Dynamic Memory Management for GPU-based training of Deep Neural Networks

Shriram S B, Anshuj Garg and Purushottam Kulkarni
Department of Computer Science and Engineering
Indian Institute of Technology Bombay
{*shriramsb, anshujgarg, puru*}*@cse.iitb.ac.in*

***Abstract*—Deep learning has been widely adopted for different applications of artificial intelligence—speech recognition, natural language processing, computer vision etc. The growing size of Deep Neural Networks (DNNs) has compelled the researchers to design memory efficient and performance optimal algorithms. Apart from algorithmic improvements, specialized hardware like Graphics Processing Units (GPUs) are being widely employed to accelerate the training and inference phases of deep networks. However, the limited GPU memory capacity limits the upper bound on the size of networks that can be offloaded to and trained using GPUs. vDNN addresses the GPU memory bottleneck issue and provides a solution which enables training of deep networks that are larger than GPU memory. In our work, we characterize and identify multiple bottlenecks with vDNN like delayed computation start, high pinned memory requirements and GPU memory fragmentation. We present vDNN++ which extends vDNN and resolves the identified issues. Our results show that the performance of vDNN++ is comparable or better (up to 60% relative improvement) than vDNN. We propose different heuristics and order for memory allocation, and empirically evaluate the extent of memory fragmentation with them. We are also able to reduce the pinned memory requirement by up to 60%.**

***Keywords*-Deep Learning; Deep Neural Networks; GPU; Memory management;**

I. Introduction

Deep neural networks are a growing force for machine learning based applications in multiple domains—natural language processing [1], speech recognition [2], computer vision [3] etc. The efficacy of a deep neural network rests on the amount of data available for training and the compute power available for the training process. A popular trend to address the compute requirements has been to use Graphics Processing Units (GPUs) to harness the parallel computation model. In fact, all of the popular software frameworks for deep learning—TensorFlow [4], PyTorch [5], etc.— provide support for the usage of GPUs for training neural networks.

As the application domain and relevance of deep learning increases, the structure of neural networks—number of layers, interconnection topology, computation at each layer etc.— is also rapidly getting more and more sophisticated. This trend can be observed with the ImageNet Challenge [6] winning networks, with AlexNet(2012) [3] having 8 layers and ResNet(2015) [7] having about hundred layers. The memory consumption of these networks is also increasing rapidly—AlexNet(128) requiring 1.1GB and ResNet-152(4) requiring 5GB, where number within (.) represent batch size. Current deep learning frameworks expect the entire training data to be present in main memory, a requirement extended to GPU based processing setups as well. This poses a potential bottleneck with GPUs, as they have smaller memory capacities as compared to conventional CPU based server systems. Typical memory capacities of GPUs are in the range of 2 GB to 24 GB [8], while requirements of deep neural networks can be more than 20 GB [9]. As a result, network combinations during training that have memory requirements greater than GPU memory capacity are discarded and do not contribute to the learning procedure.

An approach to address the memory capacity bottleneck of GPUs is to exploit the layer-wise computation based procedure employed by neural networks for learning. At any point of time, the GPU processes only one layer's computation due to limited number of cores. So, at any point of time, space for only a single layer's parameters, its inputs and outputs are required for computation. Previous work, vDNN [10], SuperNeurons [11], ooc_cuDNN [12] and moDNN [13] have exploited this peculiarity, and proposed dynamic management of GPU memory by moving to and fro the intermediate state of the network (inputs and intermediate outputs) between system memory (CPU DRAM) and the GPU memory.

The central idea of vDNN's approach is to offload data of a network layer when it is not required in the near future and make space for data of other layers to occupy GPU memory. The offloaded state is later fetched back to the GPU when required. To save time, vDNN transfers data of a layer while the computation associated with a layer (not necessarily the same layer) is in progress. But, vDNN synchronizes computation and offload/prefetch of data at the end of each layer; i.e. it continues to the next layer's computation only after both computation and transfer have been completed. This can lead to the GPU waiting for offload to finish before next layer's computation commences. We demonstrate that there are several cases in which it faces performance loss ("performance" is in the sense of training/inference time unless otherwise specified, throughout this paper) and propose a fix that begins computation of a layer at the earliest possible instance.

Further, vDNN uses the CNMeM library [14], which manages the GPU memory via the CUDA API. We demonstrate

that the memory juggling technique (offloading to system memory and copying back to GPU memory) results in fragmentation and leads to higher peak memory requirements compared to the peak aggregate memory of layers in the GPU. We explore heuristics which reduce fragmentation using the knowledge of layers chosen to be offloaded and the order in which memory can be allocated for each layer.

Offloading data from GPU memory to CPU memory frees GPU memory but increases the CPU memory footprint of DNN. Moreover vDNN uses pinned memory to speed up the data transfer. Pinned memory is non-pageable and hence it cannot be allocated by CPU to other processes. Unlike previous work [15] which proposed hardware changes to GPU, we show feasibility of compressing data using CPU cores. This could be used to reduce pinned memory consumption and train on systems with small CPU DRAM or provide memory for other processes. We also show that CPU overheads for compression are not high and that sufficient amount of processing power is available to run other processes.

Towards improving the efficacy of using GPUs for deep learning by exploiting dynamic memory offloading, following are important contributions of our work:

- Identification and demonstration of drawbacks of vDNN's approach and design of solutions to resolve them.
- Design of heuristics to reduce memory fragmentation by accounting for memory offload decisions.
- vDNN++ — the implementation of the proposed improvements and empirical demonstration.
- Exploring the feasibility of (de)compressing offloaded data using CPU to reduce vDNN's CPU memory footprint.

The rest of the paper is organized as follows. Section II briefs about Deep Neural Networks (DNN) and vDNN's approach. Section III details drawbacks of vDNN. In Section IV, we present design and implementation of vDNN++. Section V contains the experimental results of vDNN++. Section VI discusses the related work and Section VII concludes the paper.

