Sparse Tensor Core: Algorithm and Hardware Co-Design for Vector-wise Sparse Neural Networks on Modern GPUs

Maohua Zhu*
University of California, Santa Barbara
maohuazhu@ece.ucsb.edu

Tao Zhang
Alibaba DAMO Academy
t.zhang@alibaba-inc.com

Zhenyu Gu
Alibaba DAMO Academy
zhenyu.gu@alibaba-inc.com

Yuan Xie
University of California, Santa Barbara
yuanxie@ece.ucsb.edu

ABSTRACT

Deep neural networks have become the compelling solution for the applications such as image classification, object detection, speech recognition, and machine translation. However, the great success comes at the cost of excessive computation due to the over-provisioned parameter space. To improve the computation efficiency of neural networks, many pruning techniques have been proposed to reduce the amount of multiply-accumulate (MAC) operations, which results in high sparsity in the networks.

Unfortunately, the sparse neural networks often run slower than their dense counterparts on modern GPUs due to their poor device utilization rate. In particular, as the sophisticated hardware primitives (e.g., Tensor Core) have been deployed to boost the performance of dense matrix multiplication by an order of magnitude, the performance of sparse neural networks lags behind significantly.

In this work, we propose an algorithm and hardware co-design methodology to accelerate the sparse neural networks. A novel pruning algorithm is devised to improve the workload balance and reduce the decoding overhead of the sparse neural networks. Meanwhile, new instructions and micro-architecture optimization are proposed in Tensor Core to adapt to the structurally sparse neural networks. Our experimental results show that the pruning algorithm can achieve 63% performance gain with model accuracy sustained. Furthermore, the hardware optimization gives an additional 58% performance gain with negligible area overhead.

CCS CONCEPTS

• **Computer systems organization** → *Single instruction, multiple data.*

KEYWORDS

neural networks, graphics processing units, pruning

*This work is done during his internship in Alibaba.

MICRO-52, October 12–16, 2019, Columbus, OH, USA

ACM ISBN 978-1-4503-6938-1/19/10…$15.00
https://doi.org/10.1145/3352460.3358269

ACM Reference Format:
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 *The 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52), October 12–16, 2019, Columbus, OH, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3352460.3358269

1 INTRODUCTION

Deep neural networks (DNNs) have achieved state-of-the-art performance in many different tasks, such as image recognition [31, 39, 67], speech recognition [62], and natural language processing [15, 36, 71]. The underlying representational power of these neural networks comes from the huge parameter space which results in an extremely large amount of computation and memory usage. There have been plenty of prior works to improve both performance and energy efficiency of neural networks on various hardware platforms, such as GPUs [9, 18, 38, 44, 60, 73, 78], FPGAs [19, 28, 81], and ASICs [2, 3, 11–14, 20, 21, 23, 32, 34, 35, 45, 47, 49, 54, 57, 59, 61, 63, 65, 68, 69, 75, 82, 83]. Among these prior arts, sparsity-centric optimization techniques [4, 28–30, 56, 64], which exploit the sparsity in weights and activations, have achieved outstanding results for both Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs).

By leveraging the intrinsic redundancy in the weights of neural networks, various sparsifying techniques have been discussed for weight pruning. Those techniques can result in very high sparsity in the weight matrices. For instance, prior work [28–30] has reported that top-K sparsifying and retraining could result in more than 90% sparsity with negligible impact on the model accuracy. Nonetheless, such high sparsity does not necessarily guarantee that the sparse neural networks can be more performant than their dense counterparts due to the irregularity of data layout. In particular, the sparse neural networks can hardly obtain performance gain on Graphics Processing Units (GPUs). The state-of-the-art sparse library, CUSPARSE, encodes a sparse weight matrix to Compressed Sparse Row (CSR) format [8]. Since a sparse weight matrix pruned by the top-K sparsifying has a random number of non-zero elements in a row, the CSR format often leads to poor workload balance. As a consequence, the GPU is extremely underutilized when running the sparse library kernels.

On the other hand, general matrix multiplication (GEMM) has seen contiguous optimization on modern GPUs, as it is one of the

fundamental primitives of many popular neural networks. For example, NVIDIA has built high performance GEMM kernels with a hand-tuned machine code entity in the state-of-the-art CUBLAS library [51]. In addition, Tensor Core has been introduced in Volta architecture [53] to provide 8× peak TFLOPs than the FP32 CUDA Core (112TFLOPs v.s. 14TFLOPs). Unfortunately, Tensor Core focuses only on the acceleration of dense matrix multiplication. Since sparse GEMM cannot take advantage of Tensor Core, we have seen little speedup when running sparse neural networks by top-K sparsifying on it.

To address the performance issue, structural sparsifying methods [70, 76] have been proposed by removing entire rows or columns from a matrix. As a result, the structurally pruned matrices are able to sustain their denseness and thus use the Tensor Core to achieve high performance. However, such coarse-grained pruning on the weight matrices has a negative impact on the model accuracy. Although prior studies [70, 76] have reported comparable accuracy of the structurally pruned networks on small datasets (e.g., MNIST [40]), we have observed significant accuracy drop from large-scale neural networks when the structural pruning is applied (See Section 6 for the detail).

In this paper, we propose *VectorSparse*, a SIMD (Single Instruction, Multiple Data)-friendly sparsifying method, to tackle the problem. VectorSparse divides a weight matrix into multiple vectors and prunes each vector to the same sparsity. The sparse weight matrices generated by VectorSparse exhibit a better workload balance and higher parallelism than the top-K pruned weight matrices. To further improve the performance, we extend the instruction set of Volta to allow the VectorSparse neural networks to run on Tensor Core. The extension requires only minor changes to enable the necessary indexing to the register files. The simulation results show that VectorSparse neural networks are faster than either the dense or top-K sparse counterparts with negligible accuracy impact.

To the best of our knowledge, this is the first work to exploit the efficiency of sparse neural networks on Tensor Core. The contributions of this paper include:

- We go through a comprehensive performance analysis to demonstrate the inefficiency of GPU when running the sparse neural networks.
- We propose VectorSparse as a novel sparsifying algorithm that can achieve 63% performance improvement with negligible accuracy drop.
- We further extend the instruction sets of the Volta GPU to support the operand indexing in the register file.
- We also show the details of the micro-architecture design to mitigate the performance bottleneck, which achieves 58% performance gain with negligible area overhead.

2 BACKGROUND AND MOTIVATION

In this section, we first review some prior work on sparsity-centric optimization for neural networks, and then describe the existing sparsifying techniques in detail.

2.1 Sparsity-Centric Optimization for DNNs

Recently, DNNs have demonstrated significant redundancy in the parameterization [17]. The over-sized parameter space results in high sparsity in a neural network. In addition to the weight parameters, the activations of each layer of a network also possess sparsity, a factor that stems mainly from the activation functions (e.g. ReLU) [39].

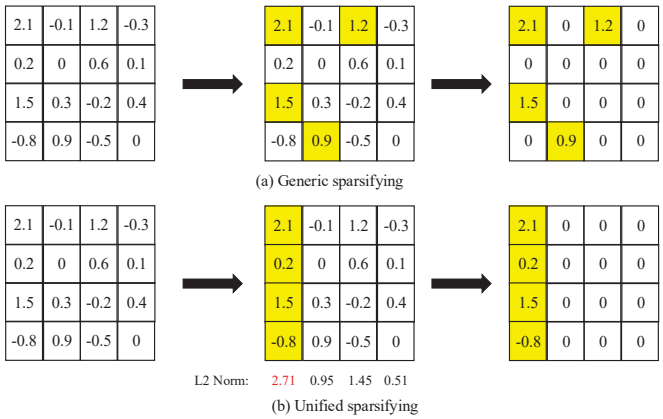


Figure 1: Examples of (a) generic sparsifying and (b) unified sparsifying. Both examples enforce 75% sparsity on a 4x4 matrix.

As the sparsity in weight parameters does not depend on the input data, it is often referred to as *static sparsity*. On the other hand, the sparsity in the activations depends on not only the weight, but also the input data. Therefore, such sparsity in the activations is denoted as *dynamic sparsity*. In this work, we focus on exploiting the static sparsity in neural networks to accelerate the inference phase of applications.

Early efforts that exploit the static sparsity concentrate on pruning the weights of the neural networks with top-K sparsifying [4, 28–30, 82]. The top-K pruning achieves great success in terms of compression ratio. However, the randomness in the positions of the non-zero elements in the top-K pruned weight matrices makes them unable to leverage sophisticated software libraries, e.g. CUBLAS [51], or hardware resources, e.g. Tensor Core [53] on modern GPUs. Hence they exhibit far lower data throughput than the corresponding dense neural networks.

To improve the efficiency of the sparse neural networks on GPUs, some work has proposed the structural sparsifying methods [70, 76]. The structural sparsifying puts certain spatial constraints on the non-zero elements to keep the denseness of the output matrices after sparsifying. The generated dense weight matrices have less parameters and can take full advantage of dense GEMM libraries. Even though such structural sparsifying has high performance, it incurs severe accuracy drop for large commercial models[1]. This is because the restricted spatial constraints of the weights make it hard to train the networks.

2.2 Existing Sparsifying Methods on GPUs

In general, the sparsifying methods [27, 30, 70, 76] can be classified into two categories, *generic sparsifying* and *unified sparsifying*. The generic sparsifying method is illustrated in Figure 1(a), which

[1]We have also observed that small neural networks could achieve good speedup and similar accuracy with structural sparsifying. These small models, however, are out of our interests since they cannot be widely adopted by the industry.

