SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks

Angshuman Parashar[†] Minsoo Rhu[†] Anurag Mukkara[‡] Antonio Puglielli[*]
Rangharajan Venkatesan[†] Brucek Khailany[†] Joel Emer[†‡] Stephen W. Keckler[†] William J. Dally[†◇]

NVIDIA[†] Massachusetts Institute of Technology[‡] UC-Berkeley[*] Stanford University[◇]
aparashar@nvidia.com

ABSTRACT

Convolutional Neural Networks (CNNs) have emerged as a fundamental technology for machine learning. High performance and extreme energy efficiency are critical for deployments of CNNs, especially in mobile platforms such as autonomous vehicles, cameras, and electronic personal assistants. This paper introduces the Sparse CNN (SCNN) accelerator architecture, which improves performance and energy efficiency by exploiting the zero-valued weights that stem from network pruning during training and zero-valued activations that arise from the common ReLU operator. Specifically, SCNN employs a novel dataflow that enables maintaining the sparse weights and activations in a compressed encoding, which eliminates unnecessary data transfers and reduces storage requirements. Furthermore, the SCNN dataflow facilitates efficient delivery of those weights and activations to a multiplier array, where they are extensively reused; product accumulation is performed in a novel accumulator array. On contemporary neural networks, SCNN can improve both performance and energy by a factor of 2.7× and 2.3×, respectively, over a comparably provisioned dense CNN accelerator.

CCS CONCEPTS

• **Computer systems organization** → **Architectures**; **Parallel architectures**; **Special purpose systems**;

KEYWORDS

Convolutional neural networks, accelerator architecture

ACM Reference format:
A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S.W. Keckler, and W.J. Dally. 2017. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. In *Proceedings of ISCA '17, Toronto, ON, Canada, June 24-28, 2017,* 14 pages.
https://doi.org/10.1145/3079856.3080254

This research was, in part, funded by the U.S. Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

ISCA '17, June 24-28, 2017, Toronto, ON, Canada

ACM ISBN 978-1-4503-4892-8/17/06…$15.00
https://doi.org/10.1145/3079856.3080254

1 INTRODUCTION

Driven by the availability of massive data and the computational capability to process it, deep learning has recently emerged as a critical tool for solving complex problems across a wide range of domains, including image recognition [22], speech processing [3, 14, 18], natural language processing [9], language translation [11], and autonomous vehicles [23]. Convolutional neural networks (CNNs) have become the most popular algorithmic approach for deep learning for many of these domains. Employing CNNs can be decomposed into two tasks: (1) training — in which the parameters of a neural network are learned by observing massive numbers of training examples, and (2) inference — in which a trained neural network is deployed in the field and classifies the observed data. Today, training is often done on GPUs [27] or farms of GPUs, while inference depends on the application and can employ CPUs, GPUs, FPGAs or specially-built ASICs.

During the training process, a deep learning expert will typically architect the network, establishing the number of layers, the operation performed by each layer, and the connectivity between layers. Many layers have parameters, typically filter weights, which determine their exact computation. The objective of the training process is to learn these weights, usually via a stochastic gradient descent-based excursion through the space of weights. This process typically employs a forward-propagation calculation for each training example, a measurement of the error between the computed and desired output, and then back-propagation through the network to update the weights. Inference has similarities, but only includes the forward-propagation calculation. Nonetheless, the computation requirements for inference can be enormous, particularly with the emergence of deeper networks (hundreds of layers [19, 20, 29]) and larger inputs sets, such as high-definition video. Furthermore, the energy efficiency of this computation is important, especially for mobile platforms, such as autonomous vehicles, cameras, and electronic personal assistants.

Recent published works have shown that common networks have significant redundancy and can be pruned dramatically during training without substantively affecting accuracy [17]. Our experience shows that the number of weights that can be eliminated varies widely across the layers but typically ranges from 20% to 80% [16, 17]. Eliminating weights results in a network with a substantial number of zero values, which can potentially reduce the computational requirements of inference.

The inference computation also offers a further optimization opportunity, as many networks employ as their non-linear operator

the ReLU (rectified linear unit) function which clamps all negative activation values to zero. The activations are the output values of an individual layer that are passed as inputs to the next layer. Our experience shows that for typical data sets, 50–70% of the activations are clamped to zero. Since the multiplication of weights and activations is the key computation for inference, the combination of these two factors can reduce the amount of computation required by over an order of magnitude. Additional benefits can be achieved by a compressed encoding for zero weights and activations, thus allowing more to fit in on-chip RAM and eliminating energy-costly DRAM accesses.

This paper introduces the Sparse CNN (SCNN) accelerator architecture, a new CNN inference architecture that exploits both weight and activation sparsity to improve the performance and power of DNNs. Our SCNN accelerator is designed to optimize the computation of the convolutional layers as state-of-the-art DNNs for computer vision are primarily dominated by these compute-intensive layers [24, 31]. Previous works have employed techniques for exploiting sparsity, including saving computation energy for zero-valued activations and compressing weights and activations stored in DRAM [7, 8]. Other works have used either a compressed encoding of activations [1] or compressed weights [34] in parts of their dataflow to reduce data transfer bandwidth and save time for computations of some multiplications with a zero operand. While these prior architectures have largely focused on eliminating computations and exploiting some data compression, SCNN is the first sparse CNN accelerator that effectively handles both the ineffectual activations and weights at the same time. Furthermore, SCNN employs both an algorithmic dataflow that eliminates all multiplications with a zero and a compressed representation of both weights and activations through almost the entire computation.

At the heart of the SCNN design is a processing element (PE) with a multiplier array that accepts a vector of weights and a vector of activations. Unlike previous convolutional dataflows [6, 8, 12, 28], the SCNN dataflow only delivers to the multiplier array weights and activations that can all be multiplied with one another in the manner of a *Cartesian product*. To reduce data accesses, the activation vectors are reused in an *input stationary* [7] fashion while being multiplied with a series of weight vectors. Finally, only non-zero weights and activations are fetched from the input storage arrays and delivered to the multiplier array. As with any CNN accelerator, SCNN must accumulate the partial products generated by the multipliers. However, since the products generated by the multiplier array cannot be directly summed together, SCNN tracks the output coordinates associated with each multiplication and sends the coordinate and product to a scatter accumulator array for summation.

To increase performance and capacity beyond a single PE, multiple PEs can run in parallel, each working on a disjoint 3D *tile* of input activations. The compression and tiling of the CNN data enables two energy-saving optizations. First, maintaining the weights and activations in a compressed form throughout the pipeline reduces energy-hungry data staging and transmission costs. Second, the entire volume of activations of larger CNNs can remain in on-die buffers between layers, entirely eliminating expensive cross-layer DRAM references for a large number of networks. Overall, this design provides efficient compressed storage and delivery of input operands, exploits high reuse of the input operands in the multiplier array, and spends no time on multiplications with zero operands.

Table 1: Network characteristics. Weights and activations assume a data-type size of two bytes.

Network	# Conv. Layers	Max. Layer Weights	Max. Layer Activations	Total # Multiplies
AlexNet [22]	5	1.73 MB	0.31 MB	0.69 B
GoogLeNet [31]	54	1.32 MB	1.52 MB	1.1 B
VGGNet [30]	13	4.49 MB	6.12 MB	15.3 B

To evaluate SCNN, we developed a cycle-level performance model and a validated analytical model that allows us to quickly explore the design space of different types of accelerators. We also implemented an SCNN PE in synthesizable System C and compiled the design into gates using a combination of commercial high-level synthesis (HLS) tools and a traditional Verilog compiler. Our results show that a 64 PE SCNN implementation with 16 multipliers per PE (1,024 multipliers in total) can be implemented in approximately $7.4mm^2$ in a 16nm technology, which is a bit larger than an equivalently provisioned dense accelerator architecture due to the overheads of managing the sparse dataflow. On a range of networks, SCNN provides a factor of 2.7× speedup and a 2.3× energy reduction relative to a comparably provisioned dense CNN accelerator.

2 MOTIVATION

Convolutional Neural Network algorithms (CNNs) are essentially a cascaded set of pattern recognition filters that need to be trained [23]. A CNN consists of a series of layers, which include convolutional layers, non-linear scalar operator layers, and layers that downsample the intermediate data, for example by pooling. The convolutional layers represent the core of the CNN computation and are characterized by a set of filters that are usually 1×1 or 3×3, and occasionally 5×5 or larger. The values of these filters are the *weights* that are learned using a training set for the network. Some deep neural networks (DNNs) also include fully-connected layers, typically toward the end of the DNN. During inference, a new image (in the case of image recognition) is presented to the network, which classifies into the training categories by computing each of the layers in the network, in succession. The intermediate data between the layers are called *activations*, and the output activations of one layer becomes the input activations of the next layer. In this paper, we focus on accelerating the convolutional layers as they constitute the majority of the computation [10].

Table 1 lists the attributes of three commonly used networks in image processing: AlexNet [22], GoogLeNet [31], and VGGNet [30], whose specifications come from the Caffe BVLC Model Zoo [5]. The increasing layer depth across the networks represents the successively more accurate networks in the ImageNet [21] competition. The Maximum Weights and Activations columns indicate the size of the largest weight and activation matrices in the network. The last column lists the total number of multiplies required to compute a single inference pass through all of the convolutional layers of the network. These data and computational requirements are derived from the standard ImageNet inputs images of 224×224 pixels. Processing larger, higher resolution images will result in greater computational and data requirements.

