13%	3x3x512	512	32%
Layer11	14x14x512	22%	3x3x512	512	29%
Layer12	14x14x512	28%	3x3x512	512	36%

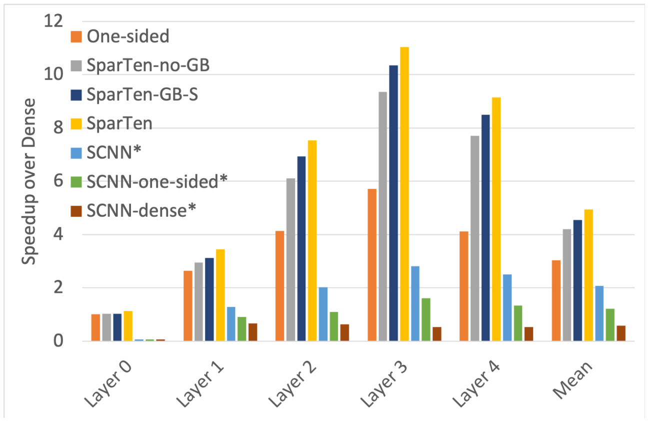


Figure 7: AlexNet Speedup (* mean excludes Layer0)

Area/Power/Clock speed estimates from ASIC synthesis: We performed synthesis of our System Verilog implementation of one 32-CU *SparTen* cluster using Synopsys's Design Compiler. We used the 45nm technology FreePDK45 library [39]. Because the FreePDK45 does not include a memory compiler, the buffers are synthesized using flip-flops which are expensive in area and power, rather than SRAM arrays. To avoid this artificial bloat that would not exist in a real implementation, we separately model the area and power of the buffers using Cacti 6.5 [32]. For Cacti power estimation, we assume one read and one write per cycle, the expected activity in *SparTen*.
Benchmarks: Table 3 shows our benchmark networks which are based on previous work [21, 33]. To obtain the sparse versions of the networks, we apply pruning [21] to the networks' filters using per-layer sparsity information after retraining for accuracy [33]. We ensure that the resulting sparsity matches previously-reported results [21]. Our simulations use a mini-batch of 16.

5 RESULTS

We show several results: (1) performance comparison of several schemes, (2) execution time breakdown for those schemes to explain their performance, (3) impact of greedy balancing, (4) energy comparison, and (5) *SparTen*'s speedup on our FPGA implementation.

5.1 Performance

We compare the performance of a dense architecture (*Dense*), a one-sided sparse architecture where only the input maps are sparse like Cnvlutin (*One-sided*), *SparTen* without either variant of greedy balancing (*SparTen*-no-GB), *SparTen* with the software-only variant of greedy balancing (*SparTen*-GB-S), *SparTen* with the hybrid variant of greedy balancing (*SparTen*), SCNN, one-sided SCNN where only the input maps are sparse (*SCNN-one-sided*), and dense SCNN (*SCNN-dense*). SCNN-one-sided and SCNN-dense incur SCNN's overheads (Section 2.1.1) whereas Dense and One-sided do not. Because SCNN compares to the SCNN variants, we include them as sanity checks. Figure 7, Figure 8 and Figure 9 show the above schemes' performance normalized to that of *Dense* for AlexNet, GoogLeNet and VGGNet, respectively. For each network, the figures show the geometric mean for each scheme; SCNN's mean for AlexNet excludes layer 0 where SCNN and its derivatives perform poorly due to non-unit convolutional stride.

For AlexNet (Figure 7), One-sided is better than Dense as expected. *SparTen*-no-GB is better than One-sided by exploiting two-sided sparsity. By alleviating load imbalance, *SparTen*-GB-S is better than *SparTen*-no-GB. *SparTen* is even better than *SparTen*-GB-S due to better fine-grained load balance. Overall, *SparTen* achieves 4.7x mean speedup over Dense (1.8x over One-sided) due to its efficient microarchitecture for *inner-join* and greedy balancing for load balance – our two contributions (Section 1). SCNN is hindered by its overheads (discussed next) and falls behind even One-sided. Consistent with expectations and in line with SCNN's results, however, SCNN is better than SCNN-one-sided and SCNN-dense both of which inherit SCNN's overheads. VGGNet and GoogLeNet follow the same trends, except in Layer 0 of VGGNet where the shallow channel depth (3 channels) hurts *SparTen*. *SparTen*-no-GB is better than *SparTen*-GB-S and *SparTen* for GoogLeNet layers *5x5_reduce* layers of both *Inception 3a* and *Inception 5a* which have 16 and 48 filters (see Table 3), respectively. Because the filter counts are not multiples of 32 in these layers, the collocation in *SparTen*-GB-S and *SparTen* leaves half the compute units unutilized. Removing the whole-filter collocation from *SparTen*-GB-S results in worse performance in most other benchmarks (not shown).

These improvements closely track the per-benchmark density listed in Table 3. For example, AlexNet layers 2 through 4, VGGNet layers 7 through 12,and GoogLeNet layers *Inception 5a 1x1* through *Inception 5a 5x5_reduce* and *Inception 5a_pool_proj*, have low density in input map and/or filter, resulting in higher improvements for the *SparTen* variants.