II. Background

A. Deep Neural Networks

Figure 1 shows a neural network composed of several layers. Training a neural network involves learning parameters (W) of each layer (which compute a function) to give a final output close to the desired value for each input. During training, in each iteration, a batch of input data is fed to the first layer. At each layer, forward propagation is performed in which, the inputs (X) determine an output (Y) using the parameters (W) of that layer. The output of each layer is used as one of the inputs for another layer. The final layer produces an output which is compared with the desired

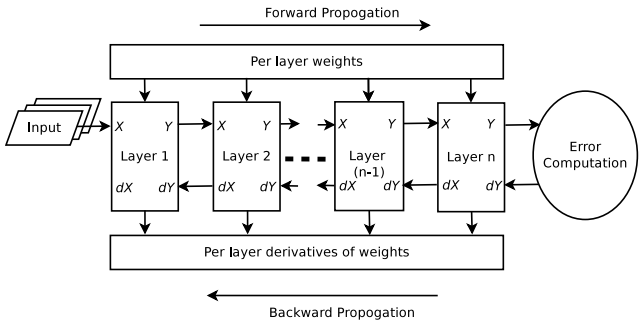

Figure 1. High level architecture of Deep Neural Networks (DNNs). X & Y represent input and output of a layer for forward propagation, dY and dX represent input and output of a layer for backward propagation respectively.

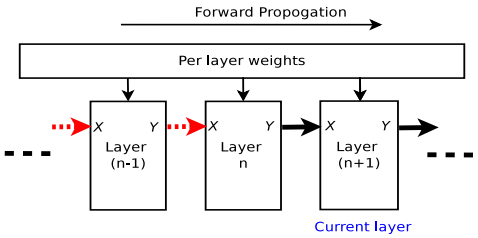

Figure 2. GPU memory snapshot during vDNN's forward propogation phase.

value to determine the error (a measure of the closeness of the two values). Gradient of the error is used to update the parameters to improve prediction accuracy. Gradient of error w.r.t output is calculated and is propagated back till the first layer. At each layer, backward propagation is performed in which the gradients of inputs (dX) and parameters of each layer (dW) are computed using its input (X), output of forward propagation (Y), parameters of current layer (W) and gradients of error w.r.t its output (dY). Derivative of the error is propagated one layer at time with input gradient of further layers (dX) fed to the previous layer (dY). Multiple iterations of the above are performed till the error is minimized.

The two most popular deep neural networks (DNNs) are Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs). CNNs are widely used for image recognition tasks [16] whereas RNNs are used for natural language processing, speech recognition and video captioning [17]. These two neural network are designed using a combination of four types of layers, viz., the convolution layer, the activation layer, the pooling layer and the fully-connected layer. The two main functions of deep networks are feature extraction and feature classification, with the former being implemented using convolution and pooling layers and the latter with fully connected layers. Activation layers are used to estimate non-linear functions w.r.t inputs. While we discuss details of our work using CNNs the concepts are general and can be extended to apply to other layer types as well.

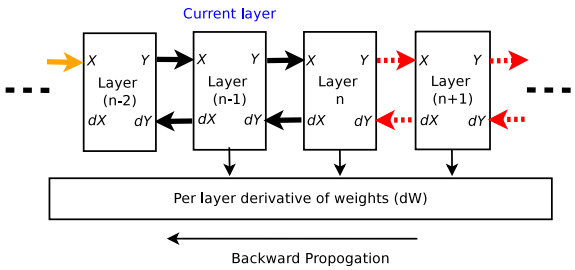

Figure 3. GPU memory snapshot during vDNN's backward propogation phase.

B. virtualized DNN (vDNN)

The current deep learning frameworks require the entire data needed by the neural network to be present in GPU memory. This constrains the size of neural networks that can be trained using GPUs. If the size of network exceeds the GPU memory, the DNN architects either have to compromise with the training time and use slower memory optimal convolution algorithms or less memory consuming undesirable network architecture or use multiple GPUs for training.

vDNN, virtualized Deep Neural Networks [10], in their work, showed that only a small percentage of the entire data is used at any point of time while training the network. Moreover, the training proceeds sequentially, processing one layer at a time. Hence, only data required by the current layer needs to be present in memory. vDNN offloads the data which is not required in the near future to the CPU memory. Also, if the data required by the current layer is present in the CPU memory, it fetches the data from CPU memory to GPU memory. The CPU-GPU memory transfers in forward and backward propagation phase of vDNN work as follows:

Forward Propagation: Once a layer's forward computation is complete, its input X is not needed till the layer is considered again during backward propagation. vDNN utilizes this fact and offloads the input X to the CPU memory. Figure 2 shows that inputs to $layer_{(n-1)}$ and $layer_{(n)}$ (red dotted arrows) have been offloaded to GPU, whereas the input to the current $layer_{(n+1)}$ is in memory (denoted by bold arrows). The offload of X of $layer_{(n+1)}$ is overlapped with its computation using CUDA streams [18].

Backward propagation: In the backward propagation phase, each layer requires gradient of the output (dY) back-propagated from its next layer, output Y generated and input X fed to it in forward propagation phase. Since the X and Y might be offloaded to CPU memory, they need to be fetched to GPU memory before current layer's backward computation can begin. Also, after the backward propagation of a layer is complete, its output Y and its gradient dY can be freed. Referring to Figure 3. solid black arrow indicate data which is present in the GPU memory, the solid orange arrow indicates the data which needs to be prefetched from CPU memory, and dotted red arrows indicate the data which is freed from GPU memory. Just after completion of back propagation of $layer_{(n)}$, its Y and dY are freed. Fetch of X of $layer_{(n-2)}$ from CPU is overlapped with back propagation of $layer_{(n-1)}$ to minimize memory fetch stalls during execution. In general, vDNN overlaps the prefetch of input of $\text{layer}_{(n)}$ with computation of $\text{layer}_{(m)}$ (for $n < m$) to reduce performance loss.

Offloading all layers is inefficient as for some networks, time taken for transfer might be more than the time taken for computation. Convolution (CONV) layers of deep networks are generally slower and hence, vDNN makes two choices of layers to offload. It either chooses to offload ALL layers ("all" offload scheme) or only all of CONV layers ("conv" offload scheme). Offloading ALL layers is most memory efficient. Offloading CONV layers is beneficial as computation of CONV can effectively cover transfer and will not affect performance much. There are multiple CONV algorithms, some of which consume less memory, taking more time and vice versa. Again, vDNN makes two choices. It offers a choice between performance optimal CONV algorithm ("p" scheme), which improves training time or memory optimal CONV algorithm ("m" scheme) which aims to reduce memory requirements during training. The above two give rise to 4 modes, namely all(m), all(p), conv(m), conv(p). Here, Offload-Scheme(Algorithm-Scheme) is used as short-hand for denoting the layers chosen to offload and CONV algorithm used. vDNN also provides a dynamic mode ("dyn" or "dy" scheme) in which it runs few profile passes and uses a heuristic to chooses the mode which fits in the memory and is the fastest. Tables and graphs use notation of Offload-Scheme(Algorithm-Scheme) to denote vDNN's mode. We also try another choice of layers to offload, namely alternate CONV layers ("aconv" scheme), denoted as aconv(p) and aconv(m) for performance and memory optimal algorithms for computation respectively. Alternate CONV offload consumes more memory than CONV offload but is more likely to be faster due to less transfer of data.