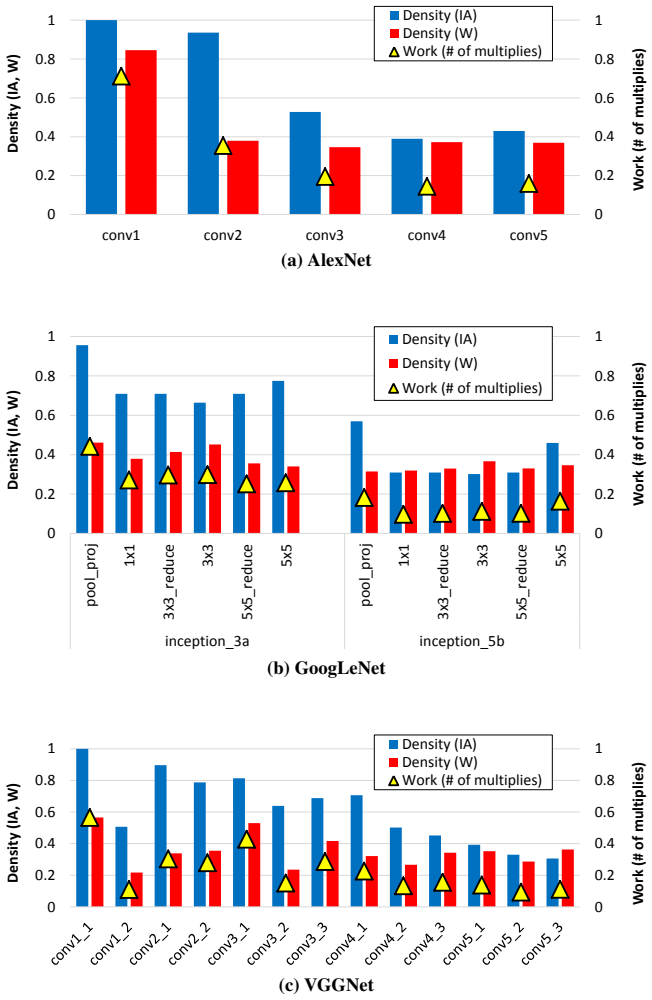


Figure 1: Input activation and weight density and the reduction in the amount of work achievable by exploiting sparsity.

Sparsity in CNNs. Sparsity in a CNN layer is defined as the fraction of zeros in the layer's weight and input activation matrices. The primary technique for creating weight sparsity is to prune the network during training. Han, et al. developed a pruning algorithm that operates in two phases [17]. First, any weight with an absolute value that is close to zero (e.g. below a defined threshold) is set to zero. This process has the effect of removing weights from the filters, sometimes even forcing an output activation to always be zero. Second, the remaining network is retrained, to regain the accuracy lost through naïve pruning. The result is a smaller network with accuracy extremely close to the original network. The process can be iteratively repeated to reduce network size while maintaining accuracy.

Activation sparsity occurs dynamically during inference and is highly dependent on the data being processed. Specifically, the rectified linear unit (ReLU) function that is commonly used as the non-linear operator in CNNs forces all negatively valued activations to be clamped to zero. After completing computation of a convolutional layer, a ReLU function is applied point-wise to each element in the output activation matrices before the data is passed to the next layer.

Table 2: Qualitative comparison of sparse CNN accelerators.

Architecture	Gate MACC	Skip MACC	Skip buffer/ DRAM access	Inner spatial dataflow
Eyeriss [7]	A	–	A	Row Stationary
Cnvlutin [1]	A	A	A	Vector Scalar + Reduction
Cambricon-X [34]	W	W	W	Dot Product
SCNN	A+W	A+W	A+W	Cartesian Product

To measure the weight and activation sparsity, we used the Caffe framework [4] to prune and train the three networks listed in Table 1, using the pruning algorithm of [17]. We then instrumented the Caffe framework to inspect the activations between the convolutional layers. Figure 1 shows the weight and activation density (fraction of non-zeros or complement of sparsity) of the layers of the networks, referenced to the left-hand y-axes. As GoogLeNet has 54 convolutional layers, we only show a subset of representative layers. The data shows that weight density varies across both layers and networks, reaching a minimum of 30% for some of the GoogLeNet layers. Activation density also varies, with density typically being higher in early layers. Activation density can be as low as 30% as well. The triangles show the ideal number of multiplies that could be achieved if all multiplies with a zero operand are eliminated. This is calculated by by taking the product of the weight and activation densities on a per-layer basis.

Exploiting sparsity. Since multiplication by zero just results in a zero, it should require no work. Thus, typical layers can reduce work by a factor of four, and can reach as high as a factor of ten. In addition, those zero products will contribute nothing to the partial sum it is part of, so the addition is unnecessary as well. Furthermore, data with many zeros can be represented in a compressed form. Together these characteristics provide a number of opportunities for optimization:

- **Compressing data:** Encoding the sparse weights and/or activations provides an architecture an opportunity to reduce the amount of data that must be moved throughout the memory hierarchy. It also reduces the data footprint, which allows larger matrices to be held in a storage structure of a given size.
- **Eliminating computation:** For multiplications that have a zero weight and/or activation operand, the operation can be data gated, or the operands might never be sent to the multiplier. This optimization can save energy consumption or both time and energy consumption, respectively.

Table 2 describes how several recent CNN accelerator architecture exploit sparsity. Eyeriss [7] exploits sparsity in activations by storing them in compressed form in DRAM and by gating computation cycles for zero-valued activations to save energy. Cnvlutin [1] is more aggressive—the architecture moves and stages sparse activations in compressed form and skips computation cycles for zero-valued activations to improve both performance and energy efficiency. Both

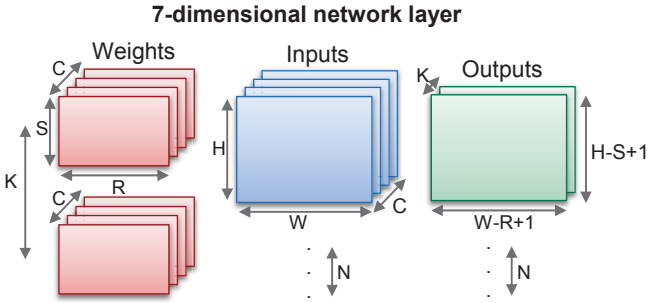


Figure 2: CNN computations and parameters.

```
for n = 1 to N
  for k = 1 to K
    for c = 1 to C
      for w = 1 to W
        for h = 1 to H
          for r = 1 to R
            for s = 1 to S
              out[n][k][w][h] +=
                in[n][c][w+r-1][h+s-1] *
                filter[k][c][r][s];
```

Figure 3: 7-dimensional CNN loop nest.

these architectures are also able to partially elide inner-buffer accesses for *weights* if those weights were only to be multiplied with a zero-valued activation. Conversely, the Cambricon-X [34] architecture exploits sparsity by compressing the pruned weights, skipping computation cycles for zero-valued weights, but it still suffers from wasted computation cycles when the non-zero weight is to be multiplied with zero-valued activations.

In addition to the different approaches of exploiting sparsity, these architectures also employ distinct *dataflows* [7] to execute a sparse convolutional layer. The most relevant distinction among these architectures' dataflows is how the innermost computation datapath exploits *spatial* reuse and sparsity patterns. Eyeriss uses a row-stationary dataflow, multicasting weights and activations across multiple scalar processing elements (PEs), with each PE independently performing zero-activation detection. Cnvlutin multiplies a single scalar non-zero activation across a vector of weights (organized by output-channel), and then reduces these output vectors across different input-channels. Cambricon-X fetches activation vectors across input-channels based on non-zero weight vectors and computes their dot product, including unnecessary work for zero-valued elements of the activation vector.

SCNN's objective is to exploit sparseness in both activations and pruned weights to eliminate as many computation cycles and data movement and storage operations as possible. SCNN employs a dense encoding of both sparse weights and activations so that only non-zero data values are retrieved from DRAM and on-chip buffers. Unfortunately, orchestrating a dataflow to deliver these sparse datasets to an array of multipliers while maximizing data reuse and multiplier utilization is non-trivial. Instead of coercing any of the previously proposed dataflows to suit our purpose, we employ a novel *Cartesian product* dataflow that exploits both weight and activation reuse while delivering only non-zero weights and activations to the multipliers. This dataflow performs an all-to-all multiply of non-zero weight and activation vector elements that can avoid any arithmetic based on zero-valued operands and achieve full multiplier utilization in steady-state.

3 SCNN DATAFLOW

While the inner core of the dataflow in SCNN is based on a spatial Cartesian product, the complete dataflow requires a deep nested loop structure, mapped both spatially and temporally across multiple processing elements. We call the full dataflow *PlanarTiled-InputStationary-CartesianProduct-sparse*, or PT-IS-CP-sparse. This section first describes a simple CNN convolutional layer to provide context for a detailed discussion of the construction of PT-IS-CP-sparse.

The core operation in a CNN convolutional layer is a 2-dimensional sliding-window convolution of an $R \times S$ element *filter* over a $W \times H$ element *input activation* plane to produce a $W \times H$ element *output activation* plane. The data can include multiple (C) input activation planes, which are referred to as *input channels*. A distinct filter is applied to each input activation channel, and the filter outputs for each of the C channels are accumulated together element-wise into a single output activation plane or *output channel*. Multiple filters (K) can be applied to the same volume of input activations to produce K output channels. Finally, a batch of N groups of C channels of input activation planes can be applied to the same volume of filter weights. Figure 2 shows these parameters applied to the computation of a single CNN layer.

The set of computations for the complete layer can be formulated as a loop nest over these 7 variables. Because multiply-add operations are associative (modulo rounding errors, which we ignore in this study), all permutations of these 7 loop variables are legal. Figure 3 shows an example loop nest based on one such permutation. We can concisely describe this nest as $N \rightarrow K \rightarrow C \rightarrow W \rightarrow H \rightarrow R \rightarrow S$. Each point in the 7-dimensional space formed from these variables represents a single multiply-accumulate operation. For the remainder of this paper, we assume a batch size of 1 ($N = 1$), which is common for inferencing tasks.