5.2 Execution time breakdown

To understand the speedups, Figure 10, Figure 11, and Figure 12 break down the above architectures' execution time normalized to that of Dense for AlexNet, GoogLeNet, and VGGNet, respectively. We omit AlexNet's Layer 0 because of SCNN's non-unit stride issue mentioned earlier in Section 5.1. Because SCNN-one-sided and SCNN-dense are only for sanity checking, we do not analyze them any further. We break down execution time into (a) non-zero computation, (b) zero computation, (c) intra-cluster (intra-PE in SCNN) loss, and (d) inter-cluster (inter-PE in SCNN) loss. Intra-cluster loss captures different effects in the *SparTen* variants and SCNN. In the *SparTen* variants, intra-cluster loss includes within-cluster load imbalance and underutilization due to lack of filters (*SparTen*-no-GB has only the former) whereas intra-PE idling in SCNN is due to too few non-zero values to be multiplied (Section 2.1.1). As before, VGGNet's Layer 0 suffers from high intra-cluster loss because of shallow channel depth. On the other hand, inter-cluster loss in the *SparTen* variants is due to insufficient amount of input and/or number of filters to keep all the 32 clusters busy. Inter-PE loss in SCNN occurs due to load imbalance exposed by the global, inter-PE barriers (Section 2.1.1).

For AlexNet (Figure 10), Dense incurs many zero computations which is the key motivation for the sparse architectures. Due to inevitably imperfect inter-cluster load balance, Dense incurs inter-cluster loss in layers with insufficient work causing some clusters to idle for a significant fraction of the overall runtime while other clusters are busy (layers with more work have less idling relative to the overall runtime). While One-sided reduces the zero computations, a significant fraction remains. The *SparTen* variants eliminate the

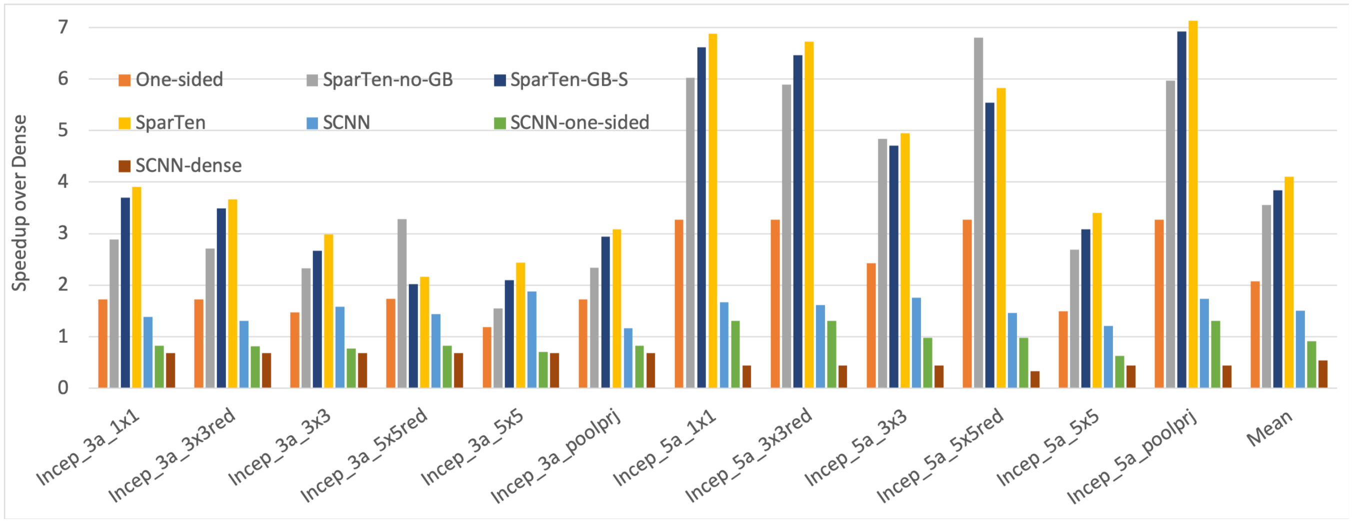


Figure 8: GoogLeNet Speedup

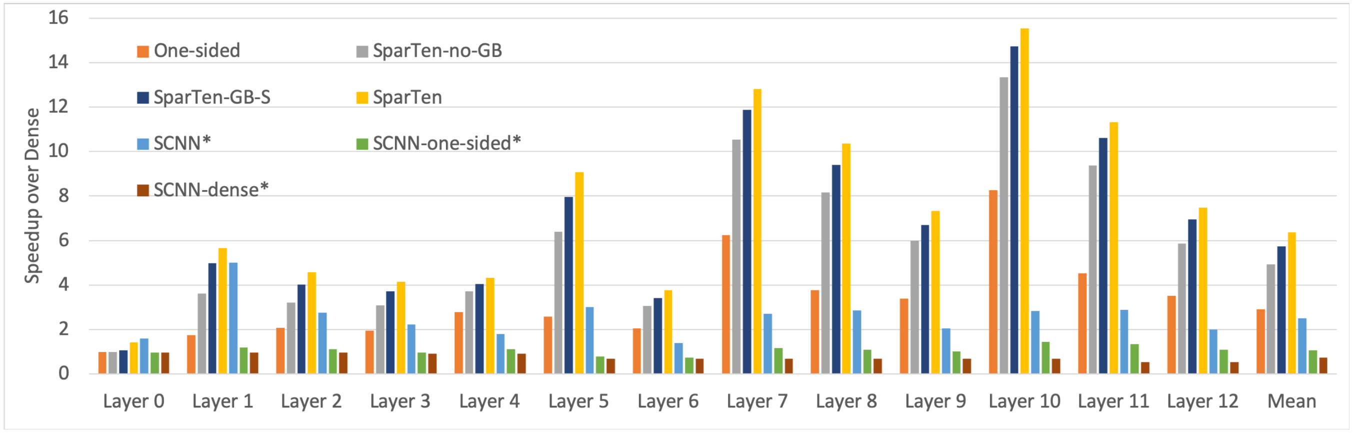


Figure 9: VGGNet Speedup (* mean excludes Layer0)

