Compressing DMA Engine: Leveraging Activation Sparsity for Training Deep Neural Networks

Minsoo Rhu*, Mike O'Connor†, Niladrish Chatterjee†, Jeff Pool†, Youngeun Kwon*, and Stephen W. Keckler†
*POSTECH, †NVIDIA

***Abstract*—Popular deep learning frameworks require users to fine-tune their memory usage so that the training data of a deep neural network (DNN) fits within the GPU physical memory. Prior work tries to address this restriction by virtualizing the memory usage of DNNs, enabling both CPU and GPU memory to be utilized for memory allocations. Despite its merits, virtualizing memory can incur significant performance overheads when the time needed to copy data back and forth from CPU memory is higher than the latency to perform DNN computations. We introduce a high-performance virtualization strategy based on a "compressing DMA engine" (cDMA) that drastically reduces the size of the data structures that are targeted for CPU-side allocations. The cDMA engine offers an average $2.6\times$ (maximum $13.8\times$) compression ratio by exploiting the sparsity inherent in offloaded data, improving the performance of virtualized DNNs by an average 53% (maximum 79%) when evaluated on an NVIDIA Titan Xp.**

I. Introduction

Deep neural networks (DNNs) are now the driving technology for numerous application domains, such as computer vision, speech recognition, and natural language processing. To facilitate the design and study of DNNs, a large number of machine learning (ML) frameworks [1], [2], [3], [4], [5], [6], [7], [8] have been developed in recent years. Most of these frameworks have strong backend support for GPUs. Thanks to their high compute power and memory bandwidth [9], GPUs can train DNNs orders of magnitude faster than CPUs. One of the key limitations of these frameworks however is that the limited physical memory capacity of the GPUs constrains the algorithm (e.g., the DNN layer width and depth) that can be trained.

To overcome the GPU memory capacity bottleneck of DNN training, prior work proposed to *virtualize* the memory usage of DNNs (vDNN) so that ML researchers can train larger and deeper neural networks beyond what is afforded by the physical limits of GPU memory [10]. By copying GPU-side memory allocations in and out of CPU memory via the PCIe link, vDNN exposes both CPU and GPU memory concurrently for memory allocations, which improves user productivity and flexibility in studying DNN algorithms (detailed in Section III). However, in certain situations, this memory-scalability comes at the cost of performance overheads resulting from the movement of data across the PCIe link. When the time needed to copy data back and forth through PCIe is smaller than the time the GPU spends computing the DNN forward and backward propagation operations, vDNN does not affect performance. However, for networks whose memory copying operation is bottlenecked by the data transfer bandwidth of PCIe, vDNN can incur significant overheads with an average 51% performance loss (worst case 63%, Section III). The trend in deep learning is to employ larger and deeper networks that lead to large memory footprints that oversubscribe GPU memory [11], [12], [13]. Therefore, ensuring the performance scalability of the virtualization features offered by vDNN is vital for the continued success of deep learning training on GPUs.

Our goal is to develop a virtualization solution for DNN training that satisfies the dual requirements of memory-scalability and high performance. To this end, we present a *compressing DMA engine* (cDMA), a general purpose DMA architecture for GPUs that alleviates PCIe bottlenecks by reducing the size of the data structures copied in and out of GPU memory. Our proposal minimizes the design overhead by extending the (de)compression units already employed in GPU memory controllers as follows. First, cDMA requests the memory controller to fetch data from the GPU memory at a high enough rate (i.e., effective PCIe bandwidth $\times$ compression ratio) so that the compressed data can be generated at a throughput commensurate with the PCIe bandwidth. The cDMA copy-engine then initiates an on-the-fly compression operation on that data, streaming out the final compressed data to the CPU memory over PCIe. The key insight derived from our analysis is that the data (specifically the *activation* maps of each DNN layer) that are copied across PCIe contain significant *sparsity* (i.e., fraction of activations that are *zero*-valued) and are highly compressible. Such sparsity of activations primarily comes from the ReLU [14] layers that are extensively used in DNNs. We demonstrate sparsity as well as compressibility of the activation maps through a data-driven application characterization study. While recent prior work [15], [16], [17] explored network sparsity in the context of DNN *inference*, our work is the first to provide a detailed analysis of DNN sparsity during *training* and how it can be used to overcome the data transfer bandwidth bottlenecks of virtualized DNNs.

II. Background

A. Deep Neural Networks

Today's most popular deep neural networks can broadly be categorized as convolutional neural networks (CNNs) for image recognition, or recurrent neural networks (RNNs) for

video captioning, speech recognition, and natural language processing. Both CNNs and RNNs are designed using a combination of multiple types of layers, most notably the convolutional layers (CONV), activation layers (ACTV), pooling layers (POOL), and fully-connected layers (FC). A deep neural network is divided into two functional modules: (a) the *feature extraction layers* that learn to extract meaningful features out of an input, and (b) the *classification layers* that use the extracted features to analyze and classify the input to a pre-designated output category. "Deep learning" refers to recent research trends where a neural network is designed using a large number of feature extraction layers to learn a deep hierarchy of features. The feature extraction layers of a CNN are generally composed of CONV/ACTV/POOL layers whereas the classification layers are designed using FC/ACTV layers.

Convolutional layers. A convolutional layer contains a set of filters to identify meaningful features in the input data. For visual data such as images, 2-dimensional filters (or 3-dimensional when accounting for the multiple input channels within the input image) are employed which slide over the input of a layer to perform the convolution operation.

Activation layers. An activation layer applies an element-wise activation function (e.g., `sigmoid`, `tanh`, and ReLU [14]) to the input feature maps. The ReLU function in particular is known to provide state-of-the-art performance for CNNs, which allows positive input values to pass through while thresholding all negative input values to *zero*.

Pooling layers. Pooling layers perform a spatial-downsampling operation on the input data, resulting in an output volume that is of smaller size. Downsampling is done via applying an average or max operation over a region of input and reducing it into a single element.

Fully-connected layers. Fully-connected layers (or classifier layers) constitute the final layers of the network. Popular choices include multi-layer perceptrons, although other types of FC layers are based on multi-nomial logistic regression. Key functionality of this layer is to find the correlation between the extracted features and the output category.

B. Training versus Inference

DNNs require *training* to be deployed for an *inference*. Training involves learning and updating the *weights* of the network, which is typically done using the backpropagation algorithm [18]. Figure 1 shows the three-step process for each training pass: (1) forward propagation, (2) deriving the magnitude of error between the network's inference and the ground truth, and (3) propagating the inference error backwards across the network using backward propagation.

Forward propagation. Forward propagation is a serialized, layer-wise computation process that is performed from the first (input) layer to the last (output) layer in a sequential manner (from left to right in Figure 1). Each layer applies a mathematical operation (such as a convolution

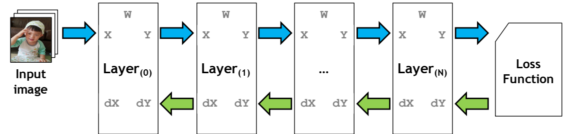


Figure 1: Training a DNN.

operation for CONV layers) to the input *activation maps*[1] (`X`) and generates/stores the results of this operation as output activation maps (`Y`).

Calculating the loss value. Forward propagation produces a classification of the input image which must be compared to the ground truth. The *loss function* is defined to calculate the magnitude of this error between classification and ground truth, deriving the *gradients* of the loss function with respect to the final layer's output. In general, the loss value drops quickly at the beginning of training, and then drops more slowly as the network becomes fully trained.

Backward propagation. Backward propagation is performed in the inverse direction of forward propagation, from the last layer to the first layer (from right to left in Figure 1), again in a layer-wise sequential fashion. During this phase, the incoming gradients (`dY`) can conceptually be thought of as the inputs to this layer which generate output gradients (`dX`) to be sent to the previous layer. Using these gradients, each layer adjusts its own layer's weights (`W`), if any (e.g., CONV and FC layers), so that for the next training pass, the overall loss value is incrementally reduced.

With sufficient training examples, which may number in the millions, the network becomes incrementally better at the task it is tasked to learn. A detailed discussion of the backpropagation algorithm and how contemporary GPUs implement each layer's DNN computations and memory allocations can be found in [19], [10].

C. Data Layout for Activation Maps

For training CNNs, the (input/output) activation maps are organized into a 4-dimensional array; the number of images batched together (`N`), the number of feature map channels per image (`C`), and the height (`H`) and width (`W`) of each image. Because the way this 4-dimensional array is arranged in memory address space has a significant effect on data locality, different ML frameworks optimize the layout of their activation maps differently. For instance, the CNN backend library for Caffe [1] is optimized for `NCHW` (i.e., the `N` and `W` in the outermost and innermost dimension of the array, respectively) whereas cuDNN [9] provides support for both `NCHW` and `NHWC`. Neon [5] and `cuda-convnet` [7] on the other hand is optimized for `CHWN`. We elaborate on the sensitivity of our proposal on activation data layout in Section VII-A.