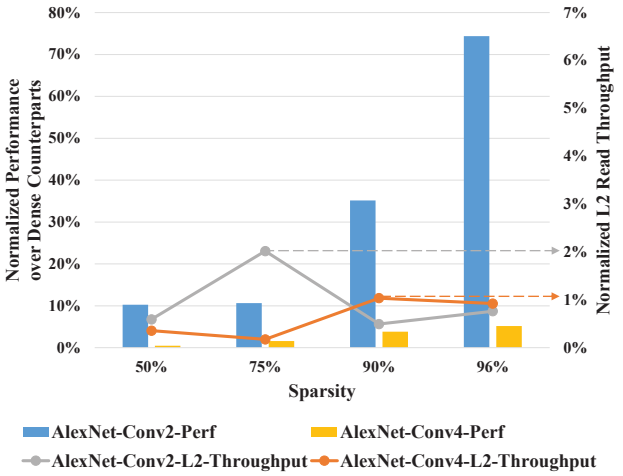


Figure 2: The normalized performance and L2 cache throughput of generic sparse CONV layers over dense CONV layers on a Tesla V100 GPU.

picks the largest four elements in absolute value as key elements in the matrix (highlighted in yellow). The key elements are kept unchanged while the rest of the elements are forced to be zero, which results in 75% sparsity in the matrix. Note that the coordinates of the key elements have to be kept along with the values as they can be arbitrary due to the flexibility of sparsifying.

Although the generic sparsifying achieves a very high compression ratio, it exposes several inefficiencies on GPUs [66]. Firstly, the variation of the row/column length of a generic sparse matrix makes it difficult to partition the workload evenly into GPUs [43]. Secondly, the number of non-zero elements in each row is unknown until runtime, leaving it difficult to choose an optimal tiling scheme for data reuse. Thirdly, the computation amount of a highly sparse matrix is not enough to hide the long memory access latency, and therefore the benefit from the high sparsity vanishes.

To reveal the problem, we run two sparse convolutional layers (Conv2 and Conv4) in AlexNet [39] on an NVIDIA Tesla V100 GPU. The convolution operation in these two layers is converted to GEMM by *im2col* transformation [10]. The implementation of the sparse layers is based on CUSPARSE, the state-of-the-art GPU library for sparse linear algebra[2]. Figure 2 shows the performance in GPU execution time and the read throughput of L2 cache, which are normalized to the dense implementations on CUBLAS.

Intuitively, layers with high sparsity should have better performance since they need less computations. However, as shown in Figure 2, the sparse layers are less performant than dense layers. Even when the sparsity is 96%, the sparse layers can only achieve 73% of the dense layer performance. The low L2 read throughput indicates that the device memory bandwidth is underutilized, which means the performance of the sparse layers is bounded by computation. We figure out that the compute units are underutilized in this case. The low utilization is due to the poor workload balance because we have observed better performance from Conv2 layer when it exhibits better workload balance.

[2]Please refer to Section 5 for the detail of the experiment setup.

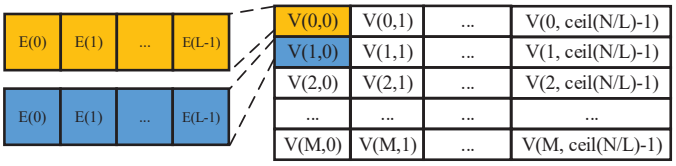


Figure 3: Dividing a $M \times N$ matrix into L-dim vectors for locality characterization.

On the other hand, the unified sparsifying is illustrated in Figure 1(b). Distinct from generic sparsifying, 75% sparsity is achieved by a column-wise sparsifying. The values of an entire column remain unchanged while the rest are forced to be zero. Therefore, it is easy to encode/decode the coordinate information. In this example, the unified sparsifying evaluates the L2 norm of each column and picks the column with the largest result.

A consequence of this selection process is the high probability that some key elements that would be kept in the generic sparsifying are removed in the unified sparsifying. Similar to the unified sparsifying, coarse-grained sparsifying in large blocks [27] can even remove an entire block out of a matrix at a time. Even though this approach allows more flexibility by adjusting the block size, it cannot use the highly optimized dense libraries to get good performance boost over the dense counterparts.

Therefore, we would like to see if there is an opportunity to get the best from both worlds. That is, we try to find a sparsifying method to achieve better performance than generic sparsifying and meanwhile eliminate the accuracy drop brought by the unified sparsifying. For this purpose, a highly flexible structural sparsifying method is desirable to preserve a comparable model accuracy with the generic sparsifying while the workload is balanced enough to ensure high performance on modern GPUs.

2.3 The Characterization of Sparsity

To find such a sparsifying algorithm, we first characterize the spatial locality of the non-zero values in the sparse neural networks. We prune ResNet-18 [31] and NMT [50] with the generic sparsifying method used in Deep Compression [30]. The two pruned networks have more than 90% sparsity and comparable accuracy with that of the dense references.

After a network is pruned, we split each row of its weight matrices into multiple L-dim vectors. Note that the vectors do not overlap with each other. Figure 3 shows an example of the split. Vector $V(y, x)$ contains the elements with row index y, and column indices from $x \times L$ to $(x + 1) \times L - 1$ in the $M \times N$ weight matrix W. If N is not divisible by L, the residue vectors are padded with zero. For each vector within the sparse weight matrix, we count the total number of zeros in the vector (in the range $[0,L]$), and then compute the *local sparsity degree* of each vector. The local sparsity is defined as the number of zeros divided by L.

Figure 4 shows the cumulative distribution of local sparsity degree with three vector sizes, 4, 8, and 16. As shown, only less than 30% of 4-dim vectors have $\leq$ 75% sparsity. As there are only 4 elements in a 4-dim vector, this result indicates that more than 70% of the 4-dim vectors do not have any non-zero elements. Moreover, only less than 2% of the 4-dim vectors have $\leq$ 25% sparsity. In other

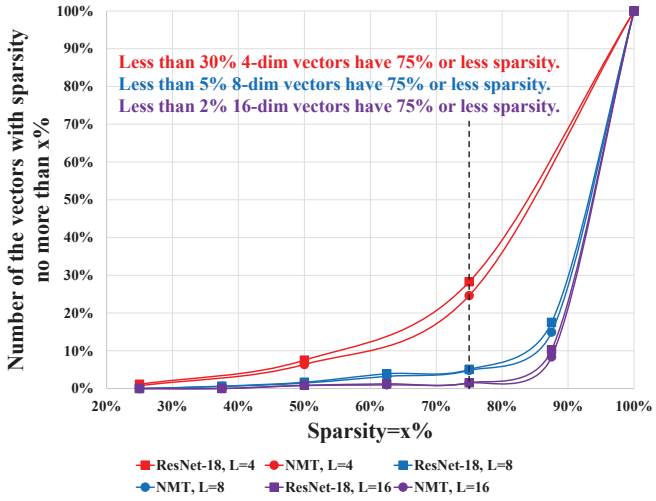


Figure 4: The cumulative distribution of the vectors vs. the local sparsity degree in a vector. The vertical axis is the number of the vectors with the sparsity not greater than x%. The horizontal axis is the sparsity x%.

words, very few of the vectors have more than 2 non-zero elements. Based on this observation, the spatial distribution of the non-zero elements in the sparse neural networks are generally retained if we only keep up to 2 non-zero elements in each vector. Instead of the location-unaware element selection used in the generic sparsifying, this approach generates a 50% sparse matrix with a balanced spatial distribution of non-zero weights.

Similarly, the experimental results for 8-dim and 16-dim vectors demonstrate that more than 95% of vectors have $\geq$ 75% sparsity. The cumulative distribution shows that there are less low sparsity vectors when the vector size is increasing. This is because the local sparsity in a larger scope more likely resembles the global sparsity. According to the observation, it inspires us to divide a weight matrix into L-dim vectors so that each vector can be sparsified independently to achieve both sparsity balance and comparable model accuracy.

3 VECTORSPARSE PRUNING

The characterization of spatial locality opens a great opportunity to avoid accuracy penalty by splitting weight matrices into vectors and sparsifying each vector to the same sparsity. However, the encoding formats for generic sparse matrices, e.g. CSR format [8], do not contain the information associated with the vectors. In this section, we first propose a balanced vector-wise encoding format for sparse matrices that simplifies the workload partitioning on GPUs. Then, we design a novel vector-wise sparsifying algorithm to prune a trained dense network to a sparse network that can maximize the vector-wise encoding efficiency.

3.1 Vector-wise Sparse Matrix Encoding

To improve the workload balance of the sparse neural networks on GPUs, we propose a three-phase vector-wise encoding method to sparsify a matrix. In the first phase, we divide a matrix into L-dim vectors, as shown in Figure 3. An $M \times N$ matrix thus has

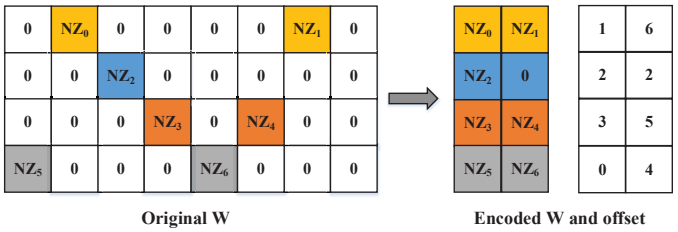


Figure 5: An example of vector-wise sparse matrix encoding with L=8 and K=2. Two non-zero elements in a row vector are compressed into one compact vector associated with their indices. All row vectors are encoded to the same length. If a row vector has less non-zero elements than the compact vector length K, the empty entries are padded with zeros.

$M \times \lceil N/L \rceil$ vectors. In case N is not divisible by L, the residue vectors are padded with zero. In the second phase, we count the number of non-zero elements N_{nz} in each vector, and then denote the maximum N_{nz} of all the vectors in this matrix as K ($K \leq L$). In the third phase, we compress each vector into a K-dim vector along with their associated indices in the original vector. Note that a vector might have less than K non-zero elements after compression. For example, the second row vector in Figure 5 has only one non-zero element NZ_2. For such vectors, the empty entries are filled with zeros to assure those vectors' K-dim. The vector-wise encoding can either be column-wise or row-wise. Without losing generality, we use column vectors unless specifically illustrated.