This simple loop nest can be transformed in numerous ways to capture different reuse patterns of the activations and weights and to map the computation to a hardware accelerator implementation. A CNN's *dataflow* defines how the loops are ordered, partitioned, and parallelized [7]. Prior work has shown that the choice of dataflow has a significant effect on the area and energy-efficiency of an architecture [7]. In fact, the choice of dataflow is perhaps the single most significant differentiator between many prior works on CNN architectures.

While the concept of dataflow has been studied for dense architectures, sparse architectures can also employ various alternative dataflows, each with its own set of trade-offs. While an exhaustive enumeration of sparse dataflows is beyond the scope of this paper, we present a specific dataflow called *PlanarTiled-InputStationary-CartesianProduct-sparse*, or PT-IS-CP-sparse. After examining a range of different dataflows, we selected PT-IS-CP-sparse because it enables reuse patterns that exploit the characteristics of sparse weights and activations. This section first presents an equivalent

dense dataflow (PT-IS-CP-dense) to explain the decomposition of the computations and then adds the specific features for PT-IS-CP-sparse.

3.1 The PT-IS-CP-dense Dataflow

Single-multiplier temporal dataflow. The *IS* term in PT-IS-CP-dense describes the *temporal* component of the dataflow. First, consider the operation of a scalar processing element (PE) with a single multiply-accumulate unit. We employ an *input-stationary* (IS) computation order in which an input activation is held stationary at the computation units as it is multiplied by all of the filter weights needed to make all of its contributions to each of the the K output channels (a $K \times R \times S$ sub-volume). Thus each input activation will contribute to a volume of $K \times W \times H$ output activations. This order maximizes the reuse of the input activations, while paying a cost to stream the weights to the computation units. Accommodating multiple input channels (C) adds an additional outer loop and results in the loop nest $C \rightarrow W \rightarrow H \rightarrow K \rightarrow R \rightarrow S$.

The PT-IS-CP-dense dataflow requires input buffers for weights and input activations, and an accumulator buffer to store the *partial sums* of the output activations. The accumulator buffer must perform a read-add-write operation for every access to a previously-written index. We call this accumulator buffer along with the attached adder an *accumulation unit*.

One of the objectives of the SCNN architecture is to maximize opportunities to store compressed activations on-die between network layers. This requires a moderately large input buffer, which can be energy-expensive to access. The input-stationary temporal loop nest amortizes the energy cost of accessing the input buffer over multiple weight and accumulator buffer accesses. More precisely, the register in which the stationary input is held over $K \times R \times S$ iterations serves as an inner buffer that filters accesses to the larger input buffer.

Unfortunately, the stationarity of input activations comes at the cost of more streaming accesses to the weights and to the partial sums in the accumulator buffer. Blocking the weights and partial sums in the output channel (K) dimension can increase reuse of these data structures and improve energy efficiency. We therefore factor the K output channels into K/K_c *output-channel groups* of size K_c, and only store weights and outputs for a single output-channel group at a time inside the weight and accumulator buffers. Thus the sub-volumes that are housed in buffers at the computation unit are:

- Weights: $C \times K_c \times R \times S$
- Inputs: $C \times W \times H$
- Partial Sums: $K_c \times W \times H$

An outer loop over all the K/K_c output-channel groups results in the complete loop nest $K/K_c \rightarrow C \rightarrow W \rightarrow H \rightarrow K_c \rightarrow R \rightarrow S$. Each iteration of this outer loop will require the weight buffer to be refilled and the accumulator buffer to be drained and cleared, while the contents of the input buffer will be fully reused because the same input activations are used across all output channels.

Intra-PE parallelism. The *CP* term in PT-IS-CP-dense describes how parallelism of many multipliers within a PE can be exploited while maximizing *spatial* reuse. A vector of F filter-weights fetched from the weight buffer and a vector of I inputs fetched from the input activation buffer are delivered to an array of $F \times I$ multipliers to compute a full Cartesian product (CP) of output partial-sums. This *all-to-all* operation has two useful properties. First, each fetched weight is reused (via wire-based multicast) over all I activations; each activation is reused over all F weights. Second, each product yields a useful partial sum such that no extraneous fetches or computations are performed. PT-IS-CP-sparse will exploit these same properties to make computation efficient on compressed-sparse weights and input activations.

The multiplier outputs are sent to the accumulation unit, which updates the partial sums of the output activation. Each multiplier output is accumulated with a partial sum at the matching output coordinates in the output activation space. These coordinates are computed in parallel with the multiplications. The accumulation unit must employ at least $F \times I$ adders to match the throughput of the multipliers.

Inter-PE parallelism. Finally, the *PT* term in PT-IS-CP-dense describes how to scale beyond the practical limits of multiplier count and buffer sizes within a PE. We employ a spatial tiling strategy to spread the work across an array of PEs so that each PE can operate independently. The $W \times H$ element activation plane is partitioned into smaller $W_t \times H_t$ element *planar tiles* (PT) that are distributed across the PEs. Each tile extends fully into the input-channel dimension C, resulting in an input-activation volume of $C \times W_t \times H_t$ assigned to each PE. Weights are broadcast to the PEs, and each PE operates on its own subset of the input and output activation space.

Unfortunately, strictly partitioning both input and output activations into $W_t \times H_t$ tiles does not work because the sliding-window nature of the convolution operation introduces cross-tile dependencies at tile edges. These data *halos* [13] can be resolved in one of two ways:

- **Input halos:** The input buffers at each PE are sized to be slightly larger than $C \times W_t \times H_t$ to accommodate the halos. These halo input values are replicated across adjacent PEs, but outputs are strictly private to each PE. Replicated input values can be multicast when they are being fetched into the buffers.
- **Output halos:** The accumulation buffers at each PE are sized to be slightly larger than $K_c \times W_t \times H_t$ to accommodate the halos. The halos now contain incomplete partial sums that must be communicated to neighbor PEs for accumulation, which occurs at the end of computing each output-channel group.

Our PT-IS-CP-dense dataflow uses output halos, though the efficiency difference between the two approaches is minimal.

Figure 4 shows pseudo-code for a single PE's loop nest in the PT-IS-CP-dense dataflow, including blocking in the K dimension (A,C), fetching vectors of input activations and weights (B,D), and computing the Cartesian product in parallel (E,F). X and Y coordinates for the accumulation buffer that are either negative or greater than $W_t - 1$ and $H_t - 1$ correspond to the locations of incomplete partial sums in the halo regions. Communication of these halos to neighboring PEs is not shown in the figure. The $Kcoord()$, $Xcoord()$, and $Ycoord()$ functions compute the k, x, and y coordinates of the uncompressed output volume using a de-linearization of the temporal loop indices a and w, the spatial loop indices i and f, and the known filter width and height. Overall, this PT-IS-CP-dense dataflow is simply a reordered, partitioned, and parallelized version of Figure 3.

```
    BUFFER wt_buf[C][Kc*R*S/F][F];
    BUFFER in_buf[C][Wt*Ht/I][I];
    BUFFER acc_buf[Kc][Wt+R-1][Ht+S-1];
    BUFFER out_buf[K/Kc][Kc*Wt*Ht];
(A) for k' = 0 to K/Kc-1
    {
      for c = 0 to C-1
        for a = 0 to (Wt*Ht/I)-1
        {
(B)       in[0:I-1] = in_buf[c][a][0:I-1];
(C)       for w = 0 to (Kc*R*S/F)-1
          {
(D)         wt[0:F-1] = wt_buf[c][w][0:F-1];
(E)         parallel_for (i = 0 to I-1) x (f = 0 to F-1)
            {
              k = Kcoord(w,f);
              x = Xcoord(a,i,w,f);
              y = Ycoord(a,i,w,f);
(F)           acc_buf[k][x][y] += in[i]*wt[f];
            }
          }
        }
      out_buf[k'][0:Kc*Wt*Ht-1] =
        acc_buf[0:Kc-1][0:Wt-1][0:Ht-1];
    }
```

Figure 4: PT-IS-CP-dense dataflow, single-PE loop nest.

3.2 PT-IS-CP-sparse Dataflow

PT-IS-CP-sparse is a natural extension of PT-IS-CP-dense that exploits sparsity in the weights and input activations. The dataflow is specifically designed to operate on compressed-sparse encodings of the weights and input activations and produces a compressed-sparse encoding of the output activations. At a CNN layer boundary, the output activations of the previous layer become the input activations of the next layer. While prior work has proposed a number of compressed-sparse representations [1, 15, 34], the specific format used is orthogonal to the sparse architecture itself. The key feature is that decoding a sparse format ultimately yields a non-zero data value and an index indicating the coordinates of the value in the weight or input activation matrices.

To facilitate easier decoding of the compressed-sparse blocks, weights are grouped into compressed-sparse blocks at the granularity of an output-channel group, with $K_c \times R \times S$ weights encoded into one compressed block. Likewise, input activations are encoded at the granularity of input channels, with a block of $W_t \times H_t$ encoded into one compressed block. At each access, the weight buffer delivers a vector of F **non-zero** filter weights along with each of their coordinates within the $K_c \times R \times S$ region. Similarly, the input buffer delivers a vector of I **non-zero** input activations along with each of their coordinates within the $W_t \times H_t$ region. Similar to the dense dataflow, the multiplier array computes the full cross-product of $F \times I$ partial sum outputs, with no extraneous computations. Unlike a dense architecture, output coordinates are not derived from loop indices in a state machine but from the coordinates of non-zero values embedded in the compressed format.