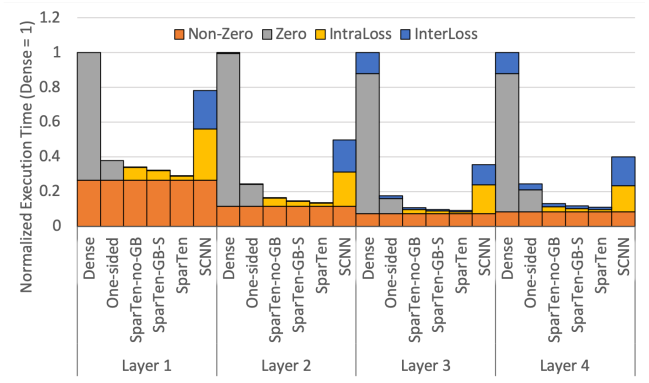


Figure 10: AlexNet Execution Time Breakdown

remaining zeros. The variants' less work also leads to less work for the busy clusters and hence less idling for the idle clusters. *SparTen*-no-GB's main overhead is within-cluster load imbalance (no intra-cluster loss due to lack of filters) which is reduced by *SparTen*-GB-S and nearly eliminated by *SparTen*. SCNN removes all zero computation but incurs significant intra- and inter-PE losses.

The same trends hold for GoogLeNet 9Figure 11) and VGGNet (Figure 12). In GoogLeNet, the *SparTen* variants incur some intra-cluster loss in (a) the *5x5_reduce* layers in both *Inception 3a* and *Inception 5a* due to the number of filters being a non-multiple of 32 which interacts poorly with collocation and (b) *5x5* layer in *Inception 3a* due to residual load imbalance even after greedy balancing. The variants also incur inter-cluster loss due to insufficient work in the small *Inception 5a* layers.

5.3 Energy

To isolate the impact of buffering on Dense's energy, we consider Dense-naive which is Dense configured with *SparTen*'s buffering instead of Dense's extremely low buffering (Table 2). Figure 13 shows the FPGA energy for Dense-naive, Dense, Ones-sided, *SparTen*-no-GB, *SparTen*-GB-S, and *SparTen* with GB-H) normalized to that of Dense-naive. We do not show SCNN because it performs worse than One-sided and its complexity is hard to model in enough detail for meaningful energy results. Due to lack of space, we show the average energy over the layers for each model. Further, our Verilog synthesis toolchain does not estimate DRAM energy which cannot