III. Motivation

In our work, we identify multiple opportunities for improvements beyond vDNN's approach. This section discusses them in detail.

A. Stalling of forward and backward propagation

vDNN synchronizes computation and offload at the end of each layer in both forward and backward propagation phases. Authors of vDNN state that synchronization ensures that offloaded layer is safely released from the memory pool before the next layer's computation begins, maximizing the memory saving benefits. While it is safe, it is not efficient. In most of the networks, "fast compute" layers are interspersed among "long compute" layers. For example, POOL (fast

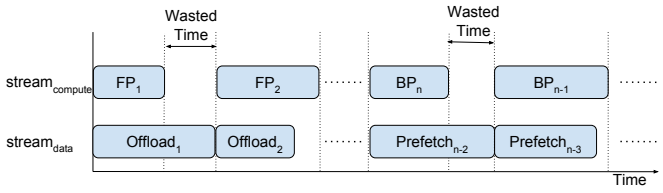

Figure 4. Delayed forward computation in case of vDNN. In forward propagation, computation of layer$_{(2)}$ is stalled till offload of layer$_{(1)}$'s input is complete. In back propagation, computation of layer$_{(n-1)}$ is stalled till prefetch of layer$_{(n-2)}$'s input is complete.

Table I
TIME FOR DIFFERENT COMPUTATIONS AND TRANSFER

Operation	Time(ms)
POOL	3
1x1 CONV	19
3x3 CONV	80
5x5 CONV	200
Transfer	62

compute) layers between 3x3 CONV (long compute) layers in AlexNet [3] and VGG [16] networks and 1x1 CONV (fast compute) between 3x3 or 5x5 CONV (long compute) in GoogLeNet [19] and ResNet [7] networks. This leads to loss of performance as shown in Figure 4.

Table I shows time for computation and transfer of inputs for different types of layers with input size 112x112x64, batch size of 128. POOL uses 2x2 filter, stride of 2, with 0 padding. CONV uses the designated filter size, stride 1, required padding to keep output dimension same as that of input and memory optimal algorithm. Timings were measured with Nvidia 1080Ti GPU. It shows that waiting for offload to complete after completion of 1x1 CONV and POOL will result in stalling of the computation stream and eventually a performance loss. With 3x3 and 5x5 CONV, offload would have completed before computation is done, and the CPU-GPU link remains unutilized. Instead, it would be better use of resources if computation of next layer is started as soon as computation of the previous layer is done. This would overlap part of the offload of "fast compute" layer with computation of "long compute" layers increasing the utilization of resources and thus, the performance.

The total space allocated to training process is equal to peak aggregate memory consumed at any point of time. The process memory requirement depends on memory for both forward and backward propagation. As seen in Figure 5, the peak GPU memory consumed in forward propagation in vDNN is much less than the peak GPU memory consumed in back-propagation. Backward propagation of each layer requires allocation of gradients of inputs and outputs apart from space for inputs and outputs and hence double the memory compared to forward propagation. Hence, there is enough space for the allocation of output of next layer and

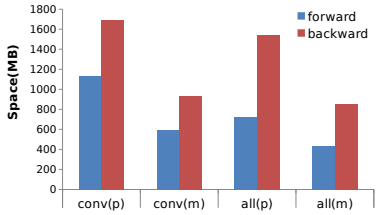

Figure 5. Comparison of aggregate peak GPU memory consumed in forward and backward propagation for AlexNet for batch size 256 in vDNN.

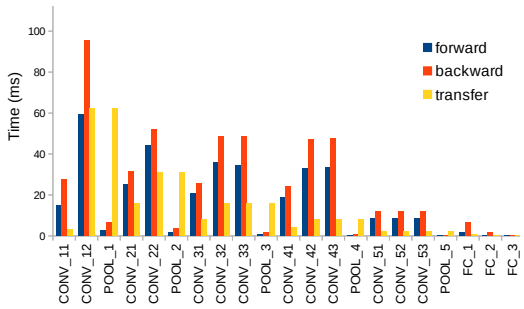

Figure 6. Computation vs transfer time for VGG-16, batch size 64 using memory optimal CONV algorithm. Experimented on Nvidia 1080 Ti.

start its forward propagation without waiting for offload of current layer's input to complete.

A similar observation can be made in the back-propagation phase. vDNN waits for memory prefetch of layer$_{(n)}$ to complete when overlapped with layer$_{(m)}$'s computation ($m > n$), even if the computation is completed. Instead, vDNN could start the computation of layer$_{(m-1)}$ without waiting for prefetch of layer$_{(n)}$. Computation has to be stalled only when $(m-1) = n$ as input of layer$_{(m-1)}$ will not be available in this case. Also, note that the order of allocation and free requests does not change. If prefetch of layer$_{(n)}$ is not complete after computation of layer$_{(m)}$, then just before starting computation of layer$(m-1)$, (i) layer$_{(m)}$'s output and its gradient are freed, (ii) space for, say layer$_{(p)}$ (whose prefetch is overlapped with layer$_{(m-1)}$) is allocated and prefetch command given before start of layer$_{(m-1)}$'s computation. This is the exact same order for allocation and free even if prefetch of layer$_{(n)}$ is synchronized with computation of layer$_{(m)}$. So, asynchronous prefetch in backward propagation doesn't require any extra GPU memory.

As an example, in VGG-16 (Figure 6), POOL_3's forward compute time is much lower compared to transfer time, while next layer's (CONV_41) forward compute time is higher than transfer time. If POOL_3's input is decided to be offloaded, then forward compute of CONV_41 could cover the transfer time for POOL_3 and its own. It isn't necessary to wait for offload of POOL_3 before starting next layer's computation.