Even though calculating output coordinates is trivial, the multiplier outputs are not typically contiguous as they are in PT-IS-CP-dense. Thus the $F \times I$ multiplier outputs must be scattered to discontiguous addresses within the $K_c \times W_t \times H_t$ output range. Because any value in the output range can be non-zero, the accumulation buffer must be kept in an uncompressed format. In fact, output activations will probabilistically have high density even with a very low density of weights and input activations, until they pass through a ReLU operation.

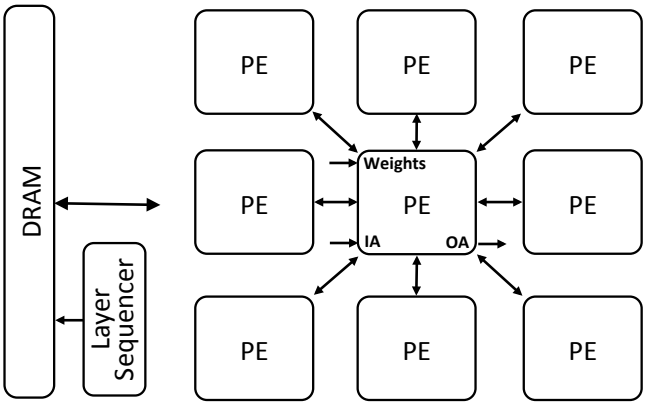


Figure 5: Complete SCNN architecture.

To accommodate the needs of accumulation of sparse partial sums, we modify the monolithic $K_c \times W_t \times H_t$ accumulation buffer from the PT-IS-CP-dense dataflow into a distributed array of smaller accumulation buffers accessed via a scatter network which can be implemented as a crossbar switch. The scatter network routes an array of $F \times I$ partial sums to an array of A accumulator banks based on the output index associated with each partial sum. Taken together, the complete accumulator array still maps the same $K_c \times W_t \times H_t$ address range, though the address space is now split across a distributed set of banks. PT-IS-CP-sparse can be implemented via small adjustments of Figure 4. Instead of a dense vector fetches, (B) and (D) fetch the compressed sparse input activations and weights, respectively. In addition, the coordinates of the non-zero values in the compressed-sparse form of these data structures must be fetched from their respective buffers (not shown). Then the accumulator buffer (F) must be indexed with the computed output coordinates from the sparse weights and activations. Finally when the computation for the output-channel group has been completed, the accumulator buffer is drained and compressed into the output buffer.

4 SCNN ACCELERATOR ARCHITECTURE

CNNs typically consist of a series of layers, including convolution, non-linear, pooling, and fully-connected. As the convolution layers typically dominate both arithmetic and computation time, the SCNN architecture is optimized for efficiency on these layers. For example, on GoogLeNet, the number of multiplies in the fully connected layers only account for 1% of the total computation. However, SCNN also includes dedicated logic for the simple localized non-linear and pooling layers. The non-linear layer is applied on a per-element basis at the end of a convolution or fully-connected layer, and is often implemented using the ReLU operator. A pooling layer can be applied after the ReLU layer; a typical 2×2 max pooling operator retains the maximum value in a window of four elements, thus reducing the volume of data passed to the next layer. While fully

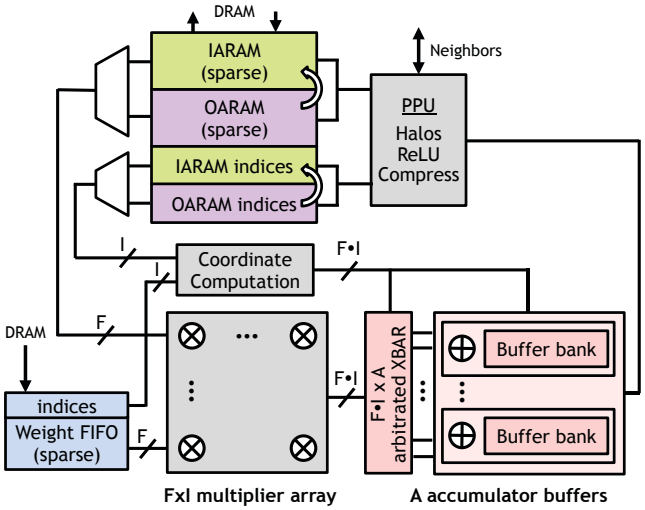


Figure 6: SCNN PE employing the PT-IS-CP-sparse dataflow.

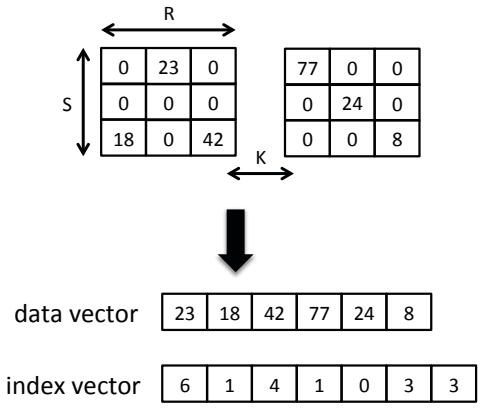


Figure 7: Weight compression.

connected layers are similar in nature to the convolution layers, they do require much larger weight matrices. However, recent work has demonstrated effective DNNs without fully connected layers [24]. Section 4.3 describes further how FC layers can be processed by SCNN.

4.1 Tiled Architecture

A full SCNN accelerator employing the PT-IS-CP-sparse dataflow of Section 3 consists of multiple SCNN processing elements (PEs) connected via simple interconnections. Figure 5 shows an array of PEs, with each PE including channels for receiving weights and input activations, and channels delivering output activations. The PEs are connected to their nearest neighbors to exchange halo values during the processing of each CNN layer. The PE array is driven by a layer sequencer that orchestrates the movement of weights and activations and is connected to a DRAM controller that can broadcast weights to the PEs and stream activations to/from the PEs. SCNN can use an arbitrated bus as the global network to facilitate the weight broadcasts, the point-to-point delivery of input activations (IA) from DRAM, and the return of output activations (OA) back to DRAM. The figure omits these links for simplicity.

4.2 Processing Element (PE) Architecture

Figure 6 shows the microarchitecture of an SCNN PE, including a weight buffer, input/output activation RAMs (IARAM and OARAM), a multiplier array, a scatter crossbar, a bank of accumulator buffers, and a post-processing unit (PPU). To process the first CNN layer, the layer sequencer streams a portion of the input image into the IARAM of each PE and broadcasts the compressed-sparse weights into the weight buffer of each PE. Upon completion of the layer, the sparse-compressed output activation is distributed across the OARAMs of the PEs. When possible, the activations are held in the IARAMs/OARAMs and are never swapped out to DRAM. If the output activation volume of a layer can serve as the input activation volume for the next layer, the IARAMs and OARAMs are logically swapped between the two layers' computation sequences. Each layer of the CNN has a set of parameters that configure the controllers in the layer sequencer, the weight FIFO, the IARAMs/OARAMs, and the PPU to execute the required computations.

Input weights and activations. Each PE's state machine operates on the weight and input activations in the order defined by the PT-IS-CP-sparse dataflow to produce a output-channel group of $K_c \times W_t \times H_t$ partial sums inside the accumulation buffers. First, a vector F of compressed weights and a vector I of compressed input activations are fetched from their respective buffers. These vectors are distributed into the $F \times I$ multiplier array which computes a form of the Cartesian product of the vectors, i.e, every input activation is multiplied by every weight to form a partial sum. At the same time, the indices from the sparse-compressed weights and activations are processed to compute the output coordinates in the dense output activation space.

Accumulation. The $F \times I$ products are delivered to an array of A accumulator banks, indexed by the output coordinates. To reduce contention among products that hash to the same accumulator bank, A is set to be larger than $F \times I$. Our results show that $A = 2 \times F \times I$ sufficiently reduces accumulator bank contention. Each accumulator bank includes adders and small set of entries for the output channels associated with the output-channel group being processed. The accumulation buffers are double-buffered so that one set of banks can be updated by incoming partial sums while the second set of banks are drained out by the PPU.

Post-processing. When the output-channel group is complete, the PPU performs the following tasks: (1) exchange partial sums with neighbor PEs for the halo regions at the boundary of the PE's output activations, (2) apply the non-linear activation (e.g. ReLU), pooling, and dropout functions, and (3) compress the output activations into the compressed-sparse form and write them into the OARAM. Aside from the neighbor halo exchange, these operations are confined to the data values produced locally by the PE.

Compression. To compress the weights and activations, we use variants of previously proposed compressed sparse matrix representations [15, 33]. Figure 7 shows an example of SCNN's compressed-sparse encoding for $R = S = 3$ and $K = 2$ with 6 non-zero elements. The encoding includes a data vector consisting of the non-zero values

and an index vector that includes the number of non-zero values followed by the number of zeros before each value. The 3-dimensional $R \times S \times K$ volume is effectively linearized, enabling full compression across the dimension transitions. The activations are encoded in a similar fashion, but across the $H \times W \times C$ dimensions. As the activations are divided among the PEs, each tile of compressed activations is actually $H_t \times W_t \times C$.

We use four bits per index to allow for up to 15 zeros to appear between any two non-zero elements. Non-zero elements that are further apart can have a zero-value placeholder without incurring any noticeable degradation in compression efficiency. The original (dense) coordinates of the weights and activations are recomputed by keeping a running sum of the number of zero and non-zero elements and the dividing by the appropriate dimension. Determining the coordinates in the accumulator buffer for each multiplier output requires reconstructing the coordinates from index vectors for F and I and combining them with the coordinates of the portion of the output activation space currently being processed. The encoding scheme can be enhanced with optimizations such as rounding up the dimension of each weight filter to a power of two to make division easier or using the four index bits per entry to encode a non-uniform number of zeros. While these optimizations would increase density somewhat, they do not substantively affect the SCNN architecture or our observed results.