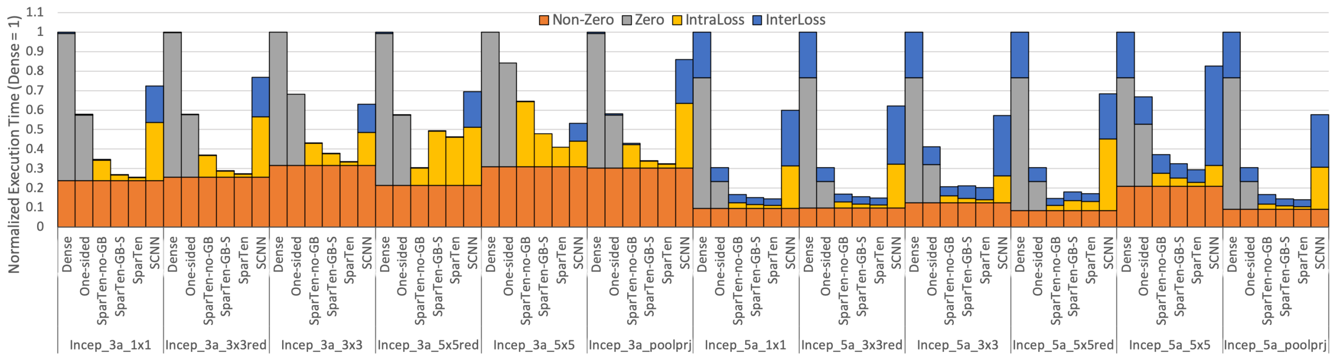


Figure 11: GoogLeNet Execution Time Breakdown

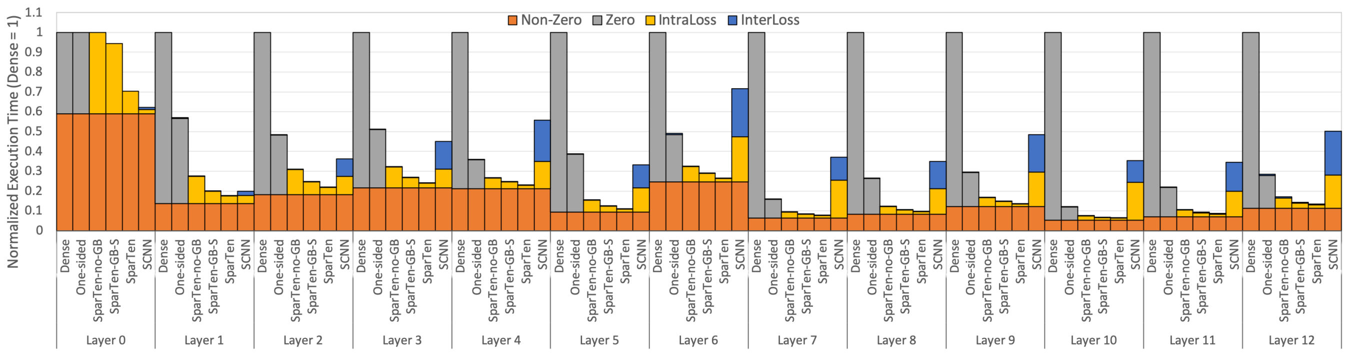


Figure 12: VGGNet Execution Time Breakdown

be normalized easily against the accelerator's energy. Therefore, we show compute and memory energies separately. We further break down compute and memory into zero and non-zero components.

For AlexNet, Dense'e compute energy is lower than that of the others due to Dense's low buffering needs, as highlighted by the drop from Dense-naive to Dense. Even so, Dense's compute energy is dominated by the zero component which is reduced by One-sided and eliminated by the *SparTen* variants. However, non-zero computation incurs more energy in these architectures than in Dense because of sparse computation overheads. That is, extra buffering, inner-join and output compaction (to a much smaller extent) incur more energy than Dense's simple multiply-accumulate. The *SparTen* variants have more non-zero compute energy than One-sided because two-sided inner-join takes more energy than one-sided inner-join. Though we account for *SparTen*'s permutation network energy (due to GB-H), it is relatively too small to be visible (recall from Section 3.3 that GB-H needs only a low-bandwidth network due to low demand). Note that the sparse computation latency overheads do not hurt performance due to simple pipelining (Section 5.1) but the energy overheads cannot be pipelined away. Overall, *SparTen* incurs 2x compute energy increase over Dense and achieves 1.5x reduction over One-sided. Nevertheless, *SparTen* is better than Dense in performance per Joule (4.7x better in performance and 2x worse in compute energy, ignoring *SparTen*'s memory energy advantage over Dense discussed next).

Because buffering affects only compute energy and not memory energy, the memory energies for Dense-naive and Dense are identical. Unlike Dense's compute energy, its memory energy is dominated by non-zeros because while compute volume reduces quadratically over sparsity memory volume reduces only linearly. Both One-sided and *SparTen* incur bit-mask and pointer overheads (Section 3.1) for their non-zero data. As in compute, One-sided reduces the zero component and the *SparTen* variants eliminate it. The *SparTen* variants have the same memory energy due to their identical memory volumes. While the same trends in compute and memory energies hold for GoogLeNet and VGGNet, *SparTen* is better where Dense's zero components for compute and memory, and hence the opportunity for sparse architectures, are higher. Overall, *SparTen* achieves 1.4x memory energy reduction over Dense and 1.3x over One-sided.

It may seem that Dense's lower buffering would imply an area advantage over *SparTen*. However, based on *SparTen*'s memory energy advantage over Dense, *SparTen*'s on-chip SRAM would be 25-30% smaller than Dense's, offsetting *SparTen*'s buffering bloat. Given that the on-chip SRAM tends to be large (e.g., TPU's 20 MB), the offset would likely be substantial.

5.4 Impact of greedy balancing

Figure 14 (red curve) plots the fraction of non-zeros in a given chunk of a filter (i.e., density) on the Y-axis for each of the 384 filters (X-axis) of AlexNet's 'Layer 2' which is representative. The filters are sorted by density for visual clarity. We make two observations. First, the absolute density is fairly low (*e.g.* the median density is approximately 24%). Second, there is significant variation in the density from under 10% to over 40%. Such variance leads to load imbalance.

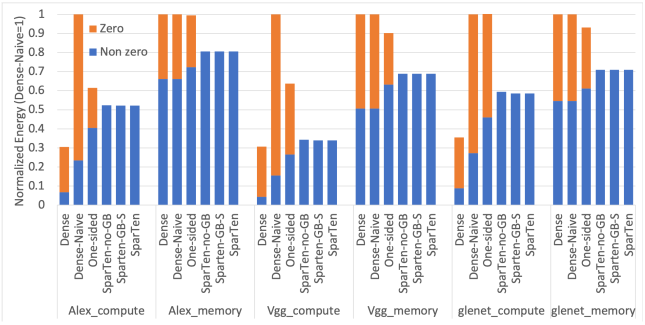


Figure 13: SparTen's energy savings

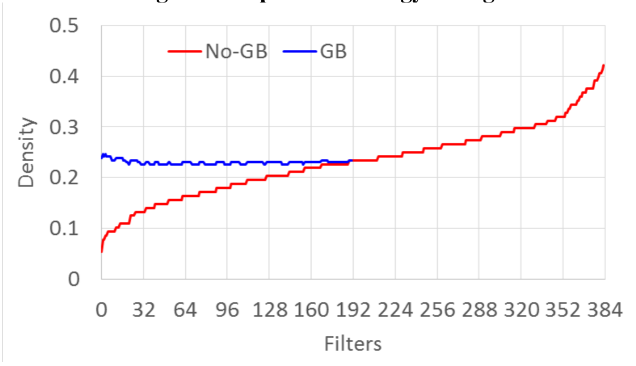


Figure 14: Impact of Greedy Balancing

Figure 14 (blue curve) displays the distribution after the hybrid variant of greedy balancing (GB-H). Note that the X-axis should now be treated as "filter-pairs" because of GB-H's pair-wise collocation (Section 3.3). As such, the X-axis extent is cut in half from 384 filters to 192 filter-pairs. The filter-pairs' density has much less variation indicating improved load balance.