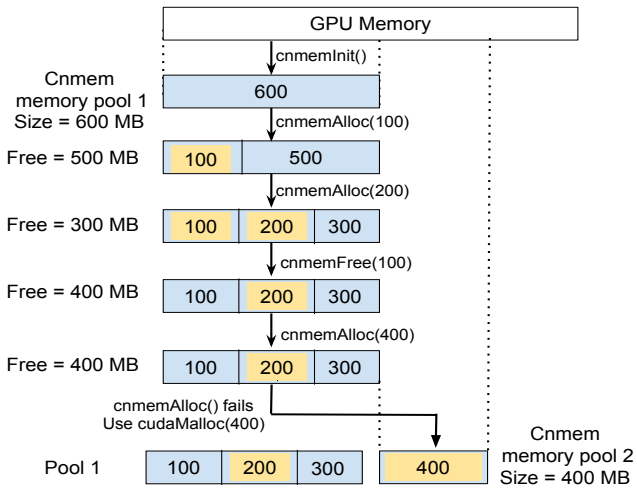

Figure 7. Fragmentation with CNMeM's memory management.

B. Memory fragmentation

vDNN uses CNMeM [14] library for managing memory. CNMeM is built over CUDA [20], which is an API for programming NVIDIA GPUs. CNMeM operates by pre-allocating a large chunk of memory from the GPU and then allocating it via the CNMeM API to CUDA applications. Due to numerous interspersed (de)allocation operations (caused by memory offload/prefetch) in a single iteration of training, memory fragmentation occurs. An example is shown in Figure 7—CNMeM initially obtains a pool of size 600MB from the GPU and subsequently a series of allocation and free requests are processed. For the allocation request of 400 MB, the aggregate free pool size is 400 MB but still it is not possible to serve the request due to fragmentation. CNMeM requests a new pool of size 400 MB from the GPU and serves the request.

Another problem with CNMeM is that if it runs out of pool memory, it allocates a new pool with size equal to the required size using `cudaMalloc()` and this new pool is never merged with the old one, causing further fragmentation across memory pools. So, reducing the total GPU memory consumption by starting with a small pool size and allowing CNMeM to grow incrementally by calling `cudaMalloc()` is not a viable solution. An extra pool of size 100 MB would have sufficed for the example shown in Figure 7 if the two pools were merged. Table II (refer Section II-B for vDNN mode terminology) compares the aggregate peak memory consumed and the actual pool size required due to fragmentation. CNMeM uses the best-fit heuristic and initial CNMeM pool size is equal to the aggregate peak memory (obtained by assuming no fragmentation). It can be seen that fragmentation causes the memory requirement to be 300-500 MB above the expected aggregate peak.

Starting at a small pool size and allowing CNMeM to grow has a problem even if the new pool is merged with the older pool. It can shown that, there are sequences of (de)allocation for which starting CNMeM with a small pool size and allowing it to grow incrementally till some fixed maximum value will not be able to serve memory requests due to fragmentation. Alternatively, starting with that fixed maximum pool size could in fact meet requirements.

Table II
EXPECTED AND ACTUAL GPU MEMORY CONSUMED IN ALEXNET(128) AND VGG-16(16)

	Alexnet (128)		**VGG-16 (16)**	
vDNN mode	Expected (MB)	Actual (MB)	Expected (MB)	Actual (MB)
dyn	1188	1803	1853	2423
conv(p)	1188	1844	1610	2231
conv(m)	710	1121	1323	1893
all(p)	1113	1769	1414	2035
conv(m)	667	987	1323	1697

There are sequences of allocation/release for which the required pool size is strictly more than the aggregate peak memory consumed by the sequence, rendering fragmentation unavoidable. Here, we assume that data cannot be moved after allocating. One such sequence is as follows.

$$a_{11}, a_{12}, a_{13}, a_{14}, d_1, a_{22}, a_{23}, a_{24}, d_2, a_{33}, a_{34}, d_3, a_{44}, d_4$$

where a_{ij} represents that an allocation request is made before d_i and after $d_{(i-1)}$, and it is to be freed during d_j. We say that a_{ij} is allocated in block i. For a given i, the size requested by a_{ij} is the same for all j. a_{1j} have size 1 unit for all j. Sizes of other allocate requests are set to value which ensures that just before d_j, the aggregate memory request sums up to 4 units, for every j. Sizes which satisfy those are a_{1j} = 1 unit, a_{2j} = 1/3 units, a_{3j} = 2/3 units, a_{4j} = 2 units, for all j. So, on d_1, block of size 1 unit is deallocated, on d_2, blocks of sizes 1, 1/3 units are deallocated. On d_3, blocks of sizes 1, 1/3, 2/3 units are deallocated. Just after d_2, if deallocated space is fragmented as 1 unit and 1/3 units instead of a single 4/3 units free block, then a_{33} and a_{34} cannot be served and same applies for d_3 and d_4. So, blocks a_{ij} for all $i \leq j$ should be contiguous just before d_j, for all j. It can be seen that the above property gets violated for one of j = 2, 3, 4.

This example shows that there is no algorithm which avoids fragmentation completely. CNMeM gets a pool of virtual addresses. Fragmentation happens at the virtual address level. If page tables could be modified, then data can be moved around efficiently at virtual address level without actually copying and fragmentation can be avoided completely. But there is no CUDA API, nor any easy way to modify Page Table entries within the GPU. When peak memory consumed is close to the total GPU memory capacity, this fragmentation might render it non-trainable. Also, this reduces the space available for other processes using the same GPUs. This raises the following problems (i) finding the initial pool size which could serve all memory

requests (ii) heuristics for memory management to reduce the amount of fragmentation.

C. High pinned memory requirement

vDNN uses pinned CPU memory to accelerate data transfer between CPU memory and GPU memory. Pinned memory is non-pageable and high pinned memory requirement is undesirable as cannot be used by CPU for any other purpose. In our experiments we found out that the pinned memory requirement by AlexNet, VGG-16 and ResNet-152 for batch size 128 are 453 MB, 7436 MB and 10193 MB respectively. It can be seen that the amount of memory consumed by networks have increased drastically over the years and can be as high as 10 GB. Such high amount of pinned memory reduces memory available for other processes. Previous work [15], which is an extension of vDNN, proposed hardware modifications to GPU which compress data on the fly while sending to CPU memory and decompress while receiving from CPU memory. While this reduces the data to offload and reduces pinned memory, it is hard to implement due to architectural changes required in hardware. As stated in that paper, the data transferred to CPU are outputs of ReLU units and has a huge scope for compression due to the presence of lots of zeros. Only one core of CPU is used during training process to execute vDNN and all others are idle during the training process. The other cores could be used to compress the data in the pinned memory. Even compressing a part of data will reduce memory considerably.