4.3 Fully-connected Layers

While our Cartesian product-based SCNN dataflow dramatically improves its efficiency at the convolutional layers, it does impose some challenges when handling fully-connected layers. Concretely, unlike a convolution filter, an individual weight connection in a fully-connected layer is not reused across multiple input activations. Thus, the Cartesian product approach of SCNN does not automatically align non-zero weights and activations that must be multiplied. As a result, the SCNN 4×4 multiplier array can only operate at a peak rate of 4 multiplies per cycles (25% of peak throughput) because the 4 input activations and 4 weights can produce only 4 useful products. Another challenge with fully-connected layers is aligning the sparse weights and the sparse activations so that the appropriate non-zero values are delivered into the multiplier array at the same time. The SCNN's logic for processing the activation and weight indices can be reused to determine the alignment, but some additional multiplexing hardware would be required to move the non-zero weights into position.

While the loss of average throughput at the fully-connected layers would make SCNN unattractive for networks that are dominated by fully-connected layers, state-of-the-art CNNs for image classification, detection, and segmentation are primarily dominated by the convolutional layers; in the networks we used in our study, the fully-connected layers accounted for only 8%, 1%, and 2% of the multiplication operations in AlexNet, GoogLeNet, and VGGNet, respectively. In addition, networks for computer vision are decreasing their reliance of fully-connected networks and in fact, recent networks eliminate these layers completely [24]. Furthermore, the fully-connected layers are generally memory-bandwidth limited as they spend most of their execution time delivering network weights from DRAM. While SCNN's noticeable throughput reduction on fully-connected layers is not ideal, it is not a significant performance limiter for these memory-hungry layers. We argue that systems desiring optimal efficiency for both convolution and fully-connected layers should consider employing both SCNN and an architecture such as EIE that is optimized for fully-connected layers [15].

Table 3: SCNN design parameters.

PE Parameter	Value
Multiplier width	16 bits
Accumulator width	24 bits
IARAM/OARAM (each)	10KB
Weight FIFO	50 entries (500 B)
Multiply array ($F \times I$)	4×4
Accumulator banks	32
Accumulator bank entries	32
SCNN Parameter	**Value**
# PEs	64
# Multipliers	1,024
IARAM + OARAM data	1MB
IARAM + OARAM indices	0.2MB

4.4 Temporal Tiling for Large Models

SCNN compresses weights and activations to reduce both arithmetic operations and data movement. Ideally, the degree of compression and the capacity of the IARAMs and OARAMs are large enough so that the activations are never evicted to outer layers of the memory hierarchy. While we size our activation RAMs to capture the capacity requirements of nearly all of the layers in the networks we examined, a few layers of VGGNet require activations to be saved to and restored from DRAM. Like other accelerator architectures, SCNN can temporally tile the activation space so that the collection of PEs operate on a sub-volume of the activations at a time. This temporal tiling can be applied in addition to the *spatial* tiling that SCNN already employs to partition the activation volume across the PEs.

While a temporally tiled convolution layer still broadcasts weights to all of the PEs, the input activation planes are partitioned into coarse-grained tiles (across all channels) that each fit into the total IARAM capacity of the accelerator. The output activation tile is offloaded to DRAM and then reloaded into the IARAM when the data is needed as the input activation to the next layer. This type of tiling leads to a small halo at the edge of each input activation tile, resulting in a few additional input activation fetches from DRAM. The temporally tiled PT-IS-CP-sparse dataflow still exploits the reuse of each input activation value from the IARAM $R \times S \times K_c$ times.

In the networks we analyzed, only 9 of the 72 total layers fail to fit entirely within the IARAM/OARAM structures, with all of them coming from VGGNet. Our analysis shows that for both dense and sparse architectures, the DRAM accesses for one temporal tile can be hidden by pipelining them in tandem with the computation of another tile. Our nominal DRAM bandwidth configuration of 50 GB/s provides ample bandwidth to absorb the additional activation traffic. Only when DRAM bandwidth drops to around 4 GB/s does performance degrade. On these 9 layers, the per-layer energy penalty of activation data transfer ranges from 5–62%, with a mean of 18%. This overhead is fairly low and will be born by all CNN architectures with similar levels of on-chip RAM for activations; overheads will

Table 4: SCNN PE area breakdown.

PE Component	Size	Area (mm^2)
IARAM + OARAM	20 KB	0.031
Weight FIFO	0.5 KB	0.004
Multiplier array	16 ALUs	0.008
Scatter network	16×32 crossbar	0.026
Accumulator buffers	6 KB	0.036
Other	—	0.019
Total	—	0.123
Accelerator total	64 PEs	7.9

be higher for accelerators that do not compress activations. While the tiling approach is attractive, we expect some low power deployment scenarios to motivate neural networks designers to size them so that they fit completely in the on-chip SRAM capacity provided by the accelerator implementation.

4.5 SCNN Architecture Configuration

While the SCNN architecture can be scaled across a number of dimensions, Table 3 lists the key parameters of the SCNN design we explore in this paper. The design employs an 8×8 array of PEs, each with a 4×4 multiplier array, and an accumulator buffer with 32 banks. We chose a design point of 1,024 multipliers to match the expected computation throughput required to process HD video in real-time at acceptable frame rates. The IARAM and OARAM are sized so that the sparse activations of AlexNet and GoogLeNet can fit entirely within these RAMs so that activations need not spill to DRAM. The weight FIFO and the activation RAMs each carry a 4-bit overhead for each 16-bit value to encode the coordinates in the compressed-sparse format. In total, the SCNN design includes a total of 1,024 multipliers and 1MB of activation RAM. At the synthesized clock speed of the PE of slightly more than 1 GHz in a 16nm technology, this design achieves a peak throughput of 2 Tera-ops (16-bit multiplies plus 24-bit adds).

Area Analysis. To prototype the SCNN architecture, we designed an SCNN PE in synthesizable SystemC and then used the Catapult high-level synthesis (HLS) tool [25, 26] to generate Verilog RTL. During this step, we used HLS design constraints to optimize the design by mapping different memory structures to synchronous RAMs and latch arrays and pipelining the design to achieve full throughput. We then used Synopsys Design Compiler to perform placement-aware logic synthesis and obtain post-synthesis area estimates in a TSMC 16nm FinFET technology. Table 4 summarizes the area of the major structures of the SCNN PE. A significant fraction of the PE area is contributed by memories (IARAM, OARAM, accumulator buffers), which consume 57% of the PE area, while the multiplier array only consumes 6%. IARAM and OARAM are large in size and consume 25% of the PE area. Accumulator buffers, though smaller in size compared to IARAM/OARAM, are heavily banked (32 banks), contributing to their large area.

5 EXPERIMENTAL METHODOLOGY

CNN performance and power measurements. To model the performance of the SCNN architecture, we rely primarily on a custom-built cycle-level simulator. This simulator is parameterizable across dimensions including number of processing element (PE) tiles, RAM capacity, multiplier array dimensions (F and I), and accumulator buffers (A). The SCNN simulator is driven by the pruned weights and sparse input activation maps extracted from the Caffe Python interface (`pycaffe`) [4] and executes each layers of the network one at a time. As a result, the simulator precisely captures the effects of the sparsity of the data and its effect on load balancing within the SCNN architecture.

Table 5: CNN accelerator configurations.

	# PEs	# MULs	SRAM	Area (mm^2)
DCNN	64	1,024	2MB	5.9
DCNN-opt	64	1,024	2MB	5.9
SCNN	64	1,024	1MB	7.9

We also developed TimeLoop, a detailed analytical model for CNN accelerators to enable an exploration of the design space of dense and sparse architectures. TimeLoop can model a wide range of dataflows, including PT-IS-CP-sparse and PT-IS-CP-dense, as well as the dataflows described in Table 2. Architecture parameters to TimeLoop include the memory hierarchy configuration (buffer size and location), ALU count and partitioning, and dense/sparse hardware support. TimeLoop also includes DRAM bandwidth and energy models for off-chip accesses. TimeLoop analyzes the input data parameters, the architecture, and the dataflows, and then computes (1) the number of cycles to process the layer based on a bottleneck analysis, and (2) the counts of ALU operations and accesses to different buffers in the memory hierarchy. We apply an energy model to the TimeLoop events derived from the synthesis modeling to compute the overall energy required to execute the layer. TimeLoop also computes the overall area of the accelerator based on the inputs from the synthesis modeling. For SCNN, the area model includes all of the elements from the synthesizable SystemC implementation. For dense architectures, area is computed using area of the major structures (RAMs, ALUs, and interconnect) derived from the SystemC modeling.

Architecture configurations. Table 5 summarizes the major accelerator configurations that we explore, including both dense and sparse accelerators. All of the accelerators employ the same number of multipliers so that we can compare the performance of the accelerators with the same computational resources. The dense DCNN accelerator operates solely on dense weights and activations and employs a *dot-product* dataflow called PT-IS-DP-dense. Dot products are usually efficient for dense accelerators because of reduced accumulation-buffer accesses, although this comes at the cost of reduced spatial reuse of weights and input activations. The optimized DCNN-opt architectures have the same configuration as DCNN but employ two optimizations: (1) compression/decompression of activations as they are transferred out of/into DRAM, and (2) multiply ALU gating to save energy when a multiplier input is zero. The DCNN architectures are configured with 2MB of SRAM for holding inter-layer activations, and can hold all of them for AlexNet and GoogLeNet. The SCNN configuration matches the architecture described in Section 4, and includes a total of 1MB of IARAM + OARAM. Because the activations are compressed, this capacity enables all of the activation data for the two networks to be held on

chip, without requiring DRAM transfers for activations. The larger VGGNet requires the activation data to be transferred in and out of DRAM. The last column of the table lists the area required for each accelerator; for simplicity, we total only the area for the PE array and SRAM banks, and omit any area for wiring among the PEs. As the interconnect bandwidth requirements for SCNN are less than for dense architectures due to the compression, including interconnect area for both dense and sparse architectures would close the area gap somewhat between the two. While SCNN has smaller activation RAM capacity, its larger size is due to the banked accumulator buffers, as described in Section 4.