[1]Following prior literature, we use the terms input/output feature maps and input/output activation maps interchangeably.

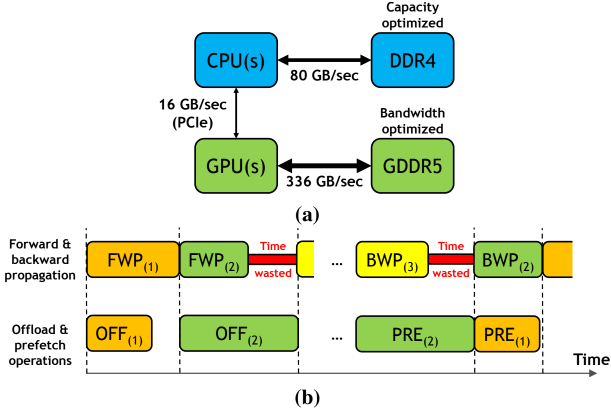


Figure 2: (a) PCIe attached CPUs and GPUs, and (b) vDNN memory management using (GPU-to-CPU) offload and (CPU-to-GPU) prefetch operations. $\text{OFF}_{(n)}$ and $\text{PRE}_{(n)}$ corresponds to the offload and prefetch of $\text{layer}_{(n)}$'s activation maps, respectively.

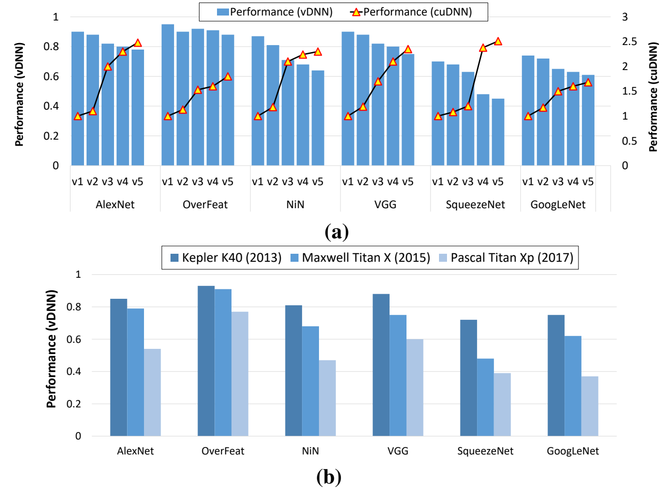


Figure 3: (a) Speedups offered by versions of cuDNN (right-axis), and the performance degradations incurred by vDNN (left-axis). (b) Performance of vDNN (using cuDNN, v5) across different generation of GPUs, normalized to an oracular GPU (Section VII).

D. Related Work

DNNs are generally over-parameterized, stemming from a significant redundancy in the way the parameters are used to represent the approximating model. As a result, a series of proposals have aimed to reduce DNN memory usage by alleviating network redundancy. Network pruning strategies in particular have been studied extensively by prior literature [20], [21], [22]. Pruning helps reduce the memory allocated for model weights by removing redundant network connections that satisfy a given pruning criteria. These proposals provide limited opportunity for saving memory usage, as weights only account for a small fraction of overall memory allocations needed for training DNNs. Hauswald et al. presented DjiNN and Tonic [23], an open source DNN service and applications suite for GPU server designs in warehouse scale computers. A series of accelerator designs have also been proposed for CNNs recently [24], [25], [26], [27], [28], [29], [30], [31], [15], [16], [17]. The scope of these prior proposals is in the domain of DNN *inference* while our work focuses on DNN *training*. More importantly, none of these prior works address the communication bottleneck that arise due to DNN memory virtualization.

III. Motivation

Several techniques have been proposed for supporting virtual memory on GPUs. Pichai et al. [32] and Power et al. [33] proposed TLB designs that leverage the unique memory access patterns of GPUs for optimizing the throughput of memory address translations. Zheng et al. [34] studied architectural solutions for closing the performance gap between page-migration based virtual memory and software-directed direct-memory-access (DMA) copy operations. Nonetheless, the performance overheads of these fine-grained, page-based virtual memory solutions are high because of the low throughput and high latency of page-migration on discrete GPU systems. Rhu et al. [10] therefore proposed an application-level virtual memory management solution specifically tailored for DNNs (vDNN). Figure 2 provides a high-level overview of a state-of-the-art DNN training platform containing CPUs and GPUs, and how the vDNN memory manager orchestrates the data copy operations across the CPU and GPU memories. Figure 2(a) illustrates a discrete GPU card (Maxwell Titan X) with 336 GB/sec of GPU DRAM bandwidth, connected to a host CPU via a PCIe channel, which provides a maximum data transfer bandwidth of 16 GB/sec for PCIe gen3.

Figure 2(b) shows how vDNN virtualizes memory by proactively offloading the inter-layer *activation maps* out to CPU memory during forward propagation and later prefetching them back into the GPU, just before they are reused during backward propagation. For training DNNs, these activation maps occupy more than 90% of the GPU-side memory allocations [10]. Thus vDNN offers significant reduction in the average GPU memory usage by offloading activations to the CPU. vDNN also provides much higher PCIe bandwidth utilization and performance than page-migration based virtual memory (i.e., 12.8GB/sec [10] versus 200 MB/sec [34]) as the data movements are orchestrated by GPU's DMA copy-engine. Therefore, major ML frameworks, such as TensorFlow and Chainer have been employing a vDNN-style solution as means to enhance both system memory scalability and the productivity of algorithm developers [35], [36], highlighting the importance of DNN memory virtualization. However, vDNN can still incur non-negligible performance overheads when the time needed to move data in and out of the CPU memory takes longer than computing DNN's backpropagation algorithm.

Figure 3 illustrates the extent of this bottleneck on the performance of DNNs. The right-axis on Figure 3(a) shows the performance improvements offered by successive ver-

Table I: GPU math throughput (TFLOPS) and memory bandwidth (GB/sec) scaling trends.

GPU generation	TFLOPS	GB/sec
Kepler K40 (2013)	5.0	288
Maxwell M40 (2015)	6.8	288
Pascal P100 (2016)	10.6	720
Volta V100 (2017)	15.0	900

sions of NVIDIA's deep learning library cuDNN [9], which effectively reduces the time spent computing each CONV layer. The more recent version of cuDNN (v5) offers an average 2.2× the performance of the first version (v1) released in 2014 across a range of different DNNs. At the same time, the maximum math throughput and memory bandwidth provided by state-of-the-art GPUs have scaled up rapidly over four years, by a factor of 3× and 3.2× in terms of math throughput[2] and memory bandwidth, respectively (Table I). Consequently, the overall latency incurred to compute a DNN algorithm is proportionally scaling down as faster GPUs and high-performance backend GPU libraries are introduced to the market. Unfortunately, the data transfer bandwidth offered by the state-of-the-art PCIe link (gen3) has remained unchanged at 16 GB/sec. This divergence is the key reason behind the steadily increasing performance overheads of `vDNN` on successive generations of GPUs and its backend DNN libraries (Figure 3). Note that recent high-end HPC systems for DNN acceleration are employing multiple GPUs per server node (e.g., up to 8 GPUs on a dual-socket motherboard), which reduces the the CPU-GPU communication bandwidth allocated per each GPU. For instance, 4 GPUs connected to a single CPU socket are given 32 GB/sec of communication bandwidth to the CPU memory over two PCIe switches, leaving only (32/4) = 8 GB/sec of CPU-GPU bandwidth available per GPU. In general, the growing performance gap between GPU performance and CPU-GPU communication bandwidth renders an urgent need for hardware/software solutions that can remedy the aforementioned communication bottleneck issue. In Section VIII, we discuss the implication of NVIDIA's high-bandwidth communication soluton NVLINK [38] and next generation PCIe on our `cDMA` proposal.

Our compressing DMA engine is based on the key observation that the activation maps, which account for the majority of GPU-side memory allocations for training deep networks [10], are amenable for compression, which will drastically alleviate the PCIe bottleneck of virtualized DNNs. A significant fraction of each layer's activations are *zero*-valued, meaning these data structures are *sparse* and are highly compressible. As noted by multiple prior works [39], [15], such sparsity of activations are originated by the extensive use of ReLU [14] layers that follow (almost) every single layer for feature extraction. We first provide a data-

[2]The math throughput of Volta V100 for mixed precision FP16/FP32 training can increase up to 125 TFLOPS when using its TensorCore [37].