IV. Design and implementation

This section describes details of the three main solution components of vDNN++: (i) improved asynchronous transfer (ii) heuristics to address fragmentation (iii) usage of compression to reduce pinned main memory footprint.

A. Improved Asynchronous Memory Transfer

The main design idea of vDNN++ is as shown in Figure 8 — commencing computation of layer operations without stalling.

Forward propagation. During forward propagation, layer$_{(n+1)}$'s computation is started as soon as layer$_{(n)}$'s computation is completed irrespective of the completion of offload of layer$_{(n)}$. The allowed peak memory of vDNN++ is maintained at the same level as consumed by vDNN for sake of comparison. If at some point, memory request made exceeds the desired peak memory, computation is stalled till memory is made available due to completion of offload of required number of previous layers. Release of offloaded layer$_{(n)}$'s memory on GPU is done only after the completion of both offload and computation of layer$_{(n)}$.

Backward Propagation. During backward propagation, if layer$_{(n)}$'s prefetch is decided to be overlapped with layer$_{(m)}$'s computation , then space for layer$_{(n)}$'s X (Refer Figure 1 for terminology of "X") is allocated and memory transfer is initiated just before start of computation of layer$_{(m)}$. After completion of layer$_{(m)}$'s computation, layer$_{(m-1)}$'s computation is started without waiting for prefetch to be completed. Computation is stalled only if prefetch of input of the layer to be computed has not been completed yet.

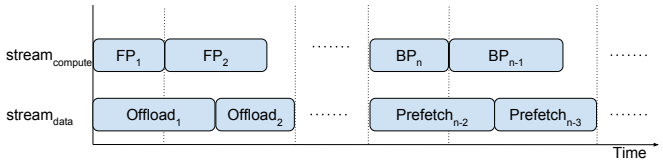

Figure 8. Working of vDNN++. In forward propagation, computation of layer$_{(2)}$ is started as soon as layer$_{(1)}$'s computation is completed, without waiting for offload of layer$_{(1)}$'s input. In backward propagation, layer$_{(n-1)}$'s computation is started as soon as layer$_{(n)}$'s computation is done without waiting for overlapped prefetch of layer$_{(n-2)}$ to complete.

As part of its implementation, vDNN++ uses two CUDA streams: (i) stream$_{compute}$ used by cuDNN [21] (Nvidia CUDA neural network library) to handle and sequence forward and backward propagation across network layers (ii) stream$_{memory}$ to manage the memory allocation/release, offload and prefetch. We modified the default vDNN implementation to minimize the computation stalling issue described earlier.

Memory Offload. If a layer (say n) has been chosen for offloading, then in stream$_{memory}$, non-blocking memory transfer of this layer's X to the pinned memory is issued via `cudaMemcpyAsync()`. This is followed by registering an event — `(event_offload_done[n])` in stream$_{memory}$, the occurrence of which denotes that the offload has been completed. After completion of computation of layer$_{(n)}$ (ensured by calling `cudaStreamSynchronize(`stream$_{compute}$`)`), a separate thread is launched which releases layer$_{(n)}$'s X after the occurrence of the `cudaEvent` launched previously `(event_offload_done[n])`. To keep track if the layer$_{(n)}$'s X is released, a semaphore `(sem_sync_offload)` is used and the thread increments the semaphore. This thread is detached so that thread resources are freed up as soon as it completes without calling `pthread_join()`. After launching this thread, the main thread moves on to the execution of subsequent layers, repeating the whole process. Finally, after forward propagation of all layers, main thread decrements the semaphore to wait for the release of all offloaded layers. This ensures that all layers chosen to be offloaded are offloaded and released from GPU memory before start of backward propagation. Also, call to `cnmemMalloc()` has to wait if serving the request exceeds the desired memory consumption. During a call to `cnmemMalloc()`,

if memory is not available, the call is blocked using a conditional statement and every time `cnmemFree()` is called, it broadcasts on this conditional statement. This helps to start computation as soon as memory is available.

Memory Prefetch. If prefetch of layer$_{(n)}$'s X is chosen to be overlapped with layer$_{(m)}$'s computation, GPU memory is allocated for layer$_{(n)}$'s X. Just after call to `cudaMemcpyAsync()` to move layer$_{(n)}$'s X from host to device, an event `(event_prefetch_done[n])` is registered in the stream$_{memory}$. This event occurs just after prefetch of layer$_{(n)}$ is done. Just before start of layer$_{(i)}$'s computation, if its X was offloaded, then the program is synchronized with the `cudaEvent (event_prefetch_done[i])` to ensure that layer$_{(i)}$'s X is available in the GPU before starting the computation of that layer.

B. Heuristics to Reduce Memory Fragmentation

The change of order of allocation/free which does not affect training, can impact the amount of fragmentation. For example, in backward propagation, space for input of layer to be prefetched could be allocated before/after allocating space for gradient of input of layer to be back-propagated.

Forward propagation requires much less memory compared to backward propagation (Section III-A). Same is observed in fragmentation. Even though free space is fragmented, forward propagation never required any more memory than aggregate peak memory consumed by the network in the experiments.

We use these insights to reduce the fragmentation at the end of forward propagation. We propose a heuristic– non-offloaded high end allocation, which works as follows. CNMeM follows best-fit heuristic, in which for an allocation request, smallest free block of size larger than requested size is broken into "used block" and a smaller "free block". The "used block" occupies the lower address. We propose an allocation scheme as follows. For all allocations except for layers chosen to be offloaded, we follow the default best-fit algorithm. For allocation of layer's input which is chosen to be offloaded to CPU, we allocate it at the highest available address. Memory gets assigned at the two ends of the memory range, with layers to be offloaded (temporary data) on the higher memory range. This ensure that there is a single used block (lower address) and a single free block (higher address) at the end of forward propagation. Further, according to this scheme, layers' data will be in an order such that during back propagation, layers' data are freed from higher to lower address. Therefore, as back propagation proceeds, no fragmentation is caused by non-offloaded layers. Only the (de)allocations for prefetched memory and for gradients of layer inputs leads to fragmentation.