Benchmarks. As described in Section 2, we use AlexNet and GoogLeNet for the bulk of our experiments. For GoogLeNet, we primarily focus on the convolutional layers that are within the *inception* modules [31]. VGGNet is known to be over-parameterized, which results in an excessively large amount of inter-layer activation data (6 MB or about 4× the largest GoogLeNet layer). Nonetheless, we use VGGNet as a proxy for large input data (such has high-resolution images) to explore the implications of coarse-grained temporal tiling on accelerator architectures. We leverage two different types of benchmarks to evaluate SCNN's efficiency. First, we developed synthetic network models where we can adjust the degree of sparsity of both weights and activations. These synthetic models are used to explore the sensitivity of the architectures to sparsity parameters (detailed in Section 6.1.). Second, we generate the actual sparse network models using the CNN pruning algorithm proposed by Han et al. [17], which we employ in cycle-level performance simulation. The pruned models have been retrained to achieve the same level of classification accuracy provided by the dense model, and we use this pruned model to obtain the post-ReLU activation maps to feed it into our performance simulator.

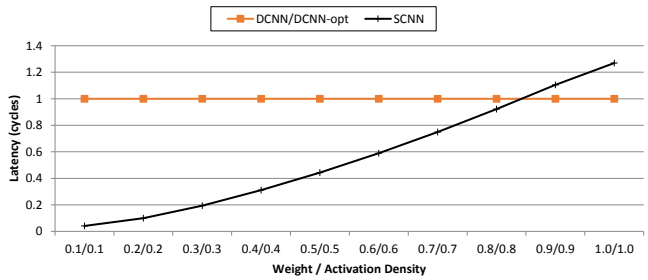
6 EVALUATION

This section first evaluates the sensitivity of SCNN to the sparseness of weights and activations using a synthetic CNN benchmark. We then measure the performance and energy-efficiency of SCNN versus a dense CNN accelerator, using AlexNet, GoogLeNet, and VGGNet. For brevity, all the inception modules in GoogLeNet are denoted as `IC_id` in all of the figures discussed in this section.

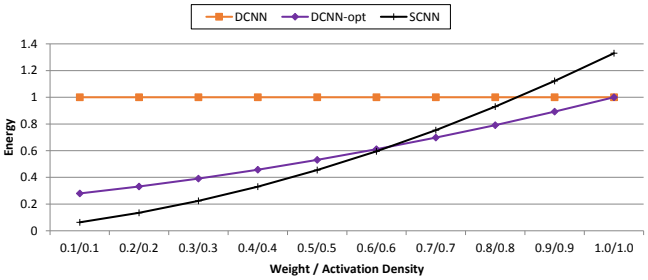
6.1 Sensitivity to CNN Sparsity

We first compare the performance and energy-efficiency of the `SCNN`, `DCNN`, and `DCNN-opt` architectures as we artificially sweep the weight and activation densities in GoogLeNet's layers from 100% (fully dense) down to 10%. The X-axis of Figure 8 simultaneously scales both weight and activation density. The 0.5/0.5 point corresponds to 50% weight density, 50% activation density, and 25% of the multiplication operations relative to the fully-dense 1.0/1.0 point. Figure 8a shows that at full density, `SCNN` only achieves about 79% of the performance of `DCNN/DCNN-opt`[1] because of `SCNN`'s dataflow is more susceptible to certain multiplier underutilization effects than `DCNN`'s dot-product dataflow. As density decreases to about 0.85/0.85, `SCNN` starts to perform better than `DCNN`, ultimately reaching a 24× improvement at the sparsest evaluated point with a density of 0.1/0.1.

[1] `DCNN-opt` performance is identical to `DCNN` because it only optimizes for energy.



(a) Performance



(b) Energy

Figure 8: GoogLeNet performance and energy versus density.

Figure 8b first shows that `DCNN-opt`'s energy optimizations of zero gating and DRAM traffic compression enable it to be better than `DCNN` at every level of density. These energy optimizations are surprisingly effective despite their minimal effect on the design of the accelerator. At full density, `SCNN` consumes 33% more energy than either dense architecture due to the overheads of storing and maintaining the sparse data structures. `SCNN` becomes more efficient than `DCNN` at about 0.83/0.83 density and more efficient than `DCNN-opt` at 0.6/0.6 density. At the sparsest evaluated point of 0.1/0.1 density, `SCNN` consumes 6% of the energy of `DCNN` and 23% of the energy of `DCNN-opt`. Given the density measurements of the networks in Figure 1, we expect `SCNN` to (a) significantly outperform the dense architectures on nearly all the layers of the networks we examined, (b) surpass the energy efficiency of `DCNN` on a majority of layers, and (c) stay roughly competitive with the energy-efficiency of `DCNN-opt` across most layers.

6.2 SCNN Performance and Energy

Performance. We compare the performance of SCNN to the baseline dense DCNN accelerator and to an *oracular* SCNN design (`SCNN(oracle)`) that represents an upper bound on performance. The performance of `SCNN(oracle)` is derived by dividing the number of multiplication operations required for a Cartesian product-based convolution (Section 3) by 1,024, the number of multipliers in the architectures we examine. Figure 9 summarizes the speedups offered by `SCNN` versus a dense CNN accelerator. Overall, `SCNN` consistently outperforms the DCNN design across all the layers of AlexNet, GoogLeNet, and VGGNet, achieving an average 2.37×, 2.19×, and 3.52× network-wide performance improvement, respectively.

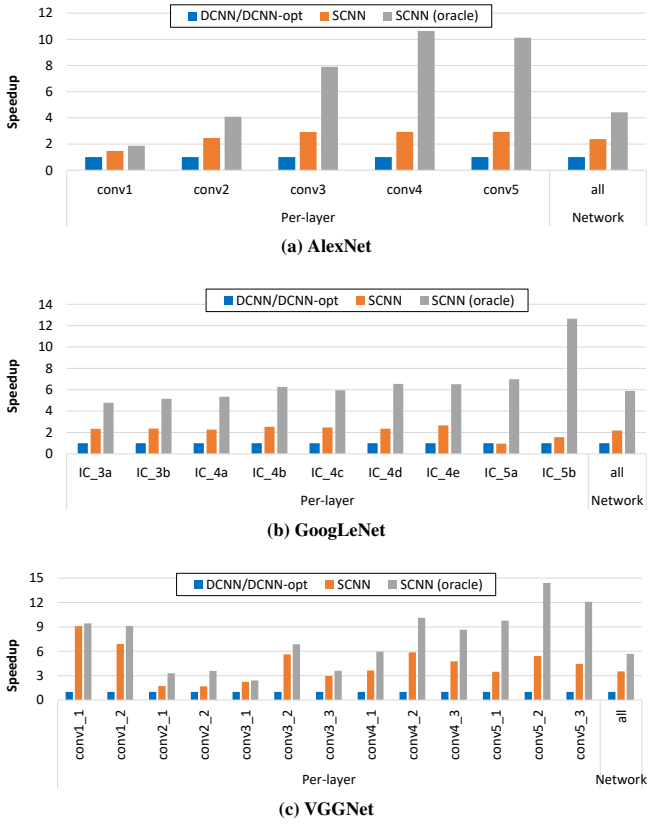


Figure 9: SCNN performance comparison.

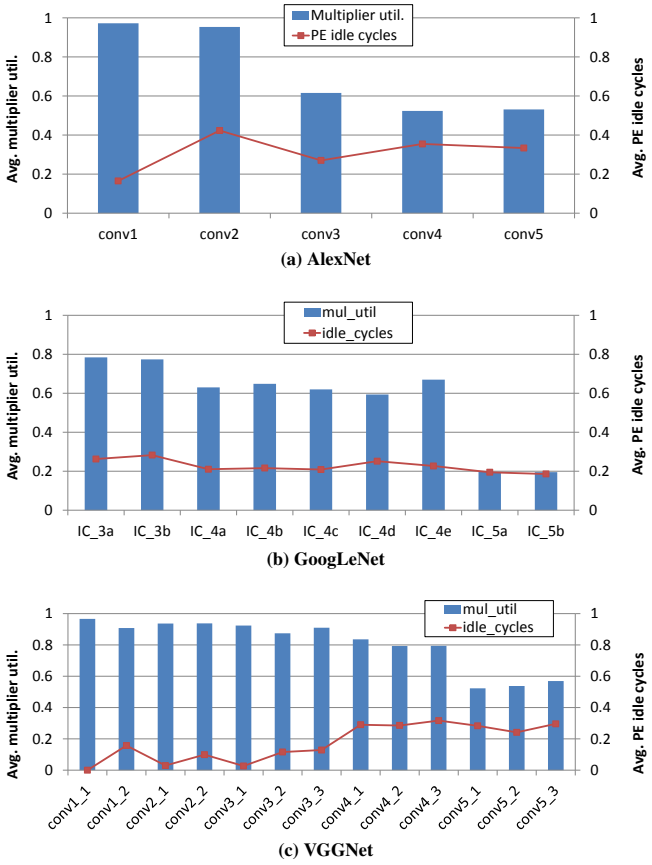


Figure 10: Average multiplier array utilization (left-axis) and the average fraction of time PEs are stalled on a global barrier (right-axis), set at the boundaries of output channel groups.