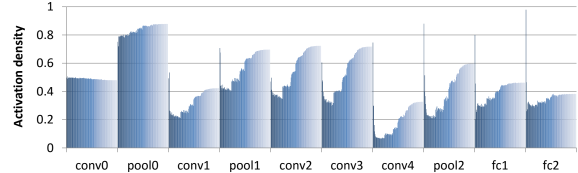


Figure 4: Average activation density of each layer in AlexNet over time during training (going from dark to light blue colored bars, per layer). Activation density is sampled at every 2K iterations of training, and a total of 226K iterations were spent to reach the fully trained model (53.1%/75.1% top-1/top-5 accuracy).

driven, in-depth DNN characterization study in Section IV that motivates our work, followed by our compressing DMA architecture in Section V. As the effectiveness of our proposition (i.e., compression) is highly correlated with the actual data that are input into the neural network, this paper primarily focuses on convolutional neural networks (CNNs) owing to their publicly available, realistic datasets (e.g., ImageNet [40]) for computer vision tasks. Nonetheless, we believe our proposal is equally applicable for some popular recurrent neural networks that extensively employ sparsity-inducing ReLU layers, including the GEMV-based (general-matrix-vector-multiplication) RNNs employed by Baidu for speech recognition [41] and language translation [42] services. At present, we cannot study these RNN applications applications as there are no publicly available training datasets. `cDMA` is less well-suited for RNNs based on LSTMs [43] or GRUs [44], as they employ `sigmoid` and `tanh` activation functions rather than ReLUs.

IV. Sparsity of DNN Activations

The focus of this paper is on DNN training, which involves learning and updating the weights of a neural network using the backpropagation algorithm. As discussed in Section II-B, the values in the output activation maps (`Y`) are derived as a function of both the input activation maps (`X`) and the layer weights (`W`). The sparsity of each layer's output activations will therefore change as the training progresses, during which not only will the layer be continuously fed with new input activation maps, but the weights for the same layer will undergo changes as a result of backpropagation. For our compressing DMA engine to be effective, it is crucial that the activation sparsity, and accordingly its compressibility, remains consistently high throughout the entire training process. This section analyzes the effect of training on activation sparsity by using AlexNet [14] as a running example. We detail our training methodology in Section VI.

A. Case Study: Activation Sparsity in AlexNet

Figure 4 shows the change in each layer's average output activation density over time, as the network is trained for better image classification. We define the per-layer average output activation density ($\texttt{AVG}_{density}$) as the number of non-zero output activations divided by the total number of output activations, which is measured across the minibatch of the

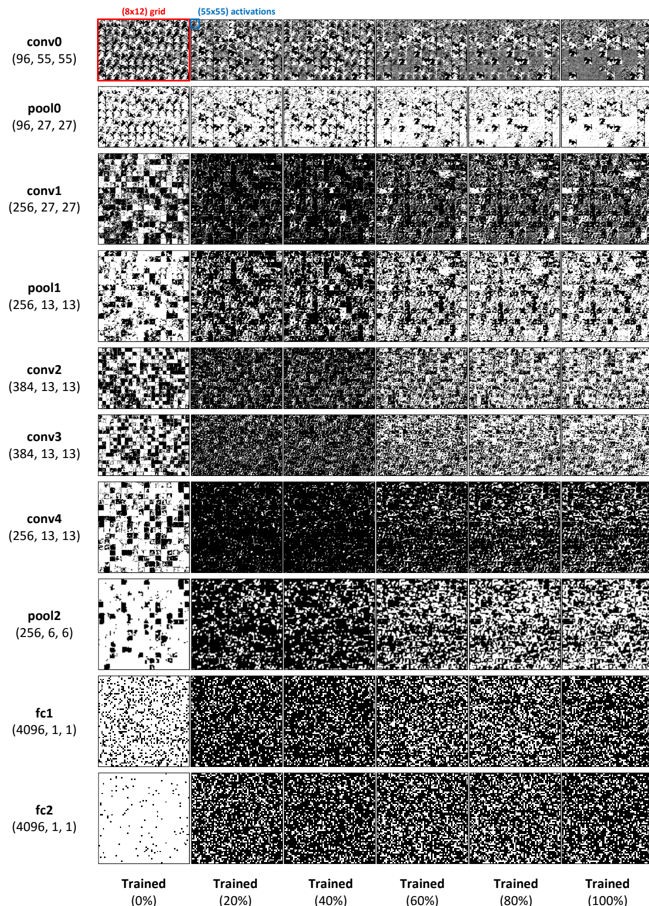


Figure 5: Change in AlexNet's activation sparsity as the network is trained. "Trained (0%)" corresponds to point in time when the weights of AlexNet were initialized at the onset of training, whereas "Trained (100%)" designates a fully-trained AlexNet. The three numbers (`C, H, W`) below the names of each layer represent the number of channels, the height, and the width of the output activation maps. A zero value in an activation map is represented as a black pixel and white otherwise. The `RGB` image of the boy shown in Figure 1 was used to derive these results.

same 50 images. Accordingly, average activation sparsity is equal to $(1-\text{AVG}_{density})$. Figure 5 shows a visualization of sparsity across time (x-axis), layer (y-axis), and spatially within each activation map. For brevity, we only show the layers that are immediately followed by ReLU layers and would exhibit sparsity. For instance, the output activation maps of the first convolutional layer of AlexNet (`conv0`) contain 96 channels, each of which can conceptually be thought of as a 2-dimensional, (55×55) array of activations per channel. The 96 channels are arranged as a (8×12) grid, with each grid corresponding to a single channel with (55×55) activations (i.e., the top-leftmost image in Figure 5). Each of the activations are displayed as black and white pixels depending on whether they are zero-valued (*sparse*, black) or not (*dense*, white).

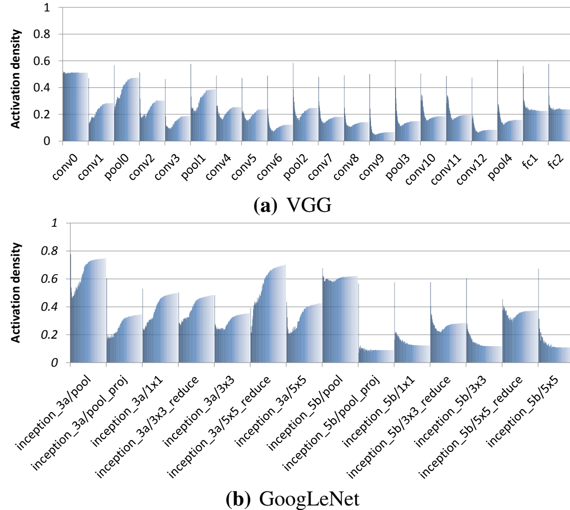
Based on this analysis, we can draw the following key observations. First, the first convolutional layer (`conv0`), regardless of the iterations of training it has gone through, is neither sparse nor dense, always falling within $\pm 2\%$ of 50% average activation sparsity (or density). Second, pooling layers always increase activation density, i.e., activation maps always get brighter after going through the pooling layers. This result is expected as pooling layers either pick the highest value (when we use max pooling) or derive the average value (average pooling) within the pooling window. Thus, a pooling layer is likely to generate a dense output unless all the input activations within the pooling window are all zero-valued. Third, with the exception of the first convolutional layer, the change in average activation density exhibits a `U`-shaped curve during training; the number of non-zero activations rapidly decreases during the initial training periods but gradually increases back during the latter stages of training as the model accuracy improves. This `U`-shaped curve is also reflected in Figure 5 where the activation maps quickly turn extremely dark during the first 40% of the training period but gradually becoming lighter as the layer enters the mid-to-end stages of the training process. Finally, layers located towards the end of the network are generally more sparse than the earlier layers with the fully-connected layers exhibiting much higher sparsity than the convolutional layers. Overall, AlexNet exhibits an average 49.4% activation sparsity across the entire network when accounting for the size of each of the layer's activations (e.g., the sizes of the activations in the earlier layers are generally larger than those located at later layers). Thus, a compression algorithm that can perfectly compress out all the zeros can reduce the activation size by about half.

B. *Effects of Training on Sparsity*

In addition to AlexNet, we examined the sparsity of activations for larger, deeper, and more recent CNNs, including OverFeat [45], NiN [46], VGG [11], SqueezeNet [47], and GoogLeNet [12]. Figure 6 shows that the per-layer sparsity measurements of these networks are very similar in nature to AlexNet, reinforcing the observations listed above [3]. In the six networks that we study in this paper, we observe an average 62% network-wide activation sparsity (maximum of 93%) across the entire training periods. Figure 7 shows the behavior of AlexNet as a function of training time, including the loss value computed by the loss function at the end of the network and the activation densities of the four convolutional layers. The graph demonstrates four key observations about the effect of training on per-layer activation density, as described below.

- When training begins, activation density drops dramatically for all of the layers. This drop correlates with the dramatic improvement in the loss function. We believe that this drop in density is due to the network

[3] We omit the results for OverFeat, NiN, and SqueezeNet due to space constraints. All these networks exhibit the `U`-shaped curve similar to AlexNet, VGG, and GoogLeNet.



(a) VGG

(b) GoogLeNet

Figure 6: Effect of training on layer activation density.

quickly adapting from its randomly initialized weights to learning what features of the input data are not important for classification.