Theoretically, $\lceil log_2 L \rceil$ bits are required to encode each index in a L-dim vector. Consequently, the overall compression ratio of this encoding is $\frac{P \times L}{(P+\lceil log_2 L \rceil) \times K}$, where P stands for the number of bits used to store the value of an element. Figure 5 shows an example of the encoding for a 4×8 matrix, where L=8, K=2. In this 4×8 FP16 matrix, we observe that each element of the offset index array can be represented in only 3-bit, as the index is in the range $[0, 7]$. Therefore, the compression ratio is 3.37x. As K is the maximum number of non-zero elements in a vector, this encoding could achieve an ideal compression ratio when all vectors have the same number of non-zero elements. However, if the number of non-zero elements vary too much, the compression ratio could be far from the ideal case.

Fortunately, neural network pruning allows us to tailor the topology of weights to achieve the spatial distribution for the ideal case of the vector-wise encoding. Instead of pursuing high overall sparsity, the pruning method for the vector-wise encoding tries to minimize the maximum number of non-zero elements in a vector.

3.2 VectorSparse: a Methodology of Iterative Vector-wise Sparsifying and Retraining

To achieve a spatially even distribution of non-zero elements in a neural network weight matrix, we propose VectorSparse: a pruning methodology with iterative vector-wise sparsifying and retraining for CNNs and RNNs. The vector-wise sparsifying can be considered as local sparsifying, which takes advantage of the aforementioned vector-wise sparse encoding. For convolutional layers in CNNs, we refer to the $N \times (CHW)$ matrix generated by the *im2col* transformation [10] as the weight matrix, where N is the number of filters,

Algorithm 1: VectorSparse algorithm: pruning with vector-wise sparsifying.

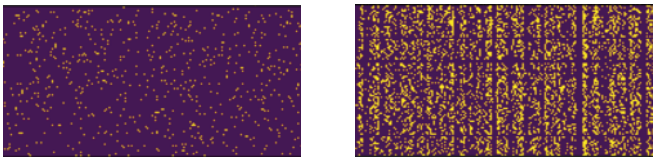
input : Weight matrix of a trained NN layer, W_0;
The vector size, L;
Maximum accuracy drop, E_δ.
output : Pruned vector-wise sparse weight matrix, W_s

1 W=W_0;
2 Divide W into $vector_{i,j}$;
3 N_{zero}=0;
4 E_0 = ValidationError(W);
5 E = E_0;
6 **while** $\frac{E-E_0}{E_0} < E_\delta$ **do**
7 N_{zero}=N_{zero} + 1;
8 **for** *each i, j* **do**
9 Sort absolute values of all elements in vector[i][j] in ascending order and save sorted elements in $sorted[L]$;
10 T_{ij}=$sorted[N_{zero}]$;
11 **for** *each element in vector[i][j]* **do**
12 Remove *element* if abs(*element*) < T_{ij};
13 **end**
14 **end**
15 Fine tune the pruned W;
16 E = ValidationError(W);
17 **end**
18 W_s=W;

C is the number of channels of each filter, and H and W are the height and width of a filter, respectively.

The vector-wise sparsifying sorts the elements in each vector by their absolute values. The largest K elements in absolute value are kept unchanged while all other elements are pruned. After this phase, all vectors have at most K non-zero elements so that they can be encoded to K-dim vectors by our vector-wise sparsity encoding. Although setting a small K can easily increase the overall compression ratio, directly pruning a dense weight matrix to a small K vector-wise encoding could lead to significant accuracy drop of the neural networks.

To address the accuracy drop issue, we propose a progressive pruning method by gradually decreasing K in the vector-wise sparsifying. Algorithm 1 shows the flow of our VectorSparse pruning. Starting from a trained, dense neural network, our algorithm prunes the network layer by layer. Given a dense weight matrix W_0, a user-specified vector size L as the input parameter, and the maximum accuracy drop E_δ as the acceptable error rate, the sparsity of the weight matrix gradually steps up (*Line#7*) until the validation error of the pruned neural network exceeds the error rate (*Line#6*). Starting from a dense weight matrix, VectorSparse prunes the elements that do not fall into the Top-N absolute values within each vector. Then the weights are tuned based on the pruned topology with the same training dataset. After the fine tuning, the algorithm evaluates the validation error with that of the dense network, E_0. The relative difference of the validation error between the pruned network and the dense network is used to determine if the pruning process can continue. VectorSparse provides the flexibility to specify the acceptable error rate, which usually varies in different applications. If an application is more sensitive to latency rather than accuracy, the maximum accuracy drop E_δ can be set higher to gain more sparsity. Otherwise, the maximum accuracy drop should be set small enough to ensure the accuracy.



(a) **(b)**

Figure 6: The spatial distribution of non-zero weights in neural networks pruned by (a) the generic method with 96% sparsity and (b) the vector-wise approach with 75% sparsity, respectively. Each yellow pixel represents a non-zero element. It is clear that vector-wise pruning achieves better regularity.

Because of the additional spatial constraint, VectorSparse usually chooses a different set of weights from the generic pruning [30] before each fine tuning step. The prior studies have figured out that the difference in the pruning pattern has negligible impact on the speed of convergence when comparing to the generic pruning [22, 48, 80, 84]. On the other hand, the number of pruned synapses between two retraining phases does affect the accuracy of a pruned neural network. This factor is controlled by the vector size L and the sparsity N_{zero} in the algorithm. Figure 6 shows the spatial distribution of non-zero elements of pruned ResNet-50 weights by our vector-wise pruning, which leads to better workload balance than the generic pruning.

3.3 GPU Kernel Design

Being aware of the spatial distribution of the non-zero elements, our vector-wise encoding method allows the sparse matrix multiplication to be efficiently mapped on GPUs with good workload balance. Correspondingly, we make minor modification to the GPU kernel for vector-wise sparse matrix multiplication. The vector-wise encoded sparse matrix multiplication kernel defines a user-interface function similar to the standard GEMM API with two extra parameters, vector size L and the maximum number of non-zero element in a vector K.

Let's assume a general matrix multiplication, **C**=**A**×**B**, where the sizes of matrices **A**, **B**, and **C** are 4×8, 8×6, and 4×6, respectively. In the traditional dense matrix multiplication kernel, the product of every row of **A** and every column of **B** needs to be computed,

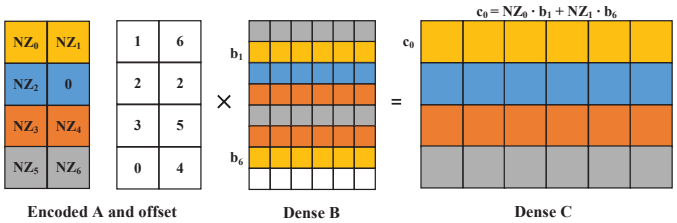


Figure 7: An example of vector-wise sparse matrix multiplication.

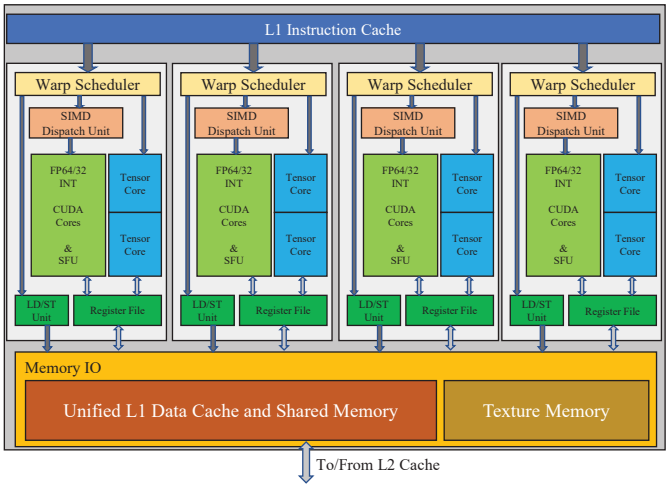


Figure 8: The architecture of a streaming multiprocessor (SM) of the Volta GPUs. Branch unit, L0 instruction cache, and constant cache are omitted for brevity.

regardless of sparsity. In particular, the first row of matrix **C** needs 48(=8×6) multiplications. On the other hand, as shown in Figure 7, after it is pruned by the VectorSparse algorithm, the 4×8 weight matrix **A** becomes a 4×2 vector-wise sparse matrix with an associated offset matrix of the same size. As a consequence, the vector-wise sparse matrix multiplication kernel only needs a subset of the elements of matrix **B** to compute the product matrix **C**. For example, since the first row of sparse matrix **A** has non-zero elements at column 1 and 6, the calculation of first row $\vec{c}_0$ in **C** is equivalent to $NZ_0 \cdot \vec{b}_1 + NZ_1 \cdot \vec{b}_6$, where $\vec{b}_1$ and $\vec{b}_6$ stand for the corresponding rows in **B**. As a result, only 12(=2×6) multiplications are executed, resulting in a 75% multiplication reduction.

In high-performance GPU matrix multiplication kernels, tiling is widely used for large size matrix multiplications. A warp is responsible for the computation of a tile of the product matrix **C**. The proposed vector-wise sparse matrix multiplication is orthogonal to the tiling techniques so that tiling can be applied to it as well. To make sure the tiled multiplication performs well, the indexing of data-dependent row in matrix **B** should be designed carefully. We will cover this topic in the next section.

4 SPARSE TENSOR CORE

So far, all benefits given by the VectorSparse pruning algorithm are in theory. As we know, generic sparse matrix suffers from the poor workload balance and cumbersome coordinate decoding for its overall performance. In fact, VectorSparse could present the same issue if the hardware design is unaware of the new algorithm. In this section, we go through the design details which make the hardware adaptive to the algorithm. In particular, we modify the Tensor Core in order to have full support for VectorSparse, which we refer to *sparse Tensor Core*.