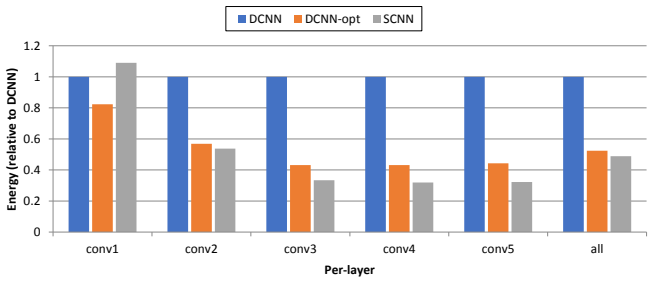
The performance gap between SCNN versus SCNN(oracle) widens in later layers of the network, i.e., the rightmost layers on the x-axis of Figure 9. SCNN suffers from two forms of inefficiency that cause this gap. First, the working set allocated to each PE tends to be smaller in the later layers (e.g., IC_5b) than in the earlier layers (e.g., IC_3a). As a result, assigning enough non-zero activations and weights in the later layers to fully utilize a PE's multiplier array becomes difficult. In other words, SCNN can suffer from *intra-PE fragmentation* when layers do not have enough useful work to fully populate the vectorized arithmetic units.

The second source of inefficiency stems from the way the PT-IS-CP-sparse dataflow partitions work across the array of PEs, which could lead to load imbalance among the PEs. Load imbalance results in under-utilization because the work corresponding to the next output-channel group K_{c+1} can only start after the PEs complete the current output-channel group K_c. The PEs effectively perform an *inter-PE synchronization barrier* at the boundaries of output-channel groups which can cause early-finishing PEs to idle while waiting for laggards.

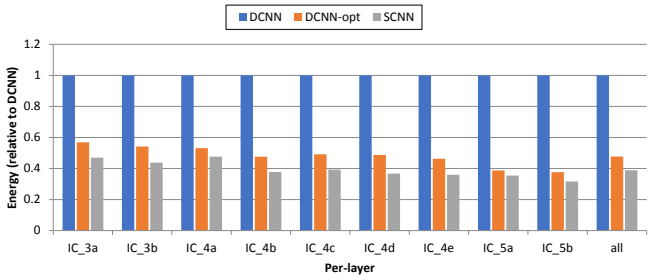
Figure 10 quantitatively demonstrates the intra-PE fragmentation in the multiplier arrays. Fragmentation is severe in the last two inception modules of GoogLeNet, with average multiplier utilization at less than 20%. In this layer, three out of the six convolutional sub-layers within the inception module have a filter size of 1×1, resulting in a maximum of 8 non-zero weights within an output-channel group with a K_c value of 8. Nonetheless, later layers generally account for a small portion of the overall execution time as the input activation volume (i.e., $H\times W\times C$) gradually diminishes across the layers.

The right y-axis of Figure 10 demonstrates the effect of load imbalance across the PEs by showing the fraction of cycles spent waiting at an inter-PE barrier. Although the inter-PE global barriers and intra-PE fragmentation prevents SCNN from reaching similar speedups offered by SCNN(oracle), it still provides an average 2.7× network-wide performance boost over DCNN across the three CNNs we examined.

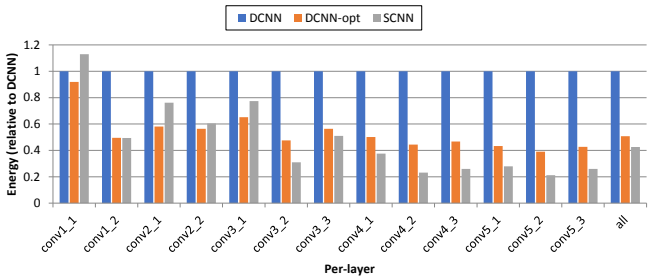
Energy-efficiency. Figure 11 compares the energy of the three accelerator architectures across the layers of the three networks. On average, DCNN-opt improves energy-efficiency by 2.0× over DCNN, while SCNN improves efficiency by 2.3× . SCNN's effectiveness varies widely across layers depending on the layer density, ranging from 0.89× to 4.7× improvement over DCNN and 0.76× to 1.9× improvement over DCNN-opt. Input layers such as VGGNet_conv1_1 and AlexNet_conv1 usually present a challenge for sparse architectures because of their 100% input activation density. In such cases, the overheads of SCNN's structures such as the crossbar and distributed



(a) AlexNet



(b) GoogLeNet



(c) VGGNet

Figure 11: SCNN energy-efficiency comparison.

accumulation RAMs overshadow any benefits from fewer arithmetic operations and data movement.

These results reveal that although the straightforward DCNN-opt architecture is unable to improve performance, it is remarkably effective at achieving good energy-efficiency on moderately sparse network layers. Nonetheless, SCNN is on average even more energy-efficient across our benchmark networks while providing a tremendous performance advantage over both DCNN and DCNN-opt.

6.3 PE Granularity

As outlined in Section 6.2, both cross-PE global barriers and intra-PE multiplier array fragmentation can contribute to degradation in the performance of SCNN. We quantify the effects of both of these factors on system performance by conducting the following sensitivity study. Assuming a fixed 1,024 multipliers for the accelerator, we sweep the total number of PEs on-chip from 64 (8×8 PEs, 16 multipliers per PE) down to 4 (2×2 PEs, 256 multipliers per PE).

Table 6: Characteristics of evaluated accelerators.

Architecture	Gate MACC	Skip MACC	Skip Buffer Access	Skip DRAM Access	Inner Spatial Dataflow
DCNN	–	–	–	–	Dot Product
DCNN-opt	A+W	–	–	A+W	Dot Product
SCNN-SparseA	A	A	A	A	Cartesian Product
SCNN-SparseW	W	W	W	W	Cartesian Product
SCNN	A+W	A+W	A+W	A+W	Cartesian Product

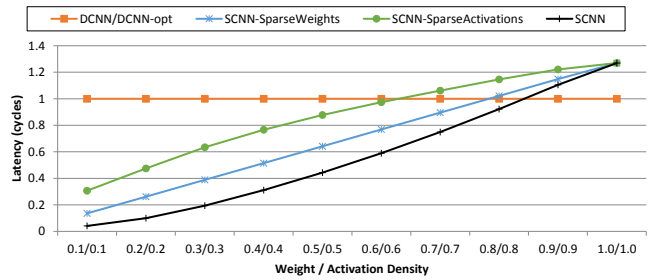
Clearly, an SCNN with 4 PEs can better sustain the effects of the global barriers than an SCNN with 64 PEs. However, the 4 PE configuration is also more likely to suffer from intra-PE fragmentation because each PE must now process a larger working set to fully utilize the math units. When evaluated on GoogLeNet, SCNN with 64 PEs achieves an 11% speedup over the one with 4 PEs as it does a better job utilizing the math arrays (average 59% math utilization versus 35%). We observed similar trends for AlexNet and VGGNet, concluding that addressing intra-PE fragmentation is more critical than inter-PE barriers for system-wide performance with the PT-IS-CP-sparse dataflow.

6.4 Effects of Weight and Activation Sparsity

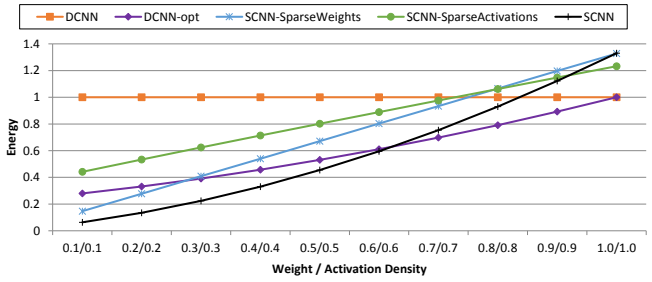
While Figure 1 shows that sparseness is abundant in both activations and pruned weights, isolating the effect of sparsity provides insight into different accelerator architecture trade-offs. We run the density-sweep experiments from Section 6.1 on two architectures derived from the SCNN design. The SCNN-SparseA architecture only takes advantage of sparsity in activations and is similar in spirit to Cnvlutin [1]. The SCNN-SparseW architecture only takes advantage of sparsity in weights and is similar in spirit to Cambricon-X [34].

Table 6 tabulates the characteristics of these new architectures alongside our baseline SCNN, DCNN, and DCNN-opt architectures. These five architectures together cover a broad design space of sparse architectures, and also encompass the types of sparsity explored in prior research, as described in Table 2. However, because of significant differences in dataflow, buffer sizing/ organization, and implementation choices (such as the use of eDRAM), our evaluated architectures cannot precisely represent those prior proposals.

Figure 12 demonstrates that SCNN is consistently superior to the SCNN-SparseA and SCNN-SparseW configurations in both performance and energy across the entire density range. The only exception is that at very high density levels (weight/activation density greater than 0.9/0.9), SCNN-SparseA is slightly more energy-efficient because of the removal of overheads to manage sparse weights. The input-stationary temporal loop around the Cartesian product makes these architectures extremely effective at filtering IARAM accesses, resulting in the IARAM consuming less than 1% of the total energy. The weight FIFO is accessed more frequently in SCNN, resulting in the weight FIFO consuming around 6.7% of total energy. Therefore, removing the weight encoding overheads in SCNN-SparseA shows a far greater benefit than removing the activation encoding overheads in SCNN-SparseW. However, as density is reduced, the filtering advantage of the input-stationary loop starts diminishing relative to the weight FIFO. At a density of 0.8/0.8, SCNN-SparseW surpasses the energy-efficiency of SCNN-SparseA, ultimately reaching a 2.5×



(a) Performance



(b) Energy

Figure 12: GoogLeNet performance and energy versus density for sparse weights, sparse activations, and both

advantage at 0.1/0.1. For a nominal density of 0.4/0.4, SCNN achieves performances advantages of 1.7× and 2.6× over SCNN-SparseW and SCNN-SparseA, respectively; SCNN achieves energy-efficiency advantages of 1.6× and 2.1× over SCNN-SparseW and SCNN-SparseA, respectively.