5.5 FPGA performance results

Figure 15, Figure 16, and Figure 17 show the speedups of One-sided, *SparTen*-no-GB, and *SparTen* over Dense obtained on the FPGA for AlexNet, GoogLeNet and VGGNet, respectively. We do not compare to other accelerators because writing working RTL for multiple, disparate designs is time-consuming. As expected, the speedups increase in the order of One-sided, *SparTen*-no-GB and *SparTen* for all three networks with the known exceptions of the GoogLeNet layers. As in the simulation (Section 5.1), *SparTen* achieves good speedups. For all the models together, *SparTen* performs 4.3x and 1.9x better than a dense architecture and a one-sided sparse architecture, respectively. while the speedup trends match those in the simulation results, the absolute speedups obtained by the FPGA are slightly lower than those reported by the simulation. This discrepancy stems from the FPGA becoming memory-bound in some cases where the computation decreases more (quadratically with sparsity) than the memory traffic (linearly with sparsity).

5.6 ASIC synthesis results

Table 4 shows the area and power estimates for one *SparTen* cluster. Our synthesis achieves 800-MHz clock speed. In comparison, SCNN achieves 1-GHz clock speed on 16-nm technology (which enjoys 23% faster fan-out-of-4 inverter (FO4) delay over our 45-nm technology [24]). As such, *SparTen* and SCNN have comparable clock speeds. In area, *SparTen* (0.766 mm^2 with 32 MAC units using 45 nm technology) compares favorably to SCNN (0.123 mm^2 with 16 MACs using denser 16 nm technology). SCNN does not provide power estimates. As mentioned above, comparing to other accelerators would be time-consuming.

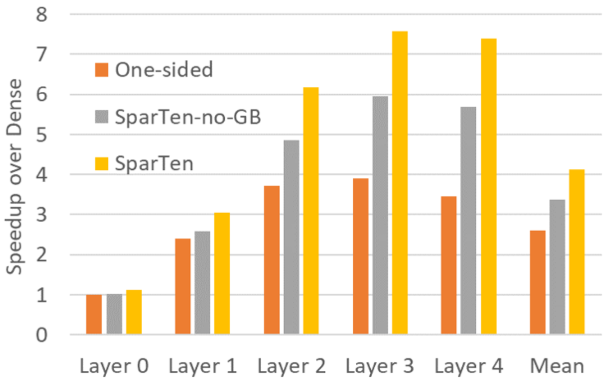


Figure 15: AlexNet Speedup on FPGA

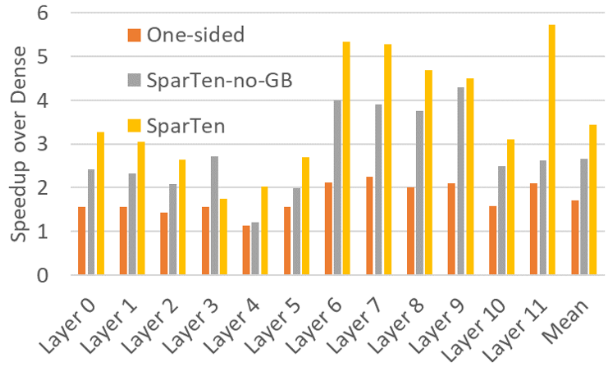


Figure 16: GoogLeNet Speedup on FPGA

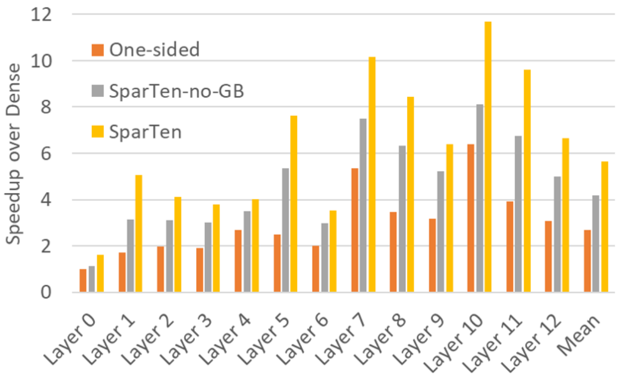


Figure 17: VGGNet Speedup on FPGA

Table 4: ASIC Area and Power for *SparTen* (45nm)

Component	Area (mm^2)	Power (mW)
Buffers	0.1	19.2
Prefix-sum	0.418	48
Priority Encoder	0.0626	6.4
MACs	0.0432	13.82
Permute Network	0.0344	10.6
Other	0.1	20.28
Total	0.766	118.30