To quantify the effect of heuristics to reduce fragmentation, smallest size of memory pool which could serve the sequence of requests is required. We approximate this using the following method. Given a sequence of requests and initial pool size, we simulate the heuristic for that sequence. If at some allocation request, the pool is not able to serve it, we start the simulation again with an extra initial pool size equal to ((request size at failure) - (size of largest free block)). Intuitively, this gives an approximation of smallest size of pool. Clearly, the pool size obtained is a sufficient pool size to serve the request in all the iterations of training as the order of request remains the same. But, there are requests which can be served with a smaller pool size but not with a larger pool size. For example, consider two pool sizes of 9 units and 11 units, respectively. Consider the sequence,

$$A_1(5), A_2(2), A_3(1), D_2, A_4(1), D_1, D_3, A_5(8)$$

where A$_i$(j) denotes the ith allocation request of j units and D(i) denotes freeing up memory of the ith request. It is easy to see that for this sequence, the best-fit algorithm will be able to serve with 9 units initial pool but not with 11 units initial pool. This shows that the above algorithm to find smaller size of memory pool might fail and report that memory required exceeds GPU capacity while actually it could have served the sequence. Increasing the pool size one byte at a time when the pool fails to serve the sequence takes a large amount of time and is not practical.

C. Compression to reduce Pinned Memory requirements

We use Zero-value compression [15] to reduce pinned memory requirements of the CPU. Parallel compression on multiple cores is implemented with multiple threads. These threads compress different parts of the offloaded network layer data in parallel. While compressing on CPU, the compressed size is not known without counting the number of zeros. Counting number of zeros and allocating it requires almost twice the time taken by just compression. Instead, space allocation is discretized with a fixed fraction (say FRAC) of layer's size chunks. Space is allocated in chunks of (original layer size) / (FRAC * (number of threads for compression)) when more memory is required for compressed data. For decompression, output size is known and the above problem does not occur.

V. Evaluation

Our experimental setup had two GPUs — NVIDIA GTX 970 and GTX 1080Ti — having 4 GB and 11 GB GPU memory respectively. Only one of them was used at any point of time. The machine with GTX 970 had 3.4 GHz Intel i7-3770 CPU (8 cores) and 32 GB CPU memory, CUDA 9.1 and CuDNN 7. The machine with GTX 1080Ti had 3.5 GHz Intel Core i5-7400 (8 cores) and 32 GB CPU memory, CUDA 9.2 and CuDNN 7.1.

As baseline for comparison, we use "base" memory manager which allocates memory for the outputs of all layers, network parameters and its gradients. It allocates reusable memory for gradients of layers, as at any time, only two

Table III
NETWORK ARCHITECTURE OF CONSTRUCTED NETWORK. CONV_2 HAS 1X1 CONV (FAST COMPUTE) INTERSPERSED AMONG 5X5 CONV (LONG COMPUTE)

Layer Name	Output size	Filter size
CONV_1	224 × 224 × 64	3×3, 64
CONV_2	224 × 224 × 64	$\begin{bmatrix} 1\times1 & 64 \\ 5\times5 & 64 \end{bmatrix} \times 5$
Max Pool_1	1×1 × 64	224×224
CONV_3	1×1×3	1×1, 3
Fully Connected_1	2	2 output neurons
Softmax_1	2	N.A.

Table IV
EXPERIMENTS ON VGG-116, VGG-191 ON NVIDIA 1080 TI - EXECUTION TIME OF ONE ITERATION OF TRAINING STEP IN MS

	VGG-116 (32)		**VGG-191 (32)**	
Mode	**vDNN**	**vDNN++**	**vDNN**	**vDNN++**
dyn	2717	**2468**	5139	**5097**
conv(p)	3055	**3018**	5139	**5097**
all(p)	3170	**3129**	5244	**5209**
aconv(p)	2717	**2468**	Out of Memory	

chunks of memory is required for gradient of layers. All allocations are on GPU DRAM with no transfer to CPU memory. This manager is exactly same as the one described in Section IV-A of [10]. We implemented a simple linear neural network library using CUDA and CuDNN for the baseline manager, with interface similar to nn.Sequential class of PyTorch [5]. We implemented vDNN++ and vDNN (its code hasn't been released yet) using this library.

A. Improvement in training time of different networks

We experimented on networks shown in Figure 9. Further, to demonstrate that vDNN++ enables training of networks which do not fit in memory while performing better than vDNN, we experimented on VGG-116 and VGG-191 (Table IV) (obtained by adding 20 and 35 CONV layers respectively to each CONV layer group of VGG-16). They require about 20GB and 30GB respectively. In each Figure/Table, quantity inside (.) in NetworkName(.) represents batch-size. Refer Section II-B for terminology on (m), (p), all, conv, aconv, dyn. Figure 9 shows that performance of vDNN++ is at least as good as vDNN. The best improvement can be seen in case of ALL offload due to the POOL layers. This is a special case of "fast compute" layers interspersed among "long compute" layers. Offload of "fast compute" layer's input is covered by computation time of further "long compute" layers. To better demonstrate this, we constructed a network with structure given by Table III. With Nvidia 1080Ti, input size of 224x224x3 and batch size 256, base(m) takes 2091ms, vDNN(conv(m)) takes 2491ms, vDNN++(conv(m)) takes 2261ms. This difference between vDNN and vDNN++ increases when the repetitions in CONV_2 block of Table III is increased. Since our implementation supports only linear networks currently, we couldn't demonstrate on networks such as ResNet [7] and GoogleNet [19], which have skip/branch connections. But such networks have alternate "fast" and "long" compute layers, similar to that of our constructed network. Hence, idea of vDNN++ could be expected to give similar performance improvement. In the case of alternate CONV offload, vDNN++ has the potential to perform much better than vDNN. In this case, to avoid stalling, it is enough for offload time of a layer to be compensated by two CONV layers' computation time in vDNN++, whereas in vDNN, each offload has to be compensated by that CONV layer's computation due to synchronization. A significant improvement of 250ms out of 2700ms could be observed in Table IV in aconv(p) of VGG-116.

There are situations where vDNN++ might not provide significant improvement over vDNN. One such case is when each layer's computation is slower than transfer. Total time of execution taken by both vDNN and vDNN++ will be the sum of computation times of individual layers. This is the situation in GTX 970 conv(p) or conv(m). Similar situation arises when all layer's computation is faster than transfer.