7 RELATED WORK

Previous efforts to exploit sparsity in CNN accelerators have focused on reducing energy or saving time, which will invariably also save energy. Eliminating the multiplication when an input operand is zero is a natural way to save energy. Eyeriss [8] gates the multiplier when it sees an input activation of zero. Because the sparsity of weights is not significant for non-pruned networks, Eyeriss opted not to gate the multiplier on zero weights. This gating approach will save energy, but not execution time. Not only does SCNN obtain energy efficiency by eliminating unnecessary multiplications, it also reduces execution time by eliminating the dead multiplier cycles inherent in zero-gating approaches.

Another approach to reducing energy is to reduce data transfer costs when the data is sparse. Eyeriss uses a run length encoding scheme when transferring activations to and from DRAM. This approach saves energy (and time) by reducing the number of DRAM accesses. However because the data is kept in an expanded form in the on-chip memory hierarchy, such architectures cannot completely eliminate energy on the data transfers from one internal buffer to another internal buffer or to the multipliers. Eyeriss does, however, save the energy for accessing the weight buffer when the associated activation is zero. SCNN also uses a compressed representation for all data coming from DRAM, but also maintains that compressed representation in all the on-die buffers.

Other sparse CNN accelerator architectures do not exploit all of the sparsity opportunities leveraged by SCNN. For example, Cnvlutin compresses activation values based on the ReLU operator, but it does not employ pruning to exploit weight sparsity [1]. Cambricon-X employs weight sparsity to keep only non-zero weights in its internal buffers [34]. However, it does not compress activation data between DRAM and the accelerator. Nor does it keep activations in a compressed form in the internal buffers, except in the queues directly delivering activations to the multipliers. In contrast, SCNN keeps both weights and activations in a compressed form in both DRAM and internal buffers. This approach saves data transfer time and energy on all data transfers and allows the chip hold larger models for a given amount of internal storage.

Avoiding delivery of zero-valued activations or weights to the multipliers can save time by eliminating ineffectual multiplier cycles. Cnvlutin selects only non-zero activation values for delivery as multiplier operands, but does occupy a multiplier with zero-valued weights. Cambricon-X can save a compute cycle for zero-valued weights, but still wastes times computing multiplications for operands with zero-valued activations. The Deep Learning Accelerator Core (DLAC) architecture paper mentions that it can skip computation on zero-valued operands to improve performance, but the architecture does not appear to employ zero compression [32]. SCNN does not deliver either zero activations or weights to the multipliers and maximally exploits opportunities of CNN sparsity.

The EIE CNN accelerator uses a compressed representation of both activations and weights, and only delivers non-zero operands to the multipliers [15]. However, EIE is designed for the fully connected layers of a CNN model, while SCNN targets the convolutional layers, which encompass the vast majority of the computations in CNNs [10, 24]. Finally, Alwani et al. propose to fuse adjacent layers in a dense CNN accelerator so that intermediate activations between layers can be kept on chip [2]. By compressing the activations, SCNN can typically keep all of the activations on-chip, without requiring a more complicated fused algorithm.

8 CONCLUSION

This paper presents the Sparse CNN (SCNN) accelerator architecture for inference in convolutional neural networks. SCNN exploits sparsity in both weights and activations using the sparse planar-tiled input-stationary Cartesian product (PT-IS-CP-sparse) dataflow. This approach enables SCNN to use a novel Cartesian product-based computation architecture that maximizes reuse of weights and activations within a set of distributed processing elements. In addition, it allows the use of a dense compressed representation for both weights and activations to be used through almost the entire processing flow. This allows for reduced data movement and increased on-die storage capacity than alternatives approaches. Our results show that with equivalent area, the SCNN architecture starts to beat an energy-optimized dense architecture on energy efficiency when the weights and activations are each less than 85% dense. On three contemporary networks (AlexNet, GoogLeNet, and VGGNet) SCNN achieves performance improvements over a comparably provisioned dense CNN accelerator by a factor of 2.7×, while still being 2.3× more energy-efficient.

REFERENCES

[1] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. 2016. Cnvlutin: Ineffectual-Neuron-Free Deep Convolutional Neural Network Computing. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 1–13.

[2] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. 2016. Fused-Layer CNN Accelerators. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*.

[3] Dario Amodei, Rishita Anubhai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Jingdong Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, Erich Elsen, Jesse Engel, Linxi Fan, Christopher Fougner, Tony Han, Awni Hannun, Billy Jun, Patrick LeGresley, Libby Lin, Sharan Narang, Andrew Ng, Sherjil Ozair, Ryan Prenger, Jonathan Raiman, Sanjeev Satheesh, David Seetapun, Shubho Sengupta, Yi Wang, Zhiqian Wang, Chong Wang, Bo Xiao, Dani Yogatama, Jun Zhan, and Zhenyao Zhu. 2015. Deep Speech 2: End-To-End Speech Recognition in English and Mandarin. https://arxiv.org/abs/1512.02595. (2015).

[4] Caffe 2016. Caffe. http://caffe.berkeleyvision.org. (2016).

[5] Caffe 2017. Caffe Model Zoo. https://github.com/BVLC/caffe/wiki/Model-Zoo. (2017).

[6] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. DianNao: A Small-footprint High-throughput Accelerator for Ubiquitous Machine-learning. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*. 269–284.

[7] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 367–379.

[8] Yu-Hsin Chen, Tushar Krishna, Joel Emer, and Vivienne Sze. 2016. Eyeriss: An Energy-efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. In *Proceedings of the International Solid State Circuits Conference (ISSCC)*.

[9] Ronan Collobert, Jason Weston, Leon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. 2011. Natural Language Processing (Almost) From Scratch. https://arxiv.org/abs/1103.0398. (2011).

[10] Jason Cong and Bingjun Xiao. 2014. Minimizing Computation in Convolutional Neural Networks. In *Proceedings of the International Conference on Artificial Neural Networks (ICANN)*. 281–290.

[11] Gregory Diamos, Shubho Sengupta, Bryan Catanzaro, Mike Chrzanowski, Adam Coates, Erich Elsen, Jesse Engel, Awni Hannun, and Sanjeev Satheesh. 2016. Persistent RNNs: Stashing Recurrent Weights On-Chip. In *Proceedings of the International Conference on Machine Learning (ICML)*.

[12] 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 International Symposium on Computer Architecture (ISCA)*. 92–104.

[13] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. 2017. TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*. 751–764.

[14] Alex Graves and Jurgen Schmidhuber. 2005. Framewise Phoneme Classification With Bidirectional LSTM and Other Neural Network Architectures. In *Neural Networks*.

[15] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark Horowitz, and Bill Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 243–254.

[16] Song Han, Huizi Mao, and William J. Dally. 2015. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. https://arxiv.org/abs/1510.00149. (2015).

[17] Song Han, Jeff Pool, John Tran, and William J. Dally. 2015. Learning Both Weights and Connections for Efficient Neural Networks. In *Proceedings of the International Conference on Neural Information Processing Systems (NIPS)*. 1135–1143.

[18] Awni Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, and Andrew Y. Ng. 2014. Deep Speech: Scaling Up End-To-End Speech Recognition. https://arxiv.org/abs/1412.5567. (2014).

[19] Kaming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. https://arxiv.org/abs/1512.03385. (2015).

[20] Gao Huang, Yu Sun, Zhuang Liu, Daniel Sedra, and Kilian Weinberger. 2016. Deep Networks with Stochastic Depth. https://arxiv.org/abs/1603.09382. (2016).

[21] ImageNet. 2016. http://image-net.org. (2016).

[22] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of the International Conference on Neural Information Processing Systems (NIPS)*.

[23] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep Learning. *Nature* 521 (May 2015), 436–444.

[24] Jonathan Long, Evan Shelhamer, and Trevor Darrell. 2015. Fully Convolutional Networks for Semantic Segmentation. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*.

[25] Grant Martin and Gary Smith. 2009. High-Level Synthesis: Past, Present, and Future. *IEEE Design & Test of Computers* 26, 4 (July/August 2009), 18–25.

[26] Mentor 2017. Catapult High-Level Synthesis. https://www.mentor.com/hls-lp/catapult-high-level-synthesis. (2017).

[27] NVIDIA 2016. NVIDIA cuDNN. https://developer.nvidia.com/cudnn. (2016).

[28] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Saekyu Lee, Jose Miguel Hernandez Lobato, Gu-Yeon Wei, and David Brooks. 2016. Minerva: Enabling Low-Power, High-Accuracy Deep Neural Network Accelerators. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 267–278.

[29] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. 2016. vDNN: Virtualized Deep Neural Networks for Scalable, Memory-Efficient Neural Network Design. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*.

[30] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. https://arxiv.org/abs/1409.1556. (May 2015).

[31] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going Deeper with Convolutions. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*.

[32] Ganesh Venkatesh, Eriko Nurvitadhi, and Debbie Marr. 2016. Accelerating Deep Convolutional Networks Using Low-precision and Sparsity. https://arxiv.org/abs/1610.00324. (2016).

[33] Richard W. Vuduc. 2003. *Automatic Performance Tuning of Sparse Matrix Kernels*. Ph.D. Dissertation. University of California, Berkeley.

[34] 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 *Proceedings of the International Symposium on Microarchitecture (MICRO)*.