- In the second regime, the activation density increases, first somewhat rapidly and then more slowly. We believe that this increase stems from two factors. First, during the middle stages of training, the network weights are iteratively optimized to extract features that it has previously neglected, gradually improving accuracy. Second, a common heuristic in training DNNs is to reduce the learning rate, typically multiplying the original learning rate by 0.5 or 0.1, when the validation accuracy plateaus with the current value [14], [48].
- During the final fine-tuning stages of training, the weights are already close to their respective optimal points so the effect on the overall average activation sparsity is minimal.
- In general, convolution layers later in the network are sparser than earlier ones. Deep networks are known to build up a rich set of hierarchical feature extractors across the network layers. For instance, Zeiler and Fergus [48] observed that the first few layers of a CNN are generally trained to respond to corners, edges, and basic colors that commonly exist across all images in a *class-invariant* fashion. However, deeper layers are used to detect *class-specific* high-level abstractions such as common textures (e.g., mesh patterns), texts, faces of a dog, etc. We hypothesize that layers located deep in the network are trained to respond to class-specific features and have activations that only respond to a subset of classes, leading to high sparsity. In contrast, because layers positioned early in a network respond in a class-invariant manner (e.g., activations will respond to all the classes that have red-colored regions of an image), they are likely to exhibit less sparsity.

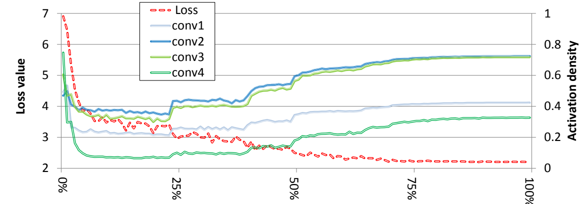


Figure 7: Change in loss value (left axis) and per-layer activation density (right axis) as the network is trained (x-axis).

V. Compressing DMA Engine

To address the performance bottlenecks associated with moving activation maps between the GPU and CPU memory, we exploit the sparsity of activation maps to compress them before transferring them across the PCIe. Our proposal is somewhat similar to compressing pages prior to moving them to backing storage in a virtual memory system [49]. Our compressing DMA engine (`cDMA`) requires choosing an efficient and effective compression algorithm, and a mechanism to employ this algorithm to compress activation maps as they are transferred between GPU and CPU memory.

A. Compression Algorithm

To compress activation maps, we need a simple compression algorithm which can sustain compression from and decompression to GPU memory at rates of 100's of GB/sec while saturating PCIe bandwidth with compressed data.

Run-length encoding compression. Early observations of the sparse activation maps demonstrated a clustering of zero-valued activations (Figure 5). As a result, we investigate a simple scheme using *run-length encoding* (`RLE`) [50] to compress the activation maps. Run-length encoding is simple to implement, and is well suited for high-bandwidth compression. Despite its simple design, the effectiveness of `RLE` highly depends on the sparsity patterns exhibited in the activation maps as compression is only effective for consecutive zeros or non-zeros. As a result, `RLE` does not offer good compression ratios across all of the activation layouts (detailed in Section VII-A).

Zero-value compression. As demonstrated in Section IV, approximately 50% to 90% of the activations are zero-valued. We therefore investigate a simple yet highly effective approach based on *Frequent-value compression* [51] that is used to compress out the zero-valued elements.

Figure 8 provides a high-level overview of our *zero-value compression* (`ZVC`) algorithm which assumes a compression window sized as 32 consecutive elements. For every 32 activation values, a 32-bit mask is generated with a '0' in a given bit position indicating the corresponding value is zero and a '1' indicating a non-zero value. After this 32-bit mask is generated, the non-zero elements are appended. Thus, 32 consecutive zero valued activations can be compressed down

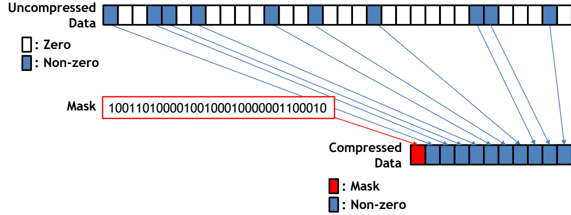


Figure 8: Zero-value compression.

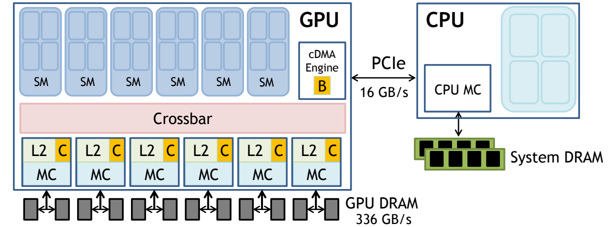


Figure 9: `cDMA` architecture overview. Box "B" indicates the location of the `cDMA` buffers whereas boxes labeled "C" indicate the location of (de)compression units.

to a single 32-bit all-zero mask (32× compression ratio). 32-consecutive non-zero elements will result in a 32-bit all-one mask, followed by the 32 non-zero activation values (a 3.1% metadata overhead, 1-bit per each single activation value). If 60% of the total activations are zero-valued, we would expect an overall compression ratio of 2.5×. Compared to `RLE`, the key advantage of `ZVC` is that it can compress out zeros equally well regardless of how the zero values are distributed in the data. Unlike `RLE`, `ZVC` works robustly across all the data layouts of the activation maps. `ZVC` can be implemented in high-bandwidth compression hardware in a straightforward manner. The hardware implementation complexity is dominated by the MUXes to gather/scatter the non-zero data elements to/from the compressed representation and the pop-count/prefix-sum operation on the mask to determine the offset to the next mask in the compressed stream. We detail the ZVC DMA engine microarchitecture in Section V-B and the area overhead in Section V-C.

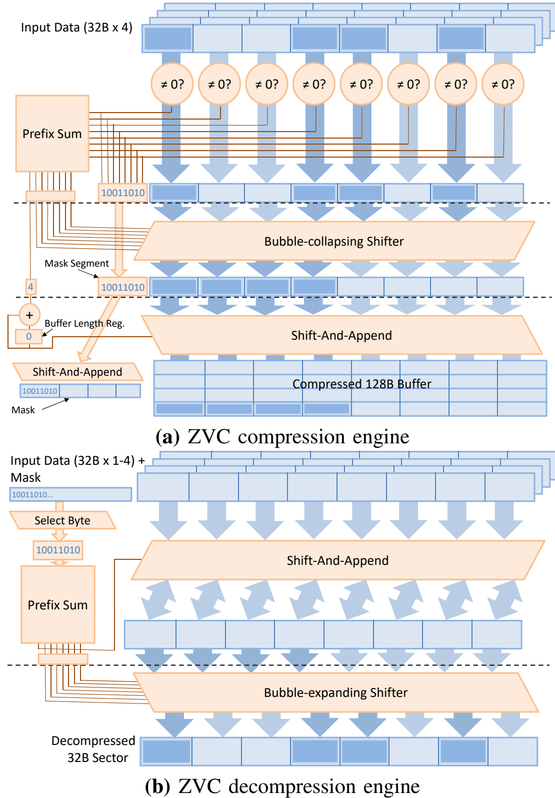
Zlib compression. The compression scheme used in the popular *gzip* utility is based on the DEFLATE algorithm [52]. This algorithm has very good performance across a range of data, but designing a high-throughput hardware to perform the compression is quite complex. Dedicated FPGA and ASIC solutions [53], [54] are capable of reaching approximately 2.5 GB/sec of throughput. While processing multiple streams in parallel with multiple compression engines can improve throughput, the hardware costs escalate linearly with increased bandwidth. Supporting this compression algorithm is impractical when the system must be capable of compressing 100's of GB/sec of data. Nonetheless, we include the results using this approach to demonstrate the upper-bound of the opportunity we may be leaving on the table by not compressing non-zero data and focusing solely on zero-value compression.

B. Compressing DMA Engine Architecture

Architecture overview. Figure 9 provides an overview of the `cDMA` architecture embedded into the memory system of a GPU. The additional hardware includes compression/decompression units adjacent to the GPU memory controllers (boxes labeled "C") and a little more data buffering storage (box labeled "B") in the existing DMA engine at the PCIe interface. GPUs already perform compression operations within the memory controllers today [55], [56], [57], but the compression operations of our `cDMA` are somewhat backwards compared to existing systems. The existing compression hardware in the GPU memory controllers compress data on the way *into* the GPU DRAM and decompress on the way out to the L2 and GPU cores (streaming multiprocessors, denoted as SMs in Figure 9) to save GPU DRAM bandwidth. Our `cDMA` architecture compresses the data coming *out* of the GPU DRAM on their way to the DMA unit, and decompresses data in the other direction. We provide a qualitative discussion on how the operation of `cDMA` can be designed to work in a conventional way (i.e., compression taking place on its way to the DRAM to save bandwidth) in Section VIII.