The order of allocation of memory in vDNN and vDNN++ differs during forward propagation. This leads to different levels of fragmentation, which sometimes can lead to vDNN++ not being able to work with same amount of memory as given to vDNN. This can be seen in 1080Ti VGG-16(64) aconv(p). In such situations, adding a little more memory can make them work.

Overall, we obtain a relative improvement of up to 60% (AlexNet(128) all(p) in Nvidia 970 and AlexNet(256) all(m) in Nvidia 1080Ti), where relative improvement is percentage reduction in training time relative to additional time taken by vDNN compared to base manager.

B. Reduction in fragmentation

Table V compares two heuristics: (i) default CNMeM best-fit heuristic (ii) non-offloaded high end allocation heuristic (Section IV-B) and two orders of allocating space: allocating overlapped prefetched layer (i) before (prefetch-first) or (ii) after (derivative-first) allocating space for input gradients for layer to currently compute. The algorithm for finding smallest pool size (Section IV-B) performs much better than allocating aggregate peak memory as initial pool size and allowing CNMeM to grow. Here, the maximum difference of actual peak size from aggregate peak size is about 343MB for AlexNet(256) in case of former. It is 656MB even for AlexNet(128) for the latter. (Comparison is done for same order of allocation, namely prefetch-first and best-fit heuristic).

From the results reported in Table V, it cannot be concluded that any one ordering or heuristic is better than the other. A better way to reduce memory consumption would be to initially simulate different orders and algorithms for allocation and choose the order/algorithm which consumes

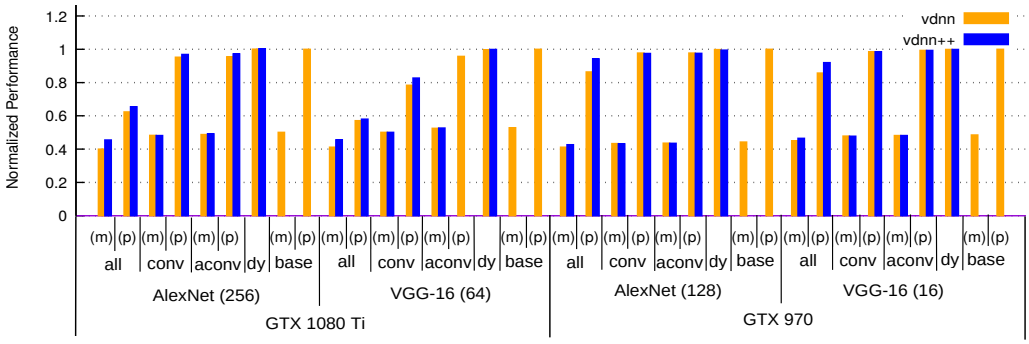

Figure 9. Performance comparison normalized with base memory manager using performance optimal CONV algorithm setup

Table V
COMPARISON OF AGGREGATE PEAK MEMORY CONSUMED AND ACTUAL CNMEM POOL SIZE REQUIRED (CALCULATED BY POOL SIZE CALCULATION ALGORITHM STATED IN SECTION IV-B) FOR BEST-FIT AND NON-OFFLOADED HIGH END ALLOCATION HEURISTIC (HIGH-END).

Network	Mode	aggregate MB)	prefetch-first best-fit (MB)	derivative-first best-fit (MB)	prefetch-first high-end (MB)	derivative-first high-end (MB)
AlexNet (256)	dyn	1934	2011	2011	**1974**	**1974**
	conv(p)	1934	**2101**	2185	2177	2114
	conv(m)	1184	1452	1476	**1427**	1465
	all(p)	1783	2126	2050	2037	**2020**
	all(m)	1100	1375	**1361**	1570	1362
	aconv(p)	1934	2162	2162	**2121**	2183
	aconv(m)	1184	**1374**	1379	1472	1534
VGG-16(64)	dyn	7243	**7243**	**7243**	7439	7439
	conv(p)	7243	7831	7439	8223	**7372**
	conv(m)	3715	4302	4107	4106	**3751**
	all(p)	7243	**7341**	8105	**7341**	7519
	all(m)	3715	**3812**	4795	**3812**	4795
	aconv(p)	7243	8027	7829	**7635**	7831
	aconv(m)	4010	4155	**4073**	4302	4106

the least GPU memory. In non-offloaded high end allocation heuristic, fragmentation is not caused by the non-offloaded layers (Section IV-B). Allocaction/free of gradients and prefetch of offloaded layers cause the fragmentation and increase memory consumption. Better algorithms for reducing fragmentation in back propagation together with non-offload high end allocation heuristic might reduce fragmentation considerably compared to other options.

C. Scope for compression

vDNN++ takes 377ms for AlexNet(128) with all(m) choice, which is comparable with the (de)compression time in CPU (Figure 10). This shows that there is scope for compression of offloaded layers on the CPU. If all the layers are compressed, we obtain an average of 36% savings in pinned memory. For VGG-16(16), this goes to 67%. It might not be feasible to compress all the layers on the CPU without affecting the performance. During training, occasionally checking the activation density (fraction of non-zero elements) [15] of each layer and choosing layers with low density to compress could reduce degradation of performance due to compression.

Only about 150%-180% of the CPU out of 8 cores (i.e. out of 800%) was being used at any point of time for (de)compression. This shows that memory, as well as processor is available for other processes.

VI. RELATED WORK

vDNN [10] is the first work that characterized the usage of GPU memory by different CNNs and explored the possibility of transferring layer-wise data across CPU and GPU memory. Our work highlights the shortcomings of vDNN and propose optimization over it. [22] distributes the DNN's computation over multiple nodes equipped with specialized processors like GPU, FPGA etc. whereas we try to optimize the resources of a single GPU for DNN's computation. [23] developed a single benchmark suite for evaluating the performance of DNNs for different processors like CPU, GPU and other accelerators. [24] rely on GPU with special memory called High Bandwidth Memory(HBM) for speeding up DNN training. The requirement of hardware with special capability limits their solution. SuperNeuron's [11] solution is similar to vDNN, however, they suggest several optimization for non-linear neural networks. Our work currently supports linear networks but it can be extended to non-linear networks. [25] utilizes dis-aggregated memory architecture to expand the memory capacity of the hardware accelerators. [26] characterize the memory access pattern of DNN layers