6 RELATED WORK

Architectures for dense CNNs have optimized compute [17, 23, 25, 28, 37], memory [7, 14, 30], and reuse [3, 8]. However, sparse architectures can significantly reduce compute and data volumes to achieve better performance and energy. Section 1 discusses the semi-sparse architectures [2, 11, 19, 42]. We have discussed extensively SCNN, a fully-sparse architecture. Diffy [31] further improves non-zero computations by exploiting the small differences in the values The bit-serial architectures [1, 11, 36] reduce compute work by leveraging Booth encoding to elide zero bits. While this approach opens a new opportunity, there are four issues. First, the schemes transfer zero values to and from memory incurring on-chip SRAM area and energy, and memory energy and bandwidth costs, unlike *SparTen*. Second, the fundamental tension between reuse and bit-level load imbalance remains. For example, Bit-Laconic's array of 16 x 9 Laconic Processing Elements [36] incurs an implicit barrier after every set of encoded-pair multiplications but does not provide load balancing like *SparTen*'s greedy balancing. Third, Bit-Laconic presents 100x or more speedup opportunity in many models. meaning that out of the possible 81 (256) radix-four (radix-two) encoded-pair multiplications for 16-bit operands, fewer than 1 (2.5) encoded pair is non-zero on average (for 8-bit operands, 0.25 (0.6) non-zero encoded pair on average). Finally, just as sparse-value schemes including *SparTen* need conservative buffering which incurs overhead compared to dense architectures (Section 4), bit-sparse schemes also need conservative buffering of full values before Booth encoding. PermDNN [12] transforms sparse filters into a permuted diagonal matrices [16] only for fully-connected layers whereas *SparTen* targets convolutional layers. CirCNN [13] uses block-circulant matrices for filters but requires complex FFT hardware and does not capture all sparsity.

In-memory accelerators [4, 9, 35, 38] leverage analog logic to achieve dense matrix multiplication. It would be hard for such accelerators to leverage sparsity. Further, analog logic incurs the well-known issues of noise, scalability, and process variation which affects speed in digital logic but value accuracy in analog logic where values change with transistor parameters. Increasing CNN inaccuracy by 10% (e.g., degrading accuracy from 90% to 89%) may be unacceptable due to undoing a year's effort by machine learning experts. To account for process variation, each chip may have to be trained separately to maintain accuracy.

Cambricon-S's coarse-grain pruning clamps to zeros the values in contiguous positions in a group of filters. The clamped values cannot be changed in retraining. Similarly, Scalpel[41] proposes coarse-grain pruning at the granularity of the hardware width to maintain regularity. In contrast, Deep Compression [20] prunes each filter value independent of neighboring values where retraining can change zeros. As such, coarse-grain pruning may degrade accuracy. Unlike Deep Compression, Cambricon-S and Scalpel do not evaluate accuracy on high-accuracy deep models. Column combining (CC) [26] proposes to merge whole sparse columns into a denser column to improve systolic array utilization. The merging prunes all but the largest non-zero value when multiple non-zero values are merged. CC lowers the accuracy from 93.75% to 93% which is not a 0.75% loss of accuracy as reported but a 12% increase in inaccuracy. As explained in Section 1, while *SparTen* GB-S's unshuffling in software is similar to CC's row permutation, the shuffling criteria of *SparTen*'s GB and CC are completely different (group by density versus jigsaw-fit to avoid conflicts), Further, GB-H explores grouping at chunk granularity instead of CC's whole-filter granularity.

Previous High-Performance Computing (HPC) work on sparse data is relevant to *SparTen*. Among sparse representations in HPC, CSR is general and widely-adopted by sparse BLAS libraries including Sparse BLAS and cuSPARSE (Nvidia's sparse BLAS library). Many implementations improve sparse BLAS's performance over general-purpose processors [5, 44]. But the implementations remain significantly slower than dense accelerators like GPUs. Instead, *SparTen* uses a bit-mask representation for efficient implementation. Further, *SparTen* proposes GB to tackle the fundamental tension between load imbalance and reuse in sparse CNNs, which is not addressed by previous work.

7 CONCLUSION

Exploiting two-sided sparsity – zeros in both the feature maps and filters – in convolutional neural networks (CNNs) to reduce compute and data volumes is challenging. Sparse vector-vector dot product, a key primitive in CNNs, requires an *inner join* of the two sparse vectors using the non-zero position as the *key* with a single *value field*. To avoid the *inner join*, SCNN performs a Cartesian product capturing the relevant multiplications. However, this approach incurs several overheads and is not applicable to non-unit-stride convolutions. Further, exploiting reuse in sparse CNNs fundamentally causes systematic load imbalance which is not addressed by SCNN.

We proposed *SparTen* which achieves efficient inner join by providing support for native two-sided sparse execution and memory storage. To tackle load imbalance, *SparTen* employs a software scheme, called *greedy balancing*, which groups filters by density via two variants, a software-only one which uses whole-filter density and a software-hardware hybrid which uses finer-grain density. Our simulations show that, on average, *SparTen* performs 4.7x, 1.8x, and 3x better than a dense architecture, one-sided sparse architecture, and SCNN, respectively. *SparTen* also achieves 1.5x and 1.3x lower compute and memory energy than the one-sided architecture. Our FPGA implementation shows that *SparTen* performs 4.3x and 1.9x better than a dense architecture and a one-sided sparse architecture, respectively. Further, *SparTen* is broadly applicable to convolutional layers using any stride, non-convolutional deep neural networks (DNNs) such as long short-term memory (LSTMs), recurrent neural networks (RNNs), and multi-level perceptrons (MLP), as well as sparse linear algebra for High-Performance Computing (HPC). We leave extending *SparTen* to these other DNNs and HPC to future work. As such, *SparTen*'s simplicity, efficiency, and high performance make it an attractive architecture for sparse matrix computations.

ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under grant numbers 1618921-CNS and 1405939-CNS.