An alternative implementation of `cDMA` would directly add the (de)compression units inside the existing DMA unit so that it compresses the data just before sending it over PCIe and decompresses the data when received over PCIe from the CPU. One key concern with this design is its effect on the bandwidth requirements of the GPU on-chip crossbar which connects the memory controllers to the SMs and DMA engine. The key design objective of our `cDMA` engine is to be able to saturate the PCIe bandwidth to the CPU with compressed data. Accordingly, the GPU crossbar bandwidth that routes uncompressed data from the L2 to the DMA engine must be high enough to generate compressed activation maps at a throughput commensurate to the PCIe link bandwidth. As detailed in Section VII-A, the maximum per-layer compression ratio observed is 13.8×. Assuming PCIe (gen3) with maximum 16 GB/sec data transfer bandwidth, up to (16×13.8)=220.8 GB/sec crossbar bandwidth must be provisioned to fully exploit the potential of sparse compression. Since the baseline DMA engine need only serve the 16 GB/sec of PCIe bandwidth, providing over 200 GB/sec of crossbar bandwidth to the DMA engine for the purposes of data offloading is unattractive. Our `cDMA` design instead augments the GPU memory controllers with the (de)compression units to compress the data read from the DRAM *before* sending it over the crossbar to the DMA. Such a design reduces the bandwidth demand on the crossbar during a compressed DMA operation back to levels similar to the baseline non-compressing DMA engine.

(De)compression microarchitecture. Figure 10(a) shows the compression engine micrarchitecture implementing the ZVC algorithm. This logic operates each cycle on a 32B (8



(a) ZVC compression engine

(b) ZVC decompression engine

Figure 10: ZVC engine microarchitecture.

word) element which corresponds both to the internal datapath width in the memory controller and to one DRAM burst. On one cycle, these eight words are compared in parallel to zero, forming the mask bits for these 8 words. A prefix sum operation, requiring eleven 3-bit adders, is performed on the mask bits to determine the number of zero-valued words in front of a given word. In the next pipeline cycle, the non-zero data elements are shifted to the correct resulting offset using the result of the prefix sum operation to drive the mux-selects. The final cycle in the pipeline steers the resulting zero-compressed data to append it to previous compressed data in the overall 128B (cache-line sized) window on which we perform ZVC compression. The 8-bit mask is also appended to the mask from previous cycles. The total latency to compress a 128B line is six cycles, four 32B sectors moving through a three-stage pipeline.

Figure 10(b) shows the ZVC decompression microarchitecture which expands one compressed item into 128B. The decompression logic also operates on 32B at a time, producing 32B of decompressed data each cycle. In the first cycle of the pipeline, an 8-bit segment of the mask is considered. A pop-count (number of ones) of this 8-bit segment of the mask determines the number of words that will be used in a given 32B sector. In parallel, the 8-bit segment of the mask is evaluated to determine the correct mux-selects (also a small prefix-sum operation). In the next cycle, the 32B decompressed value is formed by muxing payload values (or a zero) into the correct locations. The pipeline requires only two additional cycles of latency to decompress a 128B line, because decompression can start as soon as the first part of the data arrives from the crossbar.

C. *Design Overheads*

(De)compression units. While we expect that the existing GPU compression units can be leveraged for `cDMA` to minimize design overheads, we assume that the `cDMA` (de)compression hardware supplements existing hardware for a conservative area estimate. Nonetheless, our `cDMA` unit can allow existing DRAM compression schemes optimized to minimize DRAM bandwidth to also take place. We use the FreePDK [58] 45 nm process design kit and scaled the resulting area using a conservative cell size reduction of $0.46\times$ from 45 nm to 28 nm. Assuming a 50% cell area utilization due to the design being dominated by wires and MUXes, the six (de)compression units are estimated to occupy $0.31\,\text{mm}^2$.

Buffer sizing. The DMA engine must also maintain a buffer large enough to hold the bandwidth-delay product of the memory sourcing the data to prevent bubbles in the output stream. As we detail in Section VII-B, DNN computations are highly compute-bound so the required average memory bandwidth is measured at less than 100 GB/sec, leaving more than (336 – 100)=236 GB/sec for `cDMA` to fetch data without affecting performance[4]. Our experiments show that provisioning 200 GB/sec of bandwidth (i.e., the maximum per-layer compression ratio of $13.8\times$ multiplied by the max PCIe bandwidth of 16 GB/sec) for `cDMA` reaps 99% of the performance benefits of sparse compression. As a result, based on a 350 ns latency from the time the DMA engine requests data from GPU memory to the time it arrives at the DMA engine [59] and the 200 GB/sec compression read bandwidth, the DMA engine needs a 70KB (200 GB/sec$\times$350 ns) buffer, shown as block "B" in Figure 9. It may seem counter-intuitive that `cDMA` would need this large a buffer, since it is receiving only compressed requests at an overall rate of 16 GB/sec from the crossbar. The reason why the buffer must be overprovisioned is because the `cDMA` engine does not know *a priori* which responses will be compressed or not. Since it must launch sufficient requests to keep the PCIe bus busy even with highly-compressed data, a large number of requests will be in-flight. If these requests are *not* compressed, the buffer is required to hold the large amount of data returned until it can be streamed out over the PCIe interface. This buffer size

[4]For a conservative estimation, we discuss the overall design of `cDMA` architecture and its effect on DRAM bandwidth utilization in the context of Maxwell Titan X which has 40% lower memory bandwidth than the more recent Pascal Titan Xp. We quantitatively evaluate the effect of `cDMA` on both Maxwell and Pascal GPUs in Section VII.

is not a significant source of area (approximately $0.21\,\text{mm}^2$ in $28\,\text{nm}$ according to CACTI 5.3 [60]). Compared to the $600\,\text{mm}^2$ of a NVIDIA Titan X chip, the added overheads of (de)compression units and DMA buffers are negligible.

D. cDMA Policy

Performance-optimal `cDMA`. As noted above, we provisioned both the `cDMA` engine and the memory subsystem to maximally utilize and saturate the PCIe channel with compressed activations. In other words, the baseline `cDMA` policy is to read uncompressed activations at a high enough rate (commensurate with the per-layer compression ratio $\times$ PCIe bandwidth) from DRAM so that the compressed data can be transferred to the CPU via PCIe at maximum throughput.

Bandwidth-aware `cDMA`. For GPUs that are not equipped with high enough DRAM bandwidth, the near 200 GB/sec of worst-case memory bandwidth usage can be a performance limiting factor (e.g., GTX 1060 with only 192 GB/sec of DRAM bandwidth). We therefore propose a *bandwidth-aware* `cDMA` policy that throttles the DRAM bandwidth usage of `cDMA` at a statically fixed level when the compression ratio for a given layer's activation exceeds this predefined amount. As the average network-wide compression ratio is shown to be around $2.6\times$, we empirically set the bandwidth-aware `cDMA` policy's throttling threshold at 48 GB/sec (i.e., $3.0\times$ compression ratio times 16 GB/sec). For layers with a compression ratio higher than $3.0\times$, bandwidth-aware `cDMA` can only reduce the latency overheads of data movements by up to $3\times$. As we quantitatively demonstrate in Section VII-B, our bandwidth-aware `cDMA` still provides 94% of the performance of the baseline, performance-optimal `cDMA` policy, presenting a practical and cost-effective solution for DNN memory virtualization. Such a predefined threshold can be adjusted per user requirements by an extension to the CUDA driver API, similar in spirit to `cudaMemAdvise()` or `cudaMemPrefetchAsync()` used for customizing page migration policies in CUDA Unified Memory [61].

E. Software Interface

The (de)compression features of the DMA engine can be exposed to the programmer for adoption within ML frameworks and other applications. We envision that the compressed memory copy operation can be exposed to the software level using a new `cudaMemcpyCompressed()` call that enables the compression (or decompression) in the DMA engine. We expect this API will be extended beyond the typical `cudaMemcpy()` to also return the compressed size of a region on completion of the copy operation. In our experimental framework, the `cudaMemcpyCompressed` calls would easily replace the `cudaMemcpy` calls already deployed in `vDNN`.

VI. Evaluation Methodology

Architectural exploration of `cDMA` in cycle-level simulation is challenging for two primary reasons. First, existing GPU architecture simulators (e.g., GPGPU-Sim [62]) are not able to execute the cuDNN APIs as these GPU accelerated library routines are released as pre-compiled binaries. Second, a single iteration of training can take up to tens of seconds even on the fastest GPU, so running cycle-level simulations on these ML workloads within a reasonable timeframe is likely a research project on its own. We therefore take a hybrid approach in evaluating the effect of `cDMA` on training performance. Specifically, we measure DNN applications on a real GPU while properly penalizing the system performance based on an analytical model of the GPU memory subsystem, as summarized below.

