Anticipating and Eliminating Redundant Computations in Accelerated Sparse Training

Jonathan S. Lew
jonathan@ece.ubc.ca
Electrical and Computer Engineering
University of British Columbia
Vancouver, BC, Canada

Yunpeng Liu
yunpengl@cs.ubc.ca
Computer Science
University of British Columbia
Vancouver, BC, Canada

Wenyi Gong
wgong@ece.ubc.ca
Electrical and Computer Engineering
University of British Columbia
Vancouver, BC, Canada

Negar Goli*
Huawei Technologies
Vancouver, BC, Canada

R. David Evans*
Borealis AI
Vancouver, BC, Canada

Tor M. Aamodt
aamodt@ece.ubc.ca
Electrical and Computer Engineering
University of British Columbia
Vancouver, BC, Canada

ABSTRACT

Deep Neural Networks (DNNs) are the state of art in image, speech, and text processing. To address long training times and high energy consumption, custom accelerators can exploit sparsity, that is zero-valued weights, activations, and gradients. Proposed sparse Convolution Neural Network (CNN) accelerators support training with no more than one dynamic sparse convolution input. Among existing accelerator classes, the only ones supporting two-sided dynamic sparsity are outer-product-based accelerators. However, when mapping a convolution onto an outer product, multiplications occur that do not correspond to any valid output. These Redundant Cartesian Products (RCPs) decrease energy efficiency and performance. We observe that in sparse *training*, up to 90% of computations are RCPs resulting from the convolution of large matrices for weight updates during the backward pass of CNN training.

In this work, we design a mechanism, ANT, to anticipate and eliminate RCPs, enabling more efficient sparse training when integrated with an outer-product accelerator. By anticipating over 90% of RCPs, ANT achieves a geometric mean of 3.71× speed up over an SCNN-like accelerator [67] on 90% sparse training using DenseNet-121 [38], ResNet18 [35], VGG16 [73], Wide ResNet (WRN) [85], and ResNet-50 [35], with 4.40× decrease in energy consumption and 0.0017mm^2 of additional area. We extend ANT to sparse matrix multiplication, so that the same accelerator can anticipate RCPs in sparse fully-connected layers, transformers, and RNNs.

CCS CONCEPTS

• **Computer systems organization → Systolic arrays**; • **Computing methodologies → Machine learning**.

*Work performed while at the University of British Columbia

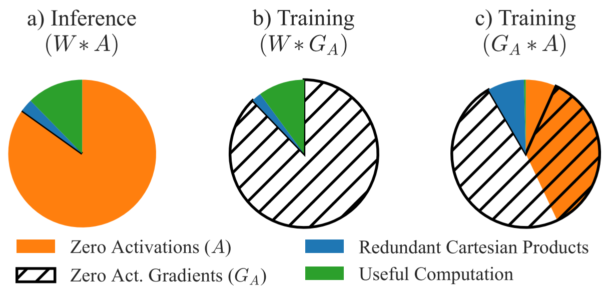

Figure 1: On an SCNN-like sparse accelerator [67], the proportion of useful computations during the $G_A * A$ phase of CNN training (c) is vanishingly small. This is in contrast to $W * A$ inference (a) and $W * G_A$ training phases, where useful computation forms a large portion of the non-zero computation being performed.

KEYWORDS

Sparse CNN Training, Sparse Matrix Multiplication, Hardware Acceleration

ACM Reference Format:
Jonathan S. Lew, Yunpeng Liu, Wenyi Gong, Negar Goli, R. David Evans, and Tor M. Aamodt. 2022. Anticipating and Eliminating Redundant Computations in Accelerated Sparse Training . In *The 49th Annual International Symposium on Computer Architecture (ISCA '22), June 18–22, 2022, New York, NY, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3470496.3527404

1 INTRODUCTION

Deep Neural Networks (DNNs) have rapidly become the state-of-the-art implementation approach in many fields such as image, speech, and text processing [4, 21, 77]. Typically, larger networks (wider, deeper, or more neurons) lead to improved accuracy [35, 39, 85]. However, accuracy improvements often demand significantly increased computation resources for training. We focus on Convolutional Neural Networks (CNNs), which consume 24% of training time in Google datacenters [44]. CNNs are used in diverse applications such as self-driving cars, image processing, and playing board games [6, 35, 72]. Training demands are further compounded by network architecture search, which trains and searches among thousands of similar models [77, 90].

While Graphics Processor Units (GPUs) currently dominate training hardware due to their flexibility, rapidly growing computational demands require new solutions to train DNNs efficiently. Hardware acceleration and sparse training are two such solutions. Training accelerators exchange flexibility for considerable improvements in performance and energy efficiency [45]. Sparse accelerators [10, 17, 57, 67] can improve performance by skipping zero multiplications which occur during DNN inference and training.

Inference acceleration is a well-studied challenge, and both dense and sparse computation accelerators have been proposed and developed in hardware [1, 8, 9, 54]. Taxonomies of dataflows for dense computation have been investigated [8, 84]. Numerous proposals for sparse inference acceleration have been studied including those taking advantage of sparse weights [17, 86, 89], activations [3], both weights and activations [31, 33, 67], and bit sparsity [2, 17]. Recent studies have also examined sparse recurrent networks [32]. Training has different constraints compared to inference. Significant adaptation of inference accelerators is required for training acceleration, e.g., Bit–Tactical [17] to TensorDash [57] required handling dynamic sparsity, tensor reuse, and storage.

Dense training accelerator designs have been proposed [10, 47, 66] and are being implemented, sold and/or deployed by industry including designs by Intel/Habana [60], Cerebras [26], and Google [43–45, 65]. The existence of dense training accelerator products implies a significant demand for training acceleration. Despite this, there are few accelerators supporting sparse training, notably NVIDIA's Ampere architecture, which supports a restricted 50% structured sparsity [1], and Cerebras Wafer Scale Engine (CWSE), which uses sparsity harvesting technology [24]. Recent work has shown training algorithms where sparsity can be introduced in activations and weights in the training process, resulting in up to 10× reduction in total compute [69]. However, works that investigated sparse training in hardware have focused on exploiting sparsity in only one of the inputs to the convolution [57, 68, 83]. In this work, we examine exploiting sparsity in *both* inputs to the convolution and make further optimizations for training.

We focus our efforts on outer-product accelerators, as they are currently the only option that already supports the two-sided dynamic sparsity encountered during training (Section 2.2). Our key insight is that training using an outer-product accelerator with sparse convolutions results in many unused computations, which we refer to as Redundant Cartesian Products (RCPs). During sparse convolution, the product of every non-zero kernel value with every non-zero image value is computed (a cartesian product). RCPs are cartesian products that do not correspond to any valid output index. Figure 1 shows the partial product breakdown for ImageNet/ResNet18 [19, 35] convolutions on an SCNN-like sparse accelerator [67]. A partial product is any potential multiplication between a weight (W), activation (A), or activation gradient (G_A). As products can involve both a zero activation (A) and gradient (G_A), there is overlap between the products with zero activations and zero gradients (shaded+hatched in c). The primary takeaway is that RCPs represent a large portion of the non-zero computations (blue, Figures 1a-c), and that sparse training conditions drastically increase the number of RCPs, up to 96% of useful computations in the $G_A * A$ phase of training (Figures 1a-b vs. Figure 1c). We explore the causes for this in Section 3. Most prior works examine inference, where RCPs are fewer or non-existent [1, 8, 9, 17, 54, 67, 84]. Addressing RCPs is a critical problem for efficient sparse training.

Our contributions are as follows:

- We describe and characterize Redundant Cartesian Products (RCPs) occurring in outer-product-based convolution accelerators, including an algorithm to detect and eliminate RCPs in outer-product accelerators (Section 3). Our work is the first to analyze RCPs in a training context, where they waste over 90% of multiplications during the weight gradient calculation.
- We propose a dataflow- and memory-agnostic ANTicipator Accelerator (ANT) to identify and eliminate over 90% of RCPs, resulting in a geometric mean of 3.71× speed up over an SCNN-like accelerator on 90% sparse training using DenseNet-121 [38], ResNet18 [35], VGG16 [73], Wide ResNet (WRN) [85], and ResNet-50 [35], with 4.40× decrease in energy and 0.0017mm^2 of additional area (Sections 4, 7) .
- We extend ANT to a matrix multiplication implementation of a text translation transformer [79] and a text classification Recurrent Neural Network (RNN) [78], anticipating and eliminating over 99% of the RCPs.

2 BACKGROUND

In this section we summarize CNN training and highlight inefficiency exposed by prior sparse acceleration approaches. In Section 5 we discuss applications of ANT to other network architectures.

2.1 Sparse Convolutional Network Training

Convolutional Neural Networks (CNNs) utilize convolution layers, where a *kernel* is slid across an *image* (Figure 2a). We use this terminology instead of the common *weights* and *activations*, as, depending on the phase of training, different matrices take the role of *kernel* and *image*. By contrast, during inference, the *kernel* and *image* always correspond to the weight and activation matrices, respectively. The convolution *image* can contain visual or non-visual data, such as board game positions [72] or audio spectra [40].

Backprop [71] is the most widely used algorithm for obtaining weight gradients during DNN training. For the L^{th} convolution layer, the three phases of Backprop produce three convolution operations, expressed as:

$$A^{L+1} = W^L * A^L \qquad \text{(Forward Pass)} \qquad (1)$$

$$G_A^L = R(W^L) * G_A^{L+1} \qquad \text{(Backward Pass)} \qquad (2)$$

$$G_W^L = G_A^{L+1} * A^L \qquad \text{(Update)} \qquad (3)$$

where we use the "$*$" symbol to represent convolution, i.e. kernel $*$ image. W and A correspond to the weights and activations, and G_W and G_A correspond to the gradient of the loss with respect to the weights and activations. $R(W)$ describes a generic rotation operation, whereby the W matrix is transposed and reshaped to align its dimensions to G_A. The final step is to update the weights using

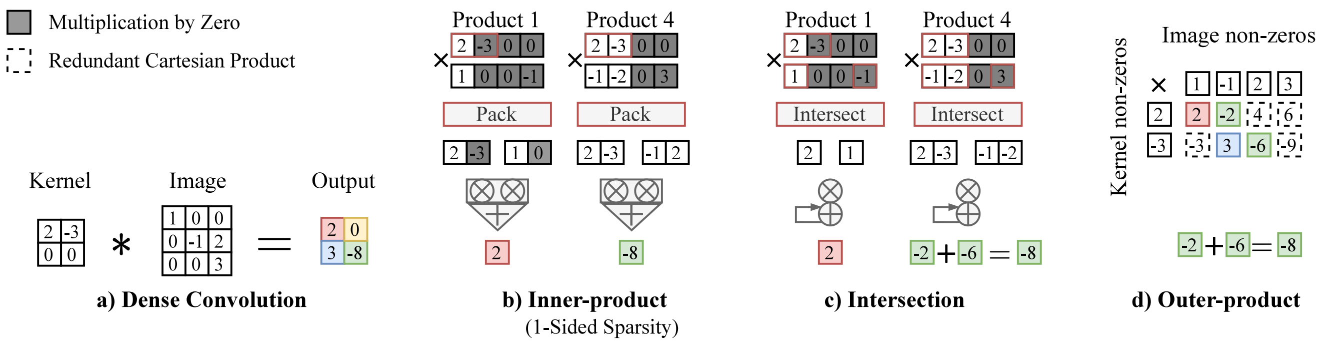

Figure 2: Convolution accelerator classes, showing zero products and Redundant Cartesian Products (RCPs). a) An example convolution of a 2×2 kernel and 3×3 image, b) Inner product/dot product, c) Intersection/streaming and, d) Outer-product.

Stochastic Gradient Descent (SGD). We refer to these convolutions as $W * A$, $W * G_A$, and $G_A * A$ henceforth.