REFERENCES

[1] Jorge Albericio, Alberto Delmas, Patrick Judd, Sayeh Sharify, Gerard O'Leary, Roman Genov, and Andreas Moshovos. 2017. Bit-pragmatic deep neural network computing. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2017, Cambridge, MA, USA, October 14-18, 2017*. 382–394. https://doi.org/10.1145/3123939.3123982

[2] Jorge Albericio, Patrick Judd, Tayler H. Hetherington, Tor M. Aamodt, Natalie D. Enright Jerger, and Andreas Moshovos. 2016. Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2016, Seoul, South Korea, June 18-22, 2016*. 1–13. https://doi.org/10.1109/ISCA.2016.11

[3] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. 2016. Fused-Layer CNN Accelerators. In *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[4] Aayush Ankit, Izzat El Hajj, Sai Rahul Chalamalasetti, Geoffrey Ndu, Martin Foltin, R. Stanley Williams, Paolo Faraboschi, Wen-mei W Hwu, John Paul Strachan, Kaushik Roy, and Dejan S. Milojicic. 2019. PUMA: A Programmable Ultra-efficient Memristor-based Accelerator for Machine Learning Inference. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. ACM, New York, NY, USA, 715–731. https://doi.org/10.1145/3297858.3304049

[5] N. Bell and M. Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. 1–11. https://doi.org/10.1145/1654059.1654078

[6] V.E. Beneš. 1965. *Mathematical Theory of Connecting Networks and Telephone Traffic*. Elsevier Science. https://books.google.com/books?id=CANltcFRRHMC

[7] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. 2014. DaDianNao: A Machine-Learning Supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. IEEE Computer Society, Washington, DC, USA, 609–622. https://doi.org/10.1109/MICRO.2014.58

[8] Yu-Hsin Chen, Tushar Krishna, Joel Emer, and Vivienne Sze. 2016. 14.5 Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. In *2016 IEEE International Solid-State Circuits Conference (ISSCC)*. 262–263. https://doi.org/10.1109/ISSCC.2016.7418007

[9] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. 2016. PRIME: A Novel Processing-in-memory Architecture for Neural Network Computation in ReRAM-based Main Memory. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*. IEEE Press, Piscataway, NJ, USA, 27–39. https://doi.org/10.1109/ISCA.2016.13

[10] C. Clos. 1953. A study of non-blocking switching networks. *The Bell System Technical Journal* 32, 2 (March 1953), 406–424. https://doi.org/10.1002/j.1538-7305.1953.tb01433.x

[11] Alberto Delmas, Patrick Judd, Dylan Malone Stuart, Zissis Poulos, Mostafa Mahmoud, Sayeh Sharify, Milos Nikolic, and Andreas Moshovos. 2018. Bit-Tactical: Exploiting Ineffectual Computations in Convolutional Neural Networks: Which, Why, and How. *CoRR* abs/1803.03688 (2018). arXiv:1803.03688 http://arxiv.org/abs/1803.03688

[12] C. Deng, S. Liao, Y. Xie, K. K. Parhi, X. Qian, and B. Yuan. 2018. PermDNN: Efficient Compressed DNN Architecture with Permuted Diagonal Matrices. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 189–202. https://doi.org/10.1109/MICRO.2018.00024

[13] Caiwen Ding, Siyu Liao, Yanzhi Wang, Zhe Li, Ning Liu, Youwei Zhuo, Chao Wang, Xuehai Qian, Yu Bai, Geng Yuan, Xiaolong Ma, Yipeng Zhang, Jian Tang, Qinru Qiu, Xue Lin, and Bo Yuan. 2017. CirCNN: Accelerating and Compressing Deep Neural Networks Using Block-circulant Weight Matrices. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50 '17)*. ACM, New York, NY, USA, 395–408. https://doi.org/10.1145/3123939.3124552

[14] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. 2015. ShiDianNao: Shifting Vision Processing Closer to the Sensor. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 92–104. https://doi.org/10.1145/2749469.2750389

[15] J. A. Farrell and T. C. Fischer. 1998. Issue logic for a 600-MHz out-of-order execution microprocessor. *IEEE Journal of Solid-State Circuits* 33, 5 (May 1998), 707–712. https://doi.org/10.1109/4.668985

[16] Norman E. Gibbs, William G. Poole, Jr., and Paul K. Stockmeyer. 1976. A Comparison of Several Bandwidth and Profile Reduction Algorithms. *ACM Trans. Math. Softw.* 2, 4 (Dec. 1976), 322–330. https://doi.org/10.1145/355705.355707

[17] V. Gokhale, A. Zaidy, A. X. M. Chang, and E. Culurciello. 2017. Snowflake: An efficient hardware accelerator for convolutional neural networks. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*. 1–4. https://doi.org/10.1109/ISCAS.2017.8050809

[18] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep Learning with Limited Numerical Precision. In *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37 (ICML'15)*. JMLR.org, 1737–1746. http://dl.acm.org/citation.cfm?id=3045118.3045303

[19] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 243–254. https://doi.org/10.1109/ISCA.2016.30

[20] Song Han, Huizi Mao, and William J. Dally. 2016. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. http://arxiv.org/abs/1510.00149

[21] Song Han, Jeff Pool, John Tran, and William Dally. 2015. Learning both Weights and Connections for Efficient Neural Network. In *Advances in Neural Information Processing Systems 28*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett (Eds.). Curran Associates, Inc., 1135–1143. http://papers.nips.cc/paper/5784-learning-both-weights-and-connections-for-efficient-neural-network.pdf