GPU node topology. Our baseline DNN training platform contains an Intel i7-5930K CPU with 64 GB of DDR4 memory communicating with an NVIDIA Maxwell Titan X containing 12 GB of GDDR5 memory with a maximum of 336 GB/sec bandwidth [63]. For performance evaluation, we also study the effects of `cDMA` on a Pascal Titan Xp which contains 12 GB of GDDR5X memory with a maximum of 548 GB/sec [64]. The PCIe switch (gen3) provides a maximum of 16 GB/sec of data transfer bandwidth.

Virtualized DNN. We modeled the `vDNN` memory management policy as described in [10], which is interfaced to the latest version of cuDNN (v5) [9]. `vDNN` is configured to offload all the layer's activation maps for memory-scalability and to maximally stress the PCIe channel. The offload and prefetch operations to and from CPU memory are initiated using `cudaMemcpyAsync()`; the memory allocation size is determined by the compression ratio observed by the `cDMA` unit, as modeled below.

Compression pipeline. We implemented our `cDMA` compression pipeline on top of Caffe [1]. We modified the Caffe Python interface (`pycaffe`) to checkpoint the target network's activations so that they can be fed to our `cDMA` compression algorithm to compress and downsize the activation maps for each layer's offloaded data. The compressed activations are then returned to the `vDNN` memory manager to measure the latency incurred during the memory copy operation to/from the CPU memory across the PCIe bus.

Effect of `cDMA` on memory bandwidth. Compared to a baseline implementation of `vDNN`, `cDMA` affects system performance based on the following two factors. First, the reduced PCIe traffic helps improve the performance of `vDNN` because the latency to move data in/out of CPU memory is significantly reduced. However, to fully saturate the PCIe link bandwidth and maximize the benefits of DNN virtualization, the compressed activations must be generated at a throughput commensurate to the PCIe transfer bandwidth. Thus the second issue is that the average DRAM bandwidth utilization of `cDMA` can exceed that of `vDNN` by a factor

Table II: Networks and trained model accuracy.

Network	Top-1/5 (%)	Batch	Trained iterations
AlexNet	53.1 / 75.1	256	226K
OverFeat	52.8 / 76.4	256	130K
NiN	55.9 / 78.7	128	300K
VGG	56.5 / 82.9	128	130K
SqueezeNet	53.1 / 77.8	512	82K
GoogLeNet	56.1 / 83.4	256	212K

of $2.6\times$ (Section VII-A), potentially interfering with the cuDNN computation and decreasing performance.

State-of-the-art DNN libraries refactor the convolution operations into a dense matrix-multiplication operation for GPU acceleration [19]. This approach allows the CNN computation to be completely *compute-bound* with high cache hit rates and low average DRAM bandwidth utilization. Using the NVIDIA CUDA profiler (`nvprof`), we observed less than an average of 100 GB/sec of off-chip memory bandwidth utilization across all six networks. This leaves more than an average $336 - 100 = 236$ GB/sec of memory bandwidth available for our `cDMA` engine to fetch activation maps from Maxwell's GPU memory without affecting the throughput of DNN computations using cuDNN.

As we are not able to model `cDMA` inside an existing GPU, evaluating the performance of cuDNN with both `vDNN` and `cDMA` in silicon is impractical. Nonetheless, as long as the GPU memory bandwidth consumption of `cDMA` (i.e., a given layer's compression ratio$\times$PCIe bandwidth, denoted as `COMP_BW` below) is smaller than the available 236 GB/sec of DRAM bandwidth (`DRAM_BW`), the compressed activations can be generated at a high enough rate to fully saturate the PCIe bandwidth while not affecting the baseline cuDNN performance. To model the bandwidth limitations on `cDMA` performance, we restrict the memory bandwidth consumption of `cDMA` to never exceed the 236 GB/sec leftover bandwidth of Titan X. For the few layers that do require a DRAM bandwidth higher than 236 GB/sec (i.e., layers with compression ratio$\times$PCIe transfer bandwidth higher than `DRAM_BW`), we assume that the compressed activations are not generated at a fast enough rate to saturate the PCIe channel. In other words, when evaluating system performance of `vDNN`, we *increase* the latency incurred when offloading the compressed activations by a factor of (`COMP_BW`/`DRAM_BW`), modeled in an existing GPU by inflating the volume of data transferred over the PCIe interface. For a conservative evaluation, we set the `COMP_BW` value, the maximum memory bandwidth `cDMA` is allowed to consume[5], to 200 GB/sec. The bandwidth-aware `cDMA` policy does not consume more than 48 GB/sec of DRAM bandwidth as discussed in Section V-D.

Training methodology. All networks are trained using stochastic gradient descent (SGD) with an initial learning rate of 0.01. We manually reduce the learning rate by factor of 0.1 or 0.5, choosing the value that provides higher improvements in validation accuracy when the validation error plateaus. Dropout [65] is employed for the fully-connected layers with a rate of 0.5. We terminate the training process when the validation accuracy does not improve further beyond a learning rate smaller than 1×10^{-5}. All of our compression algorithms are *lossless* and affect neither the functionality nor the algorithmic nature of SGD.

Networks evaluated. We study DNNs that show state-of-the-art performance in ImageNet [40]: AlexNet [14], OverFeat [45], NiN [46], VGG [11], SqueezeNet [47], and GoogLeNet [12]. We configure these networks based on the `.prototxt` files available at Caffe Zoo [1] or those available at the original authors' websites [46], [47]. Table II summarizes each network's fully trained top-1/top-5 classification accuracy, the minibatch sizes used for training, and the total number of training iterations taken to reach its final trained model.

VII. Results

This section evaluates the efficiency of `cDMA` compression, the savings in PCIe traffic, and the effect of `cDMA` on energy-efficiency and performance. The three compression algorithms discussed in Section V are denoted as `RL` (run-length encoding), `ZV` (zero-value compression), and `ZL` (zlib) in all of the figures discussed in this section. `vDNN` is evaluated with the memory management policy that provides memory-scalability, which offloads all activation maps. We also established an *oracular* baseline (`orac`) that completely removes the PCIe bottleneck by having the offload/prefetch latencies always be hidden inside the DNN computation when measuring performance.

A. Compression Efficiency

Figure 11 shows the the *maximum* per-layer compression ratio across a given network and the *average* network-wide compression ratio for each of the three compression algorithms and three data layouts. While the results presented in this section assume a 4KB compression window, we also studied windows up to 64KB and found that our results did not change much.

The maximum per-layer compression ratio determines how much DRAM bandwidth `cDMA` must provision to generate the compressed activations at a high enough rate to fully saturate the PCIe bandwidth. The average network-wide compression ratio reflects the reduction in PCIe traffic provided by `cDMA` . Overall, our `ZVC` algorithm provides the best average compression ratio across all the networks and all three data layouts (average $2.6\times$). Despite its simple design, the efficiency of `ZVC` is decoupled from the sparsity patterns in the activation maps and provides the same compression ratio regardless of how the activations are

[5] Even though the peak memory bandwidth consumption of `cDMA` can be on the order of 200 GB/sec, the *average* memory bandwidth usage will not exceed $16\times2.6 = 41.3$ GB/sec, which is the PCIe bandwidth$\times$average network-wide compression ratio of 2.6.

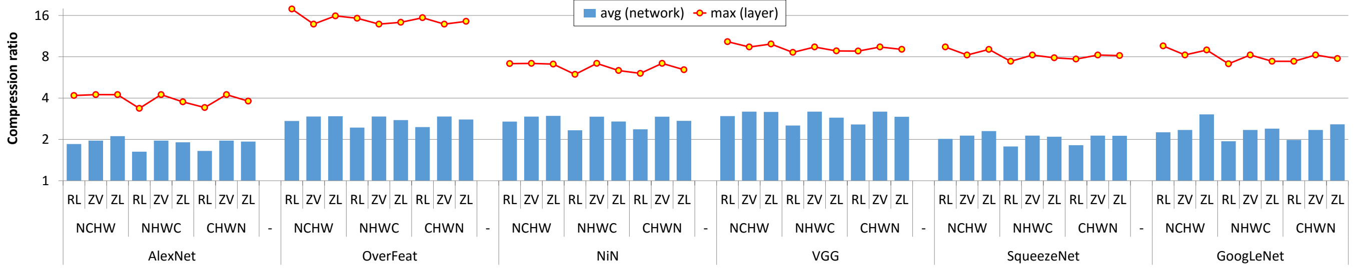


Figure 11: The average and maximum compression ratio for a given compression algorithm for different activation data layouts. The average compression ratio is weighted by the size of the activation maps that are being offloaded to the CPU so that it properly reflects the reduction in PCIe traffic. The compression window is configured as 4 KB for all three compression schemes. The y-axis is plotted on a log_2 scale.