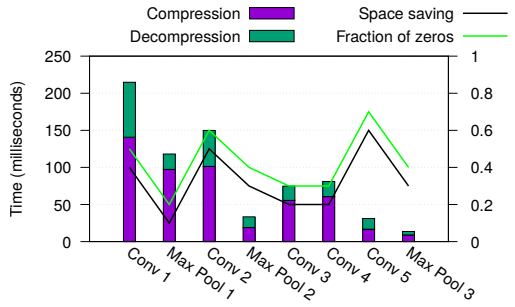

Figure 10. Results of Compression and Decompression for AlexNet(128) at a typical average activation density during training

and also explore the implication of hardware architecture on the performance of DNN. [12] uses similar approach as vDNN and they also proposed a kernel fusion technique where they fuse two or three cuDNN functions into one to avoid individual data transfers for these function. They also claim that they support training of those DNNs whose individual layer's memory requirement exceeds GPU memory. However, these optimization depend upon the plausibility of merging two kernels and likelihood of presence of highly memory intensive layers. [27] looks into the benefits of compressing the CNN layers and proposes a new hardware for compression. moDNN [13] is an extension of vDNN and proposes heuristic based offload of DNN layers to CPU memory. [15] makes use of hardware based compression of data to reduce the CPU-GPU memory transfer but it suggests changes in GPU hardware which is not always viable.

VII. Conclusion and Future Work

The size of DNNs are growing and current GPUs' memory capacity is insufficient to accomodate all the data required by them. This problem requires GPU memory management solution for training DNNs which doesn't fit in GPU memory. vDNN [10] is one such solution which proposed the idea of offloading the data from GPU memory to CPU memory which is not needed by layer being processed and prefetching it back when it's required. In our work, we highlighted the drawbacks of vDNN and presented vDNN++ which mitigates them. We characterized the cases where vDNN++ could perform much better than vDNN. vDNN++ is able to provide a relative improvement of up to 60% in training time. We showed that CPU side compression helps reduce pinned memory requirements and higher compression rates could be achieved by trading off with training time. We also proposed and evaluated multiple heuristics to reduce memory fragmentation.

The current implementation of vDNN++ supports only linear DNN but we plan to extend its functionality for non-linear networks too. We would also like to integrate compression with vDNN++ which dynamically chooses layers to compress depending on the activation density.

References

[1] R. Collobert, J. Weston, and L. Bottou, "Natural Language Processing (Almost) from Scratch," *Journal of Machine Learning Research*, vol. 12, no. Aug, pp. 2493–2537, 2011.

[2] A. Graves and J. Schmidhuber, "Framewise phoneme classification with bidirectional LSTM and other neural network architectures," *Neural Networks*, vol. 18, no. 5-6, pp. 602–610, 2005.

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet Classification with Deep Convolutional Neural Networks," in *Advances in neural information processing systems*, 2012.

[4] (2018) Tensorflow. [Online]. Available: https://www.tensor flow.org/

[5] (2018) Pytorch. [Online]. Available: http://pytorch.org/

[6] (2018) Imagenet. [Online]. Available: http://www.image-net.org/

[7] H. Kaiming, Z. Xiangyu, R. Shaoqing, and J. Sun, "Deep Residual Learning for Image Recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016.

[8] (2018) List of Nvidia graphics processing units. [Online]. Available: https://en.wikipedia.org/wiki /List_of_Nvidia_graphics_processing_units

[9] A. Canziani, A. Paszke, and E. Culurciello, "An Analysis of Deep Neural Network Models for Practical Applications," *CoRR*, vol. abs/1605.07678, 2016.

[10] M. Rhu, N. Gimelshein, and J. C. et al, "vDNN: Virtualized Deep Neural Networks for Scalable, Memory-Efficient Neural Network Design," in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2016, pp. 1–13.

[11] L. Wang, J. Ye, and Y. Zhao et al, "Superneurons: Dynamic GPU Memory Management for Training Deep Neural Networks," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2018, pp. 41–53.

[12] Y. Ito, R. Matsumiya, and T. Endo, "ooc_cuDNN: Accommodating convolutional neural networks over GPU memory capacity," in *Proceedings of the IEEE International Conference on Big Data*, Dec 2017, pp. 183–192.

[13] X. Chen, D. Z. Chen, and X. S. Hu, "moDNN: memory optimal dnn training on gpus," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2018, pp. 13–18.

[14] (2018) cnmem. [Online]. Available: https://github.com/ NVIDIA/cnmem

[15] M. Rhu, M. O'Connor, and N. C. et. al., "Compressing DMA Engine: Leveraging Activation Sparsity for Training Deep Neural Networks," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, 2018, pp. 78–91.

[16] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[17] S. Venugopalan, H. Xu, and J. Donahue et al, "Translating Videos to Natural Language Using Deep Recurrent Neural Networks," in *Proceedings of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2015, pp. 1494–1504.

[18] *NVIDIA CUDA Programming Guide*. NVIDIA, 2018. [Online]. Available: https://developer.download.nvidia.com/

[19] C. Szegedy, W. Liu, and J. Yangqing et al, "Going Deeper with Convolutions," in *Computer Vision and Pattern Recognition (CVPR)*, 2015.

[20] (2018) Cuda. [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[21] (2018) Nvidia cudnn. [Online]. Available: https://developer.nvidia.com/cudnn

[22] J.Park, H.Sharma, and D. Mahajan et al, "Scale-out Acceleration for Machine Learning," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. ACM, 2017, pp. 367–381.

[23] J.-H. Tao, Z.-D. Du, and Q. Guo et al, "BenchIP: Benchmarking Intelligence Processors," *Journal of Computer Science and Technology*, vol. 33, no. 1, pp. 1–23, Jan 2018.

[24] M. Zhu, Y. Zhuo, and C. Wang et al, "Performance Evaluation and Optimization of HBM-Enabled GPU for Data-Intensive Applications," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 5, pp. 831–840, May 2018.

[25] Y. Kwon and M. Rhu, "A Case for Memory-Centric HPC System Architecture for Training Deep Neural Networks," *IEEE Computer Architecture Letters*, vol. 17, no. 2, pp. 134–138, July 2018.

[26] S. Dong, X. Gong, and Y. Sun et al, "Characterizing the Microarchitectural Implications of a Convolutional Neural Network (CNN) Execution on GPUs," in *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, 2018, pp. 96–106.

[27] J. Kim, M. Lee, and B. Kim et al, "Data Compression Hardware of the ReLu Output in Convolution Neural Network," *Advanced Science and Technology Letters*, vol. 146, 2017.