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. *CoRR* abs/1512.03385 (2015). http://arxiv.org/abs/1512.03385

[23] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/3079856.3080246

[24] Dae Hyun Kim and Sung Kyu Lim. 2015. *Impact of TSV and Device Scaling on the Quality of 3D ICs*. Springer New York, New York, NY, 1–22. https://doi.org/10.1007/978-1-4939-2163-8_1

[25] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 1097–1105. http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[26] H. T. Kung, Bradley McDanel, and Sai Qian Zhang. 2018. Packing Sparse Convolutional Neural Networks for Efficient Systolic Array Implementations: Column Combining Under Joint Optimization. *CoRR* abs/1811.04770 (2018). arXiv:1811.04770 http://arxiv.org/abs/1811.04770

[27] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (Nov 1998), 2278–2324. https://doi.org/10.1109/5.726791

[28] Darryl D. Lin, Sachin S. Talathi, and V. Sreekanth Annapureddy. 2016. Fixed Point Quantization of Deep Convolutional Networks. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48 (ICML'16)*. JMLR.org, 2849–2858. http://dl.acm.org/citation.cfm?id=3045390.3045690

[29] Yen-Chun Lin and Chin-Yu Su. 2005. Faster Optimal Parallel Prefix Circuits: New Algorithmic Construction. *J. Parallel Distrib. Comput.* 65, 12 (Dec. 2005), 1585–1595. https://doi.org/10.1016/j.jpdc.2005.05.017

[30] Daofu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Teman, Xiaobing Feng, Xuehai Zhou, and Yunji Chen. 2015. PuDianNao: A Polyvalent Machine Learning Accelerator. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, New York, NY, USA, 369–381. https://doi.org/10.1145/2694344.2694358

[31] M. Mahmoud, K. Siu, and A. Moshovos. 2018. Diffy: a Déjà vu-Free Differential Deep Neural Network Accelerator. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 134–147. https://doi.org/10.1109/MICRO.2018.00020

[32] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman Jouppi. 2009. Cacti 6.0: A tool to model large caches. *HP Laboratories* (01 2009).

[33] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. 2017. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 27–40. https://doi.org/10.1145/3079856.3080254

[34] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael S. Bernstein, Alexander C. Berg, and Fei-Fei Li. 2014. ImageNet Large Scale Visual Recognition Challenge. *CoRR* abs/1409.0575 (2014). http://arxiv.org/abs/1409.0575

[35] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R. Stanley Williams, and Vivek Srikumar. 2016. ISAAC: A Convolutional Neural Network Accelerator with In-situ Analog Arithmetic in Crossbars. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*. IEEE Press, Piscataway, NJ, USA, 14–26. https://doi.org/10.1109/ISCA.2016.12

[36] Sayeh Sharify, Alberto Delmas Lascorz, Mostafa Mahmoud, Milos Nikolic, Kevin Siu, Dylan Malone Stuart, Zissis Poulos, and Andreas Moshovos. 2019. Laconic Deep Learning Inference Acceleration. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA '19)*. ACM, New York, NY, USA, 304–317. https://doi.org/10.1145/3307650.3322255

[37] Yongming Shen, Michael Ferdman, and Peter Milder. 2017. Escher: A CNN Accelerator with Flexible Buffering to Minimize Off-Chip Transfer. In *25th IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*.

[38] L. Song, X. Qian, H. Li, and Y. Chen. 2017. PipeLayer: A Pipelined ReRAM-Based Accelerator for Deep Learning. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 541–552. https://doi.org/10.1109/HPCA.2017.55

[39] James E. Stine, Ivan Castellanos, Michael Wood, Jeff Henson, Fred Love, W. Rhett Davis, Paul D. Franzon, Michael Bucher, Sunil Basavarajaiah, Julie Oh, and Ravi Jenkal. 2007. FreePDK: An Open-Source Variation-Aware Design Kit. In *Proceedings of the 2007 IEEE International Conference on Microelectronic Systems Education (MSE '07)*. IEEE Computer Society, Washington, DC, USA, 173–174. https://doi.org/10.1109/MSE.2007.44

[40] Mithuna Thottethodi and T. N. Vijaykumar. 2019. Why the GPGPU is Less Efficient than the TPU forDNNs. https://www.sigarch.org/why-the-gpgpu-is-less-efficient-than-the-tpu-for-dnns/.

[41] Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das, and Scott Mahlke. 2017. Scalpel: Customizing DNN Pruning to the Underlying Hardware Parallelism. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 548–560. https://doi.org/10.1145/3079856.3080215

[42] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. 2016. Cambricon-x: An Accelerator for Sparse Neural Networks. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49)*. IEEE Press, Piscataway, NJ, USA, Article 20, 12 pages. http://dl.acm.org/citation.cfm?id=3195638.3195662

[43] X. Zhou, Z. Du, Q. Guo, S. Liu, C. Liu, C. Wang, X. Zhou, L. Li, T. Chen, and Y. Chen. 2018. Cambricon-S: Addressing Irregularity in Sparse Neural Networks through A Cooperative Software/Hardware Approach. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 15–28. https://doi.org/10.1109/MICRO.2018.00011

[44] Ling Zhuo and Viktor K. Prasanna. 2005. Sparse Matrix-Vector Multiplication on FPGAs. In *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-programmable Gate Arrays (FPGA '05)*. ACM, New York, NY, USA, 63–74. https://doi.org/10.1145/1046192.1046202