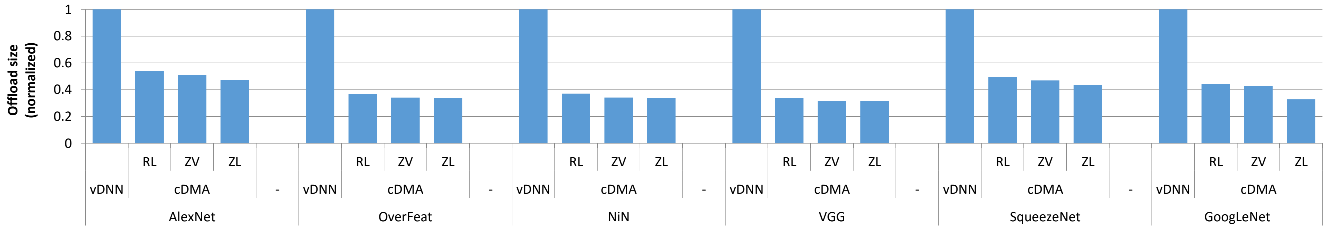


Figure 12: The size of the activation maps offloaded to CPU memory (normalized to vDNN).

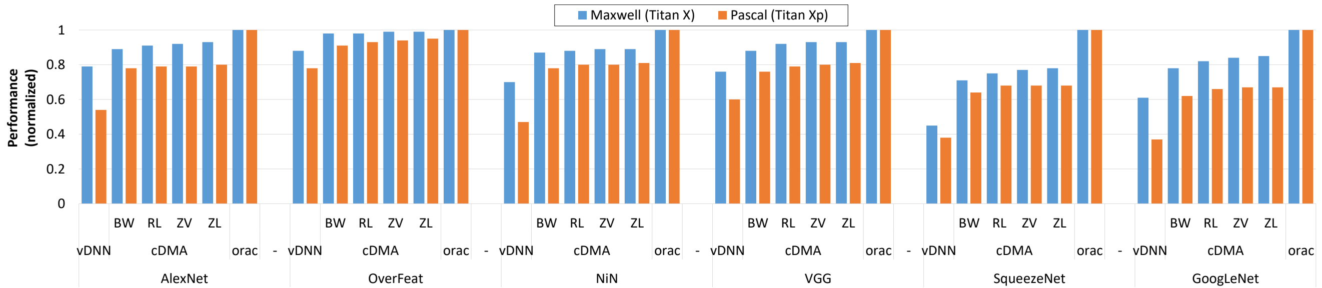


Figure 13: Overall performance of cDMA on Maxwell and Pascal GPUs (normalized to oracle baseline).

arranged in GPU memory. zlib shows the highest average compression ratio of 2.76× with NCHW but falls behind ZVC for all but GoogLeNet with NHWC and CHWN. Similar to zlib, RLE performs best with NCHW but provides the worst compression with high sensitivity to the underlying data layouts. As mentioned in Section V, zlib and RLE prefer NCHW because this layout makes it more likely to have the activation sparsity in a spatially clustered manner. In the rest of this paper, we assume the NCHW layout for both brevity and for a conservative evaluation of ZVC as both RLE and zlib perform best with NCHW.

Figure 12 shows the reduction in the size of the activations offloaded to the CPU, which directly translates into PCIe traffic reduction. Although the sophisticated zlib algorithm provides a further 30% reduction in PCIe traffic for GoogLeNet (over ZVC), the average traffic reduction across all six networks is only 3% compared to ZVC.

B. Performance

Figure 13 summarizes the performance of cDMA compared to vDNN and the oracular baseline. The performance of RLE, ZVC, and zlib are measured based on the baseline performance-optimal cDMA policy, consuming up to (compression ratio×PCIe bandwidth) of DRAM bandwidth. As discussed in Section V-C, bandwidth-aware cDMA (denoted as BW) only consumes up to 48 GB/sec of memory bandwidth; we report its effect on performance under ZVC algorithm for brevity.

While zlib provides the highest compression ratio for SqueezeNet and GoogLeNet (8% and 30% higher than ZVC), the resulting performance improvements are marginal, providing an average 0.7% speedup over ZVC (maximum 2.2% for GoogLeNet). The meager performance advantage of zlib stems from two reasons: (1) a significant fraction of the offloading latency is already being hidden by the DNN forward and backward propagation operations, and (2) the higher compression ratios zlib achieves are for layers of which RLE and ZVC already are able to mostly hide the offloading latencies. Similarly, the bandwidth-aware cDMA (BW) performs competitively against all three performance-optimal policies, reaching an average 94% of the best performing cDMA while only consuming up to 48 GB/sec

of DRAM bandwidth. Because of its simple compression algorithm and robustness across different data layouts, we conclude that ZVC with the bandwidth-aware cDMA policy is the best option for DNN virtualization.

C. Energy Efficiency

The current CUDA software stack does not provide users the ability to change the DRAM read bandwidth or PCIe transfer bandwidth, making it difficult to precisely measure the effect of cDMA on energy-efficiency. Instead, we provide a qualitative comparison of cDMA's energy-efficiency versus vDNN. The primary energy overheads cDMA imposes on vDNN are (1) the average 2.6× increase in DRAM read bandwidth, corresponding to ZVC's average network-wide compression ratio, for fetching the activations from the DRAM for cDMA compression; and (2) the (de)compression units and buffers augmented inside the GPU. Based on the examination of overheads of cDMA in Section V-C, we expect the energy costs for the additional compression logic and its buffers to be negligible as cDMA primarily leverages (de)compression units already existing in GPU memory controllers.

Also, while vDNN's offload/prefetch operations do incur 1–7% power overhead, as measured with nvprof, cDMA's average 2.6× reduction in PCIe traffic will significantly reduce the energy-consumption on the PCIe link as well as in the CPU memory subsystem. When accounting for the average 32% performance improvements (maximum 61%) provided by cDMA, we expect the overall energy consumption of cDMA to be noticeably lower than vDNN.

VIII. Discussion and Future Work

Future CPU-GPU interconnects. NVLINK is NVIDIA's proprietary, high-bandwidth interconnect that enables fast communication between GPU and CPU, and between GPUs [38]. When coupled with IBM Power systems [66], the communication bandwidth between the CPU-GPU can be up to 80 GB/sec, alleviating the communication bottleneck of virtualized DNNs. With a multi-GPU DNN platform [67] where 4 to 8 GPUs share the same communication channel, the bandwidth allocated per each single GPU is 10–20 GB/sec, which is similar to what we evaluate in this paper. Similarly, while next generation PCIe gen4 will provide 2× higher bandwidth than PCIe gen3 and increase CPU-GPU communication bandwidth to 64 GB/sec (as opposed to 32 GB/sec of gen3), communication bandwidth allocated for each single GPU will remain at (64/4)=16 GB/sec, which is equivalent to our evaluation setting. Consequently, while these future CPU-GPU interconnects can help alleviate the performance overheads of vDNN, it does not fundamentally change the problem we address in this paper. Current research trends point to even more memory hungry DNN structures with complex inter-layer dependencies; for an N layered neural network, the memory allocation size has grown from $O(N)$ for AlexNet, VGG, or GoogLeNet to $O(N^2)$ for DenseNet [68], due to its densely interconnected layers. Given such trends, the quadratically increased memory requirements of DNNs will cause much higher pressure on CPU-GPU interconnects even for PCIe gen5 (providing 4× higher bandwidth then gen3, scheduled for standardization in 2019), thereby continuing to motivate cDMA for future systems running more complex and memory-limited DNN algorithms. Overall, hardware/software co-designs like cDMA that can remedy the CPU-GPU communication bottleneck will grow in importance as DNNs become deeper and their memory requirements grow.

Design costs of cDMA. In this paper, we assumed that cDMA hardware supplements existing compression logic for a conservative area estimation, consuming a total of $0.52\,\text{mm}^2$ in a 28 nm process assuming PCIe gen3. For future high-bandwidth CPU-GPU interconnects like PCIe gen4/gen5 or NVLINK, the cDMA engine should be capable of keeping up with the higher CPU-GPU communication bandwidth and be provisioned with a proportionally larger buffer size to hold the bandwidth-delay product of the memory sourcing the data (Section V-C). For PCIe gen5, the overall area overhead of cDMA would amount to $1.5\,\text{mm}^2$ in a 28 nm process, which is negligible as current high-end GPU chips already exceed $800\,\text{mm}^2$ (e.g., Volta V100 [37]). However, we also estimate that the cDMA design complexity can be amortized by leveraging existing GPU compression logic [55], [56], [57] thereby incurring negligible design overheads on top of existing GPU microarchitecture.

Compression for GPU footprint reduction. While cDMA help reduce the PCIe traffic and CPU-side memory footprint, the amount of memory allocated inside the GPU is the same as the baseline vDNN. To reduce GPU DRAM bandwith and memory capacity requirements, the compression engine inside the GPU's memory controllers could compress and store the activations inside the GPU's DRAM. Implementing this optimization involves developing efficient memory addressing schemes that allow the memory controller to retrieve the data in its original, uncompressed form without disturbing overall performance and energy-efficiency. This future work is beyond the scope of this paper.