4.1 Baseline Tensor Core Architecture

The Tensor Core is a fast hardware functional block for dense matrix multiplication. It was first introduced in NVIDIA's Volta

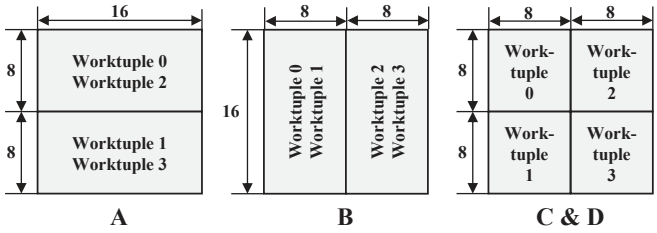


Figure 9: The mapping of a 16×16×16 matrix multiplication into four worktuples in a warp [58]. The computation task for the product matrix D is evenly partitioned into four worktuples.

architecture [53]. Each Tensor Core is able to execute a 4×4×4 matrix multiplication and addition in one cycle. The Tensor Core in Volta GPUs provides two execution modes, *FP16 mode* and *mixed precision mode*. In the FP16 mode, all matrices are in FP16. In the mixed precision mode, the Tensor Core uses FP32 accumulators and writes back the results to an FP32 matrix.

Figure 8 shows the architecture of one of the streaming multiprocessors (SMs) in Volta GPU. As illustrated, an SM consists of four subcores. In each subcore, there is a warp scheduler, a math dispatch unit, streaming processor arrays for multiple data types (a.k.a. CUDA Cores), special function units (SFUs), two Tensor Cores, LD/ST unit, and register files. The L1 data cache and shared memory are shared among the four subcores within the SM.

During program execution, two Tensor Cores are used concurrently by a warp [58]. In the CUDA programming model [53], Tensor Cores are exposed to programmers in the CUDA WMMA (Warp Matrix Multiply and Accumulate) API. The WMMA API includes dedicated matrix load and store primitives, and matrix multiply and accumulate operations for Tensor Cores. The WMMA matrix load and store operations are designed for moving data between register files and the memory hierarchy.

Given **A**, **B**, **C**, and **D** are 16×16 matrices, a warp computes a matrix multiply and accumulate, **D**=**A**×**B**+**C**. Even though NVIDIA has not disclosed the design details of Tensor Core, some work has revealed how the 32 threads within a warp collaborate to conduct the 16×16×16 matrix multiply and accumulate operation efficiently [58]. To execute a WMMA, the 32 threads in a warp are divided into 8 *threadgroups*. The *threadgroup Id* of a given thread is $\lfloor \frac{threadId}{4} \rfloor$. All threads in a threadgroup work together to compute 4×4 tile multiplications. Furthermore, for better data reuse, two threadgroups work together as a *worktuple*. *Worktuple i* consists of *threadgroup i* and *threadgroup i* + 4.

Figure 9 shows the elements processed by each worktuple in one WMMA operation. Each worktuple is responsible for computing one 8×8 tile of **D**. For example, Worktuple 0 computes $\mathbf{D}[0:7, 0:7]$. To achieve this, Worktuple 0 multiplies $\mathbf{A}[0:7, 0:15]$ and $\mathbf{B}[0:15, 0:7]$, adds the product 8×8 tile with $\mathbf{C}[0:7, 0:7]$, and saves the result to $\mathbf{D}[0:7, 0:7]$.

During the compilation time, a WMMA operation breaks down into four sets of machine-level HMMA instructions [58]. In the mixed precision mode, each set of HMMA instructions computes the product of a 4×4 tile of **A** and a 4×8 tile of **B**. Figure 10 left shows

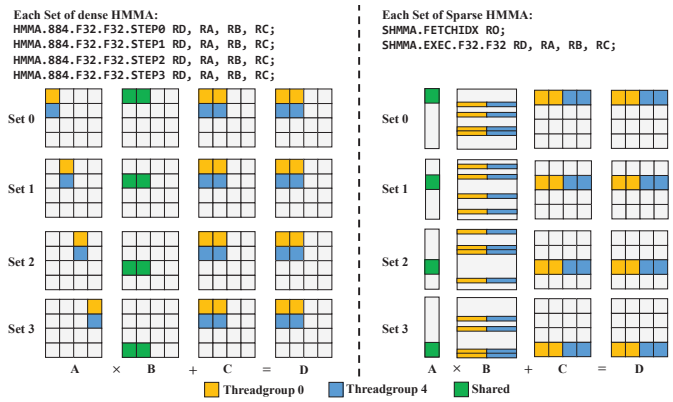


Figure 10: Elements processed by the threadgroups in Worktuple 0 in the dense mode (left) [58] and the sparse mode (right) of a dense/sparse WMMA PTX instruction, respectively.

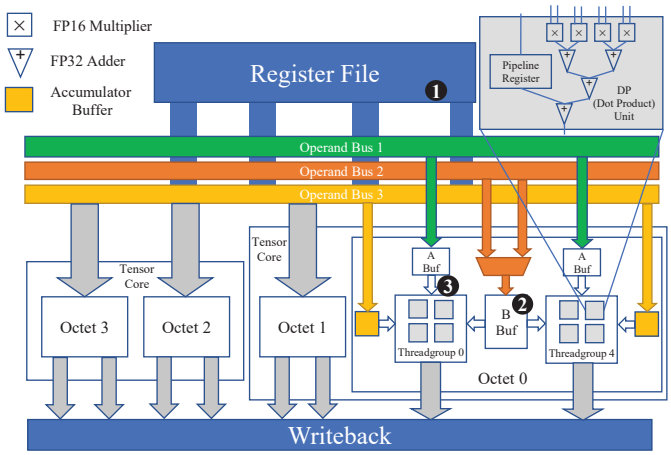


Figure 11: Tensor Core architecture [58].

the tiles processed by each set of HMMA instructions in Worktuple 0. The four sets of HMMA instructions of Threadgroup 0 and 4 compute the product submatrices $\mathbf{D}[0:3, 0:7]$ and $\mathbf{D}[4:7, 0:7]$, respectively. At the execution of one set of the HMMA instructions, the two threadgroups in Worktuple 0 share a 4×8 tile of **B** in each set. On the other hand, the 4×4 tiles of **A** are private to the threadgroups, respectively.

Figure 11 illustrates how a WMMA operation is mapped to the Tensor Core architecture [58]. There are two *octets* in a Tensor Core. Inside an octet, there are eight dot product (DP) units, each of which can compute a 4-dim vector dot product per cycle. During the execution, a worktuple is mapped to one octet and thus each threadgroup takes four DP units, respectively. The octet has operand buffers to feed the worktuple via the tiled data when executing one set of HMMA instructions. Each threadgroup has dedicated operand buffers for Operand **A** and Operand **C**. Each operand buffer can hold a 4×4 tile. On the other hand, the operand buffer dedicated to Operand **B** can hold a 4×8 tile and the data inside are shared by the two threadgroups in the same worktuple.

One threadgroup computes the multiplication of a 4×4 tile by a 4×8 tile in a set of HMMA instructions, which is 4×8=32 4-dim dot products. Because four DP units compute four 4-dim vector dot products per cycle, those set of HMMA instructions require at least 8 clock cycles to finish the computing of the 4×8×4 matrix multiplication.

4.2 The Extension of HMMA Instruction Set

With two threadgroups, a worktuple computes an 8×8×4 matrix multiplication in a set of HMMA instructions. Such a set of the HMMA instructions for the mixed precision mode are listed below. Note that the four HMMA instructions must be used together and in this particular order [53, 58].

- `HMMA.884.F32.F32.STEP0 RD, RA, RB, RC;`
- `HMMA.884.F32.F32.STEP1 RD, RA, RB, RC;`
- `HMMA.884.F32.F32.STEP2 RD, RA, RB, RC;`
- `HMMA.884.F32.F32.STEP3 RD, RA, RB, RC;`

A register name in the HMMA instructions stands for a register pair. Each register pair contains four FP16 operands. For example, if register RA is mapped to R28, it means the register pair, R27 and R28, is holding the 4×4 tile of data for **A**. The shared 4×8 tile of **B** is loaded from the RB. The result **D** is written back to RD. The instructions for FP16 mode look similar, but have FP16 accumulators instead of FP32 accumulators. Without loss of generality, we only consider the mixed precision mode in this work, as prior profiling results have shown that it has lower latency than the FP16 mode [58].

To run the vector-wise sparse matrix multiplication on the Tensor Core, we create a *vector-wise sparse mode* and refer to the original execution mode as *dense mode*. Figure 12 shows the *sparse HMMA* (SHMMA) flow on the Tensor Core to execute the vector-wise sparse matrix multiply and accumulate operation. The matrix **A** is encoded into a 16×4 matrix with the setting L=16 and K=4 so that we map four rows of the encoded **A** to each worktuple. Worktuple i computes Row $4i$ to Row $4i+3$. Therefore, in the vector-wise sparse mode, Tensor Core still computes a 16×16 matrix multiply and accumulate operation at warp level. Similar to Figure 5, the 16×16 sparse matrix **A** is encoded to a 16×4 data matrix and an associated 16×4 offset matrix. Since all the offsets in this encoding are in the range [0, 15], each offset only requires 4 bits in memory. Therefore, each row of the encoded **A** only requires 16 bits to store the 4 offsets, which means they can be stored in one register.

Different from the dense mode, the two threadgroups compute the same row of **A** in the vector-wise sparse mode, as shown in Figure 10 right. As illustrated in Figure 7, Row i of **D** is computed by multiplying the four non-zero elements in Row i of **A** with the corresponding four rows of **B**, respectively, and then accumulating the results with Row i of **C**. The four rows of **B** to be multiplied are determined by the four offset indices saved in the offset register.

To implement the vector-wise sparse mode, we extend the Tensor Core instruction set by adding two SHMMA instructions and one *offset register*:

- `SHMMA.FETCHIDX RO;`
- `SHMMA.EXEC.F32.F32 RD, RA, RB, RC;`

The instruction `SHMMA.FETCHIDX` fetches the offset indices of the four elements in a row of **A** from RO to an implicit, dedicated offset

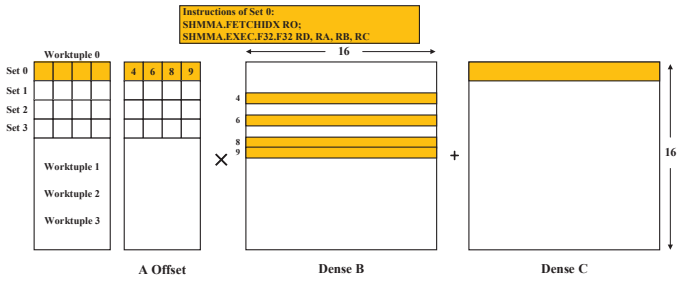


Figure 12: Sparse WMMA (SWMMA) execution flow on the Tensor Core architecture. Operand A is vector-wise encoded to a 16×4 matrix and a 16-dim offset array. Each entry of the offset array contains the 4 offset indices of the elements in the associated row.

register. The instruction `SHMMA.EXEC` first decodes the offset register to determine which rows of **B** to be fetched from RB. After the data is loaded to operand buffer *B*, instruction `SHMMA.EXEC.F32.F32` computes the 16 4-dim dot products and accumulates the results with **C**. Instead of four HMMA instructions in one set in the dense mode, two SHMMA instructions form one set in the sparse mode, where the two instructions must work together and in order.

In the vector-wise sparse mode, each threadgroup computes 8 columns of **B** by a set of SHMMA instructions. Therefore, four sets of SHMMA instructions are sufficient to complete the vector-wise sparse WMMA operation (SWMMA), denoted as one `swmma.mma` PTX instruction for sparse WMMA API. The SWMMA instructions are shown below, where `K` stands for the number of non-zero elements in each row of the *sparse* matrix **A**. Note that we do not show the load instructions for dense matrix **B** and **C** as well as the store instruction for **D** since they are the same as the APIs of the dense WMMA.

- Load **A**: `swmma.load.a.K ra, [pa];`
- Load **Offset**: `swmma.load.offset.K ro, [po];`
- Math: `swmma.mma.f32.f32.K rd, ra, rb, rc, ro;`

4.3 Micro-architecture Design for Sparse Tensor Core

The SHMMA instructions require some modifications to the original Tensor Core. We highlight the changes in Figure 11. ❶ We first add the dedicated offset registers in the register file. The offset registers can only be implicitly accessed by the SHMMA instructions.

In the baseline Tensor Core architecture, the operand buffer *B* only needs to hold 4×8 FP16 numbers as an octet loads a 4×8 tile in each set of the HMMA instructions [58]. To improve the utilization of the DP units, ❷ we not only double the buffer size to accommodate the four rows of buffer *B*, but also add another buffer to hide the load latency. In addition, ❸ we enable the broadcasting of operand buffer *A* to the four DP units it connects to so that all DP units in an octet can read the same row of **A**. By doing this, a threadgroup can compute the dot products of a row of **A** and four columns of **B** per clock cycle.

We use Figure 13, i.e. the execution timeline of each mode, to illustrate the performance benefit of SHMMA. In the dense mode (Figure 13(a)), the operand buffers *A* and *B* are filled in 2 cycles,

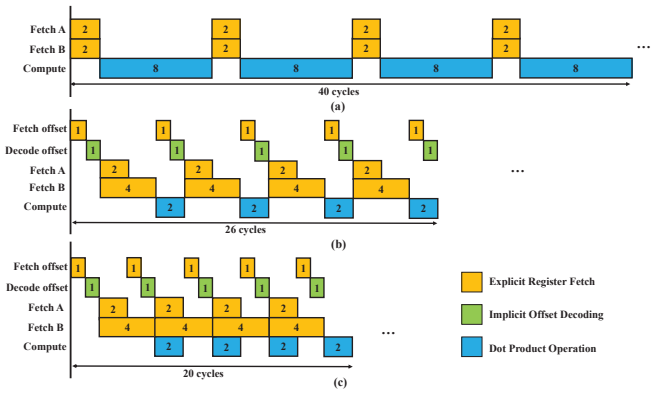


Figure 13: The execution timeline of (a) the dense mode, (b) the vector-wise sparse mode without ping-pong buffer, and (c) the vector-wise sparse mode with ping-pong buffer on the Tensor Core.

followed by the execution of HMMA instructions that takes 8(=4×2) cycles to complete a 4×8×4 GEMM. As a result, the warp has to take 40 cycles to complete the 16×16×16 GEMM computation in the dense mode.

In the vector-wise sparse mode (Figure 13(b)), the instruction `SHMMA.FETCHIDX` takes 1 cycle to load the offsets to the offset register. It then takes another cycle for the register file to decode the offset register and set up the control signals for the operand buffer *B*'s datapath. As the size of operand buffer *B* is doubled from 4×8 to 4×16 FP16 operands, it takes 4 cycles to load data to operand buffer *B*[3]. Thanks to the vector-wise sparsity, the computation time is reduced to only 2 cycles. As a result, the vector-wise sparse mode only takes 26 cycles for the computation, a 1.54× speedup.

One observation from Figure 13(a)(b) is that dense mode is compute-bound while vector-wise sparse mode is memory-bound (i.e., feeding data to operand buffer *B*). This is because the vector-wise sparse matrix multiplication has a much lower operational intensity (i.e., FLOPs/Byte) [77] than the dense matrix multiplication. It also explains why a even higher sparsity does not help to reduce the latency of the inference. To further improve the performance, we add another buffer to hide the register fetch latency. In total, the design requires a 4× larger buffer of *B*. With the ping-pong buffer design, one buffer can be read by DPs while the other is loading data from the register file. In this way, the total latency of the vector-wise sparse WMMA is further reduced from 26 cycles to 20 cycles (Figure 13(c)), an additional 1.3× speedup. In fact, the dense mode can also benefit from this larger buffer. However, since it is compute-bound, dense mode sees moderate latency reduction, from 40 cycles to 34 cycles. Considering the area overhead, it is not worthy adding a ping-pong buffer for dense mode.

5 EXPERIMENTAL METHODOLOGY

The algorithm and hardware co-design aims to accelerate the inference phase of neural networks with minimal impact on the quality of the models. To evaluate both the model accuracy and the speedup over generic sparse neural networks and dense neural networks, we

[3]We assume the same buffer load bandwidth, which takes 2 cycles to fetch 4×8 FP16 numbers to the buffer.

picked five popular neural networks in three domains: image classification, image captioning, and machine translation. We trained the neural networks with the generic sparsifying method [30] and the proposed VectorSparse method, respectively. The training was done on a single DGX-1 station with four NVIDIA Tesla V100 GPUs.

To evaluate the performance of the vector-wise sparse mode of the Tensor Core, we extended the WMMA PTX code model in the GPGPU-Sim simulator [5, 37, 42, 58]. We added the SHMMA instructions to the simulator with the timing parameters given in Figure 13. We configured the simulator to model a Tesla V100 GPU with Tensor Core [53]. The simulated V100 GPU has 80 SMs with 640 Tensor Cores and 5120 CUDA Cores inside. Equipped with 16GB HBM2 [41], V100 has 900 GB/s device memory bandwidth.

For the image classification applications, we pruned and retrained CNNs with the ImageNet [16] dataset, which comprises 1.2 million training examples and 50 thousand validation examples. To verify that our vector-wise sparsifying methods can work for commercial applications, we selected four popular CNNs, AlexNet [39], VGG-16 [67], ResNet-50 [31], and ResNeXt-50 [79] on the ImageNet ILSVRC-2012 dataset. AlexNet and VGG-16 are two popular networks that achieved high accuracy on the ILSVRC-2012 dataset. ResNet-50 has residual layers which make the training process easier for very deep neural networks. Furthermore, we also evaluated the ResNeXt-50 [79] with 32x4d configuration to show the accuracy impact on modern networks. We used the networks in TensorFlow model repository [26] as the reference.

In addition to CNNs, we also examined Long Short-Term Memory (LSTM) [33] as a representative of RNNs. We used the Show and Tell model [74] for the image captioning experiment. The image captioning model consists of an Inception V3 model [72] with an LSTM layer attached to the last layer of the CNN. The LSTM layer has 512 cells by default. Since our interest is only in the LSTM layer, we used a pre-trained Inception V3 model and randomly initialized the parameters of the LSTM layer similar to what was done in the original Show and Tell work [74]. The training dataset is MSCOCO [46] and the mini-batch size is set to 64. We trained the model for 500K steps (about 55 epochs under default configuration) in each retraining phase. To quantify the quality of generated captions, we calculated the BLEU [55] score for each training configuration on the MSCOCO test dataset. The code we used is taken from the TensorFlow model zoo [25].

For the machine translation application, we trained an encoder-decoder architecture with an attention mechanism to perform Neural Machine Translation (NMT) [7, 50]. We use an architecture with a 2-layer LSTM encoder, a 4-layer LSTM decoder, and an attention module. The first layer of the LSTM encoder is bidirectional while the rest of LSTM layers are unidirectional. Both the unidirectional and bidirectional layers have 512 LSTM cells. In the experiments, we also used the BLEU score [55] as the metric for the neural machine translation model. The WMT 16 English-German dataset [1] is used for training. We followed the instructions to reproduce the NMT training with an open source framework [24], except that only 16 epochs are used in the retraining phases. This simplification stems from our observation that the validation BLEU score becomes stable after 12 epochs.

In our experiments, the workloads were first trained with their default training methods to achieve the reference model accuracy. Then we applied our VectorSparse pruning method to the reference dense models. We use FP32 for the weights, activations, and gradients in the training process and CUDA Core based inference kernels. For Tensor Core based kernels, we dynamically downsized the FP32 weights and input activations to FP16 in each layer to avoid accuracy loss. Since we use the mixed precision mode, the output activations are still in FP32.

On the software side, we implemented vector-wise sparse matrix multiplication kernels based on CUTLASS [52], rather than CUBLAS due to the unavailability of its source code. CUTLASS is an open-source high-performance GEMM template library. It can achieve near-CUBLAS performance for most GEMM problems [52]. This library provides C++ GEMM interfaces and allows the data streams to be customized for GEMM-like computation. The CUDA Core based vector-wise sparse matrix multiplication kernels are built on the SGEMM kernels. The sparse Tensor Core based kernels are similar to the WMMA-GEMM kernels, but we call our sparse WMMA API instead of the dense WMMA API. The convolution operations in the CNN workloads are converted to GEMM by the im2col [10] method.

Besides the dense baseline, we also trained the workloads with the unified pruning method [76] and compared it against our VectorSparse method. For the sake of a fair comparison, we iteratively reduced the number of columns or channels of the weights and retrained the networks with the same setting as our VectorSparse method used. Then we evaluated the test accuracy of the trained neural network models.

6 EXPERIMENTAL RESULTS

To validate the proposed vector-wise pruning method, we first present the accuracy of the models pruned by our VectorSparse method with various configurations. And then we show the performance gain of the pruned vector-wise sparse networks with sparse Tensor Core design. Finally, we do the design overhead analysis on the sparse Tensor Core.

6.1 Impact on Accuracy

To demonstrate the generality of our work, we applied our VectorSparse pruning to various workloads. Figure 14 and 15 show the validation accuracy of the CNN and RNN models pruned by VectorSparse with L=16, the generic pruning method, and the unified pruning method, respectively. We also add the result of L=8 to show how badly the model accuracy drops for each workload when sparsity increases. In fact, we also run experiments with L >16 and found a marginal impact on the accuracy, regardless of the sparsity. Since the accuracy is insensitive to the vector size once L is greater than 16, we choose L=16 as the optimal size, which requires only 4 bits for storing the offset indices and enables finer-grained tiling strategies. During the pruning and retraining process of the VectorSparse pruning and the unified pruning, we recorded the validation accuracy of each step. The accuracy is represented in the relative deviation from the reference dense models.

As illustrated in Figure 14, all CNN models pruned with VectorSparse with L=16 can retain their accuracy until the sparsity reaches 80%. Similarly, the accuracy of the RNNs is comparable to the reference model when the sparsity does not exceed 75%, as

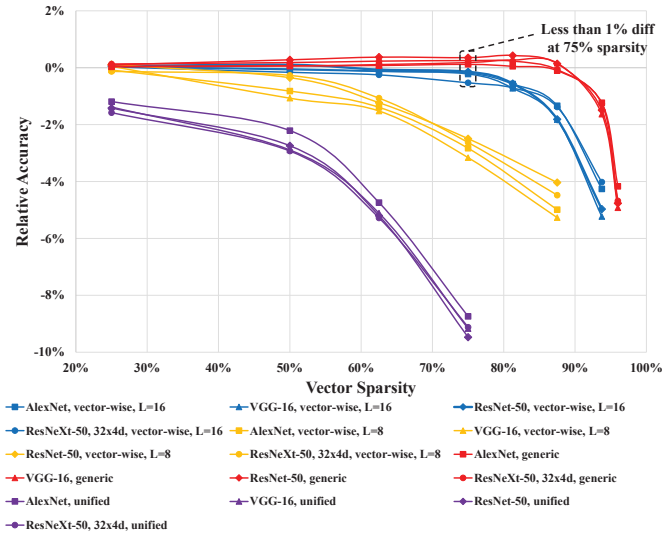


Figure 14: Accuracy vs. sparsity in the weight matrices of CNN workloads. Each workload runs with generic and unified pruning as described in Section 2.2. VectorSparse pruning is run with L=8 and 16, respectively. All accuracy results are normalized to the dense baseline.

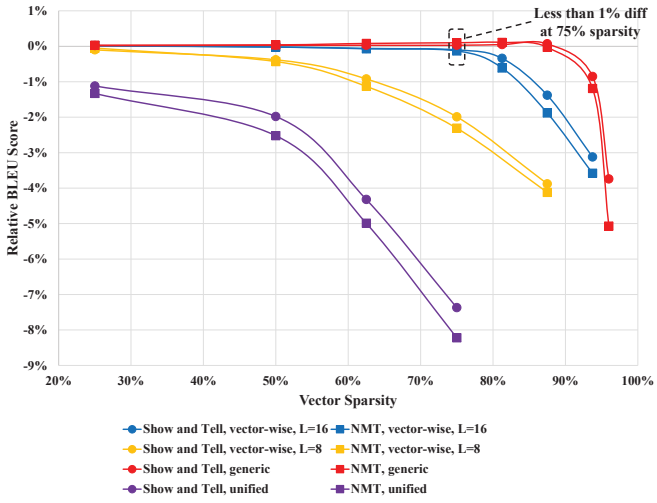


Figure 15: BLEU score vs. sparsity in the weight matrices of RNN workloads.

shown in Figure 15. Although the generic pruning method outperforms the VectorSparse with L=16 when the sparsity is higher than 80%, the generic sparse matrix is not necessarily able to be encoded into a vector-wise sparse format, if top-K elements concentrate on a few rows.

On the other hand, if vector size is set to L=8, the model accuracy drops more quickly than that of L=16. At the point of 75% sparsity, the L=8 scheme suffers more than 2% accuracy loss, which is usually unacceptable in many applications. This is because L=8 puts too much spatial constraint on the element removal. Figure 15 shows that RNNs are more resilient to the L=8 pruning than the CNNs. However, their accuracy of L=8 pruning is still incomparable to that of L=16.

Based on the results, it is clear that there is a trade-off between sparsity and accuracy. Given a vector size L, higher sparsity can achieve better performance by sacrificing the accuracy. In this work, the accuracy is the first-order metric so we opt out the L=8 scheme. However, we do believe that L=8 can be useful in some application domains. Even though it is out of the scope of this paper, we recommend that the trade-off should be done case by case.

Another observation is that unified pruning incurs the most significant accuracy drop for both CNNs and RNNs. By cutting the number of weight columns by half, the CNN and RNN models see more than 2% accuracy loss at the point of 50% sparsity. Due to the spatial constraint, the unified pruning has too few options for removing weights in each step of the retraining process. The lack of flexibility in turn limits the representative power of the network. In contrast, the vector-wise pruning has more freedom on removing weights so that it can even result in a similar topology to the generic pruning when L is not extremely small.

In summary, with L=16 vector-wise pruning, 75% sparsity is good enough to assure the accuracy. In other words, we can keep only 4 non-zero elements in each 16-dim vector. Since the warp size 32 is a multiple of 16[4], letting L=16 is also favorable for CUDA Cores. Therefore, we choose L=16 and 75% sparsity as the optimal configuration in the performance evaluation.

6.2 Performance Evaluation

We run the workloads under six different configurations to evaluate the performance of VectorSparse pruning method and the hardware design in Tensor Core. The baseline is *CUDA Core Dense*, where the dense NN workloads are running on the FP32 CUDA Cores. We also evaluate *CUDA Core Generic* and *CUDA Core Unified*, where NN workloads are pruned by generic and unified sparsifying method, respectively. The CUDA Core Generic has 96% sparsity and CUDA Core Unified has 50% sparsity. Then, we examine *Tensor Core Dense*, where the dense NNs are running on the Tensor Core. In this case, the weights and input activations are dynamically converted to FP16 in each layer while the accumulators are still in FP32. The conversion causes negligible accuracy impact.

Furthermore, *CUDA Core Vector-wise Sparse* is evaluated to justify the VectorSparse pruning method. Finally, we evaluate *Tensor Core Vector-wise Sparse* as our proposal, where vector-wise sparse NNs are running on sparse Tensor Core. Since Tensor Core does not support generic or unified sparse GEMMs, we opt out the options of running generic or unified sparse NNs on Tensor Core. Both CUDA Core Vector-wise Sparse and Tensor Core Vector-wise Sparse have 75% sparsity. Note that even with the relatively high sparsity, the accuracy, or BLEU score, of Vector-wise Sparse is still higher than CUDA Core Unified.

Figure 16 shows the inference performance after the CNNs and RNNs have been trained and pruned (if necessary). All results are normalized to CUDA Core Dense. Unsurprisingly, Tensor Core

[4]AMD uses a different terminology called wavefront. One wavefront consists of 64 threads, which is still a multiple of 16.

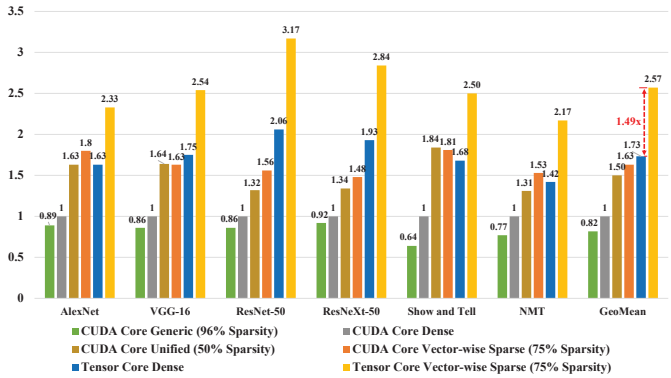


Figure 16: Normalized speedup over CUDA Core based dense NNs on V100 GPU. The vector-wise sparse NNs have 75% sparsity. The generic pruned sparse NNs have 96% sparsity. The unified pruned sparse NNs have 50% sparsity. All the vector-wise sparse NNs have better accuracy than the generic pruned NNs and the unified pruned NNs as shown in Figures 14 and 15.

Dense is faster than CUDA Core Dense since Tensor Core has higher TFLOPs than CUDA Core [53]. On the other hand, CUDA Core Generic is 18% slower than CUDA Core Dense, even if the former has 96% sparsity (i.e., only 4% computation is needed). The slowdown testifies the inefficiency of GPU to support generic sparse NNs, which is the motivation for this work. Alternatively, CUDA Core Unified on average has 1.50× speedup over CUDA Core Dense. The gain mainly comes from the half size of the dense WMMA operations.

On average, Tensor Core Vector-wise Sparse can achieve 2.57× speedup over the baseline. The root cause of the performance gain is two fold. First, with a relaxed spatial constraint, our vector-wise sparse NNs benefit from the high sparsity so that CUDA Core Vector-wise Sparse has 63% performance gain than the baseline. Secondly, with the customized SHMMA instructions and micro-architecture design, these sparse vector-wise NNs can take advantage of the powerful Tensor Core, which contributes an additional 58% performance improvement versus the CUDA Core Vector-wise Sparse. Also note that Tensor Core Vector-wise sparse has 1.49× speedup over Tensor Core Dense.

6.3 Design Overhead Analysis

In the vector-wise sparse mode of the Tensor Core, the hardware design requires a 4× large buffer for Operand *B* to hold 4×16 FP16 numbers and enable the ping-pong buffer. To support the row indexing for Operand *B*, an offset register is added for each octet.

The original size of Operand *B* buffer in each octet is 512b (=4×8×16b), and each Tensor Core has two octets, which makes the buffer size 1Kb. Our vector-wise sparse mode requires a 4× large *B* buffer so that a 4Kb buffer is added to each Tensor Core. As each SM has 8 Tensor Cores, it needs a 4KB buffer. We use CACTI7 [6] to evaluate the timing and area overhead shown in Table 1. A 4KB SRAM takes 0.019mm^2 at 22nm process node. The 0.4ns cycle time is smaller than V100's nominal cycle period (0.65ns at 1530MHz), which does not incur any timing overhead.

Table 1: Design Overhead Analysis via CACTI7 [6]

Process	SRAM Size	Area	Cycle Time
22nm	4KB	0.019mm^2 (0.069×0.275)	0.4ns

As V100 is fabricated in 12nm, we further scale the area down to 0.007mm^2. In addition, a Tensor Core needs two extra registers serving as the offset register for the two octets, so an SM needs 16 extra offset registers to fetch the operands to buffer *B*. Given V100's area is 815mm^2, the overall area overhead is negligible.

6.4 Summary and Discussion

The experimental results show that our VectorSparse pruning method could be used in the same way as the generic pruning. Although the sparsity is lower than the generic sparse NNs, the vector-wise sparse NNs enable modern GPUs to efficiently exploit the benefit from the weight pruning. Compared to the CSR format, our vector-wise encoding eliminates the row indices and allows each row to be split evenly in order to guarantee thread-level parallelism.

The evaluation of the vector-wise pruned networks also shows that the vector-wise sparse NNs with L=16 and 75% sparsity have negligible accuracy drop and promising speedup over their dense counterparts. To further boost the performance of these NNs, we added hardware support to the Tensor Core to enable the vector-wise sparse matrix multiplication. Though we suggest L=16 and K=4 as the current solution, the vector-wise sparse Tensor Core can be easily extended into many variants. As the quick evolution of the neural networks, it is possible that the sparse Tensor Core will become compute-bound again. We would like to leave this as our future work.

7 CONCLUSION

In this work, we observe that the generic sparse neural networks can hardly beat the dense neural networks on modern GPUs because of the highly optimized software and hardware support for dense GEMM. To efficiently exploit the intrinsic redundancy of the neural networks, we propose VectorSparse, a vector-wise pruning method that guarantees the pruned networks to have a balanced workload. The encoding for the vector-wise sparse matrices requires only 1-dim, fixed-length offset indices, instead of the 2-dim, variable-length indices for the generic sparse matrices. With the VectorSparse pruning method, we can prune a neural network to have 75% sparsity with negligible impact on the model accuracy. The good workload balance in the vector-wise sparse matrix multiplication makes it 63% faster than the dense counterparts on the GPU CUDA Cores. To further improve the performance of the vector-wise sparse matrix multiplication, we enabled it to run on the Tensor Core by adding a sparse mode with extended instruction set and hardware support. With negligible area overhead, our sparse Tensor Core can achieve 1.49× speedup over the dense mode of the Tensor Core with comparable model accuracy.

ACKNOWLEDGMENTS

This work was supported in part by NSF 1725447, 1730309, and 1817037.

REFERENCES

[1] [n. d.]. WMT 16 Dataset. https://www.statmt.org/wmt16/translation-task.html

[2] Vahideh Akhlaghi, Amir Yazdanbakhsh, Kambiz Samadi, Rajesh K Gupta, and Hadi Esmaeilzadeh. 2018. Snapea: Predictive early activation for reducing computation in deep convolutional neural networks. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 662–673.

[3] Jorge Albericio, Patrick Judd, Alberto Delmas, Sayeh Sharify, Gerard O'Leary, Roman Genov, and Andreas Moshovos. 2017. Bit-Pragmatic Deep Neural Network Computing. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 175–188.

[4] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. 2016. Cnvlutin: Ineffectual-neuron-free deep neural network computing. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE, 1–13.

[5] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, 163–174.

[6] Rajeev Balasubramonian, Andrew B. Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. *ACM Transactions on Architecture and Code Optimization (TACO)* 14, 2, Article 14 (June 2017), 25 pages. https://doi.org/10.1145/3085572

[7] Denny Britz, Anna Goldie, Minh-Thang Luong, and Quoc Le. 2017. Massive Exploration of Neural Machine Translation Architectures. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*. 1442–1451.

[8] Aydin Buluç, Jeremy T Fineman, Matteo Frigo, John R Gilbert, and Charles E Leiserson. 2009. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*. ACM, 233–244.

[9] Andre Xian Ming Chang, Berin Martini, and Eugenio Culurciello. 2015. Recurrent neural networks hardware implementation on FPGA. *arXiv preprint arXiv:1511.05552* (2015).

[10] Kumar Chellapilla, Sidd Puri, and Patrice Simard. 2006. High performance convolutional neural networks for document processing. In *Tenth International Workshop on Frontiers in Handwriting Recognition*. Suvisoft.

[11] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems (ASPLOS)*. ACM, 269–284.

[12] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. 2014. Dadiannao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 609–622.

[13] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. 2017. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits* 52, 1 (2017), 127–138.

[14] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. 2016. PRIME: A novel processing-in-memory architecture for neural network computation in reram-based main memory. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*. IEEE Press, 27–39.

[15] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)* (2014).

[16] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*. Ieee, 248–255.

[17] Misha Denil, Babak Shakibi, Laurent Dinh, and Nando de Freitas. 2013. Predicting parameters in deep learning. In *Advances in Neural Information Processing Systems (NeurIPS)*. 2148–2156.

[18] Greg Diamos, Shubho Sengupta, Bryan Catanzaro, Mike Chrzanowski, Adam Coates, Erich Elsen, Jesse Engel, Awni Hannun, and Sanjeev Satheesh. 2016. Persistent rnns: Stashing recurrent weights on-chip. In *International Conference on Machine Learning (ICML)*. 2024–2033.

[19] Caiwen Ding, Siyu Liao, Yanzhi Wang, Zhe Li, Ning Liu, Youwei Zhuo, Chao Wang, Xuehai Qian, Yu Bai, Geng Yuan, Xiaolong Ma, Yipeng Zhang, Jian Tang, Qinru Qiu, Xue Lin, and Bo Yuan. 2017. CirCNN: accelerating and compressing deep neural networks using block-circulant weight matrices. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. ACM, 395–408.

[20] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. 2015. ShiDianNao: Shifting vision processing closer to the sensor. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, Vol. 43. ACM, 92–104.

[21] Charles Eckert, Xiaowei Wang, Jingcheng Wang, Arun Subramaniyan, Ravi Iyer, Dennis Sylvester, David Blaauw, and Reetuparna Das. 2018. Neural cache: bit-serial in-cache acceleration of deep neural networks. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE Press, 383–396.

[22] Jonathan Frankle and Michael Carbin. 2019. The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks. In *International Conference on Learning Representations (ICLR)*.

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

[24] Google. [n. d.]. Seq2seq: Neural Machine Translation. https://google.github.io/seq2seq/nmt/

[25] Google. [n. d.]. TensorFlow Model Zoo: Im2txt. https://github.com/tensorflow/models/tree/master/research/im2txt

[26] Google. [n. d.]. TensorFlow Models. https://github.com/tensorflow/models

[27] Scott Gray, Alec Radford, and Diederik P Kingma. 2017. *GPU kernels for block-sparse weights*. Technical Report. Technical report, OpenAI.

[28] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, Huazhong Yang, and William J. Dally. 2017. ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 75–84.

[29] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. 2016. EIE: efficient inference engine on compressed deep neural network. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*. IEEE Press, 243–254.

[30] Song Han, Huizi Mao, and William J Dally. 2016. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *International Conference on Learning Representations (ICLR)* (2016).

[31] 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 (CVPR)*. 770–778.

[32] Kartik Hegde, Jiyong Yu, Rohit Agrawal, Mengjia Yan, Michael Pellauer, and Christopher W Fletcher. 2018. Ucnn: Exploiting computational reuse in deep neural networks via weight repetition. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE Press, 674–687.

[33] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[34] Animesh Jain, Amar Phanishayee, Jason Mars, Lingjia Tang, and Gennady Pekhimenko. 2018. Gist: Efficient data encoding for deep neural network training. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 776–789.

[35] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor M Aamodt, and Andreas Moshovos. 2016. Stripes: Bit-serial deep neural network computing. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–12.

[36] Andrej Karpathy, Justin Johnson, and Li Fei-Fei. 2016. Visualizing and understanding recurrent networks. *International Conference on Learning Representations (ICLR) Workshops* (2016).

[37] Mahmoud Khairy, Jain Akshay, Tor M. Aamodt, and Timothy G. Rogers. 2018. Exploring Modern GPU Memory System Design Challenges through Accurate Modeling. *arXiv preprint arXiv:1810.07269* (2018).

[38] Alex Krizhevsky. 2014. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997* (2014).

[39] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems (NeurIPS)*. 1097–1105.

[40] Yann LeCun. 1998. The MNIST database of handwritten digits. http://yann.lecun.com/exdb/mnist/

[41] Jong Chern Lee, Jihwan Kim, Kyung Whan Kim, Young Jun Ku, Dae Suk Kim, Chunseok Jeong, Tae Sik Yun, Hongjung Kim, Ho Sung Cho, Sangmuk Oh, Hyun Sung Lee, Ki Hun Kwon, Dong Beom Lee, Young Jae Choi, Jaejin Lee, Hyeon Gon Kim, Jun Hyun Chun, Jonghoon Oh, and Seok Hee Lee. 2016. High bandwidth memory (HBM) with TSV technique. In *2016 International SoC Design Conference (ISOCC)*. IEEE, 181–182.

[42] Jonathan Lew, Deval Shah, Suchita Pati, Cattell Shaylin, Mengchi Zhang, Amruth Sandhupatla, Christopher Ng, Negar Goli, Matthew D. Sinclair, Timothy G. Rogers, and Tor M. Aamodt. 2018. Analyzing Machine Learning Workloads Using a Detailed GPU Simulator. *arXiv preprint arXiv:1811.08933* (2018).

[43] Jiajia Li, Jimeng Sun, and Richard Vuduc. 2018. HiCOO: hierarchical storage of sparse tensors. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. IEEE Press, 19.

[44] Youjie Li, Jongse Park, Mohammad Alian, Yifan Yuan, Zheng Qu, Peitian Pan, Ren Wang, Alexander Schwing, Hadi Esmaeilzadeh, and Nam Sung Kim. 2018.

A Network-Centric Hardware/Algorithm Co-Design to Accelerate Distributed Training of Deep Neural Networks. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 175–188.

[45] Robert LiKamWa, Yunhui Hou, Julian Gao, Mia Polansky, and Lin Zhong. 2016. RedEye: analog ConvNet image sensor architecture for continuous mobile vision. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*. IEEE Press, 255–266.

[46] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. 2014. Microsoft coco: Common objects in context. In *European Conference on Computer Vision (ECCV)*. Springer, 740–755.

[47] Daofu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Teman, Xiaobing Feng, Xuehai Zhou, and Yunji Chen. 2015. Pudiannao: A polyvalent machine learning accelerator. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Vol. 43. ACM, 369–381.

[48] Zhenhua Liu, Jizheng Xu, Xiulian Peng, and Ruiqin Xiong. 2018. Frequency-Domain Dynamic Pruning for Convolutional Neural Networks. In *Advances in Neural Information Processing Systems (NeurIPS)*. 1051–1061.

[49] Hang Lu, Xin Wei, Ning Lin, Guihai Yan, and Xiaowei Li. 2018. Tetris: re-architecting convolutional neural network computation for machine learning accelerators. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.

[50] Thang Luong, Hieu Pham, and Christopher D Manning. 2015. Effective Approaches to Attention-based Neural Machine Translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. 1412–1421.

[51] NVIDIA. [n. d.]. CUBLAS: Dense Linear Algebra on GPUs. https://developer.nvidia.com/cublas

[52] NVIDIA. [n. d.]. CUTLASS: Fast Linear Algebra in CUDA C++. https://devblogs.nvidia.com/cutlass-linear-algebra-cuda/

[53] NVIDIA. 2017. V100 GPU architecture. the world's most advanced data center GPU. Version WP-08608-001_v1. 1. *NVIDIA. Aug* (2017), 108.

[54] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric S Chung. 2015. Toward accelerating deep learning at scale using specialized hardware in the datacenter. In *Hot Chips 27 Symposium (HCS), 2015 IEEE*. IEEE, 1–38.

[55] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*. Association for Computational Linguistics, 311–318.

[56] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. 2017. Scnn: An accelerator for compressed-sparse convolutional neural networks. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 27–40.

[57] Eunhyeok Park, Dongyoung Kim, and Sungjoo Yoo. 2018. Energy-efficient neural network accelerator based on outlier-aware low-precision computation. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 688–698.

[58] Md Aamir Raihan, Negar Goli, and Tor Aamodt. 2018. Modeling Deep Learning Accelerator Enabled GPUs. *arXiv preprint arXiv:1811.08309* (2018).

[59] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David Brooks. 2016. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*. IEEE Press, 267–278.

[60] Minsoo Rhu, Mike O'Connor, Niladrish Chatterjee, Jeff Pool, Youngeun Kwon, and Steve Keckler. 2018. Compressing DMA Engine: Leveraging Activation Sparsity for Training Deep Neural Networks. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 331–344.

[61] Marc Riera, Jose-Maria Arnau, and Antonio González. 2018. Computation reuse in DNNs by exploiting input similarity. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE Press, 57–68.

[62] Hasim Sak, Andrew W Senior, and Françoise Beaufays. 2014. Long short-term memory recurrent neural network architectures for large scale acoustic modeling.. In *Interspeech*. 338–342.

[63] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R Stanley Williams, and Vivek Srikumar. 2016. ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*. IEEE Press, 14–26.

[64] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. 2016. From high-level deep neural models to FPGAs. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 1–12.

[65] Hardik Sharma, Jongse Park, Naveen Suda, Liangzhen Lai, Benson Chau, Vikas Chandra, and Hadi Esmaeilzadeh. 2018. Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural networks. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE Press, 764–775.

[66] Seunghee Shin, Guilherme Cox, Mark Oskin, Gabriel H Loh, Yan Solihin, Abhishek Bhattacharjee, and Arkaprava Basu. 2018. Scheduling page table walks for irregular GPU applications. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE Press, 180–192.

[67] Karen Simonyan and Andrew Zisserman. 2015. Very deep convolutional networks for large-scale image recognition. *International Conference on Learning Representations (ICLR)* (2015).

[68] Linghao Song, Jiachen Mao, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. 2019. HyPar: Towards Hybrid Parallelism for Deep Learning Accelerator Array. *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2019).

[69] Mingcong Song, Jiechen Zhao, Yang Hu, Jiaqi Zhang, and Tao Li. 2018. Prediction based execution on deep neural networks. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 752–763.

[70] Xu Sun, Xuancheng Ren, Shuming Ma, and Houfeng Wang. 2017. meProp: Sparsified Back Propagation for Accelerated Deep Learning with Reduced Overfitting. In *International Conference on Machine Learning*. 3299–3308.

[71] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. *Advances in neural information processing systems (NeurIPS)* (2014), 3104–3112.

[72] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2818–2826.

[73] Vincent Vanhoucke, Andrew Senior, and Mark Z Mao. 2011. Improving the speed of neural networks on CPUs. In *Proc. Deep Learning and Unsupervised Feature Learning NeurIPS Workshop*, Vol. 1. Citeseer, 4.

[74] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. 2017. Show and tell: Lessons learned from the 2015 mscoco image captioning challenge. *IEEE Transactions On Pattern Analysis and Machine Intelligence* 39, 4 (2017), 652–663.

[75] Xiaowei Wang, Jiecao Yu, Charles Augustine, Ravi Iyer, and Reetuparna Das. 2019. Bit Prudent In-Cache Acceleration of Deep Convolutional Neural Networks. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 81–93.

[76] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2016. Learning structured sparsity in deep neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*. 2074–2082.

[77] Samuel Williams, Andrew Waterman, and David Patterson. 2009. *Roofline: An insightful visual performance model for floating-point programs and multicore architectures*. Technical Report. Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States).

[78] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, Tommer Leyvand, Hao Lu, Yang Lu, Lin Qiao, Brandon Reagen, Joe Spisak, Fei Sun, Andrew Tulloch, Peter Vajda, Xiaodong Wang, Yanghan Wang, Bram Wasti, Yiming Wu, Ran Xian, Sungjoo Yoo, and Peizhao Zhang. 2019. Machine Learning at Facebook: Understanding Inference at the Edge. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 331–344.

[79] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. 2017. Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1492–1500.

[80] Zhuliang Yao, Shijie Cao, and Wencong Xiao. 2018. Balanced Sparsity for Efficient DNN Inference on GPU. *arXiv preprint arXiv:1811.00206* (2018).

[81] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 161–170.

[82] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. 2016. Cambricon-x: An accelerator for sparse neural networks. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Press, 20.

[83] Xuda Zhou, Zidong Du, Qi Guo, Shaoli Liu, Chengsi Liu, Chao Wang, Xuehai Zhou, Ling Li, Tianshi Chen, and Yunji Chen. 2018. Cambricon-S: Addressing Irregularity in Sparse Neural Networks through A Cooperative Software/Hardware Approach. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 15–28.

[84] Zhuangwei Zhuang, Mingkui Tan, Bohan Zhuang, Jing Liu, Yong Guo, Qingyao Wu, Junzhou Huang, and Jinhui Zhu. 2018. Discrimination-aware channel pruning for deep neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*. 883–894.