Any of W, A, or G_A can be sparse during training. Weights can be sparse either by pruning small values or regularization to push values towards zero [20, 34, 52, 53, 55, 69, 87]. Sparse weights also occur when training for many other pruning techniques [20, 34, 52, 53, 55, 69, 87]. Activations are often sparse due to the use of ReLU, and sparsity can be increased using regularizers or pruning [27, 61, 64, 74]. ReLU also induces sparsity in the activation gradients (G_A). Recent work demonstrates sparsifying gradients (G_A) or activations and weights (A, W) during training significantly increases (redundant) zero multiplications on dense accelerators with little accuracy loss [28, 69].

2.2 Sparse Convolution Accelerators

The design space for accelerating sparse convolutions is vast as we aim to find a way to decrease energy consumption and improve performance. This section examines three broad classes of sparse convolution accelerators: inner-product, outer-product, and intersection. The best designs minimize memory traffic and multiplications by exploiting sparsity patterns in the convolution kernel and image matrices. However, when used for training, all classes have serious drawbacks, which lead to poor performance, area, and/or utilization. Figure 2a) illustrates a convolution operation: The kernel "shifts" horizontally and veritically over an image and for each shift the overlapping elements of kernel and image are multiplied together to form products and the products are summed together to generate an output element. For example, the output -8 in the green square in the lower right is computed as $(2 \times -1) + (-3 \times 2) + (0 \times 0) + (0 \times 3)$.

Inner-Product. Inner-product accelerators are classified by parallel multipliers followed by an accumulator tree, implementing a dot-product operation. Figure 2b) shows how the convolution in Figure 2a) is performed using an inner product. In the sparse case, identification and packing of non-zero values is required to sufficiently fill the multiplier array (Figure 2b). Packing is non-trivial: TensorDash [57] achieves this for one input by restricting to a finite number of packings and by decompressing the sparse input to determine the location of non-zeros. Enforcing structured sparsity [1, 89] simplifies packing but is incompatible with training, which has a dynamic sparsity pattern. To the best of our knowledge, no inner-product works take advantage of two-sided dynamic sparsity during training due to difficulties packing.

Additionally, inner-product accelerators require transposing using IM2COL to convert the convolution image to a dot-product-friendly format [81]. IM2COL creates duplicate multiplications, even when performed implicitly (using address conversion). Activations are larger during training vs. inference, which can further strain the memory system and decrease performance when using IM2COL.

Intersection. Figure 2c illustrates how intersection accelerators first identify matching non-zero dot-product elements before multiplying and accumulating them serially to remove unnecessary multiplications. GoSpa [18] and SparTen [31] fall under this class. The intersection operation (Figure 2c) involves identifying matching non-zero pairs in the kernel and image streams. When both kernel and image are compressed (e.g., using CSR), this operation becomes expensive. SparTen [31] creates a data dependency between the intersection and multiplication, introducing stalls. GoSpa [18] addresses this by pre-computing a larger portion of the intersection operation, improving performance over SparTen by 1.17×-1.38×.

However, all current intersection accelerators are not suitable for training due to the existence of dynamic sparsity. During inference, the weight matrix has a fixed sparsity pattern which allows a portion of the intersection to be computed ahead of time. Both kernel and image sparsity patterns change throughout training, and recomputing the entire intersection operation for every weight, activation, and gradient introduces large performance overheads. The most obvious path for GoSPA to handle dual-sided dynamic sparsity would be to re-compute the Static Sparsity Filter (SSF) every convolution, which is effectively a bitmask of nonzero values. Unpacking Compressed Sparse Row (CSR) indices and composing the bitmask would require multiple cycles (due to SRAM accesses) and/or a significant amount of area (to generate the bitmask). SparTen already dedicates over half of its area to sparsity-related logic [18] and adding handling to dynamic sparsity would mean even less area for computation to build a balanced system.

Outer-product. Figure 2d illustrates an outer-product accelerator, where the product of *all* non-zero kernel and image values is computed, followed by selecting and accumulating the required products. SCNN [67] uses this scheme, implemented as a systolic multiplier array. This class is efficient in that no zero products are created, and each product is only performed once. However, as shown in Figure 2d, some products, such as -3×1, are redundant as they do not map to any output. We label these computations as Redundant Cartesian Products (RCPs), which waste computation and data transfer.

While prior works have noted the existence of RCPs [18], it is only during training that RCPs are a major problem. During inference, RCPs are a small portion of the cartesian products, because the forward pass uses small W kernels, e.g., 3×3. In contrast, during CNN training (Section 2.1), G_A acts as a convolution kernel during Backprop. We observe that since G_A matrices are large, e.g., 224×224, the proportion of RCPs increases by 3.8× (Figure 1). RCPs are a critical problem for existing outer-product accelerators, separate from padding, stride, and sparsity. As we demonstrate in this work, creating an efficient outer-product by removing RCPs is non-trivial and results in considerable improvements in training performance and energy efficiency.

The Dual-side Sparse Tensor Core [81] (DST) avoids RCPs by modifying IM2COL for use with a sparse outer-product. As this approach uses IM2COL, values must be duplicated for each product, increasing energy consumption from data movement. We speculate that performing IM2COL serially and scheduling issues result in DST exploiting only 50%-60% of the speed up from sparsity during some layers of convolution inference.

Table 1 summarizes prior works on accelerating sparsity. We are not aware of any previous accelerator that can take advantage of all training sparsity. Training sparsity is inherently dynamic: the sparsity patterns in W, A, and G_A are constantly changing, whereas in inference, the weights are fixed so the sparsity is static. Prior sparse training accelerators *at most* take advantage of dynamic sparsity in one input (D∘-, Table 1). Dynamic sparsity poses a problem for accelerators that rely on pre-computation on a static weight matrix (D∘S, Table 1). Few accelerators can utilize dynamic two-sided sparsity, i.e., in both the kernel and image (D∘D, Table 1). Finally, the memory requirements and dimensions for the additional two training phases lead to unique constraints on training accelerators. This results in the potential for improvement in training energy and performance.

Table 1: Comparing sparse convolution acceleration approaches. DST: Dual-side Sparse Tensor Core. Accelerator classes are inner-product (IP), outer-product (OP), and intersection (INT). Support for 1-sided sparsity (D∘-), two-sided sparsity with one dynamic input (D∘S), or dynamic sparsity in both inputs (D∘D).

Work(s)	Accel. Class	Train	Sparsity Support		
			D∘-	D∘S	D∘D
TensorDash [57]	IP	✓	✓	✗	✗
Procrustes [83]	IP	✓	✓	✗	✗
SCNN [67]	OP	✗	✓	✓	✓
DST [81]	OP	✗	✓	✓	✓
GoSpa [18]	INT	✗	✓	✓	✗
SparTen [31]	INT	✗	✓	✓	✗
[3, 17, 33, 86, 89]	Many	✗	✓	✓/✗	✗
ANT (This Work)	OP	✓	✓	✓	✓

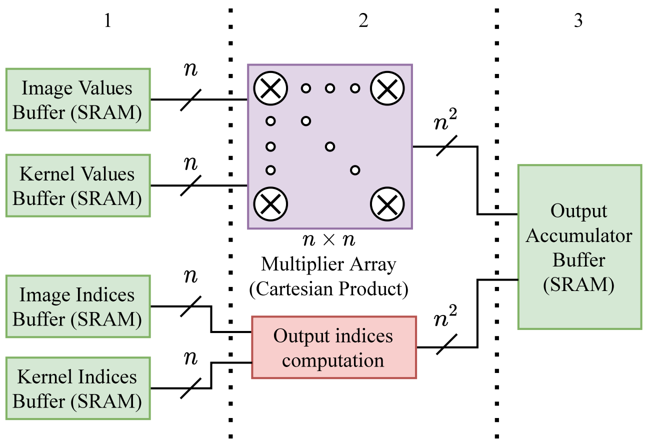
1
2
3
Image Values Buffer (SRAM)
Kernel Values Buffer (SRAM)
Image Indices Buffer (SRAM)
Kernel Indices Buffer (SRAM)
n
n
n
n
n × n
Multiplier Array (Cartesian Product)
n^2
n^2
Output indices computation
Output Accumulator Buffer (SRAM)

Figure 3: The SCNN PE microarchitecture adapted from SCNN [67] to show pipeline stages (dotted lines are pipeline registers). Each cycle, Image and Kernel Values are sent to the multiplier and indices are sent to compute Output indices. Following cycle, results of multiplier and output indices calculation are sent to the output accumulator buffer.

2.3 SCNN

SCNN, shown in Figure 3, spreads computations across an array of processing elements (PEs) such that each PE can operate independently. Since SCNN was designed for inference, its inputs are activations and weights, which correspond to image and kernel in the terminology for this work. Each $W \times H$ activation plane is partitioned into smaller $W_t \times H_t$ planar tiles (PT) that are distributed across the PEs. Thus, multiple PEs can run in parallel. Inside each PE is a multiplier array; only non-zero weights and activations are fetched from the input storage arrays and delivered to the multiplier array. The SCNN dataflow delivers weights and activations of each vector to the multiplier array. The multiplier array performs the Cartesian product of the two vectors, which is a matrix with elements formed by pairwise multiplication of one individual element selected from each of the vectors. A contribution of SCNN is showing that a convolution can be computed efficiently, by computing the Cartesian product between non-zeros values of weight and activation matrices using a systolic array of multipliers. A key observation we make in this paper is that, despite being non-zero, not all elements in such cartesian products are needed and this is especially true in the backward pass used during training of CNNs.

These pairwise multiplications can proceed in parallel within a PE. The PE fetches a set of n non-zero weights from the weight buffer and a set of n non-zero inputs from the input activation buffer. These values are multiplied in an systolic array of $n \times n$ multipliers that directly compute the Cartesian product of these two sets, which is referred to as a partial-sum. (When treating these two sets as two vectors, as in Figure 2d, the "Cartesian product" is referred to as an "outer product".) These partial-sums are accumulated by tracking the coordinates associated with them to route them to an accumulator array. Partitioning input and output activations into $W_t \times H_t$ tiles introduces cross-tile dependencies (halos) at tile edges. Resolving these dependencies requires PEs to communicate. No off-chip memory accesses are necessary if a layer's output activation can serve as the next layer's input activation. In this case, the input and output buffer are logically swapped between the layers' computation sequences.

To reduce off-chip memory accesses SCNN employs an input stationary [8] dataflow; activations are kept within the PEs and reused while different weight vectors are fetched.

3 REDUNDANT CARTESIAN PRODUCTS

In this section, we detail how RCPs occur (Section 3.1) and an algorithm to detect and eliminate them in an outer-product sparse training accelerator (Section 3.2). We leave the microarchitecture design to realize this algorithm to Section 4. Outer-product accelerators take advantage of the dynamic sparsity encountered during training, but they introduce Redundant Cartesian Products (RCPs). RCPs occur when mapping a convolution onto an outer-product datapath. RCPs did not significantly impact prior works because they avoid dynamic sparsity and/or focus on inference.

In the following descriptions, we examine a convolution of an $R \times S$ kernel with an $H \times W$ image to produce an $H_{out} \times W_{out}$ output. We follow the matrix index and dimension conventions described by Sze et al. [76], except we use x and y to denote indices in the respective W and H dimensions of the image matrix and we denote the output matrix dimensions as $H_{out} \times W_{out}$. Figure 2a has a convolution with $R, S = 2, 2$, $H, W = 3, 3$, and $H_{out}, W_{out} = 2, 2$. The output dimensions H_{out} and W_{out} are calculated from the stride, padding, and input shape. When performing a dense outer-product, all combinations of the $R \times S$ kernel values with $H \times W$ weights are multiplied. For a sparse outer-product, this reduces to the non-zero kernel values multiplied by the non-zero image values (Figure 2d).

Figure 4 illustrates the test conditions employed by ANT to detect RCPs and how these tests are related to the convolution of a kernel with an image. In this figure, blue and red squares represent image and kernel matrices as the kernel "shifts" over the image "during" the outer product (individual matrix elements not shown). For each kernel shift, overlapped kernel and image elements are multiplied and the resulting products summed to generate a single output element. This output element has indices (out_x, out_y) that are related to the indices of individual overlapping image (x,y) and kernel (s,r) elements by:

$$out_x = \frac{x - s}{stride} \tag{4}$$

$$out_y = \frac{y - r}{stride} \tag{5}$$

Where $0 \le out_x \le H_{out} - 1$ and $0 \le out_y \le W_{out} - 1$. Convolutions implemented by performing the outer product of all kernel elements times all image elements produce RCPs due to redundant combinations of kernel and image elements. These RCPs occur for kernel shifts where the kernel exceeds the boundaries of the input image, causing the output indices (out_x, out_y) to be outside the dimensions of the output matrix, as shown by position the black square relative to the green output matrix. Such kernel shifts have

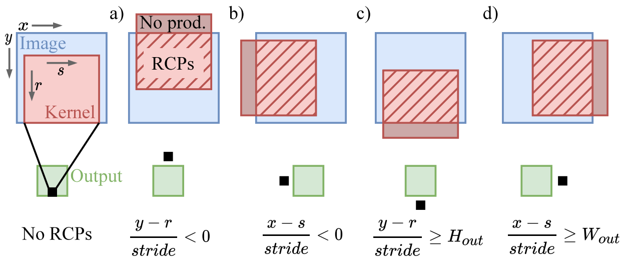

Figure 4: Four cases (a-d) where invalid kernel shifts cause RCPs and conditions to detect them for individual products. Indices used in comparison test to detect an RCP are $x, y \in [0, W), [0, H)$ (image input) and $r, s \in [0, R), [0, S)$ (kernel input). All multiplications for a given kernel shift are RCPs (hatch shaded) when that shift does not correspond to a valid output. Invalid shifts have region where kernel elements form no product (No prod.). The estimated output index of a convolution using these products is represented by the black square. In the case of RCPs, these indices are outside of the output matrix, shown by the green square.

no valid output index: they are either negative (Figure 4a and b) or exceed the dimensions of the output matrix (Figure 4c and d). Note that additional "padding" of the image would introduce additional RCPs rather than eliminate them since the indices of the associated outputs are out of range. The conditions for multiple of cases a-d in Figure 4 can occur simultaneously (e.g., when the kernel overlaps a corner of the image). The RCPs in Figure 2d correspond to each case in Figure 4 as follows: Case b) -3, Case c) 6 & -9, and Case d) 6 & 4. No Case a) RCPs occur in Figure 2d as all potential RCPs are zero, and the RCP 6 is both Cases c) and d). While perhaps easy to see visually with dense matrices, kernel-image element combinations leading to RCPs are less apparent at runtime with compressed sparse matrices. This leads to the need to develop tests based upon input indices (Section 3.2).

3.1 Modelling Dense Outer-product RCPs

We examine the influence of training on RCPs by creating an analytical model for a dense outer-product. Consider a convolution with no wasted multiplications. Every kernel position as it slides over the image matrix results in one output value and accumulates $R \times S$ products. Thus, $R \times S \times H_{out} \times W_{out}$ multiplications are required to compute the convolution.

Now, consider an outer-product with the same inputs. As there are $H \times W$ image values and $R \times S$ kernel values, this results in $R \times S \times H \times W$ multiplications. As these include products between any kernel element-image element pair, it is possible to find any product required for convolution among them. Since we need $R \times S \times H_{out} \times W_{out}$ products out of a total of $R \times S \times H \times W$, the outer-product's efficiency at computing the convolution is given by $\frac{R \times S \times H_{out} \times W_{out}}{R \times S \times H \times W}$, which simplifies to

$$\text{Outer-product Efficiency} = \frac{H_{out} \times W_{out}}{H \times W} \tag{6}$$

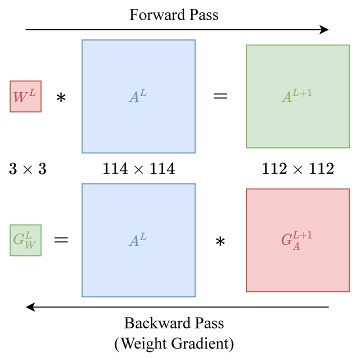

Figure 5: Relation among dimensions of matrices in forward and backward pass (weight gradient) of CNN training. Note activation gradient matrix of $L+1^{st}$ layer (G_A^{L+1}) has same dimensions as activation matrix of the $L+1^{st}$ layer (A^{L+1}).

Table 2 shows some typical dimensions of training convolutions on ImageNet/ResNet50 and/or CIFAR/ResNet18 [19, 35]. Notice that every other consecutive pair of rows share the same image matrix size with image and output matrix sizes swapped. Figure 5 provides intuition about how matrix dimensions relate to each other, using the dimensions from the first two rows of Table 2. The activation and activation gradient matrices have identical dimensions, as do the weight and weight gradient matrices. When weight matrix sizes are small, e.g. 3 × 3, the $L+1^{st}$ layer's activation and activation gradient matrix dimensions are similar to that of the L^{th} layer. Note in Figure 4 as the size of the "kernel" (red square) approaches the size of the"image" (blue square) the fraction of RCPs increases. Thus, during weight gradient computation (Equation 3, lower part of Figure 5) RCPs can dominate since the "kernel" (G_A^{L+1}) and "image" (A^L) matrices involved in the convolution have similar dimensions. This explains why there is higher efficiency in the forward pass and activation gradient calculation than in the weight gradient calculation (shown in the rightmost column of Table 2).

Table 2: Typical dimensions and outer-product efficiency for the forward ($W * A$), backward ($W * G_A$) and update ($G_A * A$) training phases on ImageNet/ResNet50 and/or CIFAR/ResNet18. The dimensions are for the kernel ($R \times S$), the image ($H \times W$), and the output ($H_{out} \times W_{out}$).

Training Phase	$R \times S$	$H \times W$	$H_{out} \times W_{out}$	Outer-product Efficiency
$W*A, W*G_A$	3 × 3	114 × 114	112 × 112	96.52%
$G_A * A$	112 × 112	114 × 114	3 × 3	0.07%
$W*A, W*G_A$	7 × 7	230 × 230	112 × 112	23.71%
$G_A * A$	112 × 112	230 × 230	7 × 7	0.09%
$W*A, W*G_A$	1 × 1	56 × 56	56 × 56	100.00%
$G_A * A$	56 × 56	56 × 56	1 × 1	0.03%
$W*A, W*G_A$	3 × 3	16 × 16	14 × 14	76.58%
$G_A * A$	14 × 14	16 × 16	3 × 3	3.53%

Algorithm 1 Ideal anticipation of RCPs. We omit the outer loops of the convolution to focus the RCP conditions

```
1: for $x, y \in (0, 1, ..., W-1) \times (0, 1, ..., H-1)$ do
2:   for $r, s \in (0, 1, ..., R-1) \times (0, 1, ..., S-1)$ do
3:     $no\_rcp_s = (y - stride \times H_{out}) + 1 \le r \le y$
4:     $no\_rcp_r = (x - stride \times W_{out}) + 1 \le s \le x$
5:     if $no\_rcp_s$ and $no\_rcp_r$ then
6:       out[$\frac{x-s}{stride}$][$\frac{y-r}{stride}$] += image[$x$][$y$] * kernel[$s$][$r$]
7:     end if
8:   end for
9: end for
```

3.2 Detecting RCPs in Sparse Outer Products

RCPs can be detected based on the indices of a given image element and kernel element. When computing the convolution output for a given kernel shift, the difference $x - s$ is constant since adjacent elements of the image and kernel are multiplied before being accumulated. If the difference $x - s$ is negative, the kernel has effectively been shifted too far to the left, corresponding to case b) in Figure 4. Similar reasoning applies to the difference $y - r$, which also remains constant across overlapping image and kernel elements for a given kernel shift. Consideration of all four cases of a kernel shifted too far up, left, down, or right leads to the conditions for cases a-d in Figure 4. The end result, taking into account convolution stride, is that a product between an image element with index (x, y) and a kernel element with index (s, r) is valid (i.e., not an RCP) for use in computing some convolution output when both of the following conditions are true:

$$(y - stride \times H_{out}) + 1 \le r \le y \tag{7}$$

$$(x - stride \times W_{out}) + 1 \le s \le x \tag{8}$$

If the constraints of an outer product accelerator are ignored, then an algorithm can be constructed which eliminates *all* RCPs by skipping computation based solely on the image index and filter index for each multiplication. Such an ideal algorithm is given by Algorithm 1. We loop through every image element-kernel element pair (lls. 1-2), check for RCPs using equations 7 and 8 (lls. 3-5), and perform the multiplication and accumulate the results to output matrix index as described in Equations 4 and 5 (ll. 6).

In an outer product accelerator, we cannot control each input to individual multiplications. For an $n \times n$ multiplier, we send two vectors of n factors to yield an outer product matrix of $n \times n$ products. Replacing a given factor from the inputs results in replacing a row or a column of outer product matrix. This row/column granularity limits the amount of RCPs that can be avoided by replacing a given factor. That is, we can only replace a factor if all of the associated column/row's products are RCPs. Otherwise, replacing the factor would mean that a useful product would be skipped.

Algorithm 2 shows how to skip input factors that result in a row (in the output matrix) of RCPs, assuming that the row factors are from the kernel matrix (indexed with s and r) and the column factors are from the image matrix (indexed with x and y). While we could equally skip input factors that results in columns of RCPs, we choose a specific case to illustrate the idea. We start by looping through the image matrix indices n at a time and compute the

Algorithm 2 Anticipation of RCPs in an outer product architecture.

```
1:  for every n indices $indices_x, indices_y \in (0, 1, ..., W-1) \times (0, 1, ..., H-1)$ do
2:      $y_{min} = min(indices_y)$
3:      $x_{min} = min(indices_x)$
4:      $y_{max} = max(indices_y)$
5:      $x_{max} = max(indices_x)$
6:      for $r \in (0, 1, ..., R-1)$ do
7:          for $s \in (0, 1, ..., S-1)$ do
8:              $valid_s = (y_{min} - stride \times H_{out}) + 1 \leq r \leq y_{max}$
9:              $valid_r = (x_{min} - stride \times W_{out}) + 1 \leq s \leq x_{max}$
10:             if $valid_r$ and $valid_s$ then
11:                 for $i \in (0, 1, ..., n-1)$ do
12:                     $x = indices_x[i]$
13:                     $y = indices_y[i]$
14:                     $\text{out}[\frac{x-s}{stride}][\frac{y-r}{stride}] \mathrel{+}= \text{image}[x][y] \times \text{kernel}[s][r]$
15:                 end for
16:             end if
17:         end for
18:     end for
19: end for
```

minimum and maximum of these indices (lls. 1-5). Here, $indices_x$ and $indices_y$ are n element vectors containing corresponding horizontal and vertical indices of non-zero image elements. Note the order image element indices are stored in $indices_x$ does not matter provided it matches the order in $indices_y$. With these minimum and maximum indices, we loop through the kernel matrix (lls. 6-7). By comparing the s and r indices of the kernel element with the minimum and maximum x and y indices of the image elements, we can determine if multiplying all the image elements with the single kernel element would result in *any* valid output elements (lls. 8-9). If multiplying the kernel element would result in useful products, then we multiply and accumulate the kernel element with the all the image elements (lls. 10-15). To perform an outer product with several kernel elements and several image elements at once, we pre-screen multiple kernel elements' indices based on the minimum and maximum x and y indices. Compared to ideal anticipation (Equations 7 and 8), the equations for determining if there are *any* valid products are less restrictive, allowing some RCPs to occur:

$$(y_{min} - stride \times H_{out}) + 1 \leq r \leq y_{max} \tag{9}$$

$$(x_{min} - stride \times W_{out}) + 1 \leq s \leq x_{max} \tag{10}$$

4 ANTICIPATOR ACCELERATOR (ANT)

In this section, we present the ANTicipator Accelerator (ANT) Processing Element (PE), which performs outer product sparse CNN training convolutions while detecting and removing RCPs using the ideas developed in the previous section. ANT builds on SCNN, adding additional logic blocks to skip RCPs and also reduce SRAM accesses for values that would lead to RCPs. We first review the sparse matrix format underpinning ANT (Section 4.1). Then, we describe the ANT PE in detail, especially the new and modified hardware blocks to anticipate and eliminate RCPs (Section 4.2). We show how the indirection of CSR allows us to avoid SRAM accesses (Section 4.3), followed by the hardware block to exploit this technique (Section 4.4). In Section 4.5, additional consideration is given to supporting rotated matrices, which are necessary for activation gradient calculation (Eqn. 2). ANT is described assuming an Image-stationary dataflow, but ANT can equally support other dataflows (Section 4.6).

4.1 Compressed Sparse Data Format

ANT exploits Compressed Sparse Row (CSR) representation of the indices to more efficiently avoid RCPs. CSR represents a matrix as three arrays: *Values*, *Columns*, and *Row-pointers*. The *Values* array holds the non-zero elements of the matrix in row-major order, *Row-pointers* array tracks the location of the start of each row within the *Values* array, and the *Columns* array indicates the locations of the nonzero values within each row. Compressed Sparse Column (CSC) is the dual, with the role of the *Columns* and *Row-pointers* swapped, becoming *Rows* and *Column-pointers*, and the *Values* array listing values in column-major order. In other words, the CSC representation of the transpose of a matrix is equivalent to the CSR representation of the matrix. Thus, CSC would work equally well with ANT.

4.2 Anticipator Processing Element

The ANT PE, shown in Figure 6, adds extra stages to the SCNN pipeline to anticipate and avoid RCPs. Stages 5 and 6 are identical to Stages 2 and 3 from SCNN (Figure 3). The PE has two parameters: n scales the $n \times n$ **Multiplier Array**, and k is the number of indices that the **First** $n+1$ **Indices within Range (FNIR)** block (Section 4.4) can analyze at once. In the ANT PE, the Image and Kernel matrix inputs are represented in CSR format, with the *Values* array in the **Image Values Buffer** and **Kernel Values Buffer**, and both the *Row-pointers* and *Columns* arrays in each of the **Image Indices Buffer** and **Kernel Indices Buffer**. Below we describe the behavior of individual hardware blocks in the ANT PE pipeline along with their interactions (circled numbers used as list headings below refer to Figure 6):

- ❶ Using a control block (not shown) the ANT PE accesses n entries stored sequentially in the **Image Values Buffer** and **Image Indices Buffer**. These n image entries will be processed with all entries in the Kernel Values and Indices Buffers, which may take several cycles. The image indices go to s **range computation** and r **range computation** blocks in the next cycle and to the **Output indices computation** block four cycles later. The image values pass through pipeline registers so they arrive at multiplier at the same time as the **Output indices computation** block provides the output indices. The contents of the Image Values and Indices Buffers are held constant until the PE has processed the entire kernel matrix (Image stationary dataflow).
- ❷ The s **range computation** block computes the minimum and maximum acceptable s indices from Equation 10, i.e.

$$\begin{aligned} s_{min} &= \min(x_0, ..., x_{n-1}) - stride \times W_{out} + 1 \\ s_{max} &= \max(x_0, ..., x_{n-1}) \end{aligned} \tag{11}$$

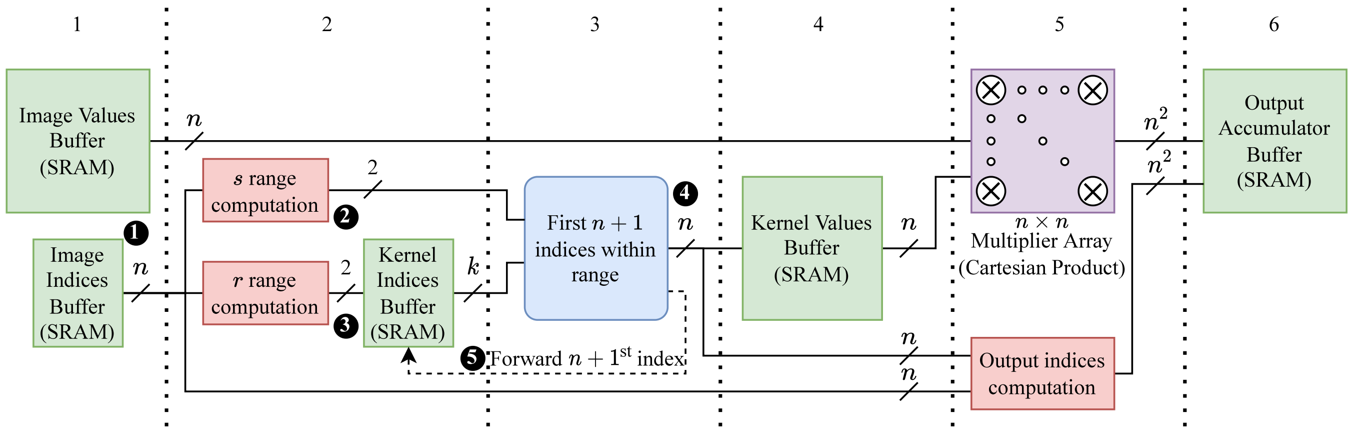
1
2
3
4
5
6
Image Values Buffer (SRAM)
Image Indices Buffer (SRAM)
s range computation
r range computation
Kernel Indices Buffer (SRAM)
First n + 1 indices within range
Kernel Values Buffer (SRAM)
n × n Multiplier Array (Cartesian Product)
Output indices computation
Output Accumulator Buffer (SRAM)
Forward n + 1st index
n
2
k
n²

Figure 6: ANT PE Microarchitecture. Dotted lines indicate location of pipeline registers, green blocks are SRAM buffers, red blocks are relatively simple arithmetic/combinational logic blocks, blue block is the novel hardware for filtering out RCPs, and the purple block represents the multiplier.

3. Similarly, the r **range computation** block computes the minimum and maximum acceptable r indices from Equation 9. Since image indices are stored in CSR format, the magnitude of the y index inputs increase monotonically. Thus, y_{min} simplifies to y_0 and y_{max} to y_{n-1} yielding:

$$\begin{aligned} r_{min} &= y_0 - stride \times H_{out} + 1 \\ r_{max} &= y_{n-1} \end{aligned} \tag{12}$$

The r range is used when accessing the **Kernel Indices Buffer** to avoid accessing indices that are not in the r range. In each cycle, the ANT PE reads k sequential column indices from the **Kernel Indices Buffer** starting from r_{min} up to and including r_{max}. These k indices are provided to the **FNIR** block.

4. The purpose of the **FNIR** block is twofold: (1) to find first n valid kernel indices so those kernel elements can be fetched and sent to the **Multiplier Array** and (2) Find the $n+1$st valid kernel index if it exists to provide feedback to the **Kernel Indices Buffer** (see 5). Each output of the **FNIR** block has an associated valid bit, in case the **FNIR** block cannot find $n+1$ valid indices from its k inputs. The **Kernel Values Buffer** fetches the values for the first n valid indices and sends them to the **Multiplier Array**. If a kernel index is invalid the associated multiplication is disabled. Since invalid r indices have already been skipped by the **Kernel Indices Buffer** control logic using the r_{min} to r_{max} range provided by the r **range computation** block, only s indices need to be checked by the FNIR block. The implementation of the FNIR block is discussed in more detail in Section 4.4.

5. The FNIR block examines $k > n$ indices and selects up to n that pass the test defined by Equations 9 and 11 to send to the multiplier array. It is possible that there are more than n valid indices in the initial k indices. To avoid "skipping" valid factors, it is important to detect when this occurs. To accomplish this, ANT employs the following approach: We have the FNIR block attempt to find one additional valid factor. This $n+1$st kernel index selected by the **FNIR** block is sent back to the **Kernel Indices Buffer** controller to help determine the starting offset into the column indices array. If the $n+1$st kernel index is valid, then the next cycle, the **Kernel Indices Buffer** will send k indices starting from that index. On the other hand, if the $n+1$st kernel index is invalid, then the **Kernel Indices Buffer** will send the next k indices since there were no additional valid factors in the last batch of k indices. This feedback mechanism allows the PE to skip past invalid kernel indices, improving the odds that the **FNIR** block will find n valid kernel indices and keep the **Multiplier Array** occupied.

The PE parameters are available in Table 4. We limit the SRAM buffer size to 8kB to enable single cycle SRAM access.

4.3 Exploiting CSR to Reduce SRAM Accesses

Figure 7 shows how CSR is exploited to reduce unnecessary SRAM lookups. The figure shows a small kernel matrix in CSR format. An image matrix (not shown) provides constraints on indices where RCPs can be avoided: r_{min}, r_{max}, s_{min}, and s_{max}. From $r_{min} = 2$ and $r_{max} = 3$, only positions 2 and 3 of the *Row-pointers* array are valid (i.e. would avoid RCPs), which point to positions 3 to 7 of the *Columns* array. After checking each of those indices against $s_{min} = 1$ and $s_{max} = 2$, only positions 3, 6, 7 are valid, so values for only those positions are looked up from the *Values*. In this simple example, *Columns* array lookups are reduced by $\frac{4}{9} \approx 44\%$ and *Values* lookups by $\frac{6}{9} \approx 67\%$.

4.4 First $n+1$ within Range (FNIR) Block

The inputs to the **FNIR** block are the *min* and *max* values and s_0 to s_{k-1} that are s indices that might be within range. The *Positions* output contains $n+1$ binary-encoded positions (0 through $k-1$), which indicate where the s indices are within range. The *Valid* output is an $n+1$-bit mask that indicates which of the *Positions* outputs are valid. The PE has two parameters: n affects number of outputs and k is the number of inputs, which follow from the

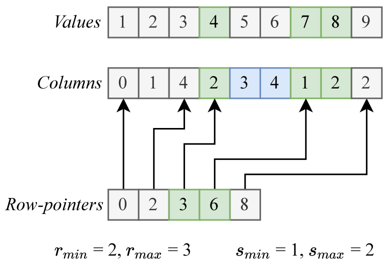

Figure 7: Example of exploiting CSR to reduce SRAM accesses. The arrows show how the CSR format uses the *Row-pointers* array to index into the *Columns* and *Values* arrays. The green boxes show indices/values that are valid and the blue boxes show locations that are accessed but are not valid.

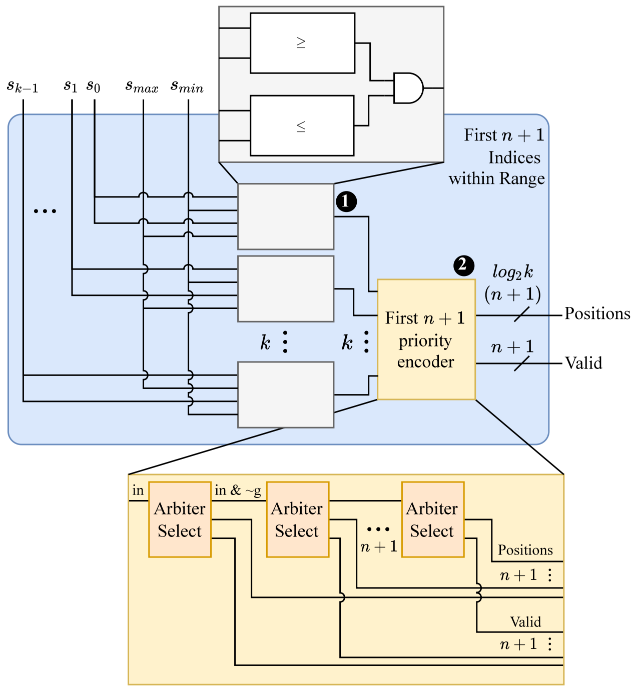

Figure 8: The First $n+1$ Indices within Range (FNIR) block is combinational logic that finds the first $n+1$ indices that are in [min, max]. The grey comparator block with the callout is repeated k times and operates in parallel on s_0 through s_{k-1}. The Arbiter Select circuit in the First $n+1$ priority encoder callout is repeated $n+1$ times.

ANT PE's parameters (Section 4.2) The **FNIR** block is illustrated in Figure 8:

1. k comparator blocks that produce a k-bit mask for whether an s index is at least *min* and at most *max*
2. The **First** $n+1$ **priority encoder** is an iterative circuit that has $n+1$ **Arbiter Select** stages and finds the first $n+1$ positions where the bit mask is 1. Given a k-bit input, **Arbiter Select** has three outputs: (1) a copy of the input bit string except with the position of the first 1 set to 0, which can be computed by bitwise complementing the one-hot encoded grant vector (g) of a combinational logic fixed-priority arbiter [15] and bitwise AND-ing with the input request vector (in); (2) the position of the first 1 it finds encoded in binary; and (3) valid, which is set to 1 if **Arbiter Select** finds a 1.

4.5 Matrix Rotation

Kernel matrix rotation is required in the backward pass ($R(W)$, Eqn. 2), so we'll explain how to do this in the CSR format. Since rotation is not always required, we use a ROTATE boolean flag input. If the flag is set, the ANT accelerator will remap the *Row-pointers* and *Columns* arrays according to Algorithm 3, which results in rotated indices. (The *Values* array does not change.) As the calculation is a transformation of indices, not data, the added overhead (area and latency) is negligible.

4.6 Dataflow

Two common alternative dataflows are kernel stationary and output stationary, we show how they can be realized in the ANT PE and enjoy the benefits of reduced RCPs.

Kernel stationary can be implemented by swapping the Image Indices and Values buffers with their Kernel counterparts, and updating the s and r range compuations to x and y range compuations. Equivalent conditions in equalities in Equations 9 and 10 can be derived by solving inequalities in Equations 7 and 8 for the minimum and maximum allowed x and y indices.

Output stationary can be implemented by changing how the PE iterates over the buffers. Rather than processing the whole Kernel matrix on every n Image elements, output stationary would involve fetching new Image elements as the PE processes more of the Kernel matrix. Admittedly, output stationary dataflow on sparse matrices is challenging since output indices are calculated on the fly, but solving this issue is beyond the scope of this work.

5 EXTENSION TO MLP, RNN, AND TRANSFORMERS

While the key motivation of this paper is exploiting the activation, weight, and gradient sparsity at minimal accuracy loss afforded by techniques such as SWAT [69] and ReSprop [28] for CNNs, 76% of Google's training workloads (as of April 2019) consist of MLP, RNN, and transformer networks [44]. These networks are key to language

Algorithm 3 Kernel matrix rotation by 180°.

```
1: Inputs: Height ($H$) and width ($W$) of the matrix
2: Inputs: Row index ($y$) and column index ($x$) into the original matrix
3: Outputs: Row index ($y_{rotated}$) and column index ($x_{rotated}$) into the rotated matrix
4: $y_{rotated} \leftarrow H - y - 1$
5: $x_{rotated} \leftarrow W - x - 1$
```

models [79]. A large part of these networks are fully-connected layers, in which multiple training samples can be batched into matrix multiplications between weight and matrices containing multiple activation vectors for inference and for the backward propagation, the weight matrix with the gradient matrix and the gradient matrix with the activation matrix. Additionally, certain transformer and RNN layers can be implemented as matrix multiplication [5, 78]. This stands in contrast to the case with SCNN [67] which suffers from the problem of aligning the input values in matrix-vector multiplication so as to produce useful products. The matrix-vector multplication results from how SCNN is used with a single input in inference. This alignment issue degrades throughput of SCNN on fully connected layers by at least 75% in a 4x4 multiplier array. In contrast during training we encounter matrix-matrix multiplications even for fully connected layers. Given this, ANT can anticipate RCPs and improve the efficiency of matrix multiplication on an outer product accelerator.

We will use the convention of a matrix multiplication between an $H \times W$ image matrix and an $R \times S$ kernel matrix, where $W = R$. Taking the dot product between each of the H rows of the image matrix and each of the S columns in the kernel matrix results in an output matrix with dimensions of $H \times S$. The output index for the product of the image element at (x, y) and the kernel element at (s, r) is

$$\begin{aligned} out_x &= s \\ out_y &= y \end{aligned} \tag{13}$$

In contrast to avoiding RCPs in convolutions, the output indices are always within the dimensions of the output matrix. Instead, we need to check that the image element's column and the kernel element's row are the same:

$$r = x \tag{14}$$

An outer product between the matrices produces $H \times W \times R \times S$ products, while the matrix multiplication actually requires $H \times W \times S$ products, so only $\frac{1}{R}$ of the products are valid. Table 3 shows the outer product efficiencies for typical dimensions on a matrix multiplication implementation of text translation transformer [5, 79] and a text classification RNN [78] trained on a large movie review dataset [56].

Given sparse activation, weight, and gradient matrices in fully-connected layers, a modified ANT accelerator can avoid these RCPs. Using r **range computation** with the **Kernel Indices Buffer** (Figure 6 ❸), the equivalent Equation 14 for RCPs is

$$\begin{aligned} r_{min} &= x_0 \\ r_{max} &= x_{n-1} \end{aligned} \tag{15}$$

For matrix multiplication, ANT completely avoids any unnecessary memory accesses due to the above calculation. Since all kernel and image index combinations are valid as long as $r = x$, ANT does not need to check the s index at all and can skip the FNIR block altogether. In summary, supporting matrix multiplication in ANT involves changing the **Output indices computation** (Equation 13) and s **range computation** (Equation 15), as well as bypassing stages 3 and 4 of the ANT PE pipeline (Figure 6).

Table 3: Typical dimensions and outer-product efficiency for the forward ($A \times W$), backward ($G_A \times W$) and update ($A \times G_A$) training phases on a text translation transformer and a text classification RNN.

Training Phase	$H \times W$	$R \times S$	Outer-product Efficiency
$A \times W$, $G_A \times W$	512×72	72×512	1.39%
$A \times G_A$	72×512	512×512	0.20%
$A \times W$	64×10	10×10	10.00%
$G_A \times W$	10×10	10×64	10.00%
$A \times G_A$	10×64	64×10	1.56%
$A \times W$	300×3	3×1200	33.33%
$G_A \times W$	1200×3	3×300	33.33%
$A \times G_A$	3×300	300×1200	0.33%
$A \times W$	300×8	8×1200	12.50%
$G_A \times W$	1200×8	8×300	12.50%
$A \times G_A$	8×300	300×1200	0.33%

6 METHODOLOGY

6.1 Simulation

We built a cycle accurate functional simulator using the DNNsim [80] framework to simulate the ANT accelerator. Since ANT is dataflow-agnostic and DRAM access patterns heavily depend on the dataflow [8, 84], we assume that the SRAM is appropriately managed to provide single-cycle memory accesses. We model a five-cycle start-up cost whenever a PE is given new image and kernel matrices. Additionally, we assume that the Output Accumulator Buffer is appropriately designed to handle the throughput from the multiplier array and the Output indices computation; other works explore how this can be done more optimally [81]. Since load balance is dataflow dependent, we also assume the existence of a perfect load balancing algorithm, which we employ to show the potential of the ANT accelerator if the workload were perfectly load balanced; other works investigate how to do this [31, 33, 81, 83]. These assumptions allow us to focus on exploring the design space of ANT PE parameters and ablation studies on elements of the microarchitecture through various workloads. We believe that there is an design space that requires further exploration in DRAM access scheduling, load balance, and appropriate dataflow choices, and we hope that our evaluation will help motivate future work. Some of the goals would be to avoid pipeline start up costs, reuse image or kernel values, and ensure that PEs finish at similar times. Some ideas for achieving this include estimating the sparsity of matrices so that PEs each have a similar amount of computation to do and finding the best way to coalesce smaller matrices together and split large matrices to avoid pipeline start up costs, reuse image or kernel values, and ensure that PEs finish at similar times.

We configure ANT training accelerator according to Table 4. Since kernel matrices can be large during the Update phase of CNNs (Equation 3), we modify the SCNN baseline to split up the kernel matrix across the 8×8 PEs and call it SCNN+. We additionally evaluate a dense inner product accelerator, DaDianNao [9], and

TensorDash, a sparse inner product accelerator [57]. These accelerators are configured with 16 floating-point multiplier PEs, and the number of tiles is configured such that the total multipliers is equivalent to ANT [9, 57, 80]. We evaluate the ANT training accelerator using the CIFAR100 [46] dataset to train ResNet18 [35], VGG16 [73], DenseNet-121 [38], and Wide ResNet (WRN) [85] networks.

6.2 Traces

We collect traces three ways: First, we train ResNet18 [35] on a current generation commodity graphics processor unit (GPU) with the ReSprop Sparse Training algorithm [28] for 100 iterations of training and collect the traces for the three convolution training phases mentioned in Section 2.1. ReSprop uses a delta-based algorithm to sparsify the activation gradients (G_A) during training. Second, we collect traces after 100 iterations of training from Sparse Weight Activation Training (SWAT) [69], which sparsifies activations in the backward pass, and weights in all phases of training. Typically, after 100 iterations, these sparsity techniques are able to bring the sparsity to near the target that we set. The traces are for $W * A$, $W * G_A$, and $G_A * A$ convolutions. We omit the update of the weights from Stochastic Gradient Descent (SGD), since they form a relatively small portion of the computation and are not efficiently computed by either ANT or SCNN+. Other methods could also be used with the ANT accelerator, for instance weight pruning [62, 63]. Thirdly, we collected traces from ResNet50 [35] trained on the Imagenet dataset [19], which due to the training time constraints, we synthetically sparsified the weights, activations, and gradients by selecting the top-K values and setting the rest to 0. We performed the same synthetic sparsification on a matrix multiplication implementation of a text translation transformer [5, 79], as well as a text classification RNN [78] trained on a large movie review dataset [56].

CIFAR100/ResNet18 used in this work has a validation accuracy of 74.84% with dense training. As ReSprop is a lossy sparsification technique, accuracy decreases slightly at high sparsities. When trained using ReSprop with sparse training, accuracy decreases by up to 0.17% at 90% sparsity. This is similar to other sparsification techniques, e.g. DropBack (-0.14% accuracy, 91% sparsity) [29, 83] and SWAT (-1.6% accuracy, 80% sparsity) [69].

6.3 Energy Estimation

Energy consumption is measured using operation counters, and multiplying by the energy-per-operation measured by Jouppi et al. [43] for a tensor processor using 7nm technology. Index comparison operations are modeled as 32-bit integer additions. All multiplication and additions are performed using Bfloat 16. In sparse format, each matrix element can be stored using 32 bits: 16 bits for the values and 16 bits for the indices, so we assume 2 elements can be fetched per 64-bit memory access.

7 EVALUATION

7.1 Results on CNN Training

Figure 9 shows the speed up and relative energy consumption relative to SCNN+ on DenseNet-121 [38], ResNet18 [35], VGG16 [73], and WideResNet (WRN) [85], on the CIFAR10 dataset sparsified using SWAT [69] and ResNet-50 [35] on ImageNet [19] sparsified synthetically. ANT achieves an average (geo-mean) of 3.71× speed up and 4.40× decrease in energy consumption. Table 5 explains the variations in speed up between the networks. Generally, the higher the proportion of RCPs that are avoided, the higher the speed up and energy savings. On average, ANT eliminates 90.3% of the RCPs. Overall, ANT is effective on both synthetic and realistic sparsity patterns.

Table 4: ANT Design Parameters

Values floating point format	Bfloat16
Indices bit width	8 bits
Max Size of SRAM Buffers	8kB
Multiply array ($n \times n$)	4x4 (default)
Number of Inputs to FNIR block (k)	16 (default)
Number of PEs	64

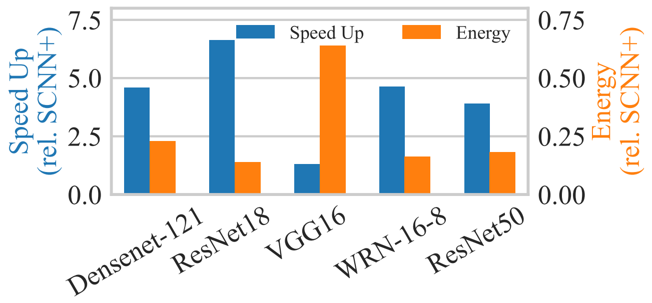

Figure 9: Speed up and energy consumption for ANT relative to SCNN+ for DenseNet-121 [38], ResNet18 [35], VGG16 [73], and WideResNet (WRN) [85], on the CIFAR10 dataset sparsified using SWAT [69] and ResNet-50 [35] on ImageNet [19] sparsified synthetically. All networks are trained to a target sparsity of 90%.

Table 5: Proportion of RCPs avoided due to ANT for DenseNet-121 [38], ResNet18 [35], VGG16 [73], and WideResNet (WRN) [85], on the CIFAR10 dataset sparsified using SWAT [69] and ResNet-50 [35] on ImageNet [19] sparsified synthetically. All networks are trained to a target sparsity of 90%.

Network	RCPs avoided
Densenet-121	93.6%
ResNet18	98.0%
VGG16	74.9%
WRN-16-8	94.8%
ResNet50	91.9%

7.2 Sensitivity to Sparsity

When evaluating ResNet18 [35] with ReSprop [28], we found that ReSprop activation gradient (G_A) sparsity targeting was imprecise and also impacted the sparsity of the activation (A) matrices. Thus, we show our measured sparsity for the gradient and activations to clarify the results. The sparsity of the weight matrix (W) is omitted since these matrices are typically much smaller and thus less impactful than the G_A and A matrices.

Figure 10 shows how well ANT is able to exploit increased sparsity to speed up computation and reduce energy consumption against a *dense* SCNN+ baseline. We see that ANT is able to achieve up to 28.1× speed up and 40× energy savings at 42%/85% sparsities. Note that sparsity does not correlate directly with speed up since sparsity distributions have some effect on the effectiveness of ANT.

While SCNN+ indeed exploits the increased sparsity, at every sparsity level, ANT is between 1.9× and 2.6× faster and uses between 2.6× and 4.4× less energy as shown in Figure 11. These savings occur from making better use of the available sparsity by avoiding RCPs and associated SRAM accesses. Although ANT does not avoid all RCPs, it avoids enough that across all sparsities we observe superior performance and energy consumption for ANT.

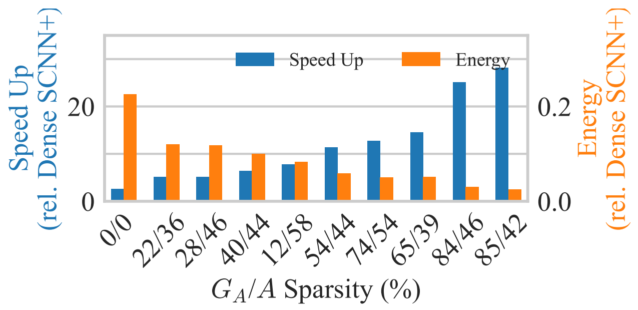

Figure 10: Speed up and relative energy consumption of ANT when compared to the *dense* (i.e. zero sparsity) SCNN+ baseline. The Network is CIFAR100/ResNet18 trained using ReSprop. Sparsity is indicated as Activation Gradient / Activation sparsity as a percentage.

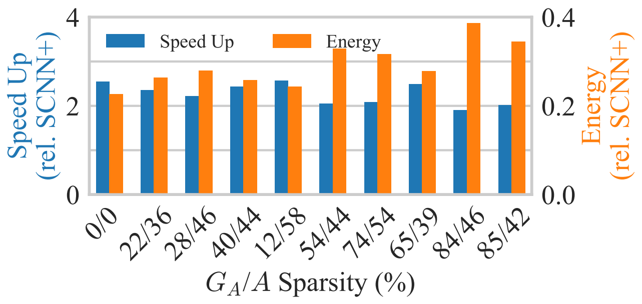

Figure 11: Speed up and energy consumption for ANT relative to SCNN+ *at the same sparsity*. The Network is CIFAR100/ResNet18 trained using ReSprop. Sparsity is indicated as Activation Gradient / Activation sparsity as a percentage.

7.3 Sensitivity to Architecture Parameters

We perform sensitivity studies on the effect of changing the size of the multiplier array and the number of inputs into the First $n+1$ Indices within Range (FNIR) block (Section 4.4). Figure 12 shows that ANT outperforms SCNN+ when the multiplier array is 4×4, 6×6, and 8×8, and thus is useful over a wide range of multiplier configurations. Figure 13 shows that ANT outperforms SCNN+ as long as the FNIR block takes at least 8 inputs. We hypothesize that with a small $k=4$ FNIR, there is no excess capability to run ahead of a 4×4 multiplier. When $k=4$ the FNIR throughput becomes the bottleneck, causing the performance decrease in Figure 13.

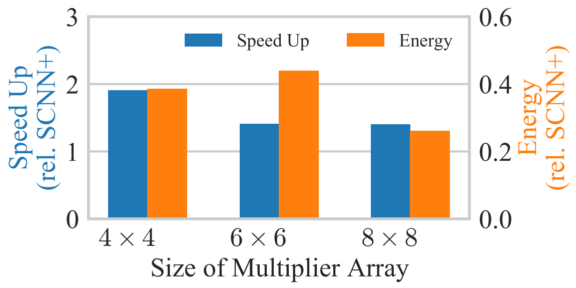

Figure 12: A ResNet18 SWAT 90% [69] study showing the effect of larger multiplier arrays on speed up and energy consumption ANT compared to SCNN+ with the same multiplier array size.

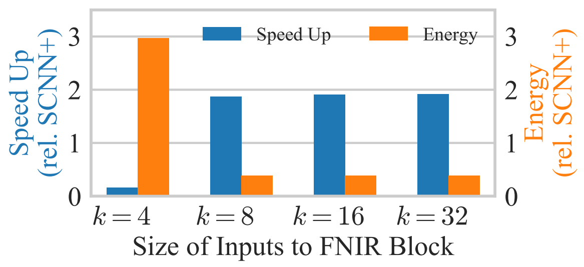

Figure 13: A ResNet18 SWAT 90% [35, 69] study showing the effect of different number of inputs to the FNIR block (Section 4.4) on speed up and energy consumption ANT compared to SCNN+ with the same multiplier array size.

7.4 Ablation Study

Figure 14 shows that eliminating RCPs based on only the r condition (Eqn. 10) or only the s condition (Eqn. 9) still yields speed up and energy savings when compared to SCNN+. Note that the number of RCPs that ANT eliminates is not the sum of the RCPs that the r and s conditions eliminate individually, since there is significant overlap in the sets of individually eliminated RCPs. Despite this, combining both conditions increases performance by 1.06× over r only.

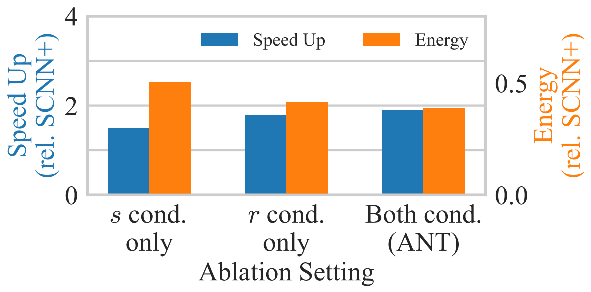

Figure 14: Ablation study showing the effect of only handling the r condition (Eqn. 10) or the s condition (Eqn. 9). The study was performed on ResNet18 SWAT [35, 69] at 90% sparsity.

7.5 Synthesis Results

The **FNIR** block (Section 4.4), the largest circuit in ANT, was implemented in RTL, and synthesized using the Synopsis Design Compiler. Note that this block is needed for efficient convolutions but not matrix multiplication (Section 5). We used FreePDK45 (45 nm) techonlogy node to synthesize our design [75]. Then, we scaled the results to the 15nm technology node, as the 15nm library is no longer available, and added a 50% wire overhead in a similar manner to prior works [59, 70]. The resulting area is 0.0017mm^2. SCNN [67] is implemented in 16nm so it is somewhat comparable to our area numbers. The **FNIR** block represents 0.02% of the area of the entire SCNN PE, or 21.25% of the 4×4 multiplier array.

7.6 Overhead and Scalability

The overhead of ANT is in the five cycle pipeline start up cost and in the area for the FNIR block. In the smaller layers, we noticed that ANT introduces a slowdown of up to 30%. Our hypothesis is that because our dataflow is distributing very little work to each PEs (10s-100s of multiplications) due to the sparsity of the matrices, the pipeline start up costs become important. This overhead becomes less important as matrices grow in size. Figure 12 shows that ANT continues to outperform SCNN as the PE is scaled up to large multiplier arrays. However, as the size of PE (number of multipliers) increases, the depth of the serial **Arbiter Selects** in the **FNIR** block (Section 4.4) would increase, leading to increased area and delay. Thus, as the size of PE increases, the trade-offs when scaling up would tend to favour the alternative of increasing the number of PEs per tile.

7.7 Performance Relative to Inner Product

We examine the relative performance of DaDianNao, TensorDash, SCNN+, and ANT, on networks with 90% sparsity (Resnet18, WRN, Densenet and VGG with SWAT, and Resnet18 with ResProp). We observe TensorDash improves performance 2.25× over dense, similar to the 1.95× reported by Mahmoud et al. [57]. However, as TensorDash only exploits one side of the approximately 90% sparsity, ANT increases performance by 8.9× over TensorDash. This clearly illustrates the advantage of using two-sided dynamic sparsity for training.

7.8 Results on Transformers and RNNs

Evaluation on a text translation transformer [79] and a text classification RNN [78] trained on a large movie review dataset [56] shows that ANT can anticipate and eliminate 99% of the RCPs at 0%, 50% and 90% sparsities.

8 RELATED WORK

Acceleration of DNNs has been studied extensively for inference. Dense DNN acceleration generally restructures dataflow and memory system around a series of multiply-accumulate units or systolic arrays [1, 8, 9, 45, 54, 84]. Other works such as OuterSPACE [66] have designed non-systolic array architectures to accelerate sparse matrix multiplications, but do not address sparse convolutions. By contrast, the techniques applied for inference acceleration vary greatly [2, 3, 17, 17, 31–33, 67, 86, 89]. Sparse models for these accelerators are obtained through the many pruning and sparsification techniques, [23, 25, 36, 48, 49, 58, 62] including channel pruning [37, 55] and advanced compression [34]. Pruning in this manner can be classified as train-for-sparsity, i.e., they have little impact on training. Often, these techniques have a dense forward pass that significantly increase training costs over non-sparse training.

Some works have examined dense training acceleration [10, 47], as well as some completed products [26, 45, 60]. Although the performance gain from sparse acceleration is much higher [67] insights from dense accelerators, such as memory hierarchy [84] can be used with ANT. Fewer works propose accelerators for sparse training. TensorDash utilizes one-sided sparsity, which achieves 1.95x throughput versus dense [57]. As ANT can take advantage of sparsity in weights, activations, and gradients simultaneously, the compute reduction is much higher. Procrustes pairs a sparsification method and accelerator, resulting in a 4.0x speedup over dense training [83]. ANT, by comparison, can utilize any of the many sparsification methods. Finally, Sigma uses a flexible adder network to extract sparsity in matrix multiplies for a 5.7x increase in performance [68]. However, Sigma does not address RCPs, and is targeted at matrix multiplies, hence ANT could potentially be combined with this work.

The accelerator proposed by this and other sparse training works is enabled by many techniques which reduce training computations. Sparsifying activations can be accomplished easily with ReLU [64], dropout [74], or by more advanced methods [27, 61]. ReLU and dropout also result in activation gradient sparsity, which is why many networks have some natural sparsity [35, 85]. By contrast, weight sparsity needs to be induced over the course of training, which has been shown to be feasible by many works [20, 34, 52, 53, 55, 69, 87]. Inducing gradient sparsity is relatively unstudied, however, some works have shown that gradient sparsity can reduce training computation significantly [11, 28, 51]. ReSprop [28] does mention a CNN training accelerator but does not describe details of the accelerator nor mention RCPs. Finally, a few works propose introducing sparsity at multiple locations during training (e.g., weights and activations) [63, 69]. These works show that a wide array of choices can be used to generate sparsity that can improve the performance of the ANT accelerator.

There are other ways of improving training throughput besides sparsity. Quantizing networks to binary [13, 14], ternary [50] or another reduced precision [12, 16, 41, 82, 88] can greatly reduce training compute requirements. This work uses 16-bit fixed point but can be easily combined with these works by reducing the precision. Finally, works have increased training performance by other means, such as reducing activation footprint [7, 22], or by creating more efficient models [30, 42]. ANT can similarly be combined with these methods, albeit with some effort.

9 CONCLUSION

We have observed and addressed an inefficiency in outer-product sparse convolution training: Redundant Cartesian Products (RCPs) are products that map to indices that are outside of the dimensions of the output matrix. Over 90% of RCPs can be detected and eliminated using our proposed ANTicipator Acclerator (ANT). We show that detecting and eliminating such products from sparse indices can be achieved using a pipelined architecture that can not only eliminate redundant computation, but also skip SRAM accesses. This architecture requires 0.0017mm^2 of additional area. We evaluate our design on training for DenseNet-121 [38], ResNet18 [35], VGG16 [73], Wide ResNet (WRN) [85], and ResNet-50 [35] and find 3.71× speed up over an SCNN-like accelerator and 4.40× decrease in energy consumption. We show that ANT can be extended to accelerate matrix multiplications, anticipating and eliminating 99% of RCPs on a text translation transformer [79] and a text classification RNN [78]. Finally, ANT is dataflow- and memory-agnostic, so it can be combined with any other outer-product-based sparse training hardware to eliminate RCPs and improve performance.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive and insightful feedback on this work. This research was supported in part by the National Sciences and Engineering Research Council of Canada (NSERC) Computing Hardware for Emerging Intelligent Sensing Applications (COHESA) program and Canada Graduate Scholarship-Master's (CGS-M) program. Tor M. Aamodt serves as a consultant for Huawei Technologies Canada Co. Ltd. and Intel Corp.

REFERENCES

[1] 2020. NVIDIA A100 Tensor Core GPU Architecture. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf.
[2] Jorge Albericio, Alberto Delmás, 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-50 '17)*. Association for Computing Machinery, New York, NY, USA, 382–394. https://doi.org/10.1145/3123939.3123982
[3] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. 2016. Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing. *ACM SIGARCH Computer Architecture News* 44, 3 (June 2016), 1–13. https://doi.org/10.1145/3007787.3001138
[4] Alexei Baevski, Yuhao Zhou, Abdelrahman Mohamed, and Michael Auli. 2020. Wav2vec 2.0: A Framework for Self-Supervised Learning of Speech Representations. *Advances in Neural Information Processing Systems* 33 (2020), 12449–12460.
[5] Christian Bartz. 2021. chainer-transformer.
[6] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. 2016. End to End Learning for Self-Driving Cars. *arXiv:1604.07316 [cs]* (April 2016). arXiv:1604.07316 [cs]
[7] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training Deep Nets with Sublinear Memory Cost. *arXiv:1604.06174 [cs]* (April 2016). arXiv:1604.06174 [cs]
[8] Y. Chen, T. Krishna, J. S. Emer, and V. Sze. 2017. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE Journal of Solid-State Circuits* 52, 1 (Jan. 2017), 127–138. https://doi.org/10.1109/JSSC.2016.2616357
[9] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam. 2014. DaDianNao: A Machine-Learning Supercomputer. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 609–622. https://doi.org/10.1109/MICRO.2014.58
[10] Y. Chen, T. Yang, J. Emer, and V. Sze. 2019. Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9, 2 (June 2019), 292–308. https://doi.org/10.1109/JETCAS.2019.2910232
[11] Brian Chmiel, Liad Ben-Uri, Moran Shkolnik, Elad Hoffer, Ron Banner, and Daniel Soudry. 2020. Neural Gradients Are Near-Lognormal: Improved Quantized and Sparse Training. In *International Conference on Learning Representations*.
[12] Jungwook Choi, Pierce I.-Jen Chuang, Zhuo Wang, Swagath Venkataramani, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. 2018. Bridging the Accuracy Gap for 2-Bit Quantized Neural Networks (QNN). *arXiv:1807.06964 [cs]* (July 2018). arXiv:1807.06964 [cs]
[13] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2016. BinaryConnect: Training Deep Neural Networks with Binary Weights during Propagations. *arXiv:1511.00363 [cs]* (April 2016). arXiv:1511.00363 [cs]
[14] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1. *arXiv:1602.02830 [cs]* (March 2016). arXiv:1602.02830 [cs]
[15] William J. Dally, R. Curtis Harting, and Tor M. Aamodt. 2016. *Digital Design Using VHDL: A Systems Approach* (1st ed.). Cambridge University Press, USA.
[16] Dipankar Das, Naveen Mellempudi, Dheevatsa Mudigere, Dhiraj Kalamkar, Sasikanth Avancha, Kunal Banerjee, Srinivas Sridharan, Karthik Vaidyanathan, Bharat Kaul, Evangelos Georganas, Alexander Heinecke, Pradeep Dubey, Jesus Corbal, Nikita Shustrov, Roma Dubtsov, Evarist Fomenko, and Vadim Pirogov. 2018. Mixed Precision Training of Convolutional Neural Networks Using Integer Operations. In *International Conference on Learning Representations*.
[17] Alberto Delmas Lascorz, Patrick Judd, Dylan Malone Stuart, Zissis Poulos, Mostafa Mahmoud, Sayeh Sharify, Milos Nikolic, Kevin Siu, and Andreas Moshovos. 2019. Bit-Tactical: A Software/Hardware Approach to Exploiting Value and Bit Sparsity in Neural Networks. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 749–763. https://doi.org/10.1145/3297858.3304041
[18] Chunhua Deng, Yang Sui, Siyu Liao, Xuehai Quan, and Bo Yuan. 2021. GoSPA: An Energy-Efficient High-Performance Globally Optimized SParse Convolutional Neural Network Accelerator. In *Proceedings of the 48th International Symposium on Computer Architecture (ISCA'21)*. Association for Computing Machinery, 1110–1123. https://doi.org/10.1109/ISCA52012.2021.00090
[19] J. Deng, W. Dong, R. Socher, L. Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A Large-Scale Hierarchical Image Database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 248–255. https://doi.org/10.1109/CVPR.2009.5206848
[20] Tim Dettmers and Luke Zettlemoyer. 2019. Sparse Networks from Scratch: Faster Training without Losing Performance. *arXiv:1907.04840 [cs, stat]* (Aug. 2019). arXiv:1907.04840 [cs, stat]
[21] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding. *arXiv:1810.04805 [cs]* (May 2019). arXiv:1810.04805 [cs]
[22] R. D. Evans, L. Liu, and T. M. Aamodt. 2020. JPEG-ACT: Accelerating Deep Learning via Transform-Based Lossy Compression. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 860–873. https://doi.org/10.1109/ISCA45697.2020.00075
[23] Utku Evci, Trevor Gale, Jacob Menick, Pablo Samuel Castro, and Erich Elsen. 2020. Rigging the Lottery: Making All Tickets Winners. In *International Conference on Machine Learning*. PMLR, 2943–2952.
[24] Andrew Feldman. 2020. Cerebras Wafer Scale Engine: Why We Need Big Chips for Deep Learning.
[25] Jonathan Frankle and Michael Carbin. 2019. The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks. *arXiv:1803.03635 [cs]* (March 2019). arXiv:1803.03635 [cs]
[26] J.P. Fricker and A. Hock. 2019. Building a Wafer-Scale Deep Learning System: Lessons Learned.
[27] Georgios Georgiadis. 2019. Accelerating Convolutional Neural Networks via Activation Map Compression. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 7085–7095.
[28] Negar Goli and Tor M. Aamodt. 2020. ReSprop: Reuse Sparsified Backpropagation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 1548–1558.

[29] Maximilian Golub, Guy Lemieux, and Mieszko Lis. 2019. Full Deep Neural Network Training On A Pruned Weight Budget. *Proceedings of Machine Learning and Systems* 1 (April 2019), 252–263.

[30] Aidan N. Gomez, Mengye Ren, Raquel Urtasun, and Roger B. Grosse. 2017. The Reversible Residual Network: Backpropagation Without Storing Activations. In *NIPS*.

[31] Ashish Gondimalla, Noah Chesnut, Mithuna Thottethodi, and T. N. Vijaykumar. 2019. SparTen: A Sparse Tensor Accelerator for Convolutional Neural Networks. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 151–165. https://doi.org/10.1145/3352460.3358291

[32] U. Gupta, B. Reagen, L. Pentecost, M. Donato, T. Tambe, A. M. Rush, G. Wei, and D. Brooks. 2019. MASR: A Modular Accelerator for Sparse RNNs. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 1–14. https://doi.org/10.1109/PACT.2019.00009

[33] 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. *ACM SIGARCH Computer Architecture News* 44, 3 (June 2016), 243–254. https://doi.org/10.1145/3007787.3001163

[34] Song Han, Huizi Mao, and William J. Dally. 2016. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *arXiv:1510.00149 [cs]* (Feb. 2016). arXiv:1510.00149 [cs]

[35] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 770–778.

[36] Yang He, Guoliang Kang, Xuanyi Dong, Yanwei Fu, and Yi Yang. 2018. Soft Filter Pruning for Accelerating Deep Convolutional Neural Networks. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI'18)*. AAAI Press, Stockholm, Sweden, 2234–2240.

[37] Yihui He, Xiangyu Zhang, and Jian Sun. 2017. Channel Pruning for Accelerating Very Deep Neural Networks. In *2017 IEEE International Conference on Computer Vision (ICCV)*. IEEE Computer Society, 1398–1406. https://doi.org/10.1109/ICCV.2017.155

[38] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. 2017. Densely Connected Convolutional Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[39] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. 2019. GPipe: Efficient Training of Giant Neural Networks Using Pipeline Parallelism. *arXiv:1811.06965 [cs]* (July 2019). arXiv:1811.06965 [cs]

[40] Zhengwei Huang, Ming Dong, Qirong Mao, and Yongzhao Zhan. 2014. Speech Emotion Recognition Using CNN. In *Proceedings of the 22nd ACM International Conference on Multimedia (MM '14)*. Association for Computing Machinery, New York, NY, USA, 801–804. https://doi.org/10.1145/2647868.2654984

[41] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2018. Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations. *Journal of Machine Learning Research* 18, 187 (2018), 1–30.

[42] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-Level Accuracy with 50x Fewer Parameters and <0.5MB Model Size. *arXiv:1602.07360 [cs]* (Nov. 2016). arXiv:1602.07360 [cs]

[43] Norman P Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, Thomas Norrie, Nishant Patil, Sushma Prasad, Cliff Young, Zongwei Zhou, and David Patterson. 2021. Ten Lessons from Three Generations Shaped Google's TPUv4i. In *Proceedings of the 48th International Symposium on Computer Architecture (ISCA'21)*. Association for Computing Machinery.

[44] Norman P. Jouppi, Doe Hyun Yoon, George Kurian, Sheng Li, Nishant Patil, James Laudon, Cliff Young, and David Patterson. 2020. A Domain-Specific Supercomputer for Training Deep Neural Networks. *Commun. ACM* 63, 7 (June 2020), 67–78. https://doi.org/10.1145/3360307

[45] 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)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3079856.3080246

[46] Alex Krizhevsky and Geoffrey Hinton. 2009. Learning Multiple Layers of Features from Tiny Images. (2009).

[47] H.T. Kung, Bradley McDanel, and Sai Qian Zhang. 2019. Packing Sparse Convolutional Neural Networks for Efficient Systolic Array Implementations: Column Combining Under Joint Optimization. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 821–834. https://doi.org/10.1145/3297858.3304028

[48] Mark Kurtz, Justin Kopinsky, Rati Gelashvili, Alexander Matveev, John Carr, Michael Goin, William Leiserson, Sage Moore, Nir Shavit, and Dan Alistarh. 2020. Inducing and Exploiting Activation Sparsity for Fast Inference on Deep Neural Networks. In *International Conference on Machine Learning*. PMLR, 5533–5543.

[49] Yann Le Cun, John S. Denker, and Sara A. Solla. 1989. Optimal Brain Damage. In *Proceedings of the 2nd International Conference on Neural Information Processing Systems (NIPS'89)*. MIT Press, Cambridge, MA, USA, 598–605.

[50] Fengfu Li, Bo Zhang, and Bin Liu. 2016. Ternary Weight Networks. *arXiv:1605.04711 [cs]* (Nov. 2016). arXiv:1605.04711 [cs]

[51] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J. Dally. 2020. Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training. *arXiv:1712.01887 [cs, stat]* (June 2020). arXiv:1712.01887 [cs, stat]

[52] Junjie Liu, Zhe Xu, Runbin Shi, Ray C. C. Cheung, and Hayden K. H. So. 2020. Dynamic Sparse Training: Find Efficient Sparse Network From Scratch With Trainable Masked Layers. *arXiv:2005.06870 [cs, stat]* (May 2020). arXiv:2005.06870 [cs, stat]

[53] Liu Liu, Lei Deng, Xing Hu, Maohua Zhu, Guoqi Li, Yufei Ding, and Yuan Xie. 2019. Dynamic Sparse Graph for Efficient Deep Learning. *arXiv:1810.00859 [cs, stat]* (May 2019). arXiv:1810.00859 [cs, stat]

[54] S. Liu, Z. Du, J. Tao, D. Han, T. Luo, Y. Xie, Y. Chen, and T. Chen. 2016. Cambricon: An Instruction Set Architecture for Neural Networks. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 393–405. https://doi.org/10.1109/ISCA.2016.42

[55] Christos Louizos, Max Welling, and Diederik P. Kingma. 2018. Learning Sparse Neural Networks through L_0 Regularization. *arXiv:1712.01312 [cs, stat]* (June 2018). arXiv:1712.01312 [cs, stat]

[56] Andrew Maas, Raymond E Daly, Peter T Pham, Dan Huang, Andrew Y Ng, and Christopher Potts. 2011. Learning word vectors for sentiment analysis. In *Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies*. 142–150.

[57] M. Mahmoud, I. Edo, A. H. Zadeh, O. Mohamed Awad, G. Pekhimenko, J. Albericio, and A. Moshovos. 2020. TensorDash: Exploiting Sparsity to Accelerate Deep Neural Network Training. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 781–795. https://doi.org/10.1109/MICRO50266.2020.00069

[58] Huizi Mao, Song Han, Jeff Pool, Wenshuo Li, Xingyu Liu, Yu Wang, and William J. Dally. 2017. Exploring the Regularity of Sparse Structure in Convolutional Neural Networks. *arXiv:1705.08922 [cs, stat]* (June 2017). arXiv:1705.08922 [cs, stat]

[59] Mayler Martins, Jody Maick Matos, Renato P. Ribas, André Reis, Guilherme Schlinker, Lucio Rech, and Jens Michelsen. 2015. Open Cell Library in 15Nm FreePDK Technology *(ISPD '15)*. ACM, 171–178. https://doi.org/10.1145/2717764.2717783

[60] E. Medina and E. Dagan. 2020. Habana Labs Purpose-Built AI Inference and Training Processor Architectures: Scaling AI Training Systems Using Standard Ethernet With Gaudi Processor. *IEEE Micro* 40, 2 (March 2020), 17–24. https://doi.org/10.1109/MM.2020.2975185

[61] Dmitry Molchanov, Arsenii Ashukha, and Dmitry Vetrov. 2017. Variational Dropout Sparsifies Deep Neural Networks. In *International Conference on Machine Learning*. PMLR, 2498–2507.

[62] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. 2017. Pruning Convolutional Neural Networks for Resource Efficient Inference. *arXiv:1611.06440 [cs, stat]* (June 2017). arXiv:1611.06440 [cs, stat]

[63] Hesham Mostafa and Xin Wang. 2019. Parameter Efficient Training of Deep Convolutional Neural Networks by Dynamic Sparse Reparameterization. In *International Conference on Machine Learning*. PMLR, 4646–4655.

[64] Vinod Nair and Geoffrey E. Hinton. 2010. Rectified Linear Units Improve Restricted Boltzmann Machines. In *ICML*.

[65] Thomas Norrie, Nishant Patil, Doe Hyun Yoon, George Kurian, Sheng Li, James Laudon, Cliff Young, Norman Jouppi, and David Patterson. 2021. The Design Process for Google's Training Chips: TPUv2 and TPUv3. *IEEE Micro* 41, 2 (March 2021), 56–63. https://doi.org/10.1109/MM.2021.3058217

[66] S. Pal, J. Beaumont, D. Park, A. Amarnath, S. Feng, C. Chakrabarti, H. Kim, D. Blaauw, T. Mudge, and R. Dreslinski. 2018. OuterSPACE: An Outer Product Based Sparse Matrix Multiplication Accelerator. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 724–736. https://doi.org/10.1109/HPCA.2018.00067

[67] 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 *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA) (ISCA'17)*. 27–40. https://doi.org/10.1145/3079856.3080254

[68] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna. 2020. SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 58–70. https://doi.org/10.1109/HPCA47549.2020.00015

[69] Md Aamir Raihan and Tor Aamodt. 2020. Sparse Weight Activation Training. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 15625–15638.

[70] M. Rhu, M. O'Connor, N. Chatterjee, J. Pool, Y. Kwon, and S. W. Keckler. 2018. Compressing DMA Engine: Leveraging Activation Sparsity for Training Deep Neural Networks. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 78–91. https://doi.org/10.1109/HPCA.2018.00017

[71] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. 1986. Learning Representations by Back-Propagating Errors. *Nature* 323, 6088 (Oct. 1986), 533–536. https://doi.org/10.1038/323533a0

[72] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. 2017. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *arXiv:1712.01815 [cs]* (Dec. 2017). arXiv:1712.01815 [cs]

[73] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv:1409.1556 [cs]* (Sept. 2014). arXiv:1409.1556 [cs]

[74] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *The Journal of Machine Learning Research* 15, 1 (Jan. 2014), 1929–1958.

[75] 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 *2007 IEEE International Conference on Microelectronic Systems Education (MSE'07)*. 173–174. https://doi.org/10.1109/MSE.2007.44

[76] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. 2017. Efficient processing of deep neural networks: A tutorial and survey. *Proc. IEEE* 105, 12 (2017), 2295–2329.

[77] Mingxing Tan and Quoc Le. 2019. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. In *International Conference on Machine Learning*. PMLR, 6105–6114.

[78] Seiya Tokui, Ryosuke Okuta, Takuya Akiba, Yusuke Niitani, Toru Ogawa, Shunta Saito, Shuji Suzuki, Kota Uenishi, Brian Vogel, and Hiroyuki Yamazaki Vincent. 2019. Chainer: A deep learning framework for accelerating the research cycle. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2002–2011.

[79] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[80] Isak Edo Vivancos, Ali Hadizaden, and Omar Mohamed Awad. 2021. DNNSim. https://github.com/isakedo/DNNsim.

[81] Yang Wang, Chen Zhang, Zhiqiang Xie, Cong Guo, Yunxin Liu, and Jingwen Leng. 2021. Dual-Side Sparse Tensor Core. In *Proceedings of the 48th International Symposium on Computer Architecture (ISCA'21)*. Association for Computing Machinery, 1083–1095. https://doi.org/10.1109/ISCA52012.2021.00088

[82] Shuang Wu, Guoqi Li, Feng Chen, and Luping Shi. 2018. Training and Inference with Integers in Deep Neural Networks. In *International Conference on Learning Representations*.

[83] D. Yang, A. Ghasemazar, X. Ren, M. Golub, G. Lemieux, and M. Lis. 2020. Procrustes: A Dataflow and Accelerator for Sparse Deep Neural Network Training. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 711–724. https://doi.org/10.1109/MICRO50266.2020.00064

[84] Xuan Yang, Mingyu Gao, Qiaoyi Liu, Jeff Setter, Jing Pu, Ankita Nayak, Steven Bell, Kaidi Cao, Heonjae Ha, Priyanka Raina, Christos Kozyrakis, and Mark Horowitz. 2020. Interstellar: Using Halide's Scheduling Language to Analyze DNN Accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 369–383. https://doi.org/10.1145/3373376.3378514

[85] Sergey Zagoruyko and Nikos Komodakis. 2017. Wide Residual Networks. *arXiv:1605.07146 [cs]* (June 2017). arXiv:1605.07146 [cs]

[86] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen. 2016. Cambricon-X: An Accelerator for Sparse Neural Networks. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12. https://doi.org/10.1109/MICRO.2016.7783723

[87] Tianyun Zhang, Shaokai Ye, Kaiqi Zhang, Jian Tang, Wujie Wen, Makan Fardad, and Yanzhi Wang. 2018. A Systematic DNN Weight Pruning Framework Using Alternating Direction Method of Multipliers. In *Proceedings of the European Conference on Computer Vision (ECCV)*. 184–199.

[88] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. 2018. DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients. *arXiv:1606.06160 [cs]* (Feb. 2018). arXiv:1606.06160 [cs]

[89] Maohua Zhu, Tao Zhang, Zhenyu Gu, and Yuan Xie. 2019. Sparse Tensor Core: Algorithm and Hardware Co-Design for Vector-Wise Sparse Neural Networks on Modern GPUs. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 359–371. https://doi.org/10.1145/3352460.3358269

[90] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. 2018. Learning Transferable Architectures for Scalable Image Recognition. *arXiv:1707.07012 [cs, stat]* (April 2018). arXiv:1707.07012 [cs, stat]