IX. Conclusion

Previous DNN virtualization solutions can incur significant performance overheads when the communication channel between the CPU and GPU is bottlenecked. We introduce a general purpose compressing DMA engine that can be used for high-performance DNN virtualization. Our proposal exploits the unique characteristics of DNNs to develop a cost-efficient compression algorithm, offering an average 2.6× (maximum 13.8×) savings in data movement on the CPU–GPU communication link. Overall, our cDMA engine improves the performance of virtualized DNNs by an average 53% (maximum 79%) on the Pascal Titan Xp

with a modest implementation overhead and can easily be adopted into existing ML frameworks.

REFERENCES

[1] Caffe, http://caffe.berkeleyvision.org, 2016.

[2] Torch, http://torch.ch, 2016.

[3] Theano, http://deeplearning.net/tutorial, 2016.

[4] Tensorflow, https://www.tensorflow.org, 2016.

[5] Nervana, https://github.com/NervanaSystems/neon, 2016.

[6] Microsoft, https://github.com/Microsoft/CNTK, 2016.

[7] A. Krizhevsky, "cuda-convnet," 2012.

[8] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems," in *Proceedings of the Workshop on Machine Learning Systems*, December 2015.

[9] NVIDIA, "cuDNN: GPU Accelerated Deep Learning," 2016.

[10] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, "vDNN: Virtualized Deep Neural Networks for Scalable, Memory-Efficient Neural Network Design," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, October 2016.

[11] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," May 2015.

[12] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going Deeper with Convolutions," in *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[13] G. Huang, Y. Sun, Z. Liu, D. Sedra, and K. Weinberger, "Deep Networks with Stochastic Depth," 2016.

[14] A. Krizhevsky, I. Sutskever, and G. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Proceedings of the International Conference on Neural Information Processing Systems (NIPS)*, December 2012.

[15] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-Neuron-Free Deep Convolutional Neural Network Computing," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2016.

[16] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-X: An Accelerator for Sparse Neural Networks," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, October 2016.

[17] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2017.

[18] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-Based Learning Applied to Document Recognition," *Proceedings of the IEEE*, November 1998.

[19] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cuDNN: Efficient Primitives for Deep Learning," in *Proceedings of the International Conference on Neural Information Processing Systems (NIPS)*, December 2014.

[20] Y. LeCun, S. Denker, and S. Solla, "Optimal Brain Damage," in *Proceedings of the International Conference on Neural Information Processing Systems (NIPS)*, November 1990.

[21] B. Hassibi and D. Stork, "Second Order Derivatives for Network Pruning: Optimal Brain Surgeon," in *Proceedings of the International Conference on Neural Information Processing Systems (NIPS)*, November 1993.

[22] S. Han, J. Pool, J. Tran, and W. Dally, "Learning Both Weights and Connections for Efficient Neural Networks," in *Proceedings of the International Conference on Neural Information Processing Systems (NIPS)*, December 2015.

[23] J. Hauswald, Y. Kang, M. A. Laurenzano, Q. Chen, C. Li, T. Mudge, R. G. Dreslinski, J. Mars, and L. Tang, "DjiNN and Tonic: DNN as a Service and Its Implications for Future Warehouse Scale Computers," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2015.

[24] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "DianNao: A Small-footprint High-throughput Accelerator for Ubiquitous Machine-learning," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, March 2014.

[25] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "DaDianNao: A Machine-Learning Supercomputer," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, December 2014.

[26] Y. Chen, T. Krishna, J. Emer, and V. Sze, "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," in *Proceedings of the International Solid State Circuits Conference (ISSCC)*, February 2016.

[27] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. Horowitz, and W. Dally, "EIE: Efficient Inference Engine on Compressed Deep Neural Network," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2016.

[28] Y. Chen, J. Emer, and V. Sze, "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2016.

[29] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. Lee, J. Miguel, H. Lobato, G. Wei, and D. Brooks, "Minerva: Enabling Low-Power, High-Accuracy Deep Neural Network Accelerators," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2016.

[30] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "A Novel Processing-in-memory Architecture for Neural Network Computation in ReRAM-based Main Memory," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2016.

[31] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, “ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars,” in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2016.

[32] B. Pichai, L. Hsu, and A. Bhattacharjee, “Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs with Unified Address Spaces,” in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, March 2014.

[33] J. Power, M. Hill, and D. Wood, “Supporting x86-64 Address Translation for 100s of GPU Lanes,” in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, February 2014.

[34] T. Zheng, D. Nellans, A. Zulfiqar, M. Stephenson, and S. W. Keckler, “Toward High-Performance Paged-Memory for GPUs,” in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, March 2016.

[35] Google, “Tensorflow: Memory-optimizer,” 2017.

[36] IBM, “Chainer: Out-of-core training,” 2017.

[37] NVIDIA, “NVIDIA Tesla V100,” 2017.

[38] ——, “NVIDIA NVLINK High-Speed Interconnect,” 2016.

[39] Y. Sun, X. Wang, and X. Tang, “Deeply Learned Face Representations Are Sparse, Selective, and Robust,” in *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[40] ImageNet, http://image-net.org, 2016.

[41] A. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates, and A. Y. Ng, “Deep Speech: Scaling Up End-To-End Speech Recognition,” 2014.

[42] G. Diamos, S. Sengupta, B. Catanzaro, M. Chrzanowski, A. Coates, E. Elsen, J. Engel, A. Hannun, and S. Satheesh, “Persistent RNNs: Stashing Recurrent Weights On-Chip,” in *Proceedings of the International Conference on Machine Learning (ICML)*, June 2016.

[43] S. Hochreiter and J. Schmidhuber, “Long Short Term Memory,” *Neural Computation*, November 1997.

[44] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, “Gated Feedback Recurrent Neural Networks,” 2015.

[45] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun, “OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks,” 2013.

[46] M. Lin, Q. Chen, and S. Yan, “Network in Network,” 2013.

[47] F. Iandola, S. Han, M. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, “SqueezeNet: AlexNet-level Accuracy with 50x Fewer Parameters and <0.5MB Model Size,” 2016.

[48] M. Zeiler and R. Fergus, “Visualizing and Understanding Convolutional Networks,” 2013.

[49] P. Wilson, S. Kaplan, and Y. Smaragdakis, “The Case for Compressed Cache in Virtual Memory Systems,” in *Proceedings of USENIX*, June 1999.

[50] A. Robinson and C. Cherry, “Results of a Prototype Television Bandwidth Compression Scheme,” *Proceedings of the IEEE*, March 1967.

[51] Y. Zhang, J. Yang, and R. Gupta, “Frequent Value Locality and Value-centric Data Cache Design,” in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, November 2000.

[52] J. Gailly and M. Adler, “The gzip Home Page.”

[53] M. Abdelfattah, A. Hagiescu, and D. Singh, “Gzip on a Chip: High Performance Lossless Data Compression on FPGAs Using OpenCL,” in *Proceedings of the International Workshop on OpenCL*, May 2014.

[54] J. D. Deaton and A. Bacon, “White Paper: Smashing Big Data Costs with GZIP Hardware.”

[55] V. Sathish, M. Schulte, and N. Kim, “Lossless and Lossy Memory I/O Link Compression for Improving Performance of GPGPU Workloads,” in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, September 2012.

[56] G. Pekhimenko, E. Bolotin, N. Vijaykumar, O. Mutlu, T. C. Mowry, and S. W. Keckler, “A Case for Toggle-Aware Compression for GPU Systems,” in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, March 2016.

[57] “NVIDIA Tegra X1: NVIDIA’s New Mobile Superchip,” 2015.

[58] NCSU, “FreePDK Process Design Kit,” 2016.

[59] H. Wong, M. M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, “Demystifying GPU Microarchitecture Through Microbenchmarking,” in *Proceedings of the International Symposium on Performance Analysis of Systems Software (ISPASS)*, March 2010.

[60] HP Labs, “CACTI: An Integrated Cache and Memory Access Time, Cycle Time, Area, Leakage, and Dynamic Power Model,” 2016.

[61] NVIDIA, “NVIDIA CUDA Programming Guide,” 2016.

[62] “GPGPU-Sim,” 2016.

[63] NVIDIA, “GeForce GTX Titan X (Maxwell),” 2015.

[64] ——, “GeForce GTX Titan Xp (Pascal),” 2017.

[65] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: A Simple Way to Prevent Neural Networks from Overfitting,” *Journal of Machine Learning Research*, June 2014.

[66] IBM, “IBM Power Systems,” 2016.

[67] NVIDIA, “The NVIDIA DGX-1 Deep Learning System,” 2016.

[68] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, “Densely Connected Convolutional Networks,” in